



**Università
di Genova**

DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Distributed Flexible Clustering

Abdul Hamid Afghan

Master's Thesis

Università di Genova, DIBRIS
Via Dodecaneso, 35 16146 Genova, Italy
<https://www.dibris.unige.it/>



Computer Science MSc
Software Security and Engineering Curriculum

Distributed Flexible Clustering

Abdul Hamid Afghan

Advisor: Matteo Dell'Amico

Examiner: Daniele D'agostino

September, 2026

Abstract

Clustering large datasets becomes difficult when their computational or memory requirements exceed the capacity of a single machine. This is particularly challenging for density-based clustering, where neighborhood information and global connectivity are needed to identify clusters and noise. FISHDBC provides a scalable approximation of HDBSCAN* by combining HNSW-based neighbor discovery with an approximate minimum spanning tree, but its original design is limited to single-machine execution.

This thesis investigates a distributed adaptation of FISHDBC. A sampled global HNSW is used to organize and partition the dataset, while multiple Workers perform local neighborhood discovery and graph processing. A central Leader combines the reduced local information, reconstructs approximate global core distances, recomputes mutual-reachability weights, and builds the global minimum spanning tree used for hierarchical clustering.

The proposed implementation is evaluated against centralized FISHDBC and HDBSCAN* under both controlled and application-specific workloads. The experiments examine clustering agreement, the effects of sampling and replication, runtime scalability, memory usage, and the computational cost of flexible distance functions. The results show that the distributed approach preserves the expected clustering structure in the evaluated configurations and improves scalability relative to centralized FISHDBC as the dataset size increases. The evaluation also identifies the centralized Indexer, particularly data routing and graph processing, as the main scalability bottleneck of the current prototype.

Acknowledgements

I would like to express my sincere gratitude to Prof. Matteo Dell’Amico for supervising this thesis and for his guidance throughout the development of this work.

I am also grateful to Prof. Marina Ribaudo for her support during the early stages of this journey, and to Prof. Giorgio Delzanno for providing access to the cluster infrastructure used to carry out the experimental evaluation.

I would also like to thank Prof. Daniele D’agostino for reviewing this thesis and for his valuable comments and suggestions.

Finally, I would like to express my deepest gratitude to my wife, Abida Nabizada, for her constant support, patience, and encouragement throughout this journey.

Table of Contents

Chapter 1	Introduction	9
1.1	Context and Motivation	10
1.2	Problem Statement	11
1.3	Proposed Distributed Approach	12
1.4	Research Questions and Objectives	13
1.5	Contributions	15
1.6	Use of LLMs	16
1.7	Thesis Organization	16
Chapter 2	Background	17
2.1	Clustering and Density-Based Clustering	17
2.2	DBSCAN	18
2.3	HDBSCAN*	20
2.4	Hierarchical Navigable Small World Graphs	23
2.5	FISHDBC	25
2.6	Graph Partitioning	28
2.7	Distributed Computing Concepts	30
2.8	Container Orchestration with Docker Swarm	32
Chapter 3	Related Work	35
3.1	Distributed Density-Based Clustering	35

3.1.1	Challenges of Distributing Density-Based Clustering	36
3.1.2	Partition-and-Merge Approaches	38
3.1.3	Graph-Based and Approximate Distributed Clustering	39
3.1.4	Limitations with Respect to FISHDBC	41
3.2	Parallel FISHDBC	42
3.3	Distributed HNSW and Pyramid	44
3.4	Positioning of This Work	48
Chapter 4	Design and Implementation	50
4.1	Design Goals	51
4.2	System Architecture	51
4.3	Global Sampling and Partitioning	52
4.4	Indexer Graph Reduction	54
4.5	Worker-Side Local Processing	56
4.6	Global Core-Distance Reconstruction	58
4.7	Mutual-Reachability Reconstruction	59
4.8	Global Minimum Spanning Tree Construction	60
4.9	Replication and Boundary Handling	62
4.10	Implementation	62
4.10.1	Software Components	63
4.10.2	Communication and Data Exchange	63
4.10.3	Configuration	65
4.10.4	Docker Swarm Deployment	66
4.10.5	Implementation Considerations	67
4.11	Design Trade-offs	69
4.12	Chapter Summary	70
Chapter 5	Evaluation	72

5.1	Evaluation Goals and Research Questions	73
5.2	Experimental Setup	74
5.2.1	Hardware and Deployment	74
5.2.2	Reference Configuration	75
5.2.3	Repetition and Measurement Procedure	75
5.3	Datasets	76
5.3.1	Synthetic Blob Dataset	76
5.3.2	Music Dataset and Arbitrary Distance Function	77
5.4	Compared Methods	78
5.5	Evaluation Metrics	79
5.5.1	Clustering Quality	79
5.5.2	Execution Time and Scalability	79
5.5.3	Graph Reconstruction	80
5.5.4	Distance Evaluations	81
5.5.5	Communication and Resource Usage	81
5.6	Synthetic Blob Experiments	82
5.6.1	B1: Reference Quality, Runtime, and Memory	82
5.6.2	B2: Sampling-Rate Sensitivity	87
5.6.3	B3: Replication and Boundary Coverage	90
5.6.4	B4: Scalability and Bottleneck Analysis	94
5.7	MusicBrainz Experiments	101
5.7.1	M1: Reference Computational Behavior	101
5.7.2	M2: Scalability with Jaro–Winkler Distance	106
5.8	Chapter Summary	114
Chapter 6 Conclusion		116
6.1	Summary of the Work	116
6.2	Answers to the Research Questions	117

6.3	Limitations	118
6.4	Future Work	119
6.5	Final Remarks	120

Chapter 1

Introduction

In Exploratory Data Analysis (EDA), clustering is widely used to identify structure in large and complex datasets by grouping similar data points without requiring predefined labels [Jai10]. Rapid technological progress has led to rapidly increasing data volumes, making large-scale data clustering an increasingly challenging task [He+10]. At sufficiently large scales, processing can require substantial computational resources and exceed the memory capacity of a single machine. This is particularly challenging for density-based clustering methods, where cluster formation depends on neighborhood information and connectivity among data points [MHA17].

To address these limitations, several scalable density-based clustering approaches have been proposed. In particular, FISHDBC [Del19] mitigates the scalability constraints of HDBSCAN* by offering an incremental, approximate alternative that avoids exhaustive pairwise distance calculations. FISHDBC combines Hierarchical Navigable Small World (HNSW) graphs for approximate neighbor discovery with approximate minimum spanning tree (MST) maintenance, enabling density-based clustering to scale efficiently while maintaining competitive clustering quality. However, its original design assumes single-machine execution.

This work explores extending FISHDBC to a distributed environment by parallelizing computation across multiple worker nodes while retaining the global structure essential for hierarchical density-based clustering.

This chapter establishes the motivation behind this work, formalizes the core problem, outlines the proposed approach, and summarizes the principal objectives and contributions of this thesis.

1.1 Context and Motivation

Clustering is an important technique in exploratory data analysis because it allows large collections of data to be organized according to similarity without requiring predefined classes. As datasets increase in size, however, the computational cost of clustering also increases [He+10]. This is particularly relevant when the algorithm depends on repeated distance evaluations or on data structures that must remain available in memory throughout the computation.

Density-based clustering methods introduce additional challenges at large scale because the assignment of a point to a cluster depends on the structure of its surrounding neighborhood. In HDBSCAN*, for example, the density of a point is represented through its core distance, while the relationship between pairs of points is expressed using mutual-reachability distance [MHA17]. The resulting hierarchy is derived from a minimum spanning tree built over these weighted relationships.

Neighborhood information and global connectivity make density-based clustering difficult to distribute directly. Simply dividing a dataset into multiple partitions may place points that are close in the original space on different workers. As a result, a worker may not observe some of the true nearest neighbors of its local points. This can affect core-distance estimates [Ank+99] and may also remove connections that are important for identifying clusters that extend across partition boundaries [Hua+17].

FISHDBC addresses some of the scalability limitations of hierarchical density-based clustering by avoiding the computation of the complete pairwise distance matrix. Instead, it uses a Hierarchical Navigable Small World (HNSW) graph to discover a subset of useful neighborhood relationships and maintains an approximate minimum spanning tree that can be updated as new points are added [Del19]. This allows FISHDBC to support incremental clustering and arbitrary distance functions while reducing the amount of distance computation required in comparison with exhaustive approaches.

The efficiency of FISHDBC is closely related to the separation between neighbor discovery and maintenance of the clustering structure. Its internal state includes the HNSW structure, the nearest neighbors discovered for each point, candidate edges, and the current approximate minimum spanning tree [Del19]. These structures are updated incrementally as new data are processed.

Despite these advantages, the original FISHDBC design assumes that these structures are maintained within a single execution environment [Del19]. When the dataset exceeds the practical capacity of a single machine, distributing the data introduces a new problem: each worker has only a partial view of the dataset, while the clustering still depends on neighborhood and connectivity information that may span several workers.

This motivates the investigation carried out in this thesis. The goal is not only to divide the dataset among multiple workers, but to determine how local computations can be combined so that sufficient global information is retained for hierarchical density-based clustering. In particular, the distributed design must account for neighbor relationships that cross partition boundaries, approximate global core distances from information obtained by different workers, and preserve enough connectivity information to construct a global minimum spanning tree.

1.2 Problem Statement

Although FISHDBC reduces the computational cost of hierarchical density-based clustering, its original design assumes that the dataset and the internal clustering structures are maintained within a single machine. This becomes a limitation when the dataset grows beyond the available memory or when the computational cost of neighborhood discovery becomes too high for a single execution environment.

A straightforward way to distribute the workload would be to partition the dataset and let each worker process one subset independently. However, this approach does not preserve the global neighborhood structure required by density-based clustering. Points that are close in the original data space may be assigned to different workers, meaning that a worker has only a partial view of the neighborhood of its local points. Since FISHDBC relies on the discovered nearest neighbors of each point to estimate local density and construct candidate edges, incomplete local neighborhoods can affect the clustering result [Del19].

The problem becomes more significant when computing core distances. In HDBSCAN* and FISHDBC, the core distance of a point is the distance to its $MinPts$ -th nearest neighbor [MHA17; Del19]. The mutual-reachability distance between two points a and b is defined as

$$d_{\text{mreach}}(a, b) = \max \{c(a), c(b), d(a, b)\}, \quad (1.1)$$

where $c(a)$ and $c(b)$ denote the core distances of the two points and $d(a, b)$ is their original pairwise distance. If some of the nearest neighbors of a point are located in another partition, the core distance computed from only the local data may not represent the corresponding value in the complete dataset. Since the mutual-reachability distance depends directly on both core distances, such local errors can change the weights of the edges used to construct the clustering hierarchy.

Another difficulty concerns connectivity between partitions. FISHDBC derives its hierarchical clustering from an approximate minimum spanning tree constructed from the

neighborhood relationships discovered during HNSW processing [Del19]. If each worker constructs a structure using only its own partition, the resulting local trees do not necessarily contain the edges required to connect points belonging to different workers. Consequently, a cluster that spans multiple partitions may become fragmented if cross-partition connections are not retained.

A distributed version of FISHDBC must therefore balance two competing requirements. On one side, the dataset and the computational workload should be divided across multiple workers in order to reduce the dependency on the resources of a single machine. On the other side, sufficient information must be preserved across partitions to approximate the global neighborhoods, core distances, and connectivity required to construct the final hierarchical clustering.

The problem addressed in this thesis is therefore not simply how to execute FISHDBC on multiple machines, but how to distribute its computational workload while retaining enough global information to reconstruct the density and connectivity relationships that would otherwise be available in a centralized execution. The proposed approach addresses this problem by separating local computation from global reconstruction, allowing workers to process subsets of the dataset while a global coordination stage combines the information required for the final clustering.

1.3 Proposed Distributed Approach

The proposed approach distributes the computational workload of FISHDBC across multiple workers while preserving the global information required for hierarchical density-based clustering. Instead of allowing each worker to cluster its local partition independently, the system separates local computation from global reconstruction. Workers process subsets of the dataset, while a central coordination stage combines their results to recover the neighborhood and connectivity information needed for the final clustering.

The current prototype distributes the neighborhood-discovery and local graph-processing stages rather than the complete data-ingestion pipeline. The Indexer still accesses the complete input dataset in order to perform sampling, partition-based routing, and distribution of the data to the Workers. The computationally expensive local HNSW construction, neighborhood discovery, and graph reduction are then performed independently across multiple machines.

The first step is to construct a smaller global HNSW structure from a sample of the complete dataset. This structure is used to capture the general organization of the data and to guide the partitioning process. The idea is inspired by Pyramid [Den+19], which uses a meta-HNSW to organize large datasets into related sub-datasets for distributed

similarity search. In this thesis, a similar principle is adapted for hierarchical density-based clustering, where the objective is not only to distribute search operations, but also to preserve enough information for global density estimation and cluster connectivity.

The sampled HNSW is partitioned so that similar points are more likely to be assigned to the same worker. The remaining data points are then routed to worker partitions according to their proximity to the global structure. To reduce the loss of neighborhood information near partition boundaries, when the replication factor is greater than one, each routed point can be assigned to multiple nearby worker partitions. This increases the overlap between local partitions and improves the possibility of discovering neighborhood relationships that would otherwise be separated by partitioning.

Each worker constructs a local HNSW using the points assigned to it and extracts local neighborhood information together with raw-distance graph edges. These local computations reduce the amount of work that must be performed on a single machine and allow the expensive neighborhood-discovery stage to be distributed across several workers. However, the local results are not considered final clustering structures because each worker has only a partial view of the complete dataset.

The information produced by the workers is sent to a central leader. Before transmission, local core-distance estimates are used to compute local mutual-reachability weights and to select a compact set of connectivity edges. These local weights are used only for graph reduction. The leader then combines the retained raw-distance edges from the workers and the indexer, reconstructs approximate global core distances, and recomputes mutual-reachability weights using the combined global information.

Unlike the original FISHDBC algorithm, the distributed prototype developed in this thesis does not preserve incremental insertion. It processes a fixed dataset through a staged batch workflow consisting of indexing, distributed worker processing, and global reconstruction.

Finally, the leader combines the local and cross-partition candidate edges, assigns mutual-reachability weights, and computes a global minimum spanning tree. The resulting tree is then used to derive the hierarchical clustering. In this way, the proposed approach distributes the most expensive local computations while retaining a global reconstruction stage for the density and connectivity information required by FISHDBC.

1.4 Research Questions and Objectives

The main objective of this thesis is to investigate whether the main principles of FISHDBC can be adapted to a distributed environment while preserving clustering quality and reducing the dependency on the computational and memory resources of a single machine. The proposed approach distributes neighborhood discovery and local graph construction

across multiple workers, while retaining a global reconstruction stage for the information required by hierarchical density-based clustering.

To evaluate whether this objective is achieved, the thesis is guided by the following research questions:

- **RQ1:** To what extent can the proposed distributed approach reproduce the clustering obtained by single-machine FISHDBC and HDBSCAN*?
- **RQ2:** How do sampling and replication affect the clustering quality of the distributed approach?
- **RQ3:** What computational trade-offs are introduced by distributing FISHDBC across multiple workers?

The first research question focuses on the quality of the clustering produced by the distributed approach. Since the proposed method divides the dataset across multiple workers and reconstructs global information from partial local views, the resulting clustering may differ from the centralized execution. Therefore, the distributed result is compared with both single-machine FISHDBC and HDBSCAN* to measure how closely the original cluster structure is preserved.

The second research question investigates the effect of two parameters introduced by the distributed architecture. Sampling controls the amount of information used to construct the global HNSW structure, while replication allows each routed point to be assigned to more than one nearby worker partition. Increasing replication creates overlap between partitions and may improve the recovery of cross-partition neighborhood relationships, at the cost of additional computation and communication.

The third research question considers the computational consequences of distribution. Dividing the workload among several workers can reduce the amount of data and computation handled locally by each machine, but it also introduces additional costs related to partitioning, replication, communication, and global reconstruction. The evaluation therefore examines the relationship between clustering quality and computational performance under different distributed configurations.

To address these research questions, this thesis pursues several concrete objectives. The proposed system is designed to partition large datasets according to their global structure, perform neighborhood discovery and graph construction locally on multiple workers, preserve useful information across partition boundaries, reconstruct approximate global core distances, and construct a global mutual-reachability minimum spanning tree for hierarchical cluster extraction. Finally, the distributed implementation is experimentally compared with centralized FISHDBC and HDBSCAN* under different sampling, replication, and dataset-size configurations.

1.5 Contributions

The main contribution of this thesis is the design and implementation of a distributed approach for FISHDBC-style hierarchical density-based clustering. The proposed solution distributes the expensive neighborhood-processing workload across multiple workers while preserving the global information required to reconstruct the final clustering.

The main contributions of this work are summarized as follows:

- **Distributed adaptation of the FISHDBC workflow.** The original FISHDBC algorithm is designed for single-machine execution. This thesis reorganizes its main processing stages so that data and computation can be distributed across multiple workers while retaining a global coordination stage.
- **Sampled global HNSW for data organization and partitioning.** A smaller HNSW structure is constructed from a sample of the complete dataset and used to guide the distribution of points among workers. This allows the system to retain a global view of the dataset without constructing a complete global index over all points.
- **Local neighborhood processing on independent workers.** Each worker constructs its own local HNSW and extracts neighborhood information and raw-distance graph edges from the points assigned to it. This makes it possible to distribute a significant part of the computational workload.
- **Replication across worker partitions.** When replication is enabled, each routed point can be assigned to up to R distinct nearby worker partitions. This introduces controlled overlap between partitions and helps recover neighborhood relationships that may otherwise be separated by partition boundaries.
- **Approximate reconstruction of global core distances.** Instead of relying only on core distances computed independently by each worker, the central leader combines neighborhood information produced by different workers and by the global routing structure to obtain approximate global core-distance estimates.
- **Local graph reduction and global mutual-reachability reconstruction.** Local core-distance estimates and mutual-reachability weights are used by the Indexer and Workers to reduce their harvested graphs before communication. The Leader subsequently combines the retained raw-distance edges, reconstructs approximate global core distances, recomputes mutual-reachability weights, and constructs the global minimum spanning tree used for hierarchical cluster extraction.

- **Experimental evaluation against centralized clustering.** The distributed implementation is evaluated against single-machine FISHDBC and HDBSCAN*. The experiments analyze clustering quality, the effects of sampling and replication, runtime scalability, communication, and the trade-offs introduced by distributed execution.

1.6 Use of LLMs

In this master’s thesis, Large Language Models (LLMs) served two primary purposes: assisting with code debugging and error resolution during the implementation phase, and supporting language refinement and formatting during the writing phase.

1.7 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 introduces the background concepts required to understand the proposed distributed FISHDBC approach. It covers density-based clustering, HDBSCAN*, Hierarchical Navigable Small World graphs, FISHDBC, graph partitioning, distributed-computing concepts, and Docker Swarm.

Chapter 3 reviews the most relevant related work. Particular attention is given to distributed density-based clustering, parallel FISHDBC, and distributed HNSW approaches such as Pyramid. The chapter also positions the contribution of this thesis with respect to these existing methods.

Chapter 4 presents the design and implementation of the proposed distributed FISHDBC approach. It describes the Indexer, Worker, and Leader roles, the sampled global HNSW used for partitioning and routing, local distance harvesting and graph reduction, replication across worker partitions, reconstruction of approximate global core distances, mutual-reachability weighting, global minimum spanning tree construction, communication between components, configuration, and Docker Swarm deployment.

Chapter 5 evaluates the proposed approach against single-machine FISHDBC and HDBSCAN*. The experiments examine clustering quality, the effects of sampling and replication, graph reconstruction, execution time, and scalability under different distributed configurations.

Finally, Chapter 6 summarizes the main findings of the thesis, discusses the limitations of the current prototype, and outlines possible directions for future work.

Chapter 2

Background

This chapter introduces the main concepts required to understand the distributed FISHDBC approach developed in this thesis. The discussion focuses on density-based clustering and its hierarchical extension, the role of neighborhood information and mutual-reachability distance, approximate nearest-neighbor search through Hierarchical Navigable Small World graphs, minimum spanning trees, graph partitioning, and the distributed-computing concepts used later in the proposed architecture.

The purpose of this chapter is to provide the technical foundation needed for the design and implementation chapters without introducing the contribution of this thesis itself. The concepts presented here are therefore limited to those necessary to understand how FISHDBC operates and why distributing its computation requires both local processing and global reconstruction.

2.1 Clustering and Density-Based Clustering

Clustering is an unsupervised learning task that aims to organize a set of data points into groups according to their similarity, without relying on predefined class labels [Jai10]. The basic assumption is that points belonging to the same cluster should be more similar to each other than to points assigned to different clusters. The notion of similarity is therefore fundamental and is commonly expressed through a distance or dissimilarity function. The choice of distance function can significantly influence both the resulting cluster structure and the computational cost of the algorithm, particularly when the data cannot be represented effectively in a standard Euclidean space.

Several families of clustering methods have been developed, including partition-based, hierarchical, and density-based approaches [Jai10]. These families differ in the assumptions

they make about the shape and organization of clusters. Partition-based methods usually divide the dataset into a predefined number of groups, while hierarchical methods represent clusters at different levels of granularity. Density-based methods instead identify clusters according to the concentration and connectivity of data points in the underlying space.

The main idea behind density-based clustering is that clusters correspond to regions where points are sufficiently close to one another, while areas containing only a few isolated points are treated as low-density regions or noise [Est+96]. This allows density-based methods to identify clusters with irregular and non-convex shapes, which may be difficult to represent using methods based on centroids or spherical assumptions. The ability to distinguish noise from clustered points is another important characteristic of this family of algorithms.

Neighborhood information plays a central role in density-based clustering [Est+96]. Whether a point belongs to a dense region depends on the points located around it and on the connectivity between neighboring regions. As a result, the clustering process depends not only on individual pairwise distances, but also on the structure formed by local neighborhoods throughout the dataset. This becomes increasingly important as dataset size grows, because discovering and maintaining neighborhood relationships can require a significant amount of computation.

These properties make density-based clustering suitable for datasets whose cluster structure is not known in advance and where clusters may have arbitrary shapes. At the same time, their dependence on neighborhood information introduces computational challenges when the number of data points becomes large. The following section introduces DBSCAN and HDBSCAN*, which formalize these density-based concepts and provide the foundation for the clustering methods considered later in this thesis.

2.2 DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm introduced by Ester et al. [Est+96]. Unlike clustering methods that represent clusters around a centroid or require the number of clusters to be specified in advance, DBSCAN identifies clusters as connected regions of sufficiently high point density. This allows the algorithm to discover clusters with irregular shapes and to distinguish isolated points from clustered regions.

DBSCAN is mainly controlled by two parameters: the neighborhood radius ε and the minimum number of points *MinPts*. Given a point p , its ε -neighborhood is defined as the set of points whose distance from p is not greater than ε :

$$N_\varepsilon(p) = \{q \in D \mid d(p, q) \leq \varepsilon\}, \quad (2.1)$$

where D is the dataset and $d(p, q)$ denotes the distance between points p and q .

The main concepts used by DBSCAN can be summarized as follows [Est+96]:

- **ε -neighborhood:** the set of points located within distance ε from a point p , as defined in Equation 2.1.
- **$MinPts$:** the minimum number of points required within the ε -neighborhood of a point for that region to be considered sufficiently dense.
- **Core point:** a point p is a core point when its ε -neighborhood contains at least $MinPts$ points.
- **Border point:** a point is considered a border point when its own ε -neighborhood contains fewer than $MinPts$ points, but the point belongs to the ε -neighborhood of at least one core point.
- **Noise point:** a point is classified as noise when it does not satisfy the core-point condition and does not belong to the ε -neighborhood of any core point.
- **Directly density-reachable:** a point p is directly density-reachable from a point q when p belongs to the ε -neighborhood of q and q is a core point.
- **Density-reachable:** a point p is density-reachable from a point q when there exists a sequence of points connecting q to p such that each point in the sequence is directly density-reachable from the preceding one.
- **Density-connected:** two points p and q are density-connected when there exists a point o from which both p and q are density-reachable.
- **Cluster:** a cluster is a maximal set of density-connected points.

Figure 2.1 illustrates the three main types of points considered by DBSCAN. A core point satisfies the minimum-density requirement within its ε -neighborhood, a border point belongs to the neighborhood of a core point without satisfying the density requirement itself, while a noise point is not connected to any sufficiently dense region.

Using these definitions, DBSCAN constructs clusters by expanding from core points. Starting from an unvisited point, the algorithm checks whether its ε -neighborhood contains at least $MinPts$ points. If the density condition is satisfied, a new cluster is created and expanded by adding points that are density-reachable from its core points. The process continues until no additional points can be added to the current cluster, after which the algorithm searches for another unvisited dense region [Est+96].

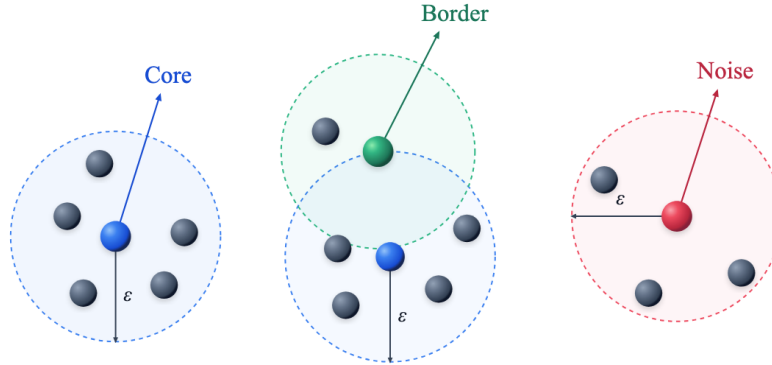


Figure 2.1: Illustration of core, border, and noise points in DBSCAN.

The resulting clusters are determined by local density and connectivity rather than by a predefined geometric model. This allows DBSCAN to discover clusters with non-convex and irregular shapes. Another important property is the explicit identification of noise, since points that do not belong to any sufficiently dense region are not forced into a cluster.

Despite these advantages, DBSCAN has several limitations. Its result depends on the choice of ε and $MinPts$, and selecting suitable parameter values can be difficult when the structure of the dataset is not known in advance. In particular, the use of a single global value of ε can be problematic when the dataset contains clusters with different densities. A small value may fragment relatively sparse clusters, while a larger value may incorrectly merge nearby dense regions. In addition, DBSCAN requires repeated neighborhood searches, which can become computationally expensive for large datasets when efficient indexing structures are not available.

These limitations motivate hierarchical density-based approaches such as HDBSCAN*, which avoid relying on a single fixed density threshold and instead construct a hierarchy over different density levels [MHA17].

2.3 HDBSCAN*

Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN*) extends the main ideas of DBSCAN by considering cluster structure over different density levels rather than relying on a single global value of ε . In DBSCAN, the choice of ε directly determines which neighborhoods are considered dense. This can be problematic when different regions of the dataset contain clusters with substantially different densities. HDBSCAN* addresses this limitation by constructing a hierarchy that represents how clusters appear, split, and disappear as the density level changes [CMS13].

A central concept in HDBSCAN* is the *core distance*. For a point p , the core distance is determined by its $MinPts$ -th nearest neighbor and can be written as

$$c(p) = d(p, N_{MinPts}(p)), \quad (2.2)$$

where $N_{MinPts}(p)$ denotes the $MinPts$ -th nearest neighbor of p , and $d(\cdot, \cdot)$ is the distance function. The core distance provides an indication of the local density surrounding a point. A small core distance generally indicates that the point lies in a dense region, while a larger value indicates a sparser neighborhood [MHA17; Del19].

HDBSCAN* uses these core distances to define the *mutual-reachability distance* between two points p and q :

$$d_{\text{mreach}}(p, q) = \max \{c(p), c(q), d(p, q)\}. \quad (2.3)$$

The mutual-reachability distance combines the original distance between two points with the local density around both endpoints. Consequently, a connection involving a point in a sparse region may receive a larger weight than a similar connection between points located in dense regions. This transformation reduces the influence of isolated or weakly connected points when the cluster structure is constructed [Del19].

Using mutual-reachability distances, HDBSCAN* can be represented as a weighted graph in which data points correspond to vertices and edge weights correspond to their mutual-reachability distances. A minimum spanning tree (MST) of this graph is then used to preserve the relevant connectivity structure with substantially fewer edges than the complete graph [MHA17; Del19].

The hierarchical clustering is obtained by examining the MST across different mutual-reachability levels. Edges with larger mutual-reachability distances correspond to weaker connections between regions. As these edges are progressively removed, the graph separates into connected components, producing a hierarchy of possible clusters. Unlike DBSCAN, which returns the clustering associated with one selected value of ε , HDBSCAN* therefore represents cluster structure across a range of density levels [MHA17].

The hierarchy can be further transformed into a *condensed cluster tree* by removing branches that do not satisfy the required minimum cluster size. HDBSCAN* then evaluates how long candidate clusters persist within this hierarchy. Clusters that remain present across a wider range of density levels are considered more stable and can be selected to produce a final flat clustering [MHA17].

Figure 2.2 illustrates this relationship between the hierarchical representation and the final clustering. The left side shows a condensed cluster tree, while the right side shows the

clusters extracted from the same hierarchy. The vertical axis of the condensed tree is expressed using

$$\lambda = \frac{1}{\varepsilon}, \quad (2.4)$$

so that increasing values of λ correspond to increasingly dense regions. The branches in the tree represent candidate clusters that exist over different density levels. The highlighted branches indicate the clusters selected from the hierarchy, and these correspond to the three groups shown in the extracted flat clustering on the right.

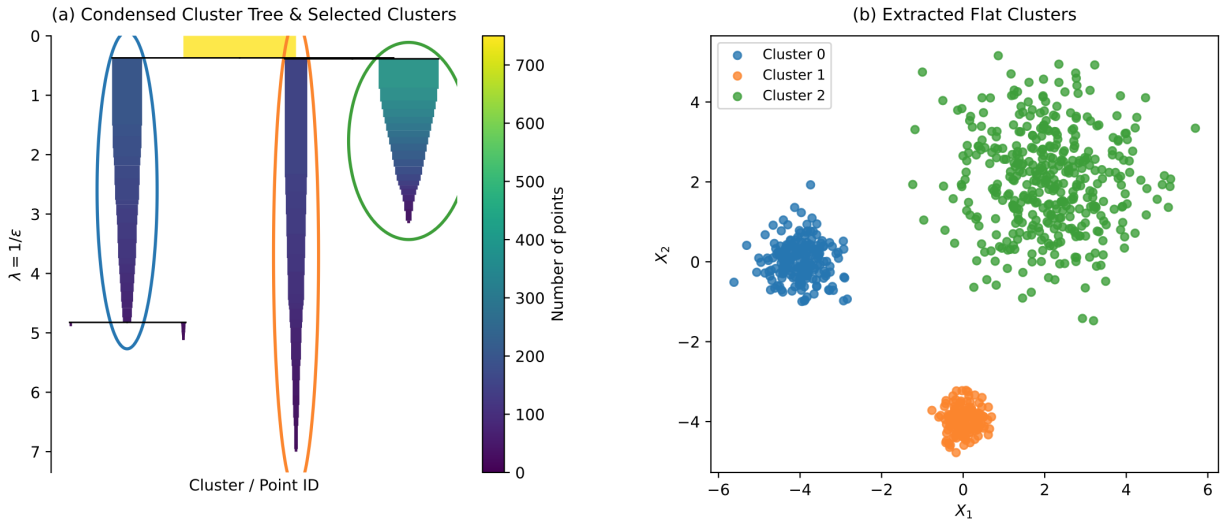


Figure 2.2: Example of an HDBSCAN* condensed cluster tree and the corresponding extracted flat clusters. The selected branches in the hierarchy correspond to the three clusters shown in the data space.

The figure also illustrates an important difference between HDBSCAN* and DBSCAN. HDBSCAN* does not require the clustering to be determined at one manually selected neighborhood radius. Instead, candidate clusters are observed over multiple density levels and the final clusters are extracted according to their persistence within the hierarchy. This makes the method better suited to datasets containing clusters with different density characteristics.

The hierarchical formulation, however, still requires neighborhood information to determine core distances and connectivity information to construct the mutual-reachability MST. For a general distance function, explicitly considering every pair of points would result in a complete graph containing $O(n^2)$ edges [Del19]. This becomes increasingly expensive as the number of data points grows.

FISHDBC addresses this computational problem by approximating the HDBSCAN* procedure rather than explicitly evaluating all pairwise relationships. It uses a Hierarchical Navigable Small World (HNSW) graph to discover useful neighborhood relationships and candidate edges while avoiding exhaustive distance computation [Del19]. Since HNSW is therefore a central component of FISHDBC and of the distributed approach developed in this thesis, the following section introduces its main structure and search mechanism.

2.4 Hierarchical Navigable Small World Graphs

Hierarchical Navigable Small World (HNSW) graphs are graph-based data structures designed for approximate nearest-neighbor search. Instead of comparing a query point with every element in the dataset, HNSW organizes the data in a hierarchy of proximity graphs that allows the search to progressively approach regions containing points similar to the query [MY20]. This makes HNSWs suitable for large datasets where exhaustive nearest-neighbor search would require a large number of distance computations.

The HNSW structure is organized into multiple graph layers. The lowest layer contains all indexed points, while the higher layers contain progressively smaller subsets determined by the probabilistic level-assignment mechanism of HNSW [MY20]. Points assigned to a higher level also appear in all lower levels. The sparse upper layers provide longer-range connections that support efficient navigation across the graph, while the lower layers, particularly Layer 0, provide more detailed local neighborhood information.

A search starts from an entry point in one of the highest layers of the hierarchy. The algorithm examines neighboring vertices and repeatedly moves toward a vertex that is closer to the query. When no closer vertex can be found in the current layer, the search continues from the corresponding position in the next lower layer. This process is repeated until the bottom layer is reached, where a more detailed search is performed to obtain the approximate nearest neighbors [MY20; Del19]. In this way, the upper layers help the search quickly approach the appropriate region of the data space, while the bottom layer refines the result.

Figure 2.3 illustrates the hierarchical organization of HNSW. The upper layers contain fewer vertices and longer-range connections, whereas the bottom layer contains the complete set of indexed points and a denser set of local connections. A search can therefore move rapidly through the sparse upper levels before performing finer exploration close to the target region.

The same hierarchical mechanism is used when new elements are inserted. A new point is first assigned to a maximum layer and is then connected to selected nearby vertices while the insertion process descends through the hierarchy. The neighborhood graph is

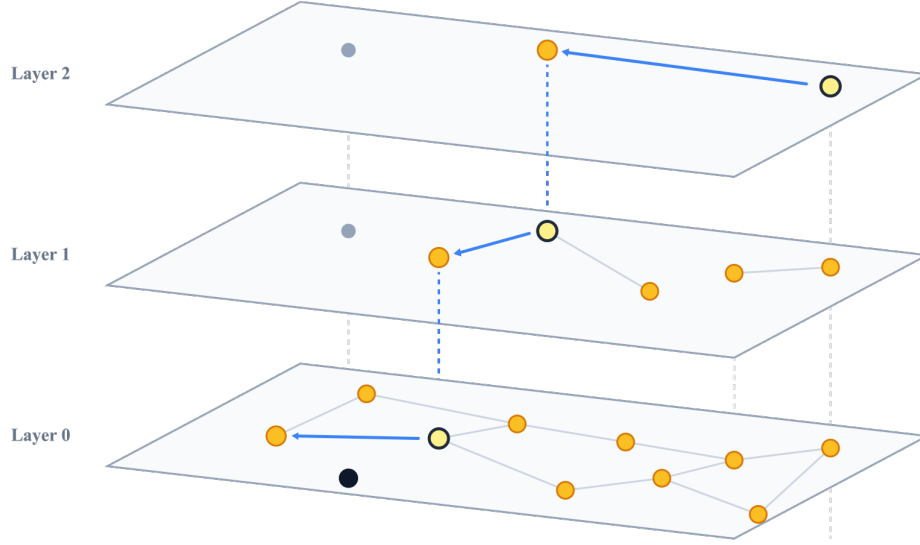


Figure 2.3: Hierarchical structure of an HNSW graph. Upper layers contain progressively fewer points and support long-range navigation, while the bottom layer contains all indexed points and provides more detailed local connectivity [MY20].

therefore built incrementally as points are added. The quality and computational cost of this construction depend on parameters controlling the graph degree and the amount of search performed during insertion and querying [MY20].

One important HNSW parameter is M , which controls the number of graph connections maintained for each vertex. Larger values generally increase graph connectivity and the probability of discovering useful neighbors, at the cost of additional memory and construction work. HNSW also uses the exploration parameter ef to control the number of candidate vertices considered during graph search. Increasing ef can improve search quality but also increases the number of distance evaluations.

In the original FISHDBC implementation, the HNSW structure is initialized using *MinPts* as its neighborhood-size parameter, together with the configured ef value [Del19]. Thus, the graph connectivity used for approximate neighbor discovery is directly tied to the same *MinPts* parameter used for core-distance estimation.

The relevance of HNSW to FISHDBC goes beyond its conventional use as a nearest-neighbor index. FISHDBC uses HNSW to discover candidate neighborhood relationships while the graph itself is being constructed. Instead of first constructing an HNSW and then issuing a nearest-neighbor query for every data point, FISHDBC observes the distance computations performed during HNSW insertion. Each computed pair (a, b) , together with its distance $d(a, b)$, provides information that can be used to update the known neighbors of the points and to generate candidate edges for the approximate minimum spanning tree

[Del19].

This design reduces the need for exhaustive pairwise distance computation. FISHDBC maintains, for each point, the *MinPts* closest neighbors discovered through HNSW processing and uses this information to estimate the point’s core distance. At the same time, the discovered point pairs contribute candidate edges for the approximate mutual-reachability minimum spanning tree. Distances that are never evaluated by HNSW are not explicitly considered by the clustering procedure [Del19].

An additional reason for using information generated during HNSW construction is that useful clustering relationships are not necessarily limited to the final nearest-neighbor edges of the bottom HNSW layer. During hierarchical search, the algorithm encounters points at different distances while progressively approaching the local neighborhood. FISHDBC uses these intermediate distance evaluations as additional connectivity information, which can help preserve connections between nearby regions that might otherwise be separated if only the final bottom-layer nearest-neighbor graph were considered [Del19].

HNSW therefore provides two important properties for the clustering approach considered in this thesis. First, it offers an efficient approximation of local neighborhoods without requiring exhaustive comparison of all pairs of points. Second, its graph structure exposes connectivity information that can be reused when constructing an approximate minimum spanning tree. These properties make HNSW a central component of FISHDBC and also motivate its use in the distributed approach developed later in this thesis.

In the distributed setting, however, a single global HNSW cannot simply be assumed to remain entirely on one machine when the objective is to distribute both data and computation. The proposed architecture therefore uses a sampled global HNSW to capture the general organization of the dataset and to guide its division among workers.

2.5 FISHDBC

FISHDBC, introduced by Dell’Amico [Del19], stands for Flexible, Incremental, Scalable, Hierarchical Density-Based Clustering. It was designed to combine several properties that are difficult to obtain simultaneously in density-based clustering: support for arbitrary data and distance functions, incremental updates, scalability, and hierarchical output [Del19]. Unlike approaches that depend on a specific geometric representation of the input data, FISHDBC accepts a user-defined symmetric distance function, which may also be non-metric. This makes the algorithm applicable to heterogeneous data for which a conventional spatial index may not be available.

FISHDBC approximates HDBSCAN* while avoiding the explicit computation of the complete pairwise distance matrix. In the generic case, constructing the full mutual-reachability

graph would require considering all pairs of points, leading to quadratic complexity. FISHDBC reduces this cost by using approximate neighborhood discovery together with incremental maintenance of an approximate minimum spanning tree [Del19]. Its design therefore separates the discovery of useful neighborhood relationships from the maintenance of the clustering structure.

Approximate neighborhood discovery is performed using a Hierarchical Navigable Small World graph. However, FISHDBC does not simply construct an HNSW structure and then query it afterward. Instead, it reuses the distance evaluations performed during HNSW insertion. Whenever the HNSW evaluates the distance between two points, that information can be used both to update their discovered neighborhoods and to generate candidate edges for the approximate minimum spanning tree [Del19]. In this way, the distance computations required for building the HNSW also contribute directly to the clustering process.

The internal state maintained by FISHDBC consists mainly of four structures: the HNSW graph, the closest discovered neighbors of each point, the current approximate minimum spanning tree, and a temporary collection of candidate edges [Del19]. For each data point, the algorithm keeps the *MinPts* closest neighbors discovered so far. The distance to the farthest point in this set represents the current estimate of its core distance. Candidate edges contain relationships whose mutual-reachability weights may be relevant for the minimum spanning tree.

Algorithm 1 summarizes the main FISHDBC procedure. The notation $c(x)$ denotes the current core distance of point x . During the insertion of a new point, the distance evaluations produced by HNSW are reused to update the neighbor information and to generate candidate mutual-reachability edges. The approximate MST is updated periodically by applying Kruskal’s algorithm to the current tree together with the newly discovered candidate edges [Del19].

When a new point x is inserted, FISHDBC first adds it to the HNSW. The distances evaluated during this operation are then examined to determine whether newly discovered points should enter the *MinPts* neighborhood of x or of previously inserted points. For each evaluated pair (x, y) , the mutual-reachability distance is computed from the pairwise distance and the current core distances of the two endpoints. The resulting edge is added to the candidate set [Del19].

The insertion of a new point may also reduce the core distance of an existing point. When this happens, the mutual-reachability weights of some previously known edges may decrease. FISHDBC therefore revisits the relevant neighbor relationships and adds the affected edges to the candidate set with their updated weights. This mechanism allows the clustering structure to evolve as new data are added without recomputing all pairwise relationships from the beginning [Del19].

Algorithm 1 FISHDBC algorithm, adapted from [Del19].

```

1: procedure SETUP( $d, MinPts, ef$ )
2:    $self.MinPts \leftarrow MinPts$ 
3:    $self.mst \leftarrow \emptyset$ 
4:    $self.neighbors \leftarrow \emptyset$ 
5:    $self.HNSW \leftarrow HNSW(d, MinPts, ef)$ 
6:    $self.candidates \leftarrow \emptyset$ 
7: end procedure
8: procedure ADD( $x$ )
9:    $self.HNSW.add(x)$ 
10:   $self.neighbors[x] \leftarrow$  closest discovered  $MinPts$  neighbors
11:  for each distance evaluation  $d(x, y) = v$  performed by HNSW do
12:     $rd \leftarrow \max(v, c(x), c(y))$ 
13:     $self.candidates[x, y] \leftarrow rd$ 
14:    if  $x$  becomes one of the  $MinPts$  closest known neighbors of  $y$  then
15:      update  $self.neighbors[y]$ 
16:      for each neighbor  $z$  of  $y$  at distance  $w < v$  do
17:        if  $c(z) < v$  then
18:           $rd \leftarrow \max(w, c(y), c(z))$ 
19:           $self.candidates[y, z] \leftarrow rd$ 
20:        end if
21:      end for
22:    end if
23:  end for
24:  if  $|self.candidates| > \alpha \cdot |self.neighbors|$  then
25:    UPDTEMST
26:  end if
27: end procedure
28: procedure UPDTEMST
29:    $self.mst \leftarrow Kruskal(self.mst \cup self.candidates)$ 
30:    $self.candidates \leftarrow \emptyset$ 
31: end procedure
32: function CLUSTER( $mcs$ )
33:   if  $self.candidates \neq \emptyset$  then
34:     UPDTEMST
35:   end if
36:   return clustering extracted from  $self.mst$ 
37: end function

```

The minimum spanning tree is not reconstructed from the complete graph after every insertion. Instead, candidate edges are accumulated temporarily. When their number reaches a specified threshold, the algorithm applies Kruskal’s algorithm to the union of the current approximate MST and the candidate edges. The candidate set is then cleared and the updated tree becomes the current clustering structure [Del19]. This incremental update strategy limits the amount of work required after each insertion.

Once clustering is requested, any remaining candidate edges are first incorporated into the MST. The resulting approximate mutual-reachability tree is then processed using the HDBSCAN*-style hierarchy construction and cluster extraction procedure. FISHDBC therefore avoids constructing the complete mutual-reachability graph while still preserving enough connectivity information to obtain an approximate hierarchical clustering.

This organization also distinguishes FISHDBC from distributed DBSCAN-style methods, which generally target flat clustering of a fixed dataset.

The scalability of FISHDBC depends strongly on the number of distance evaluations performed during HNSW construction. The algorithm maintains internal state whose size has been empirically observed to be $\mathcal{O}(n \log n)$, dominated by the HNSW structure, while the running time depends on the number of HNSW distance calls together with the cost of maintaining the approximate MST [Del19]. As a result, the method is especially effective when the HNSW discovers useful neighborhood relationships using only a small fraction of all possible pairwise comparisons.

Despite these properties, the original FISHDBC implementation maintains the HNSW, neighbor information, candidate edges, and approximate MST within a single execution environment. The algorithm therefore reduces the computational cost of hierarchical density-based clustering, but does not define how these data structures should be divided or coordinated across multiple independent machines.

2.6 Graph Partitioning

Graph partitioning is the process of dividing the vertices of a graph into a predefined number of disjoint subsets while attempting to preserve important relationships within each subset. In distributed systems, graph partitioning is commonly used to divide a large graph among multiple processing units so that each partition can be handled independently or semi-independently. A good partitioning should therefore balance the amount of work assigned to each partition while minimizing the number of relationships that cross partition boundaries [SS13].

Let a graph be represented as $G = (V, E)$, where V is the set of vertices and E is the set of edges. A k -way partition divides the vertex set into k disjoint subsets

$$V = V_1 \cup V_2 \cup \dots \cup V_k, \quad (2.5)$$

such that

$$V_i \cap V_j = \emptyset \quad \text{for } i \neq j. \quad (2.6)$$

Each subset V_i represents one graph partition. In a distributed execution, these partitions can be assigned to different workers so that each worker processes only a portion of the original graph.

Two objectives are particularly important when constructing such partitions: *balance* and *edge-cut minimization*. Balance aims to ensure that the partitions contain comparable amounts of computational work. If one partition contains substantially more vertices or workload than the others, the worker responsible for that partition may become a bottleneck while the remaining workers finish earlier and remain idle.

An edge cut occurs when an edge connects two vertices that have been assigned to different partitions. If an edge (u, v) satisfies $u \in V_i$ and $v \in V_j$ with $i \neq j$, the edge crosses a partition boundary. The number or total weight of these crossing edges is commonly referred to as the *edge cut*. Minimizing this quantity helps to keep strongly connected regions of the graph together and can reduce the amount of information that must be exchanged between workers.

When graphs represent similarity or neighborhood relationships, minimizing cross-partition edges also helps preserve locality. Vertices connected by strong neighborhood relationships are preferably assigned to the same partition. As a result, each worker retains a more complete local view of the region assigned to it. This property is particularly relevant when graph partitioning is used together with approximate nearest-neighbor structures, since splitting neighboring points across several partitions may reduce the amount of useful neighborhood information available locally.

Graph partitioning can also take edge and vertex weights into account. Vertex weights can represent computational cost or the amount of data associated with a vertex, while edge weights can represent the strength or importance of a relationship between two vertices. Weighted graph partitioning then attempts to balance the total vertex weight assigned to the partitions while avoiding cuts through strongly connected edges. This provides more information than treating all vertices and edges as equally important.

The use of graph partitioning together with HNSW has previously been explored in distributed approximate nearest-neighbor search. Pyramid, for example, constructs a smaller meta-HNSW and partitions the bottom layer of this graph into multiple sub-datasets. These partitions are then associated with different workers, each of which constructs a

local HNSW over its assigned data [Den+19]. The authors explicitly consider workload balance when constructing the partitions and aim for sub-datasets of similar size so that the distributed search workload is distributed more evenly.

In Pyramid [Den+19], the meta-HNSW acts as a global structure that guides the assignment of the original dataset to graph partitions. After the bottom layer of the meta-HNSW is partitioned, each data item is associated with the partition containing its closest representative in the meta-HNSW. Each worker subsequently builds a local HNSW over the resulting sub-dataset. This illustrates how graph partitioning can be combined with a smaller global structure to preserve locality while distributing a much larger dataset.

A related family of methods is provided by the KaHIP framework, which contains algorithms for balanced graph partitioning. Such methods aim to reduce edge cuts while respecting balance constraints between partitions [SS13]. This general objective is important in distributed processing because minimizing cut edges helps preserve local connectivity while balancing the workload assigned to different processing units.

Despite these advantages, graph partitioning cannot guarantee that all important neighborhood relationships remain inside a single partition. Points located close to a partition boundary may still be separated even when they are close in the original data space. In density-based clustering, this is especially important because missing neighboring points can affect local density estimates and connectivity information. A worker that processes only its own partition therefore has only a partial view of the complete dataset.

For this reason, partitioning alone is not sufficient for the distributed clustering problem considered in this thesis. Additional mechanisms are required to preserve information that crosses partition boundaries, for example by allowing selected data points to be processed by more than one worker or by reconstructing global information after local processing. These ideas are discussed later as part of the distributed architecture. The following section first introduces the distributed-computing concepts required to understand the interaction between workers and the global coordination stage.

2.7 Distributed Computing Concepts

Distributed computing divides a computational task across multiple processing nodes that cooperate to solve a problem that may be too large or too expensive for a single machine [TS07]. In this model, data and computation are distributed among several workers, while coordination mechanisms are used to combine partial results and maintain the overall execution. This approach can reduce the amount of memory and computation required by each individual machine and can also enable different parts of the workload to be processed in parallel.

A common strategy in distributed data processing is to partition the input dataset into smaller subsets and assign each subset to a different worker. Each worker then performs local computation only on the data assigned to it. This reduces the local memory requirement and can improve computational scalability when the work can be processed independently. However, the effectiveness of this strategy depends strongly on how the data are partitioned and how evenly the workload is distributed among workers.

Load balancing is therefore an important consideration in distributed execution, since uneven partitions can cause some workers to become bottlenecks while others remain under-utilized [TS07]. Balanced partitioning attempts to distribute the workload more evenly so that the available computational resources can be used efficiently. This idea is closely related to the graph-partitioning concepts introduced in Section 2.6.

Although local computation can reduce the workload of individual machines, each worker generally has only a partial view of the complete dataset. This distinction between *local* and *global* information is particularly important for algorithms whose result depends on relationships between data assigned to different workers. A worker may be able to compute correct relationships among points within its own partition, while remaining unaware of relevant data located in another partition.

One way to reduce the loss of cross-partition information is through *replication*. Replication allows selected data items to be assigned to more than one worker so that overlapping local views are created [Cou+11]. This can improve the probability that relationships crossing a partition boundary are observed by at least one worker. The cost is that replicated data increase memory usage, local computation, and the amount of information that may need to be transferred between components.

Distributed execution also introduces communication overhead. Workers may need to send intermediate results to other components, and a coordinating process may need to collect and combine information produced by several workers. Communication, serialization, synchronization, and data transfer therefore contribute additional costs that are not present in the same form in a single-machine execution. As a result, increasing the number of workers does not automatically produce a proportional reduction in runtime.

Many distributed systems use a *coordinator* or *leader* to manage the relationship between local processing and global computation. Workers perform tasks that can be executed independently on local partitions, while the coordinator combines their outputs or reconstructs information that cannot be obtained from a single local view. This separation is useful when the computationally expensive parts of an algorithm can be distributed, but the final result still depends on global relationships.

This distinction is particularly relevant to graph-based and density-based algorithms. If a graph is divided among several workers, each worker observes only the vertices and edges available in its own partition, possibly together with replicated boundary information.

Local graph structures can therefore be computed independently, but they may not contain all of the connections required to describe the complete graph. A later aggregation stage may be necessary to combine local structures and recover information spanning multiple partitions.

The same issue applies to neighborhood-based clustering. Local nearest-neighbor information can be computed independently on different workers, but a point's true neighborhood may include points located in other partitions. Consequently, locally computed density estimates may differ from those that would be obtained using the complete dataset. A distributed clustering method must therefore determine which computations can be performed locally and which information must later be combined globally.

The distributed approach considered in this thesis follows this general separation between local computation and global reconstruction. Data are divided among multiple workers, which independently perform neighborhood and graph-related computations on their assigned subsets. Selected data can be replicated to increase overlap between worker partitions, while a central coordination stage combines the resulting information to recover the global density and connectivity relationships needed for hierarchical clustering.

These concepts provide the systems foundation for the distributed architecture developed later in this thesis. The main challenge is not only to divide computation among several workers, but also to preserve enough information across partition boundaries so that the final result remains representative of the complete dataset. The following section introduces Docker Swarm, which provides the container-orchestration environment used to deploy the distributed prototype.

2.8 Container Orchestration with Docker Swarm

Containerization provides a way to package an application together with its runtime dependencies so that it can be executed consistently across different machines. When an application is composed of several services that must run on multiple hosts, an orchestration mechanism is needed to manage where those services execute, how they communicate, and how the desired deployment state is maintained. Docker Swarm provides these orchestration capabilities directly through Docker Engine [Docd].

A Docker Swarm consists of multiple Docker hosts operating as nodes in the same cluster. Nodes can act as *manager nodes*, *worker nodes*, or both. Manager nodes are responsible for maintaining the state of the swarm, scheduling tasks, and coordinating the execution of services, while worker nodes execute the tasks assigned to them by the managers [Docd]. A manager node also maintains the desired state of the services and can reschedule tasks when a worker becomes unavailable.

The main unit of deployment in Docker Swarm is a *service*. A service defines the container image to execute together with its runtime configuration, such as the number of replicas, resource requirements, networks, exposed ports, and placement rules. Each running instance of a service is represented by a task, and each task corresponds to a container executed on one of the swarm nodes [Docb].

Docker Swarm supports both *replicated* and *global* services. In replicated mode, a specified number of identical service instances are distributed across the available nodes. In global mode, one task is scheduled on every eligible node. Replicated services are particularly useful when several equivalent instances of the same computation must run in parallel, while global services are commonly used for components that must be present on every node [Docb].

An important feature of Swarm is the ability to control service placement. Placement constraints restrict a service to nodes that satisfy specified properties or labels. For example, a service may be configured to execute only on nodes having a particular role or custom node label. Placement preferences can also be used to encourage tasks to be distributed across selected groups of nodes. These mechanisms allow the physical deployment of services to reflect the logical roles defined by a distributed application [Doca].

Communication between services running on different swarm nodes can be provided through *overlay networks*. An overlay network creates a virtual network spanning multiple Docker hosts and allows services attached to that network to communicate even when their containers are executing on different physical machines. Docker Swarm also provides internal service discovery through DNS, allowing services to address one another by service name rather than by fixed container addresses [Docc].

Another characteristic of Swarm is its use of a declarative deployment model. Rather than manually starting individual containers on specific hosts, the desired configuration of a service is defined and submitted to the swarm manager. The manager then attempts to maintain this desired state by scheduling the required tasks on eligible nodes. If a task or worker becomes unavailable, the manager can schedule a replacement task on another suitable node [Docc]. This reduces the amount of manual coordination required when executing a distributed application across multiple machines.

These concepts are relevant to the implementation developed in this thesis because the distributed clustering system consists of components with different execution roles. Worker processes perform local computation over assigned data partitions, while coordination components handle dataset distribution and global reconstruction. Docker Swarm provides the cluster-level execution environment in which these components can be deployed across separate machines while still communicating as part of the same distributed application.

The detailed deployment configuration is presented later in the implementation chapter. In particular, the implementation uses Swarm service definitions and placement constraints

to control where the indexer, leader, and worker components are executed. This separation allows the logical architecture of the distributed clustering approach to be mapped onto a physical multi-node environment without embedding deployment-specific details into the clustering algorithm itself.

Chapter 3

Related Work

This chapter reviews the existing work most relevant to the distributed clustering approach developed in this master’s thesis. While the previous chapter introduced the technical concepts required to understand the proposed system, the purpose of this chapter is to examine how related methods address scalability, parallelism, and distribution in density-based clustering and approximate nearest-neighbor processing.

The discussion begins with existing distributed density-based clustering approaches and considers how they divide computation across multiple processing units, handle neighborhood information, and combine partial clustering results. Particular attention is given to the properties that are important for this thesis, including support for flexible distance functions, incremental processing, hierarchical clustering, and the treatment of relationships that cross partition boundaries.

Finally, distributed HNSW approaches are considered, with particular attention to Pyramid and its use of a sampled global structure, graph partitioning, and worker-local HNSW indexes. These works address complementary aspects of the problem considered in this thesis: FISHDBC provides the clustering model, parallel FISHDBC explores shared-memory parallelism, while distributed HNSW methods provide ideas for organizing neighbor-search workloads across multiple machines. The chapter concludes by positioning the proposed approach with respect to these existing methods and identifying the specific gap addressed by this thesis.

3.1 Distributed Density-Based Clustering

Distributed density-based clustering aims to extend the advantages of density-based methods to datasets that are too large or computationally expensive to process efficiently on

a single machine. The main difficulty is that density-based clustering depends on neighborhood relationships and connectivity between data points [Est+96], while distributed execution divides the dataset across multiple processing units. As a result, points belonging to the same dense region may be placed on different workers, and local computations may not observe all of the relationships required to reproduce the clustering that would be obtained from the complete dataset [Pat+12].

Several approaches have been proposed to address this problem. Some methods partition the data space and perform clustering locally before merging partial results, while others use shared or distributed data structures to parallelize neighborhood discovery and cluster construction [CGL10; Pat+12]. More recent approaches also use approximate neighbor graphs to avoid relying on geometric partitioning and to support more general distance functions [Lul+16].

The following subsections review these strategies in more detail. The discussion first considers the challenges introduced by distributing density-based clustering, then examines partition-and-merge methods and graph-based approximate approaches. The section concludes by identifying the limitations of these methods with respect to FISHDBC, particularly in terms of flexibility in the distance function, incremental processing, hierarchical output, and the reconstruction of global density and connectivity information.

3.1.1 Challenges of Distributing Density-Based Clustering

Distributing density-based clustering is more difficult than simply dividing a dataset among several workers. The reason is that density-based methods determine cluster membership from neighborhood relationships and connectivity between data points. Once the dataset is partitioned, each worker observes only a subset of the complete data and may therefore have an incomplete view of the neighborhood of points located close to partition boundaries. This can affect local density estimates and may cause points that belong to the same cluster in the complete dataset to be treated differently when processed independently [Est+96; Pat+12].

The problem is particularly evident in DBSCAN-style clustering. Whether a point is classified as a core point depends on the number of neighbors contained within its ε -neighborhood. If some of these neighbors are located in another partition, the worker processing the point may observe fewer neighbors than are present globally. A point that would satisfy the core-point condition in the complete dataset may therefore appear as a border or noise point in a local partition. Similar problems occur when a dense region extends across several partitions, since a locally identified cluster may represent only one part of a larger global cluster.

For this reason, distributed implementations must preserve or reconstruct relationships

that cross partition boundaries. One common strategy is to introduce overlapping regions or exchange boundary information between workers so that clusters formed independently can later be merged. However, this introduces additional communication and coordination costs. Earlier parallel DBSCAN implementations frequently adopted master-worker approaches in which workers performed local clustering and a central process merged the partial results. Patwary et al. observe that this design can suffer from both workload imbalance and communication overhead, particularly when the final merging stage remains centralized [Pat+12].

Data partitioning itself is another important challenge. A partitioning strategy should distribute the workload sufficiently evenly while also keeping strongly related points together. These two objectives can conflict. A partitioning that is well balanced computationally may divide dense regions across many workers, while a partitioning that preserves local structure may assign substantially different amounts of work to different nodes. This trade-off becomes more significant for density-based methods because the computational effort associated with neighborhood queries can vary substantially across regions of different density [Pat+12].

Many distributed DBSCAN implementations address partitioning by exploiting the geometric structure of Euclidean data. Spatial decomposition can make it possible to assign nearby points to the same worker and to identify regions where additional boundary processing is required. However, such strategies become less straightforward for high-dimensional or non-Euclidean data. Lulli et al. note that several distributed DBSCAN approaches partition the feature space and then reconcile the work performed at partition boundaries, but this strategy becomes problematic as dimensionality increases and cannot be directly applied when the input consists of arbitrary objects described only through a distance function [Lul+16].

This limitation is important for clustering methods that aim to support flexible distance functions. When data cannot be represented conveniently in a geometric coordinate space, there may be no natural spatial partition that preserves neighborhood relationships. In such cases, distributing the computation requires a different representation of locality, for example through a neighbor graph or another structure derived from pairwise similarity information. NG-DBSCAN follows this direction by replacing Euclidean-space partitioning with a distributed approximate neighbor graph, allowing arbitrary data to be processed as long as a symmetric distance function is available [Lul+16].

Finally, distribution introduces a distinction between *local* and *global* clustering information. A worker may construct a valid clustering or neighborhood structure for its own partition, but the final result may still depend on relationships that are visible only when information from several workers is combined. Distributed density-based clustering must therefore address not only parallel computation, but also the reconstruction of global connectivity from partial local views. This issue becomes even more significant for hierarchical

density-based methods, where global density estimates and connectivity relationships influence the hierarchy produced by the algorithm.

These challenges explain why distributed density-based clustering generally requires more than independent execution of the original clustering algorithm on separate partitions. Effective approaches must consider how the data are divided, how relationships crossing partition boundaries are preserved, how the workload is balanced, and how local results are combined into a consistent global result. The following subsections examine representative approaches that address these issues through partition-and-merge strategies and graph-based approximate methods.

3.1.2 Partition-and-Merge Approaches

A common strategy for distributing DBSCAN is to divide the dataset into several partitions, perform clustering independently on each partition, and then combine the partial results into a global clustering. This approach reduces the amount of data processed by each worker and allows local clustering tasks to be executed in parallel. However, the final merging stage must correctly reconcile clusters that extend across partition boundaries, since a dense region may be split among several workers.

One example is P-DBSCAN, proposed by Chen et al. The method partitions the input database into several parts and assigns each part to a separate computing node. Each node constructs its own spatial index and performs DBSCAN independently on its local data. After the local clustering phase, the partial results are aggregated to obtain the final clustering [CGL10]. The approach follows a relatively direct partition-and-merge model and demonstrates how DBSCAN can be parallelized in a distributed environment by reducing the amount of data handled by each individual node.

A limitation of this general strategy is that the merging phase can become a bottleneck, particularly when local results are returned to a central process. Patwary et al. addressed this issue with PDSDBSCAN, a parallel DBSCAN formulation based on the disjoint-set data structure [Pat+12]. Their approach replaces part of the sequential cluster-expansion behavior of DBSCAN with a tree-based representation in which local clusters are constructed independently and subsequently merged using disjoint-set operations.

In PDSDBSCAN, each processing unit first performs clustering on its assigned local data and represents the resulting clusters as disjoint-set trees. The local structures are then combined to form the final clusters. Unlike earlier master-worker approaches in which the master performs the final merging sequentially, PDSDBSCAN also parallelizes the merging stage. The authors implemented the method for both shared-memory and distributed-memory architectures using OpenMP and MPI, respectively [Pat+12]. This reduces the sequential bottleneck associated with centralized cluster merging and improves scalability

on large datasets.

More recent work has extended the same general idea with more sophisticated data decomposition and boundary handling. HY-DBSCAN is an exact parallel DBSCAN algorithm designed for distributed-memory systems [Wu+22]. It decomposes the data space using a modified k -d tree, distributes the resulting blocks among processing nodes, and exchanges points located in boundary regions before local clustering is performed. This allows each local process to observe points from neighboring partitions that may be required for correct ε -neighborhood computation.

HY-DBSCAN also introduces optimizations for the local clustering and global merging stages. A grid-based spatial index is used to reduce the cost of neighborhood queries, while a distributed version of Rem’s Union-Find algorithm is used to merge clusters spanning several partitions. The implementation combines MPI for communication between computing nodes with OpenMP for parallelism within individual nodes, allowing it to exploit both distributed-memory and shared-memory resources [Wu+22].

These approaches show that partition-and-merge strategies can scale density-based clustering by reducing the amount of data processed locally and by parallelizing the construction and merging of clusters. At the same time, they rely on mechanisms for recovering relationships that are lost when the original dataset is divided. Depending on the method, this may involve spatial indexing, overlapping boundary regions, communication between neighboring partitions, or distributed cluster-merging structures.

A common characteristic of these methods is that partitioning is closely connected to the geometric representation of the data. Spatial decomposition structures such as R-trees, k -d trees, and grids are effective when data points are represented in a coordinate space and neighborhood relationships can be derived geometrically. This assumption is more restrictive when the input consists of arbitrary objects described only through a distance function. In such cases, there may be no explicit feature space that can be divided using conventional spatial boundaries.

This limitation motivates a different class of distributed density-based methods that represent neighborhood relationships explicitly through graphs rather than through geometric partitions. Such approaches are especially relevant when arbitrary distance functions or non-spatial data must be supported. The following subsection therefore considers graph-based and approximate distributed clustering methods.

3.1.3 Graph-Based and Approximate Distributed Clustering

An alternative to spatial partitioning is to represent neighborhood relationships explicitly through a graph. This is particularly useful when the input data cannot be conveniently

divided using geometric coordinates or when the clustering method must support more general distance functions. Instead of assuming that nearby points can be identified through a spatial decomposition, graph-based methods construct an approximation of the neighborhood structure and perform clustering over that representation.

A representative example is NG-DBSCAN, proposed by Lulli et al. [Lul+16]. The method was designed as a distributed and approximate formulation of DBSCAN that can operate on arbitrary data using a symmetric distance function. Rather than partitioning the original feature space, NG-DBSCAN constructs a distributed neighbor graph in which vertices represent data items and edges represent approximate neighborhood relationships. The clustering is then derived from the information contained in this graph.

The construction of the neighbor graph is inspired by approximate nearest-neighbor techniques and allows the method to avoid evaluating all pairwise distances. This is important because an exhaustive comparison of all data items would require quadratic work in the number of points. By approximating the neighborhood structure, NG-DBSCAN reduces the amount of distance computation while still retaining sufficient local information to identify dense regions [Lul+16].

The distributed execution follows a vertex-centric model. Each data item is associated with a local neighborhood representation, and the neighbor graph is refined iteratively across the distributed system. Once the approximate graph has been constructed, DBSCAN-style clustering can be performed using the resulting neighborhood relationships. The implementation described by Lulli et al. uses Apache Spark to distribute the computation across multiple machines [Lul+16].

The main advantage of this formulation is that it does not require the data to be embedded in a low-dimensional Euclidean space. As long as a suitable symmetric distance function is available, the same distributed graph-based procedure can be applied to different kinds of input data. This makes NG-DBSCAN more flexible than distributed methods that depend directly on spatial decomposition structures such as grids, R-trees, or k -d trees [Lul+16].

However, NG-DBSCAN remains fundamentally a DBSCAN-style clustering method. Its objective is to identify a flat set of density-connected clusters from an approximate neighborhood graph. It does not maintain an incremental hierarchical clustering structure and does not construct the mutual-reachability minimum spanning tree required by HDBSCAN*-based methods such as FISHDBC. These differences are important because the information required for hierarchical density-based clustering extends beyond the neighborhood relationships needed for a single DBSCAN partition.

Graph-based approximation is nevertheless highly relevant to the direction of this thesis. It demonstrates that distributed density-based clustering can be performed without relying on an explicit geometric partition of the original data space, and that approximate neighborhood structures can provide a practical basis for reducing the computational cost

of clustering arbitrary data. These ideas are conceptually related to FISHDBC, which also relies on approximate neighborhood discovery rather than exhaustive distance computation.

The main difference is that FISHDBC uses the discovered neighborhood relationships not only to identify dense regions, but also to estimate core distances, generate mutual-reachability candidate edges, and maintain an approximate minimum spanning tree for hierarchical clustering. The next subsection therefore compares the distributed approaches discussed so far with the properties of FISHDBC and identifies the remaining gap addressed by the later sections of this chapter.

3.1.4 Limitations with Respect to FISHDBC

The distributed density-based clustering approaches discussed in the previous subsections demonstrate that DBSCAN-style clustering can be parallelized using several strategies, including spatial partitioning, distributed merging, disjoint-set structures, and approximate neighbor graphs. These approaches address important scalability problems, but their design goals differ from those of FISHDBC.

One important difference concerns flexibility with respect to the input data and the distance function. Several partition-and-merge approaches rely on geometric properties of the data and on spatial indexing structures such as grids, R-trees, or k -d trees. These techniques can be effective when the input can be represented in a low-dimensional coordinate space, but they are less suitable when the data are arbitrary objects and only a custom dissimilarity function is available. NG-DBSCAN reduces this limitation by operating on arbitrary data through a symmetric distance function and by constructing an approximate neighbor graph instead of partitioning the original feature space [Lul+16].

A second difference is incrementality. The distributed DBSCAN approaches considered here are mainly designed to process a fixed dataset in batch form. When new data are introduced, the neighborhood structure and clustering generally need to be recomputed or updated through another global processing phase. FISHDBC, in contrast, was specifically designed to support incremental insertion, allowing new data points to update the neighborhood information and approximate minimum spanning tree without recomputing the entire clustering structure from the beginning [Del19].

The type of clustering result also differs. Distributed DBSCAN variants generally produce a flat clustering corresponding to a particular density configuration. FISHDBC instead approximates HDBSCAN* and therefore maintains a hierarchical density-based representation. This requires not only identifying connected dense regions, but also preserving the information necessary to construct the hierarchy from mutual-reachability relationships and a minimum spanning tree [Del19].

This distinction is especially important in a distributed environment. Combining local DBSCAN cluster labels is not sufficient for reproducing a FISHDBC-style result. A distributed FISHDBC approach must recover enough neighborhood information from the workers to estimate approximate global core distances. These core distances are required to determine mutual-reachability edge weights, which in turn affect the global minimum spanning tree used to derive the final hierarchy.

Another difference concerns the role of approximate neighborhood discovery. NG-DBSCAN already demonstrates that approximate neighbor graphs can be used to support distributed density-based clustering over arbitrary data [Lul+16]. FISHDBC follows a related idea but uses the discovered neighborhood relationships for a different purpose. The distances evaluated during HNSW construction are reused to maintain nearest-neighbor information and candidate edges for the approximate minimum spanning tree [Del19]. The neighborhood structure therefore contributes directly to the hierarchical clustering process rather than being used only to recover DBSCAN connectivity.

The distributed methods reviewed in this section therefore provide several useful ideas for scaling density-based clustering, but none of the representative approaches considered here directly combines all of the properties relevant to FISHDBC: flexible distance functions, incremental updates, approximate neighbor discovery, and hierarchical density-based clustering. More importantly for this thesis, they do not directly address how these properties can be preserved when the computation is divided across independent machines and the global core-distance and connectivity information must be reconstructed from local results.

For this reason, the next section examines the previous parallel implementation of FISHDBC, which provides an important intermediate step between the original single-machine algorithm and the distributed approach developed in this thesis.

3.2 Parallel FISHDBC

A previous attempt to improve the computational performance of FISHDBC was developed by Pastorino in the master’s thesis *Flexible Clustering Parallelo* [Pas23]. The work investigates how the main computational stages of FISHDBC can be executed in parallel using multiple processes on a shared-memory machine. Rather than changing the general clustering model, the objective is to reduce execution time by exploiting the computational resources available within a single physical system.

The parallelization focuses mainly on the construction of the HNSW structure and on the computation of the minimum spanning tree. These stages represent important parts of the FISHDBC workflow because neighborhood discovery produces the relationships later used to construct the approximate clustering structure, while the MST determines the

connectivity from which the hierarchical clustering is extracted [Pas23].

For the HNSW construction, the parallel implementation allows several processes to insert data points concurrently. Since the HNSW graph must be accessible by all participating processes, its internal structures are stored in shared memory. Pastorino uses shared NumPy arrays to represent graph adjacency information and associated edge weights, allowing processes to access and modify the same graph representation without maintaining a separate copy for each process [Pas23].

A relevant aspect of this implementation is the use of a largely lock-free strategy. Introducing locks around every graph modification would provide stronger synchronization but could also reduce the benefits of parallel execution by forcing processes to wait for one another. The proposed approach therefore minimizes synchronization and allows concurrent updates to the shared structures [Pas23]. This choice favors execution speed while accepting that concurrent operations may introduce additional approximation into the constructed neighborhood graph.

The computation of the minimum spanning tree is also parallelized. Instead of constructing the complete tree through a single sequential execution, the available edge set is divided among multiple processes. Each process computes a local minimum spanning tree over its assigned edges. The locally produced trees are then collected and their edges are combined, after which a final Kruskal step is applied to obtain the global minimum spanning tree [Pas23]. In this way, a substantial part of the edge processing can be performed concurrently before the final global merge.

This strategy exploits an important property of minimum spanning trees: edges that cannot contribute to a minimum spanning tree within a local subset can be discarded before the final merge. Consequently, the final Kruskal step can operate on a reduced collection of candidate edges rather than on the entire original edge set. The approach therefore decreases the amount of work performed during the final sequential phase while preserving the overall MST construction procedure described in the parallel implementation [Pas23].

The experimental evaluation reported by Pastorino shows that parallelizing these stages can reduce the execution time of FISHDBC, particularly when the workload is sufficiently large to compensate for the overhead introduced by process creation, synchronization, and shared-memory management [Pas23]. The results also illustrate that increasing parallelism is not free: the benefits depend on the dataset, the number of processes, and the additional approximation introduced by concurrent HNSW construction.

Despite these improvements, the architecture remains based on shared memory. The participating processes execute on the same physical machine and access common data structures, including the HNSW representation and other state required by the clustering procedure [Pas23]. The approach can therefore exploit multiple CPU cores, but its total memory and computational capacity remain bounded by the resources available on that machine.

This distinction is important for the problem addressed in this thesis. Moving from shared-memory parallelism to a distributed environment requires more than assigning FISHDBC operations to additional processes. Independent machines do not share the same HNSW, neighborhood information, or MST state. The data and computation must therefore be partitioned, and sufficient information must later be exchanged to reconstruct the global density and connectivity relationships required by the clustering algorithm.

Parallel FISHDBC nevertheless provides an important intermediate step between the original centralized FISHDBC implementation and a distributed execution model. It demonstrates that both neighborhood discovery and MST computation contain substantial opportunities for parallel execution. At the same time, its dependence on shared-memory data structures highlights the additional challenges that arise when the computation is distributed across separate machines.

The following section considers distributed HNSW approaches, with particular attention to Pyramid [Den+19]. Unlike parallel FISHDBC, these methods explicitly divide an HNSW-based workload across multiple machines and therefore provide useful ideas for partitioning, routing, and replication in a distributed environment.

3.3 Distributed HNSW and Pyramid

HNSW provides efficient approximate nearest-neighbor search on a single machine, but distributing an HNSW-based workload introduces an additional problem: deciding which machine should process a query. A straightforward distributed implementation can randomly divide the dataset among several workers and construct an independent HNSW index on each partition. However, random partitioning does not preserve similarity relationships between data items. Consequently, obtaining accurate search results may require sending each query to all workers and merging their results, which limits the throughput benefits of distribution [Den+19].

Pyramid addresses this problem by organizing HNSW-based similarity search across multiple machines while attempting to preserve locality between similar items [Den+19]. Instead of randomly assigning the complete dataset to workers, Pyramid first constructs a smaller global representation of the data, referred to as a *meta-HNSW*. This structure is then used to divide the dataset into balanced sub-datasets so that similar items are more likely to be assigned to the same worker.

The HNSW structures used within Pyramid rely on the standard graph-walk mechanism for both query processing and graph construction. During a query, the search starts from an entry point in the highest layer and progressively moves toward the lower layers. At each upper layer, a narrow search is used to identify a promising entry point for the next layer.

A broader search is then performed at the bottom layer to obtain the final approximate neighbors. During graph construction, the same search procedure is reused to identify suitable neighbors for a newly inserted item [Den+19].

Figure 3.1 presents the HNSW query-processing and graph-construction procedures used by Pyramid. These procedures describe the underlying HNSW operations rather than the complete distributed Pyramid architecture.

The distributed contribution of Pyramid is built around a two-level organization. First, a random sample of the original dataset is selected and further reduced through clustering. The resulting cluster representatives are used to construct a smaller meta-HNSW, which provides an approximate global representation of the dataset. The bottom layer of this meta-HNSW is then partitioned into balanced sub-datasets using graph partitioning [Den+19].

The approach developed in this thesis differs at this stage. Rather than performing an additional centroid-based reduction, the Meta-HNSW is constructed directly from the sampled data points. This choice is consistent with the objective of retaining support for general distance functions, for which a meaningful centroid representation may not be available.

After the meta-HNSW has been partitioned, the complete dataset is assigned to the resulting partitions. Each data item is associated with an appropriate region of the meta-HNSW and is routed to the corresponding worker. The data are therefore distributed according to similarity rather than through a purely random assignment. Once this redistribution is complete, each worker constructs an independent local HNSW over the data assigned to its partition [Den+19].

The resulting architecture consists of a small global meta-HNSW and multiple worker-local HNSW structures. The meta-HNSW provides a global view that can be used to determine which partitions are likely to contain relevant data, whereas the local HNSW structures perform the more expensive search over the full data stored at each worker.

This organization is also used during query processing. A query is first evaluated against the meta-HNSW to identify the most relevant partitions. Rather than broadcasting the query to every worker, Pyramid forwards it only to the workers associated with the selected regions of the meta-HNSW. Each selected worker then searches its local HNSW and returns approximate nearest neighbors. The results produced by the workers are finally combined to obtain the result of the distributed query [Den+19].

The number of sub-datasets searched for a query introduces a trade-off between search cost and result quality. Sending a query to more partitions increases the probability that relevant neighbors are found, but also increases the amount of computation and communication required. Searching fewer partitions reduces the distributed workload, but increases the possibility of missing neighbors located in another part of the distributed index [Den+19].

(a) HNSW query processing via graph walk

```
1: Input: HNSW graph  $G$ , similarity function  $s$ , query  $q$ , search factor  $l$  for the bottom layer
2: Output:  $k$  approximate neighbors of  $q$ 
3: Set queue  $C.add(G.entryVertex, s(q, G.entryVertex))$ 

4: for  $t = G.maxLayer$  downto 1 do
5:    $C \leftarrow \text{Search-Level}(G, t, q, s, C, 1)$ 
6:    $C \leftarrow \{max(C)\}$ 
7: end for
8:  $C \leftarrow \text{Search-Level}(G, 0, q, s, C, l)$ 
9: return top  $k$  vertices with the maximum  $s(q, x)$  in  $C$ 

10: function SEARCH-LEVEL( $G, level, q, s, C, factor$ )
11:    $W = C$ 
12:   while  $C.size > 0$  &  $s(q, max(C)) \geq s(q, min(W))$  do
13:      $v_{curr} \leftarrow C.pop-max()$ 
14:     for each neighbor  $v$  of  $v_{curr}$  in  $G.level$  do
15:       if  $v$  is not checked then
16:          $C.add(v, s(q, v)), W.add(v, s(q, v))$ 
17:         Resize  $W$  to  $factor$  by keeping the most similar items
18:       end if
19:     end for
20:   end while
21:   return  $W$ 
22: end function
```

(b) HNSW graph construction

```
1: Input: dataset  $X$ , similarity function  $s$ , maximum vertex degree  $M$ , search factor  $l$  for graph walk
2: Output: HNSW proximity graph  $G$ 
3: Initialize  $G = \emptyset$ 
4: for each item  $x$  in  $X$  do
5:    $u = random()$  and  $C.add(G.entryVertex)$ 
6:   for  $t = G.maxLayer$  downto  $u + 1$  do
7:      $C \leftarrow \text{Search-Level}(G, t, x, s, C, 1)$ 
8:      $C \leftarrow \{max(C)\}$ 
9:   end for
10:  for  $t = u$  downto 0 do
11:     $C \leftarrow \text{Search-Level}(G, t, x, s, C, l)$ 
12:    Connect  $x$  to its top  $M$  neighbors in  $C$  for graph  $G.t$ 
13:     $C \leftarrow \{max(C)\}$ 
14:  end for
15: end for
16: return  $G$ 
```

Figure 3.1: HNSW query-processing and graph-construction procedures used in Pyramid, adapted from Deng et al. [Den+19].

Pyramid also introduces replication, but its purpose differs from the point-level replication used in this thesis. In Pyramid, complete sub-HNSW indexes can be replicated across machines to improve availability and mitigate the effect of slow or unavailable workers. The replication therefore concerns distributed search infrastructure rather than the creation of overlapping point-level neighborhoods near partition boundaries.

The approach proposed in this thesis instead replicates individual data points across selected worker partitions. Its purpose is to increase the probability that neighborhood relationships crossing a partition boundary are observed during worker-local processing.

The main advantage of Pyramid over a naive distributed HNSW is therefore not a modification of the basic HNSW search algorithm itself, but the way the global index is organized. The sampled meta-HNSW provides information about the global distribution of the data, graph partitioning preserves locality, and local HNSW indexes allow most of the search work to be performed independently by the workers. This avoids requiring every query to be processed by every machine.

These ideas are particularly relevant to the distributed clustering problem considered in this thesis. A sampled global structure provides a mechanism for obtaining an approximate view of the complete dataset before the full data are distributed, while graph partitioning provides a way to place similar points on the same worker. Worker-local HNSW structures can then perform neighborhood discovery independently after the partitioning phase.

However, Pyramid and FISHDBC have different objectives. Pyramid is designed for distributed approximate nearest-neighbor search. Given a query, its coordinator needs to combine the candidate neighbors returned by the selected workers. It does not perform density-based clustering and therefore does not need to reconstruct global core distances, mutual-reachability distances, or the minimum spanning tree required by an HDBSCAN*-style clustering procedure.

This distinction is especially important at partition boundaries. For nearest-neighbor search, the routing mechanism aims to access enough relevant partitions to retrieve high-quality approximate neighbors. In a FISHDBC-style clustering procedure, incomplete neighborhood information may also change the estimated core distance of a point. Since core distances contribute directly to mutual-reachability weights, missing cross-partition relationships can in turn affect the edges selected for the global minimum spanning tree.

Consequently, the distributed organization proposed by Pyramid cannot be applied directly as a complete solution for FISHDBC. Its sampled global structure, locality-aware partitioning, replication, and worker-local HNSW indexes provide useful mechanisms for distributing neighborhood discovery, but an additional global reconstruction stage is required for hierarchical density-based clustering. In particular, information obtained independently by the workers must be combined to estimate global neighborhood properties and to reconstruct the connectivity required by the final clustering hierarchy.

Pyramid therefore provides a complementary contribution to the previous work discussed in this chapter. FISHDBC provides the hierarchical density-based clustering model, while parallel FISHDBC demonstrates how some of its computational stages can be parallelized within a shared-memory machine. Pyramid, in contrast, demonstrates how HNSW-based computation can be partitioned and coordinated across independent machines. These complementary ideas form the basis for positioning the distributed approach developed in this thesis, which is discussed in the following section.

3.4 Positioning of This Work

The related work discussed in this chapter addresses different parts of the problem considered in this thesis. Distributed DBSCAN variants show how density-based clustering can be divided across multiple processing units, while FISHDBC introduces a flexible and incremental approximation of HDBSCAN* based on HNSW and minimum spanning tree maintenance. Parallel FISHDBC demonstrates that important parts of the FISHDBC workflow can be executed concurrently in a shared-memory environment, and Pyramid shows how HNSW-based similarity search can be organized across multiple machines using a sampled global structure, graph partitioning, and worker-local indexes.

These approaches are complementary, but none of them directly addresses the same combination of requirements considered in this thesis. Distributed DBSCAN methods mainly target flat clustering and typically focus on local cluster construction followed by some form of global merging. FISHDBC, in contrast, maintains an approximate hierarchical density-based clustering structure incrementally, but its main state is centralized [Del19]. Pastorino’s parallel implementation improves the execution of FISHDBC by exploiting multiple processes and shared-memory data structures, but it remains limited to a single physical machine [Pas23]. Pyramid distributes HNSW-based processing across multiple machines, but its objective is approximate nearest-neighbor search rather than density-based clustering [Den+19].

The approach developed in this thesis is positioned between these lines of work. It retains the main clustering principles of FISHDBC, particularly the use of approximate neighborhood information, core distances, mutual-reachability relationships, and a minimum spanning tree for hierarchical cluster extraction. At the same time, it adopts the general idea of using a sampled global HNSW structure to organize the dataset before distributing the computation, following the type of locality-aware organization explored by Pyramid.

The central difference is that the distributed workers do not independently produce final clustering results. Each worker operates only on a local partition and constructs a local HNSW-based view of the data assigned to it. From this local structure, the worker extracts neighborhood information and connectivity information that can later contribute to the

reconstruction of the global clustering structure.

This distinction is necessary because local density information cannot in general be treated as globally valid. A point may have relevant neighbors in another partition, particularly near partition boundaries. Computing mutual-reachability weights entirely from a local view could therefore use incorrect core-distance estimates. The proposed architecture consequently separates local neighbor discovery from global density reconstruction.

In the proposed design, the results produced by the workers are combined by a central Leader. Local mutual-reachability weights are used only to select a reduced set of connectivity edges, while the raw distances of the retained edges are transmitted. The Leader combines these edges with the information provided by the Indexer, reconstructs approximate global core distances, and recomputes mutual-reachability weights consistently over the merged candidate graph.

Replication is also used to reduce the effect of strict partition boundaries. When replication is enabled, each routed point can be assigned to up to R distinct nearby worker partitions. This introduces controlled overlap between partitions and helps recover neighborhood relationships that may otherwise be separated by partition boundaries.

The resulting design therefore separates the clustering workflow into two levels. The distributed workers perform local and computationally expensive neighbor discovery, while the leader is responsible for recovering the global information required for the final hierarchical structure. This separation is the main architectural idea developed in the next chapter.

The following chapter presents the design of the proposed distributed FISHDBC architecture in detail, including the roles of the indexer, workers, and leader, the partitioning and replication strategy, and the reconstruction of global core distances and the final minimum spanning tree.

Chapter 4

Design and Implementation

This chapter presents the design and implementation of the distributed FISHDBC approach developed in this thesis. The objective is to distribute the most computationally expensive neighborhood-discovery operations while retaining enough density and connectivity information to reconstruct the graph required by HDBSCAN*. The system separates the computation into three main roles: an indexer, a set of workers, and a leader.

The indexer constructs a sampled global HNSW, partitions its bottom layer, and uses the resulting landmark partitions to distribute the complete dataset among the workers. During HNSW construction and routing, the indexer also records the raw distances evaluated by the approximate search process. Each worker then constructs a local HNSW and performs the same type of distance harvesting on its assigned data.

Both the indexer and workers reduce their observed distance graphs before communication. They retain observed k -nearest-neighbor edges together with edges selected through a locally weighted minimum spanning tree. The local tree is constructed using mutual-reachability distances derived from locally available core-distance estimates. The original raw distance of every retained edge is preserved and transmitted to the leader.

The leader combines the edge sets produced by the indexer and workers, reconstructs approximate global neighborhoods and core distances, recomputes mutual-reachability weights using this combined information, and constructs the global minimum spanning tree. The clustering hierarchy and final labels are then derived from this tree.

The method remains approximate because HNSW evaluates only a subset of all possible pairwise distances. Consequently, the final clustering depends on the candidate relationships discovered during global and local HNSW processing rather than on a complete all-pairs distance graph.

4.1 Design Goals

The first design goal is to distribute neighborhood discovery. HNSW construction and search require a large number of distance evaluations, and performing all of these operations on one machine limits scalability. The dataset is therefore divided among multiple workers that construct and query independent local HNSW indexes.

A second goal is to preserve locality during data distribution. Random partitioning could place many nearby points on different workers. The indexer instead constructs a sampled global HNSW and partitions its bottom-layer graph, making related landmarks more likely to remain in the same partition.

A third goal is to preserve useful density and connectivity information without sending every harvested distance to the leader. Both the indexer and workers therefore retain a compact union of observed k -nearest-neighbor edges and locally selected MST edges.

A fourth goal is to avoid treating local density estimates as globally final. A worker observes only a subset of the complete dataset, so its local core distance may be larger than the estimate obtained after information from several partitions is combined. Local core distances and local mutual-reachability weights are therefore used only for selecting candidate edges. The leader later reconstructs global core-distance estimates and recomputes the final mutual-reachability weights.

Finally, the design aims to limit the effect of partition boundaries. Replication allows the same point to be processed in more than one nearby partition, increasing the possibility that relationships crossing a partition boundary are observed locally.

4.2 System Architecture

The architecture contains three roles: the *indexer*, the *workers*, and the *leader*. Their responsibilities correspond to global organization, parallel local graph construction, and global reconstruction, respectively.

The indexer samples the input dataset and builds a Meta-HNSW over the sampled points, referred to as landmarks. Its bottom layer is partitioned using KaHIP, and the resulting landmark-to-partition mapping is used to route every original data point to one or more workers. The indexer also records raw distances computed during Meta-HNSW construction and routing.

Each worker receives a locality-aware subset of the dataset. It builds a local HNSW, records the raw distances evaluated during index construction and search, derives an observed k -nearest-neighbor graph, estimates local core distances, and computes local mutual-

reachability weights. These weights are used to construct a local MST. The worker sends the union of its observed k -NN edges and MST edges to the leader while preserving the original raw distance of every edge.

The indexer applies the same graph-reduction principle to the distances observed during Meta-HNSW construction and routing. Its reduced edge set therefore provides global-scale graph evidence, while the workers provide more detailed local evidence.

The leader merges these edge sets, removes duplicates, reconstructs approximate global neighborhoods, computes global core-distance estimates, and reweights the candidate edges using mutual-reachability distance. Kruskal’s algorithm is then applied to construct the global MST, from which the HDBSCAN* hierarchy and final clustering are obtained.

Figure 4.1 illustrates the overall processing flow.

4.3 Global Sampling and Partitioning

The Indexer first constructs an approximate global representation of the dataset for partitioning and routing. Let the complete dataset be denoted by D and the sampling rate by r . A landmark set D' is drawn uniformly without replacement from D using the configured random seed, with size

$$|D'| = \lfloor r|D| \rfloor. \quad (4.1)$$

The sampled points act as landmarks and are used to construct the Meta-HNSW.

Using only a fraction of the complete dataset reduces the amount of global HNSW construction work performed before distribution, while still providing an approximate representation of the relationships between different regions of the dataset.

Landmark Graph Partitioning

After constructing the Meta-HNSW, Layer 0 is extracted as a graph. This layer contains all sampled landmarks together with their HNSW adjacency relationships. The resulting graph is divided into P partitions using KaHIP [SS13].

HNSW edges are naturally represented by distances, whereas the graph partitioner interprets larger weights as stronger relationships. The raw edge distance d is therefore converted into an integer similarity weight using

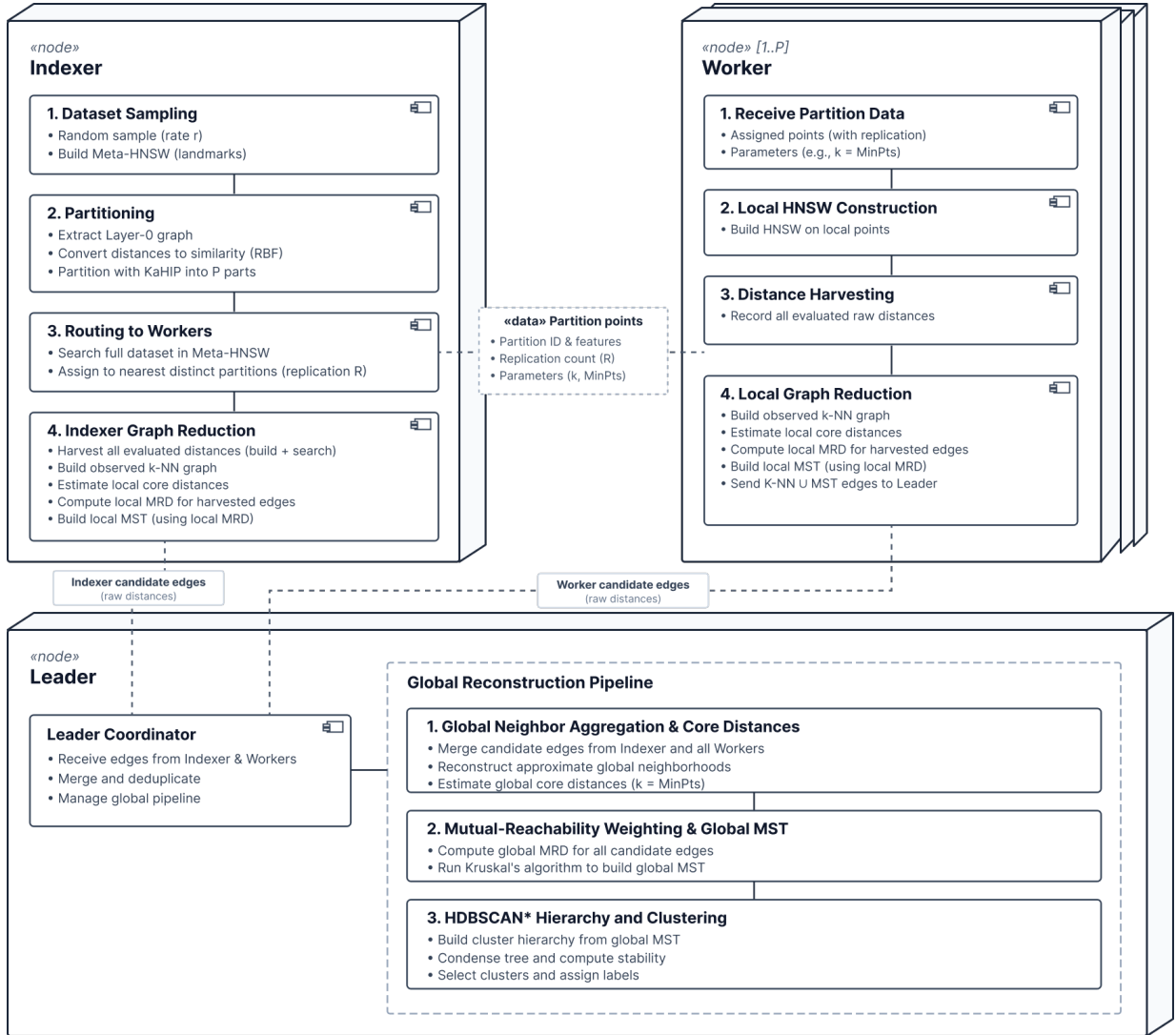


Figure 4.1: Overall architecture of the distributed FISHDBC approach.

$$w(d) = \max \left\{ 1, \left\lfloor W_{\max} \exp \left(-\frac{d^2}{2\sigma^2} \right) \right\rfloor \right\}, \quad (4.2)$$

where $W_{\max} = 10\,000$. When no explicit scale is provided, σ is derived from the median undirected edge distance of the extracted Layer 0 graph. Consequently, nearby landmarks receive larger weights and are more expensive for the partitioner to separate.

The implementation uses KaHIP version 3.25 with the **FAST** preset. The reference configuration uses an imbalance tolerance of 0.03 and partitioning seed 42. Landmark vertices receive equal node weights, so the balancing objective is based on the number of landmarks assigned to each partition.

The output of this stage is a mapping between every sampled landmark and one of the P logical worker partitions.

Routing the Complete Dataset

Every original point $x \in D$ is then searched against the Meta-HNSW. Nearby landmarks and their associated partition identifiers determine where the point will be processed.

When the replication factor is $R = 1$, a point is assigned to its primary partition. When $R > 1$, additional nearby landmarks belonging to distinct partitions can cause the point to be assigned to more than one worker.

This procedure attempts to place related points together while allowing controlled overlap near partition boundaries.

4.4 Indexer Graph Reduction

The role of the indexer does not end after partitioning and routing. During Meta-HNSW construction and searches, it records the raw distances evaluated by HNSW. These observations contain more information than the final Layer 0 adjacency because they include point pairs explicitly compared during HNSW processing.

Sending every harvested distance to the leader would introduce a large communication and memory cost. The indexer therefore reduces its observed graph before transmission.

For each point, the observed neighbors are ordered according to raw distance, and the closest k neighbors are retained. These edges form an observed k -nearest-neighbor graph. From this graph the indexer estimates

Algorithm 2 Global sampling, partitioning, and routing.

Require: dataset D , sampling rate r , number of partitions P , replication factor R

- 1: $D' \leftarrow$ sample points from D using rate r
 - 2: $H_G \leftarrow$ build Meta-HNSW on D'
 - 3: $G \leftarrow$ extract Layer 0 from H_G
 - 4: transform graph distances using Equation 4.2
 - 5: $\Pi \leftarrow$ partition G into P parts using KaHIP
 - 6: **for** each point $x \in D$ **do**
 - 7: search x in H_G
 - 8: obtain nearby landmarks and their partition identifiers
 - 9: assign x to its primary partition
 - 10: **if** $R > 1$ **then**
 - 11: assign x to additional nearest distinct partitions
 - 12: **end if**
 - 13: **end for**
 - 14: send each partition to its corresponding worker
-

$$c_I(x) = d\left(x, N_I^{(k)}(x)\right), \quad (4.3)$$

where $N_I^{(k)}(x)$ denotes the k -th closest neighbor of x among the distances observed by the indexer.

If fewer than k observed neighbors are available for a point, the Indexer assigns an infinite local core distance for the purpose of local mutual-reachability weighting. This follows the same conservative rule used by the Workers and prevents an incomplete observed neighborhood from being treated as a reliable local density estimate.

For every harvested edge (x, y) , a local mutual-reachability weight is then computed as

$$d_{\text{mreach}}^I(x, y) = \max\{c_I(x), c_I(y), d(x, y)\}. \quad (4.4)$$

Kruskal’s [Kru56] algorithm is applied to these weights to construct an indexer-side MST. The final reduced edge collection is

$$E_I = E_{kNN}^I \cup E_{MST}^I. \quad (4.5)$$

Only the original raw distance $d(x, y)$ is transmitted for each retained edge. The locally computed mutual-reachability value is used for edge selection but is not considered the final global weight.

Algorithm 3 Indexer-side graph reduction

Require: Harvested raw-distance observations E_I^{raw} , neighborhood size k

```
1: for each point  $x$  do
2:   select the  $k$  closest observed neighbors of  $x$ 
3:   if at least  $k$  neighbors are available then
4:     estimate local core distance  $c_I(x)$ 
5:   else
6:      $c_I(x) \leftarrow \infty$ 
7:   end if
8: end for
9: for each harvested edge  $(x, y)$  do
10:   $w(x, y) \leftarrow \max\{c_I(x), c_I(y), d(x, y)\}$ 
11: end for
12: construct Indexer-side MST  $T_I$  using weights  $w$ 
13:  $E_I \leftarrow E_{kNN}^I \cup T_I$ 
14: send retained edges with their raw distances to the Leader
```

The use of an MST as a reduction step follows the general property that, once the relevant edge weights are fixed, edges excluded from a minimum spanning tree of a candidate graph are not required to reconstruct an MST over that same weighted graph [Epp94]. In the proposed method, this reduction is applied using the locally computed mutual-reachability weights rather than the raw pairwise distances.

It is important to note that this reduction does not provide an exact guarantee with respect to the final global mutual-reachability graph. The Indexer-side mutual-reachability weights are computed using locally available core-distance estimates, whereas the Leader later reconstructs approximate global core distances and recomputes the edge weights. Therefore, the local MST is used as an approximate connectivity-preserving reduction mechanism rather than as a proof that all edges irrelevant to the final global MST have been removed. This follows the approximate philosophy of FISHDBC, where neighborhood observations and candidate edges are used to construct an approximate hierarchical clustering [Del19].

4.5 Worker-Side Local Processing

Each worker receives the points assigned to one logical partition, including any replicated points. It constructs a local HNSW and performs searches over the points assigned to that worker.

The HNSW implementation used by the prototype is reused from the original FISHDBC

implementation released by Matteo Dell’Amico in the `flexible-clustering` project [Del]. The implementation is included in the project source and adapted so that raw distance evaluations can be recorded during HNSW insertion and search.

This is important because FISHDBC does not rely only on the final HNSW Layer 0 adjacency. The distributed implementation likewise records all raw pairwise distances evaluated by the HNSW process and builds its candidate graph from these observations.

For each local point x , the harvested neighbors are ordered according to raw distance and the nearest k are retained. These edges form an approximate local k -nearest-neighbor graph and are used to estimate

$$c_i(x) = d\left(x, N_i^{(k)}(x)\right). \quad (4.6)$$

If fewer than the required number of neighbors have been observed, the implementation assigns an infinite local core distance for the local edge-selection procedure. This conservative value prevents an incomplete local neighborhood from being interpreted as a reliable density estimate.

Every harvested edge is then weighted using

$$d_{\text{mreach}}^i(x, y) = \max\{c_i(x), c_i(y), d(x, y)\}. \quad (4.7)$$

The worker applies Kruskal’s algorithm to these locally weighted edges and constructs a local MST. The worker output is

$$E_i = E_{kNN}^i \cup E_{MST}^i. \quad (4.8)$$

For every retained edge, the worker sends

$$(x, y, d(x, y)), \quad (4.9)$$

where x and y are represented using their global point identifiers.

The local mutual-reachability weight is therefore used to decide which connectivity edges are retained, while the original raw distance is preserved for the later global reconstruction.

The two retained edge categories have complementary roles. The k -NN edges preserve neighborhood information required for core-distance reconstruction, while the MST edges provide a compact representation of locally observed density-aware connectivity.

Algorithm 4 Worker-side local graph construction.

Require: local partition D_i , neighborhood size k

```

1:  $H_i \leftarrow$  build local HNSW on  $D_i$ 
2: harvest all raw distances evaluated by  $H_i$ 
3: for each point  $x \in D_i$  do
4:   select the  $k$  closest observed neighbors of  $x$ 
5:   if at least  $k$  neighbors are available then
6:     estimate local core distance  $c_i(x)$ 
7:   else
8:      $c_i(x) \leftarrow \infty$ 
9:   end if
10: end for
11: for each harvested edge  $(x, y)$  do
12:    $w(x, y) \leftarrow \max\{c_i(x), c_i(y), d(x, y)\}$ 
13: end for
14: construct local MST  $T_i$  using weights  $w$ 
15:  $E_i \leftarrow E_{kNN}^i \cup T_i$ 
16: send retained edges with their raw distances to the leader

```

4.6 Global Core-Distance Reconstruction

The leader receives one reduced raw-edge set from the indexer and one from each worker. These sets are combined into the global candidate collection

$$E_{\text{candidate}} = E_I \cup \left(\bigcup_i E_i \right). \quad (4.10)$$

Edges are represented as undirected pairs. If the same edge is received from more than one source, duplicate observations are removed while retaining the raw pairwise distance.

The incident candidate edges of each point define its reconstructed global neighborhood. Let $N_G(x)$ denote the resulting candidate set for point x . The leader sorts these neighbors by raw distance and, when at least k neighbors are available, estimates

$$\hat{c}(x) = d\left(x, N_G^{(k)}(x)\right). \quad (4.11)$$

When at least k candidate neighbors are available, the reconstructed core-distance estimate has a useful monotonic property. Since the candidate set available to the Leader is a subset of the complete set of possible neighbors, its k -th smallest distance cannot be smaller than

the exact k -th-neighbor distance. Therefore,

$$\hat{c}(x) \geq c(x), \quad (4.12)$$

where $c(x)$ denotes the exact core distance computed from the complete dataset.

Furthermore, adding additional candidate neighbors can only preserve or decrease $\hat{c}(x)$. This explains why combining observations from several Workers, the Indexer, and replicated partitions can improve the core-distance estimate.

This property applies only when at least k candidate neighbors are available. It does not hold for the implementation fallback used when fewer than k candidates have been reconstructed.

When fewer than k candidates are available but the point has at least one observed neighbor, the implementation uses the distance to the farthest available candidate as a fallback estimate. If no candidate neighbor is available, the implementation assigns a fallback value of zero.

These cases are explicitly recorded in the experimental statistics. In particular, the implementation reports the number of points with no candidates, the number with fewer than k candidates, and the corresponding incomplete neighborhood fraction.

The reconstructed value remains approximate because HNSW does not evaluate all possible pairwise distances. Nevertheless, it is generally more informed than a single worker-local estimate because the leader combines evidence from the indexer, different workers, and replicated observations.

4.7 Mutual-Reachability Reconstruction

After reconstructing approximate global core distances, the leader reweights every retained candidate edge using one common density estimate.

For each edge (x, y) ,

$$d_{\text{mreach}}^G(x, y) = \max \{ \hat{c}(x), \hat{c}(y), d(x, y) \}. \quad (4.13)$$

This calculation is distinct from the local mutual-reachability weighting performed by the indexer and workers. Local MRD is used only to select connectivity edges. The leader recomputes MRD because its core-distance estimates are based on a larger collection of candidate relationships.

Algorithm 5 Approximate global core-distance reconstruction.

Require: indexer edge set E_I , worker edge sets $E_1, \dots, E_P$, neighborhood size k

```

1:  $E \leftarrow E_I \cup (\bigcup_i E_i)$ 
2: remove duplicate undirected edges
3: for each point  $x$  do
4:   collect all candidate neighbors incident to  $x$ 
5:   sort candidates by raw distance
6:   if at least  $k$  candidates are available then
7:      $\hat{c}(x) \leftarrow$  distance to the  $k$ -th candidate
8:   else if at least one candidate is available then
9:      $\hat{c}(x) \leftarrow$  distance to the farthest available candidate
10:    mark neighborhood as incomplete
11:   else
12:      $\hat{c}(x) \leftarrow 0$ 
13:    mark neighborhood as incomplete
14:   end if
15: end for
16: return reconstructed core-distance estimates  $\hat{c}$ 

```

If a point appears in several worker partitions, the leader may receive additional close neighbors for that point. Its reconstructed global core distance can therefore be smaller than an estimate obtained from only one local partition. Reusing the local MRD weights would consequently be inconsistent with the merged global candidate graph.

Algorithm 6 Global mutual-reachability reconstruction.

Require: candidate raw edges $E_{\text{candidate}}$, reconstructed core distances $\hat{c}$

```

1:  $E_{\text{mreach}} \leftarrow \emptyset$ 
2: for each  $(x, y, d(x, y)) \in E_{\text{candidate}}$  do
3:    $w \leftarrow \max\{\hat{c}(x), \hat{c}(y), d(x, y)\}$ 
4:   add  $(x, y, w)$  to  $E_{\text{mreach}}$ 
5: end for
6: return  $E_{\text{mreach}}$ 

```

The result is one globally reweighted candidate graph. Its edge set remains approximate, but all retained edges are interpreted using the same reconstructed density information.

4.8 Global Minimum Spanning Tree Construction

The leader applies Kruskal’s algorithm to the reconstructed mutual-reachability graph

$$G_{\text{mreach}} = (V, E_{\text{mreach}}), \quad (4.14)$$

where V contains all data points and E_{mreach} contains the retained candidate edges with their global mutual-reachability weights.

Edges are processed in increasing order of weight. An edge is accepted when it connects two components that are not yet connected. The procedure continues until $|V| - 1$ edges have been selected or the candidate edges have been exhausted.

Algorithm 7 Global minimum spanning tree construction.

Require: global MRD edges E_{mreach} , vertex set V

```

1: sort  $E_{\text{mreach}}$  by increasing weight
2: initialize one disjoint set for each vertex
3:  $T_{\text{global}} \leftarrow \emptyset$ 
4: for each edge  $(x, y, w) \in E_{\text{mreach}}$  do
5:   if  $x$  and  $y$  belong to different components then
6:     add  $(x, y, w)$  to  $T_{\text{global}}$ 
7:     merge their components
8:   end if
9:   if  $|T_{\text{global}}| = |V| - 1$  then
10:    break
11:   end if
12: end for
13: return  $T_{\text{global}}$ 

```

The implementation verifies that the resulting tree contains $|V| - 1$ edges. This condition indicates that the reconstructed candidate graph is connected. The number of connected components and whether the MST is complete are recorded in the run statistics.

If a complete tree cannot be constructed, the candidate graph is considered disconnected and the run is treated as an unsuccessful global reconstruction rather than silently interpreting the incomplete structure as a complete MST.

The resulting global tree remains approximate because both the available candidate edges and reconstructed core distances originate from incomplete HNSW observations. The method therefore constructs the MST of the reconstructed candidate graph rather than claiming to reproduce the MST of the complete pairwise graph.

Once the global MST has been obtained, the remaining stages follow the HDBSCAN* procedure: construction of the single-linkage hierarchy, condensation according to the minimum cluster size, cluster-stability computation, EOM-based cluster selection, and assignment of labels and membership probabilities.

4.9 Replication and Boundary Handling

Partitioning introduces boundaries between worker-local views. Although the landmark graph is partitioned to preserve locality, some close points may still be assigned to different workers.

For each point, the landmarks returned by the Meta-HNSW search are considered in increasing distance order. Their partition identifiers are deduplicated, and the first R distinct partitions are selected.

Consequently, when the replication factor is greater than one, each routed point is assigned to up to R distinct nearby worker partitions, according to the partition identifiers of the landmarks returned by the Meta-HNSW search.

This allows the same point to participate in several local HNSW structures. A relationship that is not observed in one partition may therefore be discovered in another, increasing the candidate information available to the leader.

Replication can improve reconstructed neighborhoods and connectivity, but it also increases network traffic, local HNSW construction, distance evaluations, memory consumption, and worker output size. The replication factor therefore controls a trade-off between additional cross-partition evidence and duplicated distributed work.

The point-level replication used in this thesis should not be confused with the replication mechanism used by Pyramid. In Pyramid, replication is primarily applied to distributed search partitions for availability and straggler handling, whereas the present approach uses overlapping point assignments to increase neighborhood evidence near partition boundaries.

Replication does not guarantee exact reconstruction of the original global neighborhood graph. It only increases the opportunity for relevant cross-partition relationships to be observed.

4.10 Implementation

The distributed FISHDBC prototype is implemented in Python as independent Indexer, Worker, and Leader services. Each service maintains its own local state and exchanges required intermediate data explicitly rather than through shared memory.

The HNSW implementation used by both the Indexer and Workers is reused and adapted from Matteo Dell’Amico’s original FISHDBC implementation in the `flexible-clustering` project [Del]. Reusing this implementation is particularly useful because FISHDBC relies

on the distance evaluations performed during HNSW processing rather than only on the final HNSW graph.

The prototype follows a staged batch-execution model. The Indexer performs global preparation and routing, the Workers process their assigned partitions independently and in parallel, and the Leader performs global reconstruction once the required candidate-edge sets have been received.

The HNSW connectivity parameter M is distinct from the clustering neighborhood parameter $MinPts$. In the distributed prototype, $M = 5$ controls the maximum connectivity of the HNSW graph, while $MinPts = 15$ determines the number of non-self neighbors required for core-distance estimation. The graph-reduction neighborhood budget is therefore defined independently of the HNSW graph degree.

The value $M = 5$ was selected as the fixed HNSW configuration used throughout the experimental evaluation and was not varied as an experimental parameter.

4.10.1 Software Components

The three roles are implemented as independent Python services. The Indexer and Workers include the adapted FISHDBC HNSW implementation, while the Leader uses the received raw-distance edge collections to perform global graph reconstruction.

The services run in separate containers and do not rely on shared in-memory objects. This keeps the implementation aligned with the distributed architecture and allows each role to execute on a different physical machine.

The current containers use Python 3.14. NumPy is used for numerical data representation, KaHIP for graph partitioning, FastAPI for service endpoints, and the `hdbscan` package is used by the Leader for the final hierarchy and cluster-extraction operations.

4.10.2 Communication and Data Exchange

Communication between the distributed services is implemented over HTTP. FastAPI provides the Worker and Leader service endpoints, while HTTP clients are used by the Indexer and Workers to transmit intermediate results.

The main communication paths are

$$\text{Indexer} \rightarrow \text{Workers}, \tag{4.15}$$

$$\text{Indexer} \rightarrow \text{Leader}, \quad (4.16)$$

and

$$\text{Workers} \rightarrow \text{Leader}. \quad (4.17)$$

The Indexer sends each Worker the points assigned to its logical partition. Each point retains its global identifier together with the feature vector required for local distance evaluation.

The Indexer sends its reduced candidate-edge union directly to the Leader. Similarly, each Worker sends the union of its local k -NN and MST edges. A candidate edge has the conceptual representation

$$(u, v, d(u, v)), \quad (4.18)$$

where u and v are global identifiers and $d(u, v)$ is the raw pairwise distance.

The current implementation exposes, among others, the following communication paths:

- Indexer to Worker: `/ingest`;
- Indexer to Leader: `/leader/cache`;
- Worker to Leader: `/leader/results`.

Intermediate Python objects are serialized using `pickle` with the highest protocol supported by the executing Python version and transferred as `application/octet-stream` HTTP request bodies.

The Leader canonicalizes received edges as undirected point pairs and removes duplicate observations before reconstruction. Global identifiers make it possible to recognize the same points and edges when replication causes them to appear in multiple worker outputs.

Each message also contains run and partition metadata. The Leader starts global reconstruction only after the Indexer edge set and all expected Worker results for the current run have been received. Results from previous executions are not mixed with a newly initialized run.

4.10.3 Configuration

The prototype exposes the main algorithmic and distributed parameters through the experiment configuration, allowing different executions to be performed without modifying the implementation. The main parameters control sampling, partitioning, replication, neighborhood reconstruction, HNSW search, and HDBSCAN* cluster extraction.

The reference configuration uses a sampling rate of 0.10 and divides the landmark graph into ten logical worker partitions. Unless otherwise specified, the replication factor is 1, meaning that each point is assigned to one worker partition. The experiments vary these parameters independently when studying the effects of sampling and replication.

Core-distance estimation uses $MinPts = 15$, while the minimum cluster size used during HDBSCAN* extraction is also set to 15. The neighborhood budget used during graph reduction is chosen to be at least $MinPts$, ensuring that the Indexer and Workers retain enough observed neighbors to estimate a core distance whenever sufficient candidate information has been discovered.

Throughout the distributed implementation, a point is excluded from its own neighborhood. Self-distance edges are discarded by both the Indexer and the Workers, so $MinPts$ refers to the number of distinct non-self neighbors used for core-distance estimation. The centralized FISHDBC baseline follows the same convention.

The standard HDBSCAN implementation used in the evaluation follows a different parameter convention in which the point itself is included when interpreting `min_samples`. To maintain the same effective number of non-self neighbors across the compared methods, the HDBSCAN baseline is therefore configured with

$$\text{min_samples}_{HDBSCAN} = MinPts + 1. \quad (4.19)$$

For example, when the distributed implementation and centralized FISHDBC use $MinPts = 15$, the HDBSCAN baseline uses `min_samples=16`.

The HNSW search effort is controlled through the `ef` parameter, which is set to 50 in the reference configuration. Increasing this parameter causes HNSW to explore more candidate vertices during search, potentially increasing the number of useful distance observations at the cost of additional computation.

Graph partitioning is performed using KaHIP with the **FAST** preset. The reference configuration uses an imbalance tolerance of 0.03 and partitioning seed 42. Landmark sampling uses seed 42. Fixed random seeds are used where appropriate so that repeated executions can use the same sampled landmarks and partitioning configuration when required.

Each experimental execution is associated with a run identifier, and its configuration is

stored together with the generated results. This allows clustering-quality measurements, execution times, graph statistics, and resource measurements to be associated with the exact parameter values used for a particular run.

Table 4.1 summarizes the principal parameters of the reference distributed configuration.

Table 4.1: Main parameters of the reference distributed configuration.

Parameter	Meaning	Reference value
Sampling rate	Fraction used for Meta-HNSW	0.10
Partitions	Number of logical worker partitions	10
Replication	Number of worker partitions per point	1
<i>MinPts</i>	Number of non-self neighbors for core distance	15
Minimum cluster size	HDBSCAN* cluster-extraction parameter	15
Search <i>ef</i>	HNSW search effort	50
KaHIP mode	Graph-partitioning preset	FAST
KaHIP imbalance	Allowed partition imbalance	0.03
Partition seed	KaHIP random seed	42
Sample seed	Landmark-sampling random seed	42
HNSW <i>M</i>	Maximum HNSW neighbors per upper layer	5

4.10.4 Docker Swarm Deployment

The prototype is deployed using Docker containers orchestrated by Docker Swarm. Containerization provides a consistent execution environment, while Swarm maps the logical Indexer, Worker, and Leader roles onto separate physical machines.

The experimental cluster contains one front node and ten worker nodes. The front node acts as the Swarm manager and hosts the Indexer and Leader services. Worker services are placed on the remaining machines using node labels and placement constraints, with one logical worker assigned to each physical worker node.

The services communicate through an attachable Docker overlay network. Service discovery allows containers to communicate using logical service names rather than fixed container addresses.

In the reference deployment, the Indexer and Leader are each configured with an 8 GB memory limit and a 4 GB reservation. Worker containers are configured with a 12 GB memory limit and a 4 GB reservation.

Docker volumes are used to persist generated datasets and experimental results outside the lifetime of individual containers. This also allows repeated runs to reuse an immutable dataset while storing the output of each repetition separately.

The orchestration layer is kept independent from the clustering logic. Changing service placement or the number of available machines does not require changes to the Worker algorithm itself, provided that the logical partitioning and deployment configuration are adjusted consistently.

4.10.5 Implementation Considerations

Several practical aspects affect the scalability and interpretation of the current prototype.

Distance harvesting and graph reduction. The Indexer and Workers record raw distances evaluated during HNSW construction and search. These observations provide richer information than final HNSW adjacency alone, but their number can become large. Sending the complete harvested graph would substantially increase network traffic and Leader memory requirements.

The implementation therefore retains only observed k -NN edges and edges selected by the local mutual-reachability MST. This reduction is particularly important for the Indexer because routing searches over the complete dataset can generate a large number of distance evaluations.

Centralized routing. Although local neighborhood discovery is distributed, the Indexer still processes every input point to determine its worker assignment. Each point must be searched against the Meta-HNSW before being routed.

Routing therefore remains centralized in the current prototype and can become a computational and memory bottleneck as the dataset grows. This behavior is measured explicitly in the scalability evaluation.

Per-role scalability. The distributed design reduces the amount of data processed by each Worker, but it does not remove all centralized state. The Indexer currently has access to the complete dataset in order to sample, route, and distribute the input points; the current implementation does not stream the dataset incrementally through the Indexer. Each Worker stores only its assigned partition, including any replicated points, together with its local HNSW and harvested-distance cache. The Leader does not store the complete feature vectors, but it maintains the global candidate-edge collection, reconstructed core distances, and the global MST.

Table 4.2 summarizes the main computational, memory, and communication responsibilities of the three roles. The exact cost of HNSW construction and search is data dependent,

so the table describes the dominant operations rather than assigning a fixed asymptotic bound to HNSW processing.

Table 4.2: Main computational, memory, and communication responsibilities of the distributed components.

Role	Main state	Main computation	Communication
Indexer	Complete input dataset, sampled Meta-HNSW, landmark partition mapping, and harvested-distance cache	Sampling, Meta-HNSW construction, graph partitioning, routing of all n points, and Indexer graph reduction	Partitioned and replicated data to Workers; reduced candidate-edge set to Leader
Worker i	Assigned local partition, local HNSW, and local harvested-distance cache	Local HNSW construction and search, observed k -NN extraction, local core-distance estimation, MRD weighting, and local MST construction	Reduced k -NN $\cup$ MST edge set to Leader
Leader	Global candidate-edge collection, reconstructed core distances, and global MST	Edge merging and deduplication, global core-distance reconstruction, mutual-reachability weighting, Kruskal’s algorithm, and hierarchy extraction	Receives reduced edge sets from Indexer and Workers

Incomplete neighborhoods. Local HNSW observations do not guarantee that every point has at least $MinPts$ candidate neighbors. Workers treat incomplete local core distances conservatively as infinity when selecting their local MST edges.

At the Leader, incomplete reconstructed neighborhoods are handled using the farthest available candidate distance, while a point with no candidate neighbors receives a fallback value of zero. These cases are recorded in the run statistics and are treated as approximation diagnostics rather than exact density estimates.

Replication and duplicate information. When replication is greater than one, the same point can be processed by multiple Workers. All copies retain the same global identifier, which allows the Leader to merge repeated observations.

Replication may improve the recovery of relationships near partition boundaries, but increases network traffic, local HNSW processing, memory consumption, and Worker output size.

Leader-side processing. The Leader does not receive complete Worker HNSW objects or feature matrices. It operates mainly on global point identifiers, raw-distance candidate edges, reconstructed core distances, and the resulting MST.

Nevertheless, global neighborhood reconstruction and MST construction still require state covering the complete dataset. The architecture therefore distributes local graph exploration but does not eliminate all centralized processing.

Disconnected candidate graphs. The Leader records the number of connected components of the reconstructed candidate graph and verifies whether the global MST contains $|V| - 1$ edges. A run for which this condition is not satisfied is considered to have an incomplete global reconstruction and is not treated as a valid complete MST.

Approximation. The final candidate graph is incomplete with respect to the complete pairwise distance graph. HNSW evaluates only a subset of possible distances, and graph reduction removes additional observed edges before communication.

The resulting clustering must therefore be interpreted as approximate. The k -NN edges preserve density information, while local MRD-based MST edges retain a compact subset of connectivity information for global reconstruction.

Batch execution. The original FISHDBC implementation supports incremental insertion of new data. The distributed prototype developed in this thesis instead uses a staged batch execution model. The Indexer prepares the distributed workload, Workers process their partitions, and the Leader reconstructs the final clustering after all required results have been collected.

Supporting continuous incremental updates across several machines would require additional distributed state-management and coordination mechanisms and is left outside the scope of the present work.

4.11 Design Trade-offs

The proposed design introduces trade-offs between neighborhood information, computation, memory, communication, and scalability.

The sampling rate controls the size of the Meta-HNSW. A larger sample provides a more detailed global representation but increases Meta-HNSW construction, routing, and distance-harvesting cost.

Partitioning reduces the amount of data processed by each Worker but introduces boundaries that may separate relevant neighbors. Locality-aware partitioning and replication reduce this effect without eliminating it completely.

Replication increases the chance that boundary relationships are observed locally, but also duplicates data transfer, local HNSW work, memory use, and Worker output.

Graph reduction creates another trade-off. Retaining every HNSW distance evaluation would preserve more information but would substantially increase communication and Leader processing. Instead, the Indexer and Workers transmit the union of observed k -NN edges and locally selected mutual-reachability-MST edges.

The architecture also retains centralized operations. The Indexer routes the complete dataset, while the Leader performs global neighborhood reconstruction, mutual-reachability weighting, and MST construction. This simplifies the consistency of globally dependent operations but creates potential centralized scalability bottlenecks.

Finally, the distributed method remains approximate because of sampling, HNSW neighbor discovery, partitioning, graph reduction, and reconstructed rather than exact global core distances. The design therefore aims to retain sufficient density and connectivity information for hierarchical clustering rather than reproduce the complete pairwise graph.

The evaluation in Chapter 5 examines these trade-offs experimentally through clustering-quality measurements, sampling and replication studies, graph-reconstruction statistics, execution-time measurements, and scalability experiments.

4.12 Chapter Summary

This chapter presented the design and implementation of the distributed FISHDBC prototype. The architecture consists of an Indexer, parallel Workers, and a Leader. The Indexer samples and partitions the dataset, routes points to worker partitions, and records raw distances evaluated during Meta-HNSW construction and routing.

Both the Indexer and Workers reduce their harvested distance graphs by retaining observed k -nearest-neighbor edges together with edges selected using a local mutual-reachability MST. These retained edges are communicated using their original raw distances rather than the locally computed MRD weights.

The Leader merges the resulting candidate edges, reconstructs approximate global neighborhoods and core distances, recomputes mutual-reachability weights using one global density model, and constructs the global MST used for hierarchical cluster extraction.

The implementation reuses the HNSW code from the original FISHDBC `flexible-clustering`

project and deploys the Indexer, Workers, and Leader as independent Python services communicating over HTTP in a Docker Swarm environment. The implementation also records diagnostics related to incomplete neighborhoods, graph connectivity, execution time, and resource usage.

Replication provides controlled overlap between worker partitions, while graph reduction limits the amount of harvested HNSW information transmitted to the Leader. The resulting method remains approximate and batch-oriented, and both Indexer routing and centralized Leader reconstruction represent potential scalability limitations.

The following chapter evaluates the distributed approach in terms of clustering quality, parameter sensitivity, graph reconstruction, execution time, and scalability.

Chapter 5

Evaluation

This chapter evaluates the proposed distributed FISHDBC approach in terms of clustering quality, computational performance, scalability, and the effect of the main parameters introduced by the distributed architecture. The evaluation is organized around the research questions defined in Chapter 1, with particular attention to how closely the distributed result reproduces centralized clustering and to the trade-offs introduced by sampling, replication, and global reconstruction.

Two datasets are used under different computational conditions. The first is a synthetic Gaussian-blobs dataset with a conventional Euclidean distance function, used to study clustering quality, sampling sensitivity, replication, scalability, and the runtime behavior of the distributed pipeline.

The second dataset is MusicBrainz, derived from the MusicBrainz open music metadata database, which contains information about songs, artists, albums, releases, and related records. It is used to evaluate the distributed approach with an application-specific distance function, providing a workload in which distance evaluation and neighborhood discovery can have a larger impact on the overall execution time.

For the synthetic experiments, clustering quality is assessed using the Adjusted Rand Index, Adjusted Mutual Information, the number of detected clusters, and noise points. For the MusicBrainz experiments, clustering outputs are reported only as descriptive diagnostics, while the evaluation focuses primarily on computational behavior.

The chapter first describes the experimental environment, datasets, compared methods, and evaluation metrics. It then presents the synthetic-data experiments, followed by the MusicBrainz experiments, and concludes with a discussion of the results and the limitations of the current prototype.

5.1 Evaluation Goals and Research Questions

The main goal of the evaluation is to determine whether the proposed distributed FISHDBC approach can preserve the clustering behavior of centralized density-based methods while distributing the computationally expensive neighborhood-discovery stage across multiple machines.

The experiments are organized around the three research questions introduced in Chapter 1.

- **RQ1: To what extent can the proposed distributed approach reproduce the clustering obtained by single-machine FISHDBC and HDBSCAN*?**

This question focuses on clustering quality. The distributed result is compared with centralized FISHDBC and HDBSCAN* using agreement measures, the number of detected clusters, and the amount of noise. On the synthetic dataset, the known generating labels also provide an additional reference for evaluating the recovered cluster structure.

- **RQ2: How do sampling and replication affect the clustering quality of the distributed approach?**

This question investigates the effect of two main parameters introduced by the distributed architecture. The synthetic Blob experiments vary the sampling rate and replication factor while keeping the remaining configuration fixed. The resulting changes in clustering quality, graph reconstruction, runtime, and communication are then measured.

- **RQ3: What computational trade-offs are introduced by distributing FISHDBC across multiple workers?**

This question considers execution time, scalability, communication, and resource usage. The evaluation measures the separate costs of the Indexer, parallel Worker phase, and Leader reconstruction, and compares the total runtime with centralized FISHDBC and HDBSCAN*. Scalability is examined as dataset size increases, with particular attention to identifying centralized bottlenecks in the current architecture.

The two datasets provide complementary evaluation settings. The synthetic Gaussian Blob dataset is used for controlled experiments with Euclidean distance and for studying the sensitivity of the distributed parameters. MusicBrainz is used to evaluate the system with an application-specific distance function, allowing the experiments to investigate how the distributed design behaves when distance computation becomes a more significant part of the clustering workload.

Together, these experiments evaluate not only whether the distributed method produces comparable clustering results, but also under which computational conditions the additional cost of distribution is justified.

5.2 Experimental Setup

The experiments were executed on a physical Docker Swarm cluster composed of one front node and ten worker nodes. The same deployment was used throughout the evaluation in order to keep the hardware and execution environment fixed while varying the dataset and algorithmic parameters.

5.2.1 Hardware and Deployment

The front node acts as the Docker Swarm manager and hosts both the Indexer and Leader services. It is equipped with an Intel Xeon E5520 processor running at 2.27 GHz, with four processing cores, and approximately 11.6 GiB of main memory. The ten worker nodes are equipped with Intel Core i5-9500 processors running at 3.00 GHz, with six cores per machine and approximately 15.4 GiB of memory each.

One Worker service is deployed on each physical worker node, giving a total of ten parallel Worker instances. The Indexer and Leader remain on the front node, while the centralized FISHDBC and HDBSCAN* baselines are executed on a single worker machine so that their measurements represent single-machine execution.

The ten worker machines collectively provide 60 CPU cores. However, the distributed execution consists of ten independent Worker services operating on separate partitions rather than a single shared-memory 60-core process.

Table 5.1: Hardware used for the experimental evaluation.

Role	Nodes	Processor	Cores/node	Memory/node
Front	1	Intel Xeon E5520, 2.27 GHz	4	11.6 GiB
Worker	10	Intel Core i5-9500, 3.00 GHz	6	15.4 GiB

The front node is less powerful than the worker machines and hosts two centralized components simultaneously. This hardware difference is relevant when interpreting the distributed runtime, particularly because the Indexer performs sampling, partitioning, and routing of the complete dataset before the Worker phase begins.

5.2.2 Reference Configuration

Unless an experiment explicitly varies one parameter, the distributed implementation uses the reference configuration summarized in Table 5.2. The values correspond to the configuration described in Chapter 4 and are kept fixed so that individual experiments can isolate the effect of the parameter under study.

Table 5.2: Reference configuration used in the evaluation.

Parameter	Meaning	Reference value
Sampling rate	Fraction used for Meta-HNSW	0.10
Worker partitions	Number of logical partitions	10
Replication factor	Partitions assigned per point	1
<i>MinPts</i>	Non-self neighbors for core distance	15
Minimum cluster size	HDBSCAN* extraction parameter	15
HNSW <i>ef</i>	Search effort	50
HNSW <i>M</i>	Graph connectivity parameter	5
KaHIP mode	Graph partitioning preset	FAST
KaHIP imbalance	Allowed imbalance	0.03
Sample seed	Landmark sampling seed	42
Partition seed	KaHIP partitioning seed	42

Self-neighbors are excluded by the distributed implementation and centralized FISHDBC. Therefore, *MinPts* = 15 refers to fifteen distinct non-self neighbors. The HDBSCAN* baseline is configured with `min.samples=16` so that its effective neighborhood size follows the same convention.

For parameter-sensitivity experiments, only the parameter under investigation is changed whenever possible. In particular, the sampling experiment varies the fraction used to construct the Meta-HNSW, while the replication experiment varies the number of nearby worker partitions to which each point is assigned.

The B4 scalability experiment uses a separate scale-oriented configuration. To keep the experiment within the memory limits of the front node, the dimensionality is fixed at 10 and the sampling rate at 0.001 for all B4 dataset sizes. These values differ from the B1–B3 reference configuration, but remain fixed within B4 so that dataset size is the only parameter varied in the scalability study.

5.2.3 Repetition and Measurement Procedure

Each experimental configuration is executed three times when repeated measurements are available. Reported execution times are expressed as the mean over the completed

repetitions together with the standard deviation. Fixed random seeds are used for dataset generation, landmark sampling, and graph partitioning where applicable, allowing the same input and partitioning conditions to be reproduced between repetitions.

For the distributed implementation, execution time is decomposed into the Indexer, Worker, and Leader stages. Since Workers execute in parallel, the Worker contribution to distributed wall-clock time is defined as the duration of the slowest Worker rather than the sum of all Worker execution times:

$$T_{\text{workers}} = \max_i T_{\text{worker},i}.$$

The distributed execution time is therefore evaluated from the sequential pipeline stages as

$$T_{\text{distributed}} = T_{\text{indexer}} + T_{\text{workers}} + T_{\text{leader}}.$$

In addition to total runtime, the experiments record clustering outputs and internal reconstruction statistics, including the number of clusters and noise points, ARI and AMI agreement, Worker-to-Leader edge counts, incomplete neighborhoods, and global MST connectivity. Peak resident memory is also recorded for the centralized baselines and for the individual distributed components. Where relevant, the Indexer runtime is further decomposed into Meta-HNSW construction, graph partitioning, and routing in order to identify which stage limits scalability as the dataset size increases.

5.3 Datasets

The evaluation uses two datasets with different characteristics. A synthetic Gaussian Blob dataset provides a controlled numerical benchmark with known cluster labels and an inexpensive Euclidean distance function. MusicBrainz is instead used to evaluate the system on real music metadata with an application-specific distance function. The two datasets therefore exercise the distributed architecture under substantially different neighborhood-search workloads.

5.3.1 Synthetic Blob Dataset

The first dataset is generated synthetically using Gaussian blobs. This choice provides direct control over the number of points, dimensionality, number of clusters, cluster dis-

person, and random seed. It also provides known generating labels that can be used as a reference when evaluating the clustering produced by the different methods.

The reference Blob configuration contains 10 clusters generated with a fixed random seed and a cluster standard deviation of 1. Euclidean distance is used throughout the synthetic experiments. The reference experiment uses 10,000 points represented in 1,000 dimensions, while larger datasets are used in the scalability experiments.

The synthetic dataset serves several purposes. First, its known labels allow the clustering quality of Distributed FISHDBC, centralized FISHDBC, and HDBSCAN* to be compared against the same reference structure. Second, its controlled generation makes it possible to vary individual distributed parameters such as sampling and replication while leaving the underlying data unchanged. Finally, because Euclidean distance is relatively inexpensive to compute, the dataset provides a useful setting for exposing the overhead introduced by indexing, routing, communication, and global reconstruction.

For the largest scalability experiments, the dimensionality and sampling rate may be reduced to keep execution within the memory available on the front node. These changes are reported explicitly for the corresponding experiments rather than being treated as part of the reference Blob configuration.

5.3.2 Music Dataset and Arbitrary Distance Function

The second evaluation dataset is derived from the MusicBrainz database [Met26], an open music metadata database maintained by the MetaBrainz Foundation. The experiments use the artist-name field and consider subsets of different sizes.

Artist names are normalized before clustering by applying Unicode NFKC normalization, removing leading and trailing whitespace, converting the text to lowercase, and discarding null or empty values.

Pairwise dissimilarity between records is computed using a Jaro–Winkler-based distance applied to the normalized artist-name strings. Jaro–Winkler is commonly expressed as a similarity score, where larger values indicate more similar strings. For the clustering experiments, this similarity is converted into a dissimilarity as

$$d_{\text{JW}}(x, y) = 1 - s_{\text{JW}}(x, y),$$

where $s_{\text{JW}}(x, y)$ denotes the Jaro–Winkler similarity between the two normalized artist-name strings.

This dataset is used to evaluate the distributed approach in a setting where dissimilarity is defined through an application-specific distance function rather than Euclidean distance

on synthetic vectors.

5.4 Compared Methods

The evaluation compares the proposed Distributed FISHDBC implementation with two centralized baselines: the original single-machine FISHDBC algorithm and HDBSCAN*. The three methods represent different trade-offs between approximation, flexibility, and execution model.

Distributed FISHDBC. This is the approach proposed in this thesis. The dataset is organized by the Indexer, divided across multiple Workers, and processed through local HNSW-based neighborhood discovery and graph reduction. The Leader then combines the retained raw-distance edges, reconstructs approximate global core distances, recomputes mutual-reachability weights, and constructs the global MST used for hierarchical cluster extraction. The method is approximate because only relationships discovered through HNSW processing and retained after graph reduction are available to the Leader.

Centralized FISHDBC. The original FISHDBC implementation is used as the main centralized baseline. It performs approximate neighborhood discovery using HNSW and constructs an approximate mutual-reachability MST on a single machine. This comparison is particularly important because Distributed FISHDBC is derived from the same underlying clustering approach, allowing the effects of distribution, sampling, replication, and global reconstruction to be evaluated separately from the approximation already present in FISHDBC.

HDBSCAN*. HDBSCAN* is used as a second centralized reference method. Unlike FISHDBC, it does not rely on the same HNSW-based approximation strategy. On the synthetic Blob dataset it provides a useful reference for comparing the recovered hierarchical density-based clustering against a standard implementation. For the MusicBrainz experiments, the comparison is performed using the same distance definition whenever the implementation permits a fair execution with the arbitrary-distance representation.

To make the comparison consistent, the same effective neighborhood and cluster parameters are used across the methods. Distributed FISHDBC and centralized FISHDBC use $MinPts = 15$ with self-neighbors excluded. The HDBSCAN* baseline is therefore configured with `min_samples=16`, because its parameter convention includes the point itself, while the minimum cluster size is set to 15 for all methods.

The centralized baselines are executed on a single worker machine, while the distributed method uses the complete multi-node deployment. Runtime comparisons are therefore interpreted together with clustering quality and resource usage, rather than as a comparison between identical hardware allocations.

5.5 Evaluation Metrics

The evaluation considers both clustering quality and computational behavior. Since the proposed approach is approximate and distributed, no single metric is sufficient to characterize its performance. The experiments therefore combine clustering-agreement measures with execution-time, graph-reconstruction, communication, and resource-usage statistics.

5.5.1 Clustering Quality

Clustering quality is evaluated using the Adjusted Rand Index (ARI) and Adjusted Mutual Information (AMI). Both measures compare two cluster assignments while correcting for agreement that may occur by chance. Values close to one indicate strong agreement between the compared clusterings.

For the synthetic Blob dataset, the known generating labels provide an additional reference. ARI and AMI are therefore computed between the output of each clustering method and the synthetic ground-truth labels. Pairwise agreement is also measured between Distributed FISHDBC, centralized FISHDBC, and HDBSCAN*.

For the MusicBrainz experiments, no external ground-truth clustering is assumed. The number of detected clusters and noise points is therefore reported descriptively, while the primary evaluation focuses on runtime, distance-function evaluations, memory usage, communication, and distributed stage behavior.

For the synthetic experiments, the number of detected clusters and the number or fraction of points classified as noise are reported alongside ARI and AMI. For MusicBrainz, these quantities are used only as descriptive diagnostics.

5.5.2 Execution Time and Scalability

Runtime is measured as wall-clock execution time. For Distributed FISHDBC, the main stages are measured separately: Indexer processing, parallel Worker processing, and Leader reconstruction.

Since Workers execute concurrently, Worker time is defined as the duration of the slowest Worker:

$$T_{\text{workers}} = \max_i T_{\text{worker},i}.$$

The total distributed execution time is evaluated as

$$T_{\text{distributed}} = T_{\text{indexer}} + T_{\text{workers}} + T_{\text{leader}}.$$

For repeated experiments, runtime is reported as the mean together with the standard deviation. Scalability is evaluated by increasing dataset size and observing how the total runtime and the contribution of the individual distributed stages change.

Where useful, the Indexer time is further decomposed into operations such as Meta-HNSW construction, graph partitioning, and routing. This makes it possible to identify centralized stages that may limit the scalability of the current architecture.

5.5.3 Graph Reconstruction

Because the Leader reconstructs the global clustering structure from reduced candidate graphs rather than from the complete pairwise graph, additional metrics are used to evaluate the quality of this reconstruction.

The evaluation records the number of candidate edges received from the Indexer and Workers, the number of duplicate edges removed during merging, and the size of the final candidate graph. It also reports the number and fraction of points whose reconstructed neighborhood contains fewer than the required *MinPts* candidates.

Global connectivity is evaluated by recording the number of connected components in the reconstructed candidate graph and by checking whether the final MST contains exactly $n - 1$ edges. This condition confirms that the candidate graph is connected and that a complete spanning MST can be constructed; it does not imply that the reconstructed candidate graph is identical to the complete mutual-reachability graph.

These measurements complement ARI and AMI by showing whether changes in clustering quality can be related to missing neighborhood or connectivity information.

5.5.4 Distance Evaluations

The number of distance-function evaluations is recorded where available. This metric is particularly important for the MusicBrainz experiments, where similarity is computed using an application-specific distance function whose cost can be higher than that of a simple Euclidean distance.

For Distributed FISHDBC, distance evaluations can be separated between the Indexer and the Workers. This makes it possible to determine how much distance computation is performed centrally and how much is distributed across the worker nodes.

When an exact pairwise distance matrix is required by a baseline, the number of possible pairwise evaluations is

$$\frac{n(n-1)}{2}.$$

Comparing the number of evaluated pairs with this quantity provides an indication of how effectively HNSW-based neighborhood discovery avoids exhaustive distance computation.

5.5.5 Communication and Resource Usage

The distributed experiments also record the amount of information exchanged between components. In particular, Worker-to-Leader communication is measured using the number of reduced candidate edges transmitted by the Workers. Where available, the corresponding serialized payload size is also reported.

Peak resident memory (RSS) is recorded using the maximum resident-set-size reported by each process. For Distributed FISHDBC, memory is reported separately for the Indexer, Leader, and Workers because the computation spans several processes and physical machines. Worker memory is summarized using both the maximum Worker peak RSS and the mean Worker peak RSS. The component-level peaks are not summed and interpreted as a single distributed peak-memory value because the individual maxima may occur at different times.

Communication and memory measurements are interpreted together with runtime and clustering quality. This allows the evaluation to distinguish improvements obtained through parallel computation from costs introduced by replication, graph reduction, communication, and centralized reconstruction.

5.6 Synthetic Blob Experiments

5.6.1 B1: Reference Quality, Runtime, and Memory

The first synthetic experiment establishes a reference configuration for the distributed implementation and compares its clustering result and execution time with centralized FISHDBC and HDBSCAN*. Unlike the following parameter studies, no distributed parameter is varied in B1. The purpose is to determine whether Distributed FISHDBC preserves the expected clustering structure under a fixed configuration and to provide a reference point for the subsequent sampling, replication, and scalability experiments.

The reference dataset contains 10,000 points in 1,000 dimensions generated from ten Gaussian clusters. The sampling rate is set to 0.10, the dataset is partitioned across ten Workers, and the replication factor is one. The remaining clustering and HNSW parameters are kept fixed across all three repetitions.

Distributed FISHDBC is compared with centralized FISHDBC and HDBSCAN*. The centralized methods are executed on one worker machine with six available CPU cores. HDBSCAN* is configured to use all six cores during its parallel core-distance computation. Distributed FISHDBC instead uses ten Worker machines together with the Indexer and Leader on the front node. The runtime comparison therefore represents the actual experimental deployment rather than an equal-hardware speedup comparison.

Clustering quality. Table 5.3 summarizes the clustering quality obtained over the three repetitions. Distributed FISHDBC consistently identifies the expected ten clusters and remains very close to the ground-truth labels. Its mean ARI is approximately 0.9993 and its mean AMI approximately 0.9992.

Table 5.3: Clustering quality in the B1 reference experiment. Values are mean $\pm$ standard deviation over three repetitions.

Method	ARI	AMI	Clusters	Noise points
Distributed FISHDBC	0.9993 ± 0.0010	0.9992 ± 0.0011	10	6.0 ± 8.7
FISHDBC	0.9900 ± 0.0000	0.9897 ± 0.0000	11	39.0 ± 0.0
HDBSCAN*	1.0000 ± 0.0000	1.0000 ± 0.0000	10	0.0 ± 0.0

The Distributed FISHDBC repetitions contain 16, 2, and 0 noise points, respectively. Although this produces noticeable variation in the noise count, the corresponding ARI and AMI values remain close to one, indicating that the overall cluster structure is preserved.

Centralized FISHDBC shows a slightly larger deviation from the reference. It returns eleven clusters and 39 noise points in each repetition, with an ARI of approximately 0.9900 and an

AMI of approximately 0.9897. HDBSCAN* reproduces the ground-truth partition exactly in all three runs.

The pairwise agreement between the three methods is shown in Table 5.4. Distributed FISHDBC agrees most closely with HDBSCAN*, while its agreement with centralized FISHDBC remains high.

Table 5.4: Pairwise clustering agreement in the B1 reference experiment. Values are mean $\pm$ standard deviation over three repetitions.

Comparison	ARI	AMI
Distributed FISHDBC vs. HDBSCAN*	0.9993 ± 0.0010	0.9992 ± 0.0011
Distributed FISHDBC vs. FISHDBC	0.9894 ± 0.0008	0.9889 ± 0.0011
FISHDBC vs. HDBSCAN*	0.9900 ± 0.0000	0.9897 ± 0.0000

Figure 5.1 provides a two-dimensional PCA projection of the reference dataset and the three clustering outputs. A single PCA transformation is fitted to the dataset and the same projected coordinates are used in every panel, allowing the cluster assignments to be compared on a common representation.

The visualization is therefore intended only to illustrate the recovered structure. Since the original dataset is 1,000-dimensional, the quantitative agreement measures in Tables 5.3 and 5.4 provide the primary evidence for clustering quality.

Runtime comparison. Figure 5.2 compares the mean wall-clock runtime of the three methods over the three repetitions.

The measured runtimes are summarized in Table 5.5.

Table 5.5: Wall-clock runtime in the B1 reference experiment.

Method	Runtime (s)
Distributed FISHDBC	43.24 ± 0.29
FISHDBC	51.33 ± 0.17
HDBSCAN*	62.44 ± 0.02

Under the evaluated reference configuration, Distributed FISHDBC records the lowest wall-clock runtime among the three implementations. This result should not be interpreted as an equal-resource speedup, since the distributed method uses ten Worker machines while the centralized baselines execute on one six-core machine. Its role in B1 is instead to establish the observed runtime behavior under the actual deployment used throughout the evaluation.

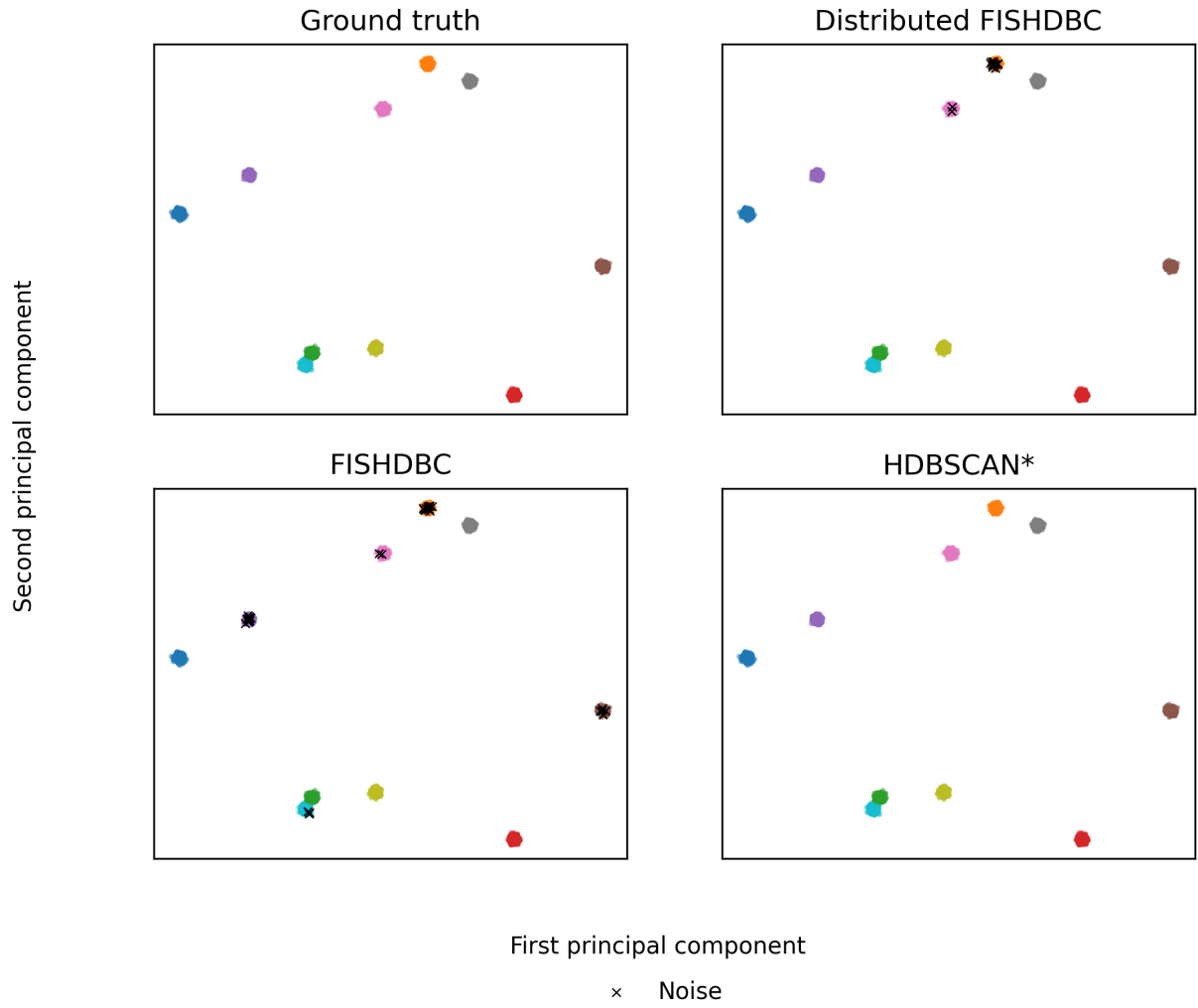


Figure 5.1: Two-dimensional PCA projection of the B1 reference dataset. The same projected coordinates are used for the ground truth and all three clustering methods. PCA is used only for visualization; clustering agreement is evaluated quantitatively using ARI and AMI.

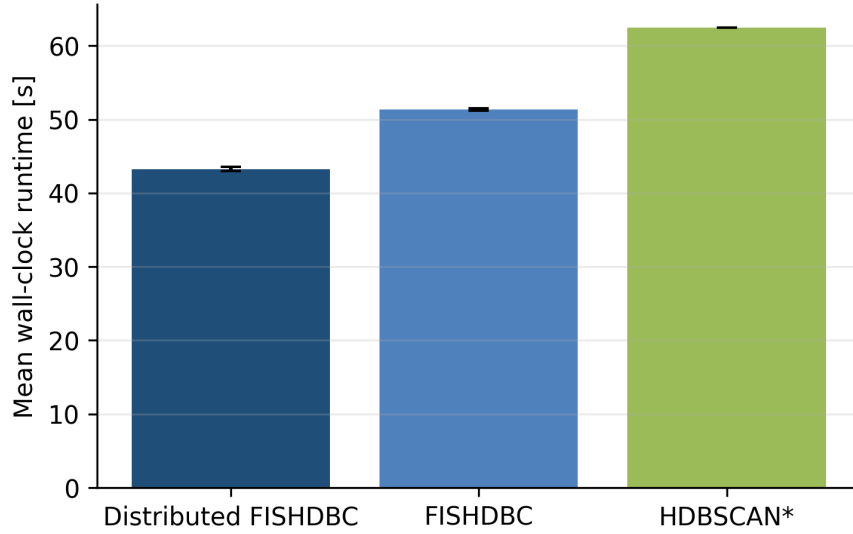


Figure 5.2: Mean wall-clock runtime in the B1 reference experiment. Error bars denote the standard deviation over three repetitions. Centralized FISHDBC and HDBSCAN* are executed on one six-core worker machine; HDBSCAN* is configured to use all six available CPU cores.

Distributed stage decomposition. The distributed execution can be decomposed into the Indexer, parallel Worker, and Leader phases. Worker wall time is defined as the maximum runtime among the ten Workers because they execute concurrently.

Table 5.6: Distributed stage runtime in the B1 reference experiment.

Stage	Runtime (s)
Indexer	31.53 ± 0.43
Worker wall	10.07 ± 0.35
Leader	1.64 ± 0.05

The Indexer is the largest contributor to the distributed runtime even at the reference scale. The three recorded stage times sum consistently to the authoritative Distributed FISHDBC wall-clock runtime in every repetition.

This stage decomposition provides an initial indication of the centralized cost within the distributed architecture. Its behavior at larger dataset sizes is examined in more detail in the B4 scalability experiment.

Peak memory usage. Peak resident memory provides a complementary view of the computational cost. For the distributed implementation, memory is reported separately for

the Indexer, Leader, and Workers rather than as a single aggregate value. The maximum Worker value characterizes the largest per-worker memory requirement, while the mean Worker value summarizes the typical memory footprint across the ten worker partitions.

Table 5.7: Peak resident memory in the B1 reference experiment. Values are mean $\pm$ standard deviation over three repetitions. Distributed component peaks are reported separately and are not summed into a single cluster-wide value.

Component / Method	Peak RSS (MiB)
Distributed Indexer	768.85 $\pm$ 5.01
Distributed Worker (maximum)	264.81 $\pm$ 29.19
Distributed Worker (mean)	257.44 $\pm$ 30.72
Distributed Leader	286.84 $\pm$ 2.15
Centralized FISHDBC	695.38 $\pm$ 0.30
HDBSCAN*	199.28 $\pm$ 0.01

At the reference scale, the Indexer has the largest individual memory footprint within Distributed FISHDBC, with a mean peak RSS of approximately 769 MiB. The maximum Worker peak is approximately 265 MiB, while the Leader reaches approximately 287 MiB. Centralized FISHDBC reaches approximately 695 MiB and HDBSCAN* approximately 199 MiB.

These values should be interpreted as per-process or per-component memory requirements. In particular, the distributed values do not represent total memory consumed by the complete Docker Swarm deployment. Their purpose is to show the maximum memory pressure placed on the individual services and machines participating in the distributed computation.

Graph reconstruction and communication. The global reconstruction succeeds in all three B1 repetitions. The Leader receives a connected candidate graph and constructs a complete MST containing

$$N - 1 = 9,999$$

edges in every run. No points are reported with fewer than the required MinPts reconstructed neighbors, and no infinite or fallback core distances are required.

The Workers transmit an average of approximately

$$130,492 \pm 130$$

reduced candidate edges to the Leader per run. These edges correspond to the union of the locally retained neighborhood and local MST information before Leader-side merging and deduplication; they do not represent the final global MST edge count.

Summary. The B1 experiment establishes that Distributed FISHDBC preserves the intended cluster structure under the reference configuration. Across the three repetitions, it consistently identifies the expected ten clusters and achieves ARI and AMI values close to one. Its clustering result is closer to HDBSCAN* than to centralized FISHDBC, while the reconstructed global graph remains connected and produces a complete MST in every run.

The reference experiment also provides the baseline computational behavior for the following parameter studies. Distributed FISHDBC records a mean wall-clock runtime of approximately 43 seconds, with the Indexer already representing the largest part of the distributed execution. The Indexer is also the largest individual distributed memory consumer in B1, reaching approximately 769 MiB peak RSS on average. The following experiments use this configuration as the reference point for studying the effect of sampling rate and replication.

5.6.2 B2: Sampling-Rate Sensitivity

The second synthetic experiment investigates the effect of the sampling rate used to construct the Meta-HNSW. The purpose is to determine whether increasing the amount of data represented in the global sample improves clustering quality and how this additional information affects the computational cost of the distributed pipeline.

The experiment uses the same 10,000-point, 1,000-dimensional Blob dataset and the same clustering configuration as the B1 reference experiment. The replication factor remains one and the dataset is partitioned across ten Workers. Only the sampling rate is changed.

The evaluated sampling rates are

$$0.01, \quad 0.025, \quad 0.05, \quad 0.10, \quad 0.20.$$

Each configuration is executed three times. The clustering result is evaluated against the known ground-truth labels using ARI and AMI, while computational cost is assessed using total Distributed FISHDBC wall-clock time and the internal Indexer timings.

Clustering quality. The clustering result remains very stable across the tested sampling rates. Table 5.8 summarizes the main quality and runtime measurements.

Table 5.8: Effect of the Meta-HNSW sampling rate on clustering quality and Distributed FISHDBC runtime. Values are mean $\pm$ standard deviation over three repetitions.

Sampling rate	ARI	AMI	Clusters	Noise	Runtime (s)
0.010	1.000 ± 0.000	1.000 ± 0.000	10.0 ± 0.0	3.0 ± 0.0	34.21 ± 0.48
0.025	1.000 ± 0.000	1.000 ± 0.000	10.0 ± 0.0	0.0 ± 0.0	35.71 ± 0.08
0.050	1.000 ± 0.000	1.000 ± 0.000	10.0 ± 0.0	1.3 ± 1.5	35.27 ± 0.94
0.100	1.000 ± 0.001	0.999 ± 0.001	10.0 ± 0.0	3.3 ± 4.9	46.08 ± 2.15
0.200	0.999 ± 0.000	0.999 ± 0.000	10.0 ± 0.0	8.7 ± 1.5	60.73 ± 0.94

ARI remains close to one for every evaluated sampling rate, and AMI follows the same behavior. Increasing the sample size therefore does not produce a consistent improvement in clustering agreement on this synthetic dataset. Even the smallest sampling rate, 0.01, reproduces the ground-truth structure almost exactly.

The number of noise points also remains very small relative to the 10,000-point dataset and does not follow a monotonic trend. The 0.025 configuration produces no noise in any of the three repetitions, whereas the largest sampling rate produces an average of 8.7 ± 1.5 noise points.

This behavior is consistent with the structure of the data. The ten Gaussian clusters are well separated, and a relatively small global sample is already sufficient to represent their large-scale organization. The result should not be interpreted as evidence that very small sampling rates are sufficient for arbitrary datasets, since more complex or overlapping structures may require a larger global sample.

Runtime effect. The computational effect of increasing the sampling rate is considerably more visible than its effect on clustering quality. Figure 5.3 shows the Distributed FISHDBC wall-clock runtime for the five configurations.

At sampling rates between 0.01 and 0.05, the mean runtime remains between approximately 34 and 36 seconds. Increasing the sampling rate to 0.10 raises the runtime to approximately 46 seconds, while a sampling rate of 0.20 requires approximately 61 seconds.

The increase is therefore concentrated at the higher sampling rates. In particular, doubling the sampling rate from 0.10 to 0.20 produces a substantial increase in execution time without a corresponding improvement in ARI or AMI.

Indexer cost. The internal Indexer measurements explain the runtime increase. A larger sampling rate increases the number of points included in the Meta-HNSW, which in turn affects both construction of the global index and subsequent routing of the full dataset.

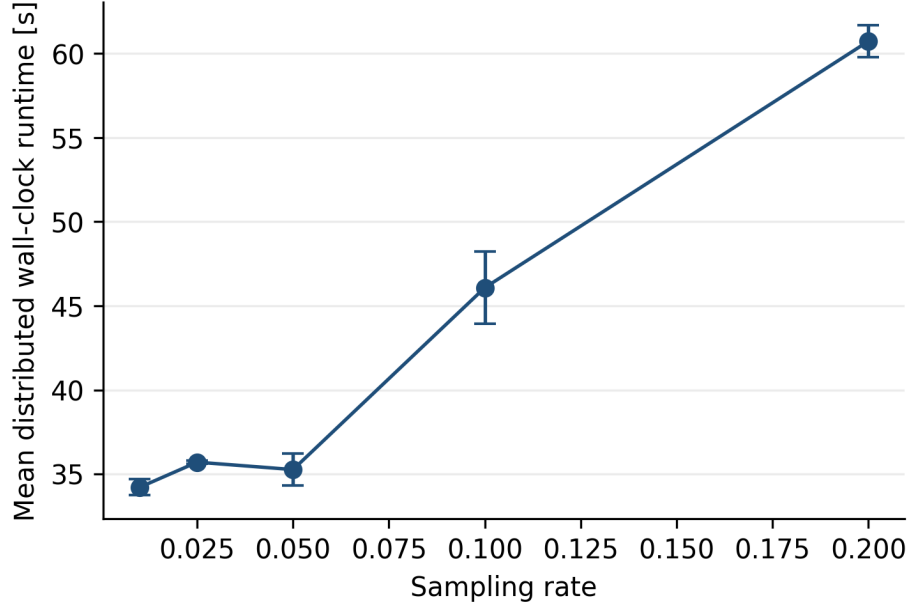


Figure 5.3: Mean Distributed FISHDBC wall-clock runtime for different Meta-HNSW sampling rates. Error bars represent the standard deviation over three repetitions.

Routing remains the largest Indexer component. Its mean runtime increases from approximately 17.3 seconds at a sampling rate of 0.01 to approximately 31.8 seconds at 0.20.

Meta-graph extraction also increases with the sample size, from approximately 4.6 seconds to approximately 10.5 seconds over the same range. Meta-HNSW construction shows an even stronger relative increase, from approximately 0.15 seconds at the smallest sampling rate to approximately 6.8 seconds at the largest one.

In contrast, graph partitioning contributes only a small fraction of the Indexer runtime throughout the experiment.

These measurements show that the additional cost of higher sampling is not limited to constructing a larger sample index. It also increases the amount of work required during global routing and graph processing.

Communication behavior. The Worker-to-Leader communication volume changes much less than the Indexer runtime. The reduced candidate-edge payload remains of the same order of magnitude across the tested sampling rates.

This indicates that the main cost of increasing the sampling rate occurs before Worker-side reconstruction, primarily within the Indexer. Sampling rate therefore has a stronger effect

on centralized preprocessing than on the amount of reduced graph information returned by the Workers.

Summary. The B2 experiment shows that the Meta-HNSW sampling rate introduces a clear trade-off between global sampling effort and computational cost. Across the tested range, clustering quality remains almost unchanged, with ARI and AMI values close to one even at the smallest sampling rates.

Higher sampling rates, however, increase the execution time of Distributed FISHDBC, particularly through additional Indexer work. Routing, Meta-HNSW construction, and meta-graph extraction all become more expensive as the sample grows.

For the evaluated synthetic Blob dataset, increasing the sampling rate beyond the smaller tested values therefore provides little measurable clustering benefit while increasing runtime. This result does not establish a universally optimal sampling rate; rather, it shows that the appropriate sampling level depends on the structure of the dataset and on the computational trade-off between global representation and Indexer cost.

5.6.3 B3: Replication and Boundary Coverage

The third synthetic experiment evaluates the effect of replicating points across multiple worker partitions. The purpose of replication is to increase the overlap between neighboring partitions and therefore provide additional opportunities for Workers to observe neighborhood relationships that would otherwise cross partition boundaries.

The experiment evaluates replication factors

$$R = 1, \quad 2, \quad 3, \quad 4.$$

The configuration with $R = 1$ corresponds to the B1 reference experiment. The analysis confirms that the B1 and B3 configurations use the same dataset and clustering parameters, with replication being the relevant parameter that changes across this experiment. The dataset therefore contains 10,000 points in 1,000 dimensions, the sampling rate is fixed at 0.10, and ten worker partitions are used.

Each replication configuration is executed three times. Clustering quality is evaluated using ARI, AMI, the number of detected clusters, and the number of noise points. The additional cost of replication is evaluated through wall-clock runtime and the number of reduced candidate edges transmitted from the Workers to the Leader.

Clustering quality. Clustering quality remains very high for every replication factor. All tested configurations identify the expected ten clusters, and ARI and AMI remain close to one. Increasing replication therefore does not produce a substantial change in the overall recovered cluster structure for this synthetic dataset.

The number of noise points shows small variations between configurations, as reported in Table 5.9. However, the behavior is not monotonic with replication. The $R = 1$ configuration produces an average of 6.0 noise points, while the corresponding means for $R = 2$, $R = 3$, and $R = 4$ are 7.0, 4.3, and 5.3 points, respectively.

Table 5.9: Effect of replication on clustering quality, runtime, and Worker-to-Leader communication. Values are mean $\pm$ standard deviation over three repetitions.

R	ARI	AMI	Noise	Runtime (s)	Worker $\rightarrow$ Leader edges
1	0.999 ± 0.001	0.999 ± 0.001	6.0 ± 8.7	43.24 ± 0.29	$130,492 \pm 130$
2	0.999 ± 0.001	0.999 ± 0.001	7.0 ± 6.6	45.46 ± 1.23	$148,912 \pm 581$
3	1.000 ± 0.000	0.999 ± 0.000	4.3 ± 3.2	46.83 ± 3.89	$152,142 \pm 8,244$
4	0.999 ± 0.001	0.999 ± 0.001	5.3 ± 7.6	46.33 ± 2.91	$150,711 \pm 6,770$

The differences in noise count are small relative to the 10,000-point dataset and show considerable variation between repetitions. In particular, increasing the replication factor does not consistently reduce the number of noise points. The experiment therefore does not provide evidence of a monotonic clustering-quality improvement as replication increases.

This result should be interpreted in the context of the synthetic dataset. The Gaussian clusters are already well separated, and the reference configuration achieves almost perfect agreement without additional replication. Consequently, there is relatively little boundary information for replication to recover.

Runtime cost. Replication increases the amount of local data processed by the Workers, because points may be assigned to more than one partition. The corresponding effect on wall-clock runtime is shown in Figure 5.4.

The reference configuration with $R = 1$ requires approximately 43.2 seconds. Increasing replication to two raises the mean runtime to approximately 45.5 seconds. The $R = 3$ and $R = 4$ configurations remain in the same range, with mean runtimes of approximately 46.8 and 46.3 seconds, respectively.

The additional runtime is therefore relatively limited for the evaluated configuration. Replication introduces extra Worker-side processing, but the increase from $R = 1$ to the higher replication factors remains only a few seconds in the 10,000-point experiment.

The runtime does not increase strictly with R . This is expected in a distributed execution where the end-to-end Worker phase depends on the slowest Worker rather than on the sum

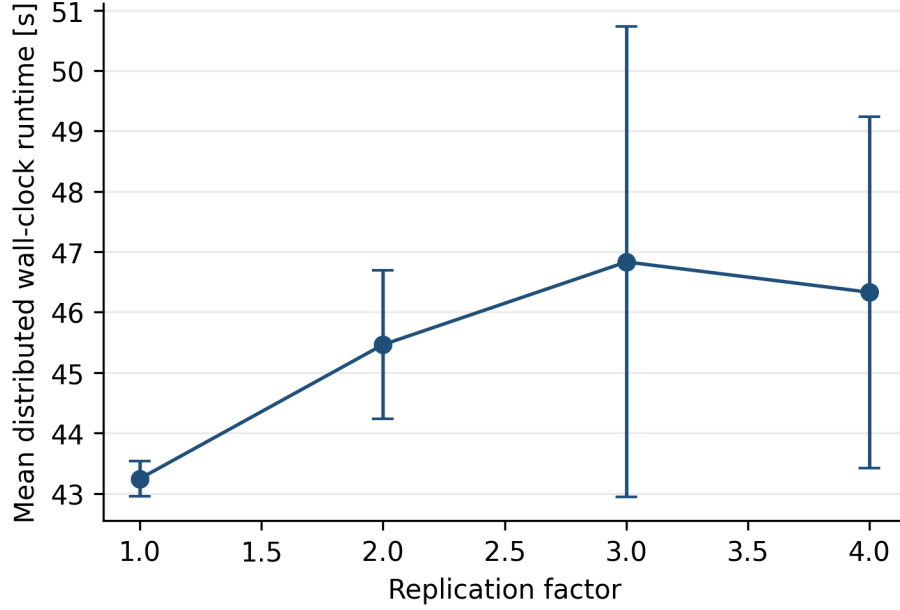


Figure 5.4: Mean Distributed FISHDBC wall-clock runtime for different replication factors. Error bars represent the standard deviation over three repetitions.

of all Worker computation. Small differences in partition sizes and execution conditions can therefore affect the measured wall-clock time.

Communication cost. The effect of replication is more visible in the amount of graph information transmitted from the Workers to the Leader. Figure 5.5 shows the mean number of reduced candidate edges sent per run.

With no additional replication, the Workers transmit approximately 130,000 candidate edges per run. This increases to approximately 149,000 edges at $R = 2$ and approximately 152,000 edges at $R = 3$. The $R = 4$ configuration remains at a similar level, with approximately 151,000 transmitted edges.

Replication therefore increases the amount of information available to the Leader, but the communication increase is not proportional to the replication factor. Although a point may be assigned to additional partitions, many of the resulting local observations overlap or are removed by the Worker-side graph reduction before transmission.

The transmitted values represent the local reduced edge unions and should not be interpreted as the number of edges in the final global MST.

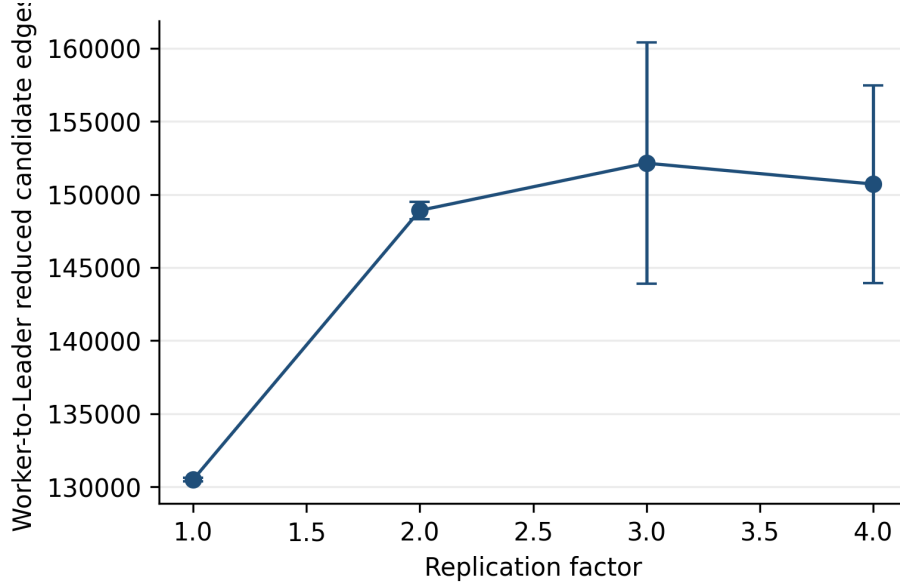


Figure 5.5: Worker-to-Leader communication for different replication factors. Values represent the reduced candidate-edge unions transmitted by the Workers before Leader-side merging and deduplication.

Boundary-coverage interpretation. The purpose of replication is to reduce the effect of partition boundaries by allowing neighboring points to coexist in more than one Worker partition. The experiment confirms that this mechanism increases the amount of candidate neighborhood information generated by the Workers. However, on the synthetic Blob dataset, this additional information does not translate into a clear improvement in ARI, AMI, or noise classification.

This does not imply that replication is unnecessary in general. Rather, the reference Blob dataset already provides a favorable partitioning problem: clusters are well separated and Distributed FISHDBC achieves near-perfect quality even at $R = 1$. A dataset containing more overlapping clusters or a larger number of neighborhoods crossing worker boundaries could exhibit a stronger dependence on replication.

Summary. The B3 experiment shows that increasing the replication factor adds overlap between worker partitions and increases the amount of candidate graph information returned to the Leader. This additional coverage introduces a moderate increase in runtime and a clearer increase in Worker-to-Leader communication.

At the same time, clustering quality remains nearly unchanged across $R = 1, \dots, 4$. All configurations recover the expected ten-cluster structure, and the small variations in noise

points do not follow a consistent trend as replication increases.

For the evaluated synthetic Blob dataset, replication therefore provides additional cross-partition neighborhood evidence, but this additional information does not produce a measurable improvement in clustering quality. The experiment instead makes the associated cost more visible: higher replication increases Worker-side processing and Worker-to-Leader communication while ARI, AMI, and the recovered cluster structure remain nearly unchanged.

5.6.4 B4: Scalability and Bottleneck Analysis

The final synthetic experiment evaluates the scalability of Distributed FISHDBC as the number of input points increases. The main objective is to determine whether distributing the neighborhood-discovery workload improves the runtime behavior of FISHDBC at larger dataset sizes and to identify the components that limit the scalability of the current implementation.

The experiment considers datasets containing

$$N = 50,000, \quad 100,000, \quad 150,000, \quad 200,000$$

points. Unlike B1–B3, B4 uses a scale-oriented configuration chosen to remain within the memory limits of the front node. All B4 runs use 10-dimensional synthetic Blob data, a sampling rate of 0.001, ten worker partitions, and a replication factor of one. These values are kept fixed for all four dataset sizes, so dataset size is the only parameter varied within the B4 experiment. With the fixed sampling rate, the Meta-HNSW sample grows from 50 landmarks at 50K points to 200 landmarks at 200K points.

Each configuration is executed three times. Reported runtimes are the mean wall-clock time together with the standard deviation over the three repetitions.

Distributed FISHDBC is compared with centralized FISHDBC and HDBSCAN*. The centralized baselines are executed on one worker machine with six available CPU cores. HDBSCAN* is configured to use all six cores during its parallel core-distance computation. Distributed FISHDBC instead uses ten Worker machines, while the Indexer and Leader execute on the front node. The comparison therefore represents the actual centralized and distributed execution environments used in the experiments rather than equal hardware allocations.

Runtime scalability. Figure 5.6 compares the wall-clock runtime of the three methods as the dataset size increases.

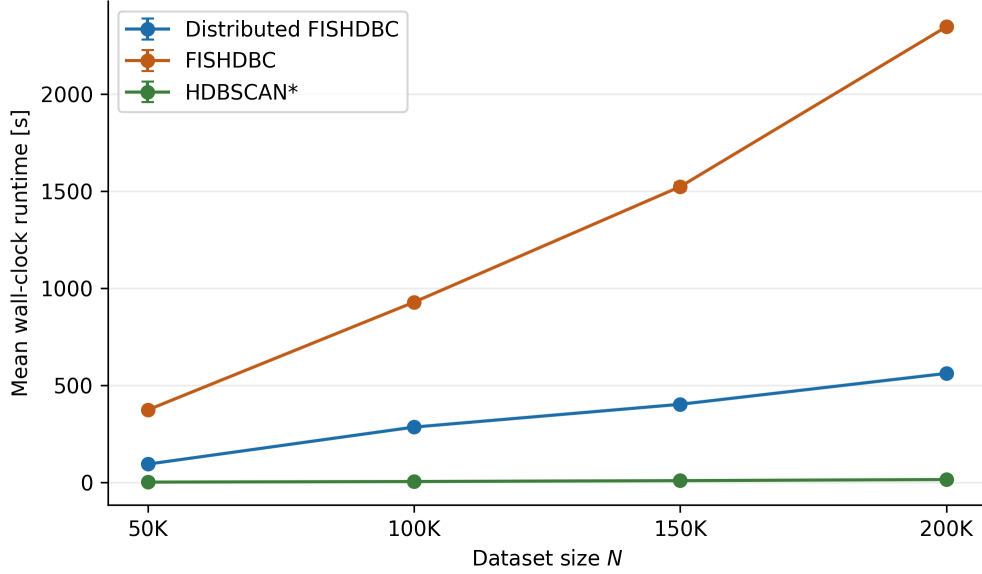


Figure 5.6: Mean wall-clock runtime as the dataset size increases. Centralized FISHDBC and HDBSCAN* are executed on one six-core worker machine, with HDBSCAN* configured to use all six available CPU cores. Error bars show the standard deviation over three repetitions.

Table 5.10: Runtime of the three clustering methods in the B4 scalability experiment. Values are mean $\pm$ standard deviation over three runs.

Dataset size	Distributed FISHDBC (s)	FISHDBC (s)	HDBSCAN* (s)
50K	95.20 $\pm$ 0.51	374.40 $\pm$ 2.91	2.65 $\pm$ 0.01
100K	285.83 $\pm$ 2.97	928.71 $\pm$ 4.27	5.74 $\pm$ 0.04
150K	403.13 $\pm$ 6.47	1523.05 $\pm$ 20.39	10.28 $\pm$ 0.02
200K	562.05 $\pm$ 1.76	2346.02 $\pm$ 14.49	15.90 $\pm$ 0.71

Distributed FISHDBC is consistently faster than centralized FISHDBC over the complete evaluated range. At 50K points, the distributed implementation requires approximately 95 seconds, compared with approximately 374 seconds for centralized FISHDBC. At 200K points, Distributed FISHDBC requires approximately 562 seconds, compared with approximately 2,346 seconds for centralized FISHDBC. More generally, the runtime gap between the two implementations remains substantial as the dataset size increases, indicating that the distributed implementation scales more effectively than the centralized FISHDBC baseline over the evaluated range.

HDBSCAN* exhibits a different runtime profile and remains substantially faster than both FISHDBC implementations. Its mean runtime increases from approximately 2.65 seconds at 50K to approximately 15.90 seconds at 200K. This result is specific to the tested workload: the synthetic data use a low-dimensional Euclidean representation, and the HDBSCAN* implementation can take advantage of optimized numerical operations and all six CPU cores of the baseline machine. The purpose of this experiment is therefore not to show that Distributed FISHDBC outperforms HDBSCAN* for inexpensive Euclidean distances, but to evaluate whether distribution improves the scalability of FISHDBC itself.

Distributed runtime decomposition. To identify where the distributed implementation spends its execution time, the distributed pipeline is decomposed into the Indexer, parallel Worker, and Leader phases. Since the Workers execute concurrently, Worker wall time is defined as the maximum runtime among the ten Workers rather than the sum of their individual runtimes.

The Indexer is the dominant stage at every evaluated dataset size. Its mean runtime increases from approximately 65.53 seconds at 50K to approximately 435.97 seconds at 200K. Over the same range, Worker wall time increases from approximately 20.83 seconds to 80.76 seconds, while Leader time increases from approximately 9.04 seconds to 45.71 seconds.

The relative contribution of the Indexer also grows with the dataset size. It accounts for approximately 68.7% of the distributed stage time at 50K, 69.4% at 100K, 73.2% at 150K, and 77.5% at 200K. At 200K, the Worker phase accounts for approximately 14.4%, while the Leader accounts for approximately 8.1%.

This behavior shows that the main scalability limitation is not the parallel Worker computation or the final Leader-side reconstruction. Instead, the centralized Indexer increasingly determines the total execution time as the dataset grows.

Indexer bottleneck. Figure 5.8 further decomposes the Indexer into its recorded internal operations.

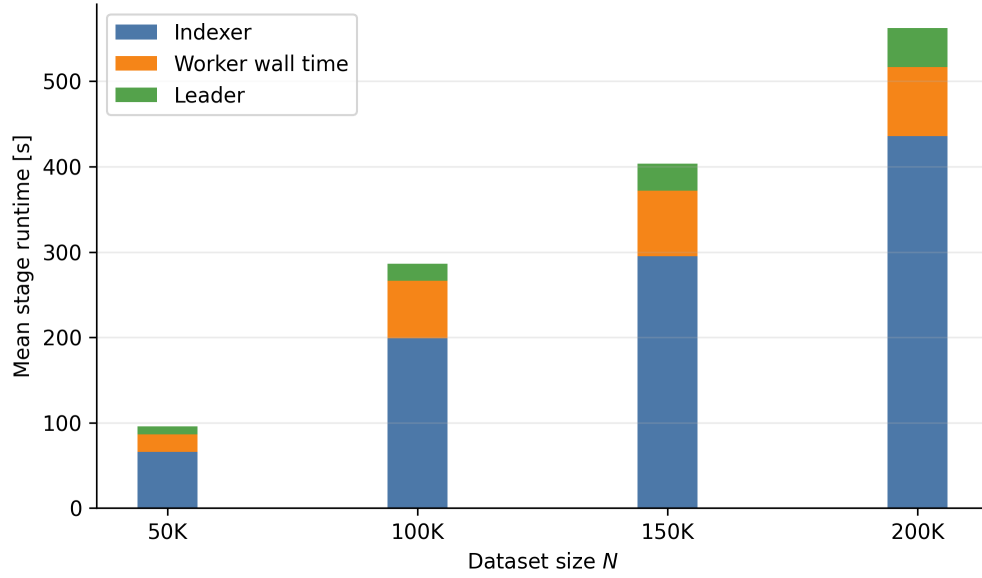


Figure 5.7: Runtime decomposition of Distributed FISHDBC into Indexer, parallel Worker, and Leader phases. Worker wall time is the maximum runtime among the ten parallel Workers.

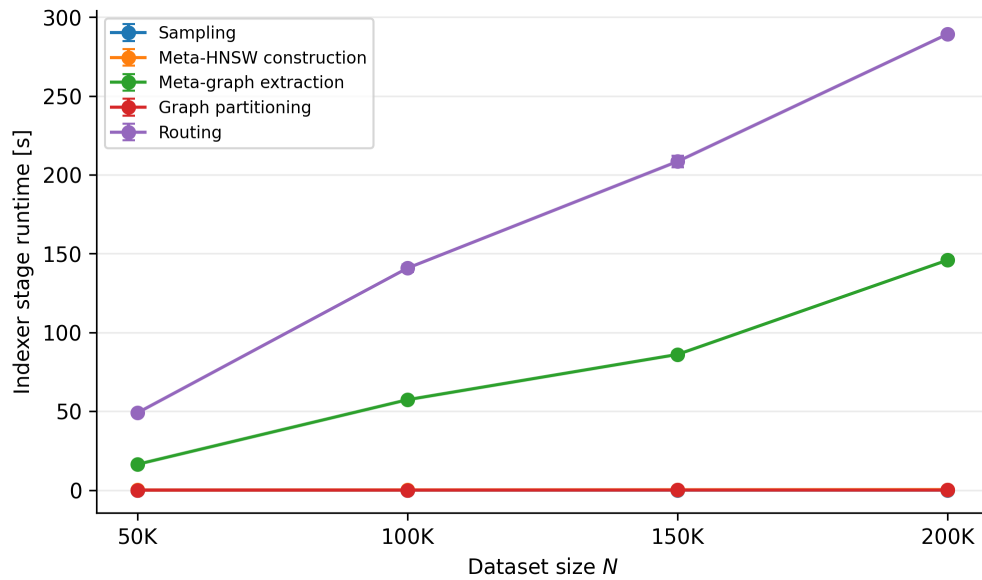


Figure 5.8: Runtime of the recorded Indexer stages as the dataset size increases. Routing is the dominant Indexer operation throughout the evaluated range.

Routing is the largest recorded Indexer component for all four dataset sizes. Its mean runtime increases from approximately 48.95 seconds at 50K to approximately 140.79 seconds at 100K, 208.47 seconds at 150K, and 289.31 seconds at 200K.

Routing accounts for approximately 74.7% of the Indexer runtime at 50K, 70.9% at 100K, 70.6% at 150K, and 66.4% at 200K. More importantly, routing alone represents approximately half of the complete distributed runtime: 51.4% at 50K, 49.3% at 100K, 51.7% at 150K, and 51.5% at 200K.

Meta-graph extraction is the main secondary Indexer cost. Its runtime increases from approximately 16.3 seconds at 50K to approximately 146 seconds at 200K, while sampling, Meta-HNSW construction, and graph partitioning remain small in comparison.

These measurements explain why the parallel Worker phase does not determine the overall scalability of the current system. Before points can be processed by the Workers, the complete dataset is routed centrally through the Meta-HNSW by the Indexer. Reducing or distributing this operation therefore represents the clearest direction for improving the scalability of the current prototype.

Communication growth. Another potential source of distributed overhead is the amount of information transferred from the Workers to the Leader. Figure 5.9 shows the size of the reduced candidate-edge payload transmitted after local processing.

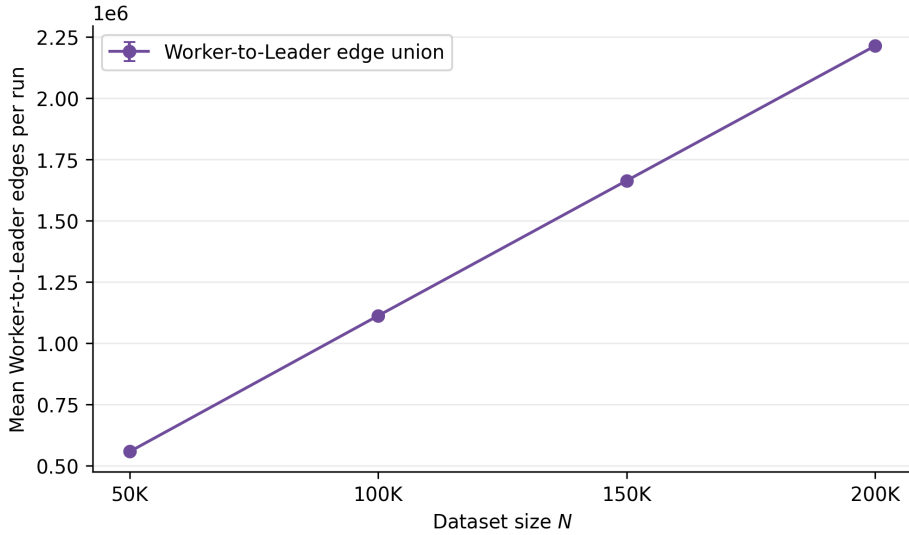


Figure 5.9: Worker-to-Leader communication as the dataset size increases. The values represent the summed reduced candidate-edge payload produced by the Workers before Leader-side deduplication and do not correspond to final MST edges.

The mean number of transmitted edges increases from approximately 557,448 at 50K points to 1.11 million at 100K, 1.66 million at 150K, and 2.21 million at 200K.

When normalized by the number of input points, the values remain nearly constant:

$$11.15, \quad 11.12, \quad 11.09, \quad 11.07$$

edges per point for 50K, 100K, 150K, and 200K respectively.

The measured communication volume therefore grows approximately in proportion to the dataset size over the evaluated range. This suggests that the local graph-reduction strategy prevents the Worker-to-Leader edge payload from growing disproportionately as the dataset becomes larger. This is an empirical observation for the tested configurations rather than a formal complexity bound.

Memory scalability. Peak resident memory is also examined as the dataset size increases. For the distributed implementation, the Indexer, Leader, and Worker processes are kept separate because they execute on different machines. Table 5.11 reports the Indexer peak RSS, the maximum Worker peak RSS observed in each repetition, the Leader peak RSS, and the corresponding centralized baseline measurements.

Table 5.11: Peak resident memory in the B4 scalability experiment. Values are mean $\pm$ standard deviation over three repetitions and are expressed in MiB. Distributed component peaks are reported separately and are not summed into a cluster-wide peak.

N	Indexer	Max Worker	Leader	FISHDBC	HDBSCAN*
50K	1458 ± 1	1573 ± 179	898 ± 1	3579 ± 2	157 ± 1
100K	4296 ± 13	4271 ± 10	1644 ± 0	7723 ± 0	198 ± 0
150K	6495 ± 63	4796 ± 2	2409 ± 17	12311 ± 0	244 ± 1
200K	8213 ± 0	4981 ± 9	3117 ± 18	12396 ± 76	288 ± 0

The Indexer memory requirement grows substantially with dataset size, from approximately 1.42 GiB at 50K points to approximately 8.02 GiB at 200K points. The maximum Worker peak also increases, from approximately 1.54 GiB to 4.86 GiB, while Leader peak RSS grows from approximately 0.88 GiB to 3.04 GiB.

Centralized FISHDBC exhibits a larger single-process memory requirement at the larger dataset sizes, reaching approximately 12.11 GiB at 200K points. By comparison, the largest individual distributed component at 200K is the Indexer at approximately 8.02 GiB. This does not imply that the distributed execution consumes less total memory across the cluster, since the memory used by different services is spread across several machines and their individual peaks may occur at different times. The measurements instead show that

the distributed design reduces the maximum memory demand placed on any single process relative to centralized FISHDBC at the larger evaluated sizes.

HDBSCAN* maintains a much smaller memory footprint on this low-dimensional Euclidean workload, increasing from approximately 157 MiB at 50K to approximately 288 MiB at 200K.

Clustering quality. The runtime behavior observed in B4 is accompanied by stable clustering quality across the evaluated dataset sizes. Distributed FISHDBC and HDBSCAN* recover the ten generating clusters exactly for all four sizes, with ARI and AMI equal to 1.0 and no points classified as noise.

Centralized FISHDBC remains close to the ground truth, although it introduces small deviations in cluster count and noise classification. Its ARI ranges from approximately 0.9949 to 0.9997 across the evaluated sizes.

Table 5.12: Clustering quality in the B4 scalability experiment. Values are means over three repetitions.

N	Distributed FISHDBC			FISHDBC			HDBSCAN*		
	ARI	Clusters	Noise	ARI	Clusters	Noise	ARI	Clusters	Noise
50K	1.0000	10	0	0.9949	12	40	1.0000	10	0
100K	1.0000	10	0	0.9980	13	22	1.0000	10	0
150K	1.0000	10	0	0.9997	10	41	1.0000	10	0
200K	1.0000	10	0	0.9992	12	53	1.0000	10	0

The deviations produced by centralized FISHDBC remain small relative to the dataset size. Distributed FISHDBC, however, maintains the generating ten-cluster structure exactly across the complete B4 range. The scalability trend observed in the distributed runtime is therefore not accompanied by a loss of clustering agreement on this controlled synthetic workload.

Summary. The B4 experiment shows that Distributed FISHDBC scales more effectively than the centralized FISHDBC implementation over the evaluated range. As the dataset size increases from 50K to 200K points, the distributed implementation maintains a substantially lower wall-clock runtime than the centralized FISHDBC baseline while preserving the generating cluster structure on the synthetic dataset.

The internal measurements also identify the main limitation of the current distributed architecture. The Indexer accounts for an increasing fraction of the total execution time as the dataset grows. Within the Indexer, centralized routing is the dominant operation

and represents approximately half of the complete distributed runtime. The memory measurements show the same general pressure on the Indexer: at 200K points it is the largest individual distributed memory consumer, although its peak RSS remains below that of centralized FISHDBC. In contrast, the parallel Worker phase, Leader-side reconstruction, and Worker-to-Leader communication remain comparatively controlled.

HDBSCAN* remains substantially faster for this low-dimensional Euclidean workload. Therefore, the main conclusion of B4 is not that Distributed FISHDBC universally outperforms other clustering methods, but that distributing FISHDBC improves its scalability relative to its centralized implementation under the evaluated configuration. The following Music experiments examine how this computational balance changes when the distance function is application-specific and more expensive to evaluate.

5.7 MusicBrainz Experiments

The synthetic Blob experiments evaluate the proposed method under controlled conditions using an inexpensive Euclidean distance function. The MusicBrainz experiments instead examine the computational behavior of the distributed architecture when similarity is defined through an application-specific string distance.

The experiments use the `artist` field of the MusicBrainz dataset. Each record is therefore represented by an artist-name string, and pairwise dissimilarity is computed using Jaro–Winkler distance. The objective is not to derive a semantic clustering of musical content, since no external ground-truth partition of the artist names is assumed. Instead, the dataset is used to evaluate the cost of neighborhood discovery when each distance evaluation requires string comparison rather than a low-dimensional Euclidean operation.

The MusicBrainz experiments therefore focus primarily on execution time, distance-function evaluations, memory usage, distributed stage behavior, graph reconstruction, and communication. The number of detected clusters and noise points is reported only as a descriptive property of the resulting clustering.

5.7.1 M1: Reference Computational Behavior

The first MusicBrainz experiment establishes a reference execution for the application-specific Jaro–Winkler distance. The experiment uses 10,000 MusicBrainz records and evaluates Distributed FISHDBC using the same general configuration adopted in the synthetic reference experiments.

The input field is the artist name and the distance function is Jaro–Winkler. The Meta-

HNSW sampling rate is set to 0.10, corresponding to 1,000 sampled landmarks. The sampled graph is divided into ten worker partitions and the replication factor is one. The clustering configuration uses $MinPts = 15$, a minimum cluster size of 15, HNSW $M = 5$, and $ef = 50$. The sampling and partitioning seeds are fixed to 42.

Overall runtime. Table 5.13 summarizes the runtime of Distributed FISHDBC over the three repetitions.

Table 5.13: Runtime of Distributed FISHDBC in the M1 MusicBrainz experiment. Values are mean $\pm$ standard deviation over three repetitions.

Stage	Runtime (s)
Indexer	58.72 ± 0.82
Worker wall	2.61 ± 0.16
Leader	1.82 ± 0.06
Total distributed runtime	63.15 ± 0.75

The Indexer dominates the execution time, requiring approximately 58.7 seconds of the 63.2-second distributed runtime. This corresponds to approximately 93% of the measured distributed execution time. In contrast, the parallel Worker phase requires approximately 2.6 seconds and the Leader approximately 1.8 seconds.

This behavior is more strongly centralized than in the synthetic B1 experiment. Although local HNSW construction and neighborhood processing are distributed across ten Workers, every input record must first be searched against the Meta-HNSW by the Indexer before it can be assigned to a Worker.

Indexer runtime decomposition. The internal Indexer timings are shown in Table 5.14.

Table 5.14: Indexer runtime decomposition in the M1 MusicBrainz experiment. Values are mean $\pm$ standard deviation over three repetitions.

Operation	Runtime (s)
Meta-HNSW construction	3.70 ± 0.10
Meta-graph extraction	22.25 ± 0.25
Graph partitioning	0.13 ± 0.01
Routing	32.63 ± 0.55

Routing is the largest recorded Indexer operation, requiring approximately 32.6 seconds on average. Meta-graph extraction is the second-largest component at approximately 22.3

seconds, while Meta-HNSW construction requires about 3.7 seconds. Graph partitioning contributes only a small fraction of the total Indexer runtime.

The results therefore confirm that the main centralized cost remains the global organization of the dataset rather than the Worker-side local computation.

Distance-function evaluations. The main purpose of the MusicBrainz experiment is to examine a workload in which distance evaluation is more computationally significant. The number of Jaro–Winkler distance evaluations performed by the Indexer and Workers is summarized in Table 5.15.

Table 5.15: Jaro–Winkler distance evaluations in the M1 MusicBrainz experiment. Indexer and Worker values are mean $\pm$ standard deviation over three repetitions; the distributed total is the sum of the mean component counts.

Component	Distance evaluations
Indexer	3,272,246 $\pm$ 10,081
Workers, aggregate	2,734,371 $\pm$ 15,626
Distributed total	6,006,617

For a dataset containing 10,000 points, the number of unique unordered pairwise comparisons is

$$\frac{N(N-1)}{2} = \frac{10,000 \times 9,999}{2} = 49,995,000.$$

Distributed FISHDBC therefore evaluates approximately 6.0 million Jaro–Winkler distances, corresponding to about 12% of the complete pairwise comparison count.

This does not imply that the distributed implementation performs fewer distance evaluations than centralized FISHDBC in every configuration, since the distributed architecture introduces additional work for Meta-HNSW construction, global routing, and independent Worker-local HNSWs. The measurement instead shows that HNSW-based neighborhood discovery avoids the need for exhaustive pairwise comparison even when using the application-specific string distance.

The measured time spent directly evaluating the distance function is approximately 18.92 ± 0.37 seconds in the Indexer. Across all Workers, the aggregate distance-evaluation time is approximately 4.66 ± 0.05 seconds. For the Indexer, Jaro–Winkler evaluation alone accounts for approximately one third of its algorithm execution time, indicating that distance computation represents a substantial component of this workload.

Memory usage. Peak resident memory is reported separately for the distributed components in Table 5.16. As in the synthetic experiments, the component peaks are not summed into a single distributed-memory value because their maxima may occur at different times and on different machines.

Table 5.16: Peak resident memory in the M1 MusicBrainz experiment. Values are mean $\pm$ standard deviation over three repetitions.

Component	Peak RSS (MiB)
Indexer	1839.51 ± 6.52
Worker, maximum	313.10 ± 52.89
Worker, mean	242.91 ± 43.66
Leader	272.26 ± 2.04

The Indexer is also the largest individual memory consumer, reaching approximately 1.80 GiB peak RSS. The largest Worker requires approximately 313 MiB on average, while the Leader remains below 300 MiB.

The result is consistent with the architectural role of the Indexer, which maintains the complete input dataset, the sampled Meta-HNSW, the routing information, and a large cache of harvested distances.

Partition balance and Worker processing. The ten Worker partitions contain an average of 1,000 records by construction, although the individual partition sizes differ because assignment follows the similarity structure of the sampled Meta-HNSW rather than an equal-size random split.

Across the three repetitions, the average minimum Worker partition size is approximately 716 points, while the average maximum size is approximately 1,215 points. This variation introduces some imbalance between Workers, but the parallel Worker phase remains small relative to the Indexer runtime.

The Worker wall-clock contribution, defined as the runtime of the slowest Worker in each repetition, remains approximately 2.61 ± 0.16 seconds.

Graph reduction and communication. The Indexer observes several million raw distance relationships but reduces this information substantially before communication with the Leader. On average, the Indexer transmits approximately

$$137,821 \pm 53$$

reduced candidate edges.

The ten Workers together transmit approximately

$$108,941 \pm 351$$

reduced candidate edges per repetition.

After Leader-side merging and deduplication, the reconstructed global candidate graph contains approximately

$$226,128 \pm 473$$

candidate edges.

The reduction therefore limits the amount of neighborhood information that must be transferred and processed globally despite the substantially larger number of raw Jaro–Winkler distance observations produced during HNSW processing.

Global reconstruction. Global reconstruction succeeds in all three repetitions. The candidate graph contains one connected component in every run, and the Leader constructs a spanning MST containing

$$N - 1 = 9,999$$

edges.

No point has fewer than the required 15 reconstructed candidate neighbors, and no point is reconstructed without candidate neighbors. The incomplete neighborhood fraction is therefore zero in all three repetitions.

These results show that the candidate information retained by the Indexer and Workers is sufficient to maintain global connectivity and reconstruct the required neighborhoods for this 10,000-record workload. The resulting MST remains the MST of the approximate reconstructed candidate graph and should not be interpreted as the exact MST of the complete pairwise graph.

Clustering output. Although clustering quality is not the primary objective of the MusicBrainz experiment, the resulting cluster and noise counts are recorded as descriptive diagnostics.

Distributed FISHDBC produces an average of

$$22.7 \pm 2.1$$

clusters across the three repetitions. The number of points classified as noise is

$$8858.7 \pm 53.8,$$

corresponding to approximately 88.6% of the dataset.

The large noise fraction should not be interpreted as evidence of poor clustering quality. In this experiment, clustering is based only on Jaro–Winkler similarity between artist-name strings, and no external ground-truth partition is assumed. The clustering statistics are therefore used only to detect abnormal or degenerate executions rather than to evaluate semantic clustering accuracy.

Summary. The M1 experiment establishes the reference computational behavior of Distributed FISHDBC on 10,000 MusicBrainz artist-name records using Jaro–Winkler distance.

The distributed execution completes in approximately 63.15 ± 0.75 seconds, with the Indexer accounting for the dominant share of the runtime. Routing and Meta-graph extraction are the main Indexer costs, while the parallel Worker phase and Leader reconstruction remain comparatively small.

Approximately 6.0 million Jaro–Winkler distance evaluations are performed, corresponding to about 12% of the approximately 50 million unique pairwise comparisons that would exist for a complete 10,000-point distance graph. Distance evaluation itself represents a substantial fraction of the Indexer runtime, confirming that the MusicBrainz workload places greater emphasis on the flexible distance-function capability of FISHDBC.

The reconstructed candidate graph is connected in every repetition, all points contain at least the required number of candidate neighbors, and a 9,999-edge spanning MST is obtained in every run. M1 therefore provides a stable reference configuration for the following MusicBrainz scalability experiments.

5.7.2 M2: Scalability with Jaro–Winkler Distance

The second MusicBrainz experiment evaluates how the three clustering approaches behave as the number of artist-name records increases. The purpose is primarily computational: the experiment studies whether the use of HNSW-based neighborhood discovery remains beneficial as the number of possible pairwise Jaro–Winkler comparisons grows.

Six nested dataset sizes are considered: 2,000, 5,000, 10,000, 15,000, 20,000, and 25,000 records. For each size, Distributed FISHDBC is executed three times, while the same dataset is also processed by centralized FISHDBC and HDBSCAN*. The distributed configuration uses ten Worker partitions, replication factor $R = 1$, $MinPts = 15$, minimum cluster size 15, HNSW $M = 5$, and $ef = 50$. The Meta-HNSW sampling rate is fixed at 0.10, so the number of sampled landmarks grows proportionally with the dataset size.

As in M1, the experiment is not intended to evaluate semantic music clustering. The input consists only of normalized artist-name strings and Jaro–Winkler distance is used as an application-specific distance function. The primary measurements are therefore execution time, distance-function evaluations, memory usage, distributed stage behavior, and graph reconstruction. Cluster and noise counts are reported only as descriptive diagnostics.

Runtime scalability. Figure 5.10 compares the complete execution time of Distributed FISHDBC, centralized FISHDBC, and HDBSCAN* as the dataset size increases.

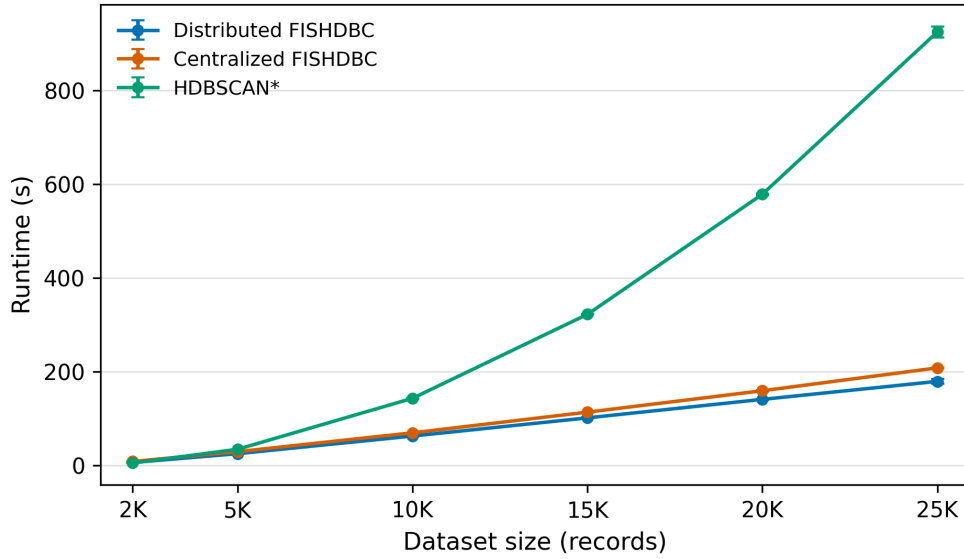


Figure 5.10: Execution-time scalability on the MusicBrainz artist-name datasets using Jaro–Winkler distance. Values for Distributed FISHDBC are mean $\pm$ standard deviation over three repetitions.

At 2,000 records, HDBSCAN* is the fastest method, completing in approximately 5.50 seconds, compared with 6.61 seconds for Distributed FISHDBC. At this small size, the complete pairwise distance matrix is still inexpensive enough that the additional organization required by the distributed architecture does not provide a runtime advantage.

Table 5.17: Execution time in the M2 MusicBrainz scalability experiment. Values are mean $\pm$ standard deviation in seconds.

Records	Distributed FISHDBC	Centralized FISHDBC	HDBSCAN*
2,000	6.61 ± 0.15	8.84 ± 0.05	5.50 ± 0.03
5,000	25.46 ± 0.27	29.71 ± 0.24	34.88 ± 0.09
10,000	63.09 ± 0.66	70.00 ± 0.05	143.65 ± 1.16
15,000	101.87 ± 0.97	114.04 ± 0.51	322.81 ± 0.77
20,000	141.32 ± 0.73	159.73 ± 1.49	578.93 ± 1.86
25,000	179.71 ± 5.47	208.52 ± 0.86	925.00 ± 12.07

The behavior changes as the dataset grows. From 5,000 records onward, Distributed FISHDBC is faster than both centralized baselines. At 25,000 records, Distributed FISHDBC completes in approximately 179.71 seconds, compared with 208.52 seconds for centralized FISHDBC and 925.00 seconds for HDBSCAN*. At this size, the distributed implementation is approximately $1.16\times$ faster than centralized FISHDBC and $5.15\times$ faster than HDBSCAN*.

The increasing difference with HDBSCAN* is particularly relevant for this workload because its precomputed Jaro–Winkler representation requires evaluation of the complete pairwise distance matrix. The HNSW-based methods instead examine only a subset of the possible pairwise relationships.

Distributed stage behavior. Figure 5.11 separates the distributed execution into the Indexer, parallel Worker phase, and Leader reconstruction.

Table 5.18: Distributed-stage runtime in the M2 experiment. Values are mean $\pm$ standard deviation in seconds.

Records	Indexer	Worker wall	Leader
2,000	5.99 ± 0.11	0.30 ± 0.03	0.32 ± 0.04
5,000	23.48 ± 0.28	1.18 ± 0.02	0.80 ± 0.00
10,000	58.57 ± 0.63	2.70 ± 0.19	1.82 ± 0.06
15,000	94.45 ± 1.15	4.58 ± 0.34	2.84 ± 0.15
20,000	131.27 ± 0.28	6.18 ± 0.51	3.87 ± 0.10
25,000	167.18 ± 6.03	7.65 ± 0.73	4.88 ± 0.10

The Indexer dominates the distributed execution at every evaluated size, accounting for approximately 91–93% of the complete runtime. At 25,000 records, for example, the Indexer requires approximately 167.18 seconds of the 179.71-second distributed execution, whereas the Worker phase requires only 7.65 seconds and the Leader approximately 4.88 seconds.

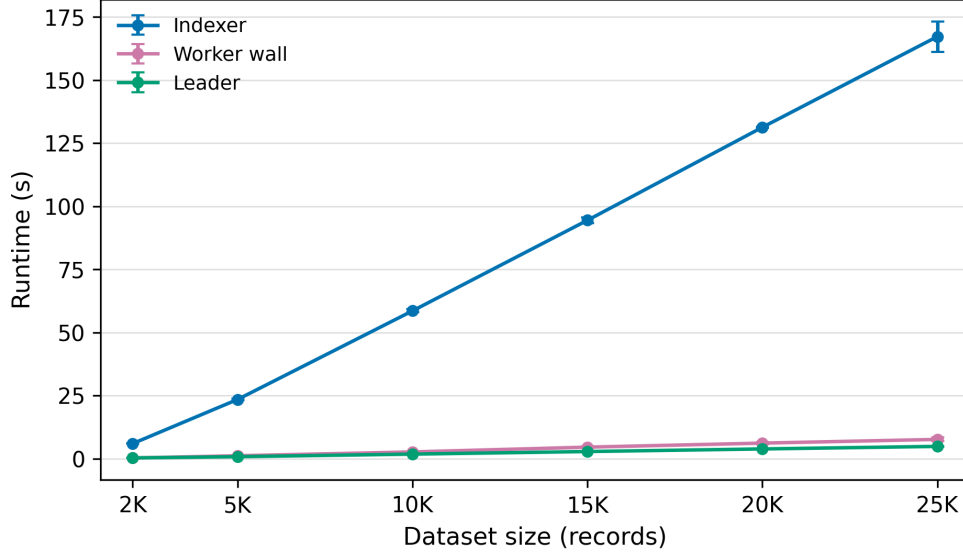


Figure 5.11: Runtime of the Indexer, parallel Worker phase, and Leader as the MusicBrainz dataset size increases. The Worker value corresponds to the wall-clock duration of the parallel Worker phase.

This result extends the behavior already observed in M1. Increasing the input size does not shift the main cost toward the parallel Worker computation. Instead, the centralized organization of the dataset remains the dominant part of the execution. The current scalability limitation is therefore mainly the Indexer, which must maintain the sampled global structure and route the complete dataset before Worker-side processing can be completed.

Distance-function evaluations. The difference between the methods becomes clearer when considering the number of Jaro–Winkler distance evaluations. For a dataset of N records, the number of unique pairs in a complete pairwise distance matrix is

$$P(N) = \frac{N(N-1)}{2}. \quad (5.1)$$

Figure 5.12 compares this value with the number of distance evaluations performed by Distributed FISHDBC and centralized FISHDBC.

At 10,000 records, the complete pairwise graph contains approximately 50 million unique pairs. Distributed FISHDBC evaluates approximately 6.00 million distances, while centralized FISHDBC evaluates approximately 4.01 million. At 25,000 records, the complete

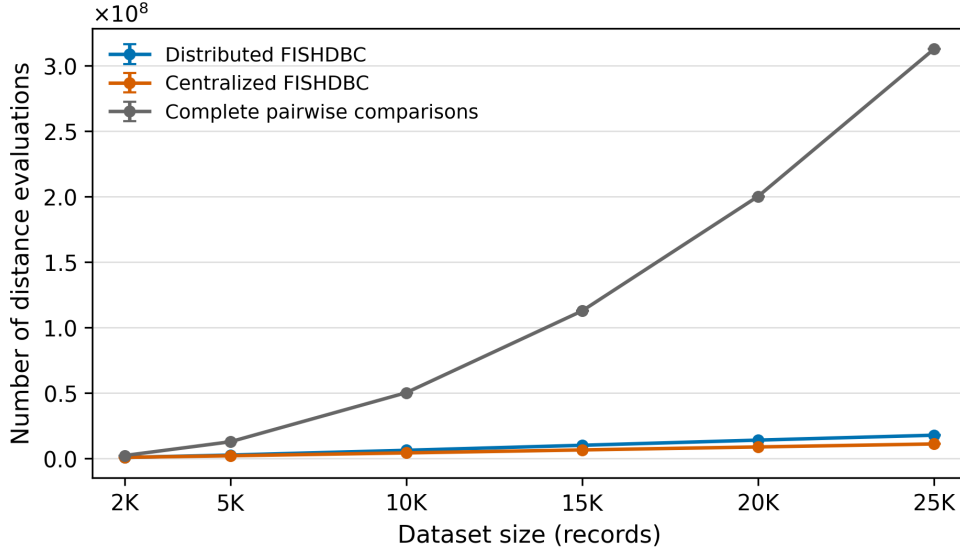


Figure 5.12: Number of Jaro–Winkler distance evaluations as the MusicBrainz dataset size increases. The complete-pairwise curve represents $N(N - 1)/2$ unique pairs.

pairwise matrix contains approximately 312.49 million unique pairs, compared with approximately 17.54 million evaluations for Distributed FISHDBC and 10.85 million for centralized FISHDBC.

Distributed FISHDBC therefore performs more distance evaluations than centralized FISHDBC. This additional work is introduced primarily by the Meta-HNSW and by routing each point through the global sampled structure before local Worker processing. Nevertheless, both HNSW-based approaches evaluate only a small fraction of the complete pairwise distance space.

Figure 5.13 shows this fraction directly.

For Distributed FISHDBC, the evaluated fraction decreases from approximately 26.76% at 2,000 records to 5.61% at 25,000 records. For centralized FISHDBC, it decreases from approximately 30.94% to 3.47% over the same range.

The absolute number of distance evaluations therefore increases with dataset size, but substantially more slowly than the number of possible pairs. This explains an important part of the increasing runtime difference between the HNSW-based approaches and the exhaustive pairwise HDBSCAN* baseline.

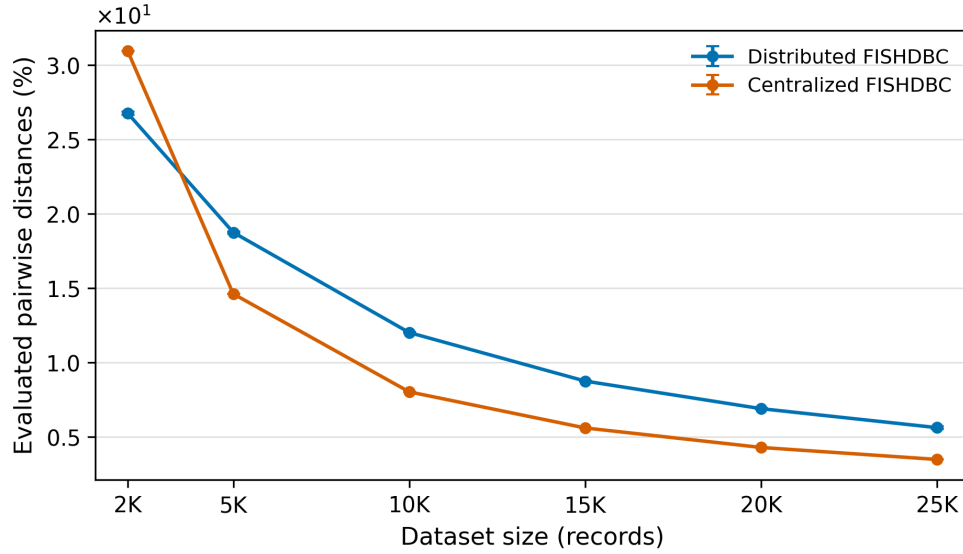


Figure 5.13: Fraction of all possible unique pairwise Jaro–Winkler distances evaluated by Distributed FISHDBC and centralized FISHDBC.

Memory usage. Figure 5.14 reports peak resident memory for the distributed components and the two centralized baselines.

The memory behavior differs substantially among the methods. The peak RSS of the Indexer increases from approximately 295 MiB at 2,000 records to approximately 4,643 MiB at 25,000 records. Centralized FISHDBC increases from approximately 247 MiB to 2,448 MiB over the same range.

The most pronounced growth is observed for HDBSCAN*. Its peak RSS increases from approximately 203 MiB at 2,000 records to 12,311 MiB at 25,000 records. This growth is consistent with the use of a precomputed pairwise Jaro–Winkler distance matrix, whose storage requirement grows quadratically with the number of records.

For the distributed implementation, memory values are reported independently for the Indexer, Leader, and maximum Worker. These values are intentionally not summed, since the processes execute on different machines and their maximum resident-memory measurements do not necessarily occur at the same time. The measurements again identify the Indexer as the component under the greatest memory pressure as the dataset size increases.

Candidate-graph reconstruction. The number of edges retained in the reconstructed global candidate graph is shown in Figure 5.15.

The candidate graph grows from approximately 43,482 edges at 2,000 records to approxi-

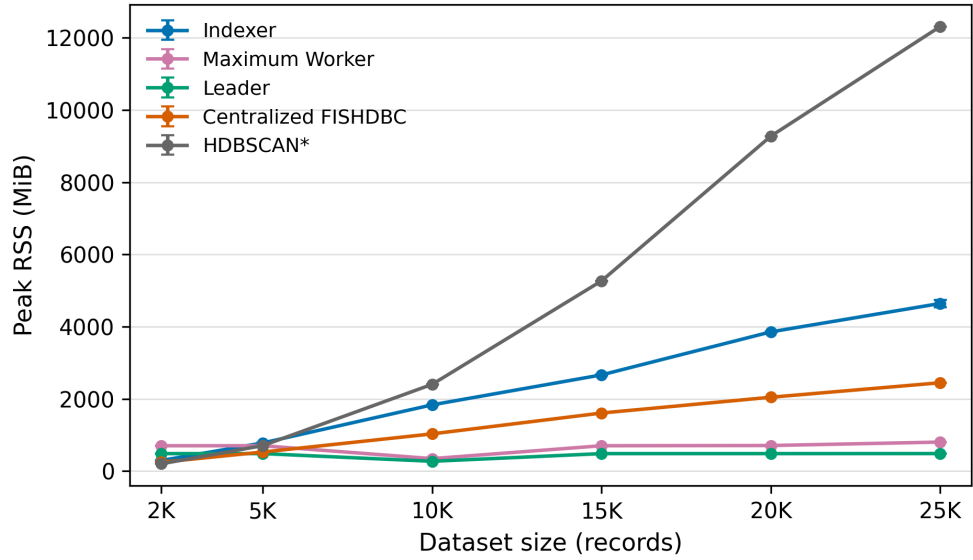


Figure 5.14: Peak resident memory during the M2 MusicBrainz scalability experiment. Distributed-component peaks are reported separately and are not summed.

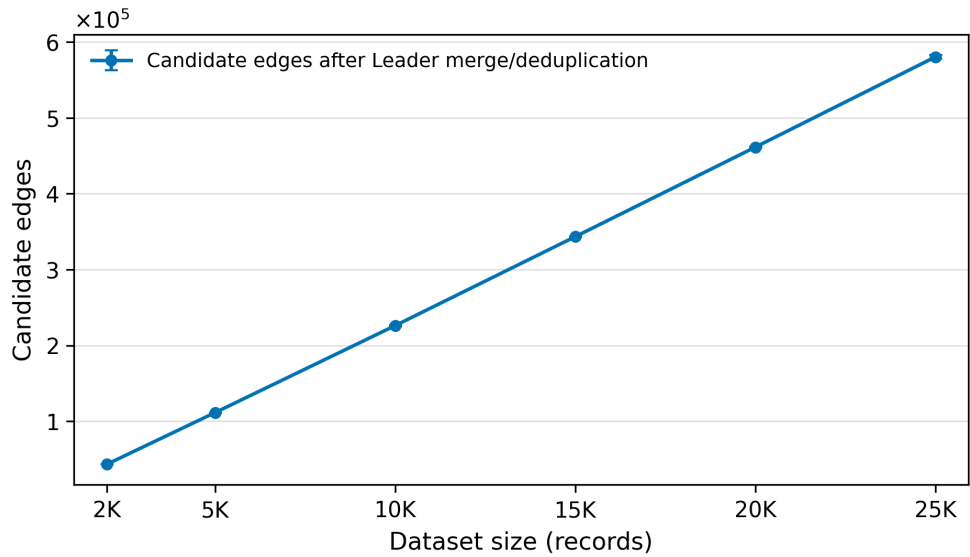


Figure 5.15: Size of the reconstructed global candidate graph as the MusicBrainz dataset size increases.

mately 580,242 edges at 25,000 records. Despite the increasing dataset size, global reconstruction succeeds in every evaluated configuration.

Table 5.19: Global graph-reconstruction diagnostics in M2. Candidate-edge counts are means over three repetitions.

Records	Candidate edges	Components	MST edges	Incomplete neighborhoods
2,000	43,482	1	1,999	0
5,000	111,475	1	4,999	0
10,000	226,111	1	9,999	0
15,000	343,337	1	14,999	0
20,000	461,319	1	19,999	0
25,000	580,242	1	24,999	0

The reconstructed candidate graph contains one connected component for every dataset size and every repetition. No point has an incomplete reconstructed neighborhood, no point has zero candidate neighbors, and the resulting MST contains exactly $N - 1$ edges in every run.

These observations indicate that the graph reduction performed by the Indexer and Workers retains sufficient candidate information for global connectivity over the complete evaluated M2 range. As in M1, however, the resulting MST is the MST of the approximate reconstructed candidate graph and should not be interpreted as the exact MST of the complete pairwise graph.

Clustering diagnostics. The cluster and noise counts follow broadly similar trends for the three methods as the dataset size increases. For example, at 25,000 records, Distributed FISHDBC produces on average 74.0 ± 6.1 clusters with approximately 20,983 noise points. Centralized FISHDBC produces 65 clusters with 21,513 noise points, while HDBSCAN* produces 79 clusters with 21,055 noise points.

One exception occurs for centralized FISHDBC at 10,000 records, where it produces two clusters and only 275 noise points, while Distributed FISHDBC produces approximately 23.7 clusters with 8,803 noise points and HDBSCAN* produces 20 clusters with 8,612 noise points. Since the centralized FISHDBC result is identical across its repetitions, this difference is not treated as random execution variability.

These clustering values are reported only as descriptive diagnostics. No external ground-truth partition is available for the artist-name strings, and the experiment does not interpret the resulting groups as semantic music clusters.

Summary. The M2 experiment shows that the computational advantage of approximate HNSW-based neighborhood discovery becomes more visible as the MusicBrainz dataset grows. HDBSCAN* is fastest for the smallest 2,000-record input, but its runtime and memory requirements grow rapidly because the Jaro–Winkler distance matrix requires all pairwise comparisons. From 5,000 records onward, Distributed FISHDBC is faster than both centralized baselines, and at 25,000 records it is approximately $5.15\times$ faster than the HDBSCAN* baseline and $1.16\times$ faster than centralized FISHDBC.

At the same time, M2 confirms the main architectural limitation identified in M1 and in the synthetic scalability experiments. Approximately 91–93% of the distributed runtime is spent in the Indexer, while the parallel Worker phase and Leader reconstruction remain comparatively small. The distributed approach therefore reduces the impact of local neighborhood processing, but its scalability is increasingly constrained by centralized sampling, graph processing, and routing.

Despite this bottleneck, the reconstructed candidate graph remains connected for every tested dataset size, all reconstructed neighborhoods contain at least the required number of candidate neighbors, and a complete $N - 1$ edge spanning tree is produced in every run. M2 therefore demonstrates both the benefit of the flexible HNSW-based distance evaluation strategy and the main centralized bottleneck that should be addressed in future versions of the distributed architecture.

5.8 Chapter Summary

This chapter evaluated the proposed Distributed FISHDBC approach using both synthetic and application-specific workloads. The experiments examined clustering agreement, the effects of sampling and replication, execution time, memory usage, graph reconstruction, communication, and scalability.

The synthetic Blob experiments showed that Distributed FISHDBC preserved the expected clustering structure across the evaluated configurations. Sampling and replication mainly affected computational cost and the amount of candidate neighborhood information, while clustering agreement remained high. The scalability experiment also showed that the distributed implementation scaled more effectively than centralized FISHDBC over the tested dataset sizes, although the centralized Indexer became the dominant runtime and memory bottleneck.

The MusicBrainz experiments extended the evaluation to an application-specific Jaro–Winkler distance. The results showed that HNSW-based neighborhood discovery evaluates only a fraction of all possible pairwise distances, which becomes increasingly important as the dataset size grows. From 5,000 records onward, Distributed FISHDBC was faster

than both centralized FISHDBC and the HDBSCAN* baseline, while HDBSCAN* showed substantially higher runtime and memory growth because of the complete pairwise distance matrix.

Across the MusicBrainz scalability experiments, the reconstructed candidate graph remained connected, all required neighborhoods were available, and a complete $N - 1$ edge spanning tree was produced in every run. At the same time, the Indexer consistently accounted for the majority of distributed execution time, confirming that global sampling, graph processing, and routing are the main scalability limitations of the current prototype.

Overall, the evaluation shows that the proposed architecture can distribute a substantial part of FISHDBC’s neighborhood-processing workload while retaining the global density and connectivity information required for hierarchical clustering. The results also identify the centralized components that should be addressed in future work to improve scalability further.

Chapter 6

Conclusion

This chapter summarizes the main contributions and findings of this thesis. It revisits the research questions introduced in Chapter 1, discusses the main limitations of the proposed distributed FISHDBC prototype, and outlines directions for future work. The chapter concludes by highlighting the main lessons learned from distributing neighborhood discovery while retaining the global density and connectivity information required for hierarchical clustering.

6.1 Summary of the Work

This thesis investigated how the main principles of FISHDBC can be adapted from a single-machine execution model to a distributed environment. The main challenge is that density-based hierarchical clustering depends on neighborhood information, core distances, and global connectivity, while distributing the dataset gives each Worker only a partial view of these relationships.

The proposed architecture separates the computation into an Indexer, multiple Workers, and a Leader. The Indexer constructs a sampled global HNSW, partitions the resulting landmark graph, and routes the complete dataset to the Workers. Each Worker builds a local HNSW, records the raw distances evaluated during neighborhood discovery, reconstructs local neighborhood information, and reduces its observed graph using local mutual-reachability weights.

The Leader combines the reduced raw-distance edges produced by the Indexer and Workers, reconstructs approximate global core distances, recomputes mutual-reachability weights using the merged candidate graph, and constructs the global minimum spanning tree used for hierarchical cluster extraction.

The resulting prototype distributes the computationally expensive neighborhood-discovery work across multiple machines while retaining a centralized stage for operations that depend on global information.

6.2 Answers to the Research Questions

RQ1: Clustering agreement

The synthetic experiments show that Distributed FISHDBC can preserve the main clustering structure obtained by centralized density-based methods under the evaluated configurations. In the B1 reference experiment, Distributed FISHDBC consistently recovered the expected ten clusters and achieved ARI and AMI values close to one with respect to the known generating labels.

The distributed result was also highly consistent with HDBSCAN*, while centralized FISHDBC showed slightly larger deviations in the number of clusters and noise points. In the B4 scalability experiment, Distributed FISHDBC recovered the ten generating clusters exactly across all evaluated dataset sizes from 50K to 200K points.

These results indicate that the distributed reconstruction of approximate global neighborhoods and mutual-reachability relationships can preserve the relevant clustering structure for the evaluated synthetic workloads. However, the method remains approximate and the experiments do not establish equivalence with the complete pairwise HDBSCAN* graph for arbitrary datasets.

RQ2: Effect of sampling and replication

The parameter-sensitivity experiments show that sampling and replication mainly affect computational cost and the amount of candidate neighborhood information available to the distributed reconstruction.

In the sampling experiment, clustering quality remained nearly unchanged across the tested sampling rates. Even the smallest evaluated global sample was sufficient to preserve the ten-cluster structure of the synthetic dataset. Increasing the sampling rate, however, increased Indexer runtime substantially because a larger Meta-HNSW must be constructed and searched during routing.

Replication increased the overlap between Worker partitions and increased the number of candidate edges returned to the Leader. Nevertheless, this additional information did not produce a measurable improvement in clustering quality on the evaluated synthetic

dataset, where the reference configuration already achieved near-perfect agreement with a replication factor of one.

The results therefore show that larger samples and higher replication are not automatically beneficial. Their value depends on whether the additional neighborhood information compensates for the extra computation, communication, and memory usage.

RQ3: Computational trade-offs

The MusicBrainz scalability experiments further show how these trade-offs change when an application-specific distance function is used. With Jaro–Winkler distance, HNSW-based neighborhood discovery evaluates only a fraction of all possible pairwise distances. From 5,000 records onward, Distributed FISHDBC was faster than both centralized FISHDBC and the HDBSCAN* baseline. At 25,000 records, it was approximately $1.16\times$ faster than centralized FISHDBC and $5.15\times$ faster than HDBSCAN*.

The experiments also confirm that the Indexer is the main scalability bottleneck of the current architecture. Across the M2 dataset sizes, the Indexer accounted for approximately 91–93% of the distributed execution time, while the parallel Worker phase and Leader reconstruction remained comparatively small. These results show that distribution improves the scalability of FISHDBC under the evaluated workloads, but that centralized sampling, graph processing, and routing increasingly limit the overall benefit as the dataset grows.

6.3 Limitations

The current prototype has several limitations that should be considered when interpreting the experimental results.

First, the distributed implementation does not preserve the incremental insertion capability of the original FISHDBC algorithm. The prototype follows a staged batch workflow in which the Indexer processes a fixed dataset, the Workers perform local computation, and the Leader reconstructs the final clustering only after all required results have been received.

Second, important parts of the workflow remain centralized. In particular, the Indexer accesses the complete dataset and performs routing for every input point. The Leader also maintains the global candidate-edge collection and constructs the final MST. These centralized stages limit the scalability that can be obtained by increasing the number of Workers.

Third, the global clustering structure is reconstructed from approximate HNSW observations and from reduced candidate graphs. The reconstructed core distances and final MST

are therefore approximate and are not guaranteed to be identical to those obtained from the complete pairwise mutual-reachability graph.

Fourth, the experimental evaluation was performed on a fixed physical cluster with one front node and ten Worker machines. The experiments therefore characterize the behavior of the proposed prototype under this deployment and do not constitute a general strong- or weak-scaling study over different numbers of machines.

Finally, the synthetic dataset contains relatively well-separated clusters, which makes the clustering problem favorable for the distributed approach. More difficult datasets with overlapping clusters, highly irregular density, or stronger partition-boundary effects may show greater sensitivity to sampling and replication.

The evaluation also does not directly measure the fraction of true k -nearest-neighbor relationships that cross Worker partition boundaries. The replication experiment therefore evaluates the effect of additional partition overlap indirectly through clustering quality, graph reconstruction, communication, and runtime. A dedicated boundary-sensitive dataset or explicit cross-partition neighbor analysis would provide a more direct evaluation of this aspect.

6.4 Future Work

A first direction for future work is to include the parallel FISHDBC approach proposed by Pastorino [Pas23] in the experimental evaluation. This would provide an additional reference between the original single-machine FISHDBC implementation and the distributed approach developed in this thesis, allowing the effect of parallelism and distribution to be compared more directly.

A second important direction is to restore the incremental behavior of the original FISHDBC algorithm. The current distributed implementation follows a batch-oriented workflow, where the complete dataset is processed before the final clustering is reconstructed. Supporting incremental insertion would require mechanisms for updating partition assignments, Worker-local HNSWs, core-distance information, and the global MST without reprocessing the entire dataset.

Finally, the current Indexer requires access to the complete dataset in order to construct the sampled global structure and route all points to the Workers. This represents one of the main centralized limitations of the current architecture. Future work should therefore investigate alternative approaches that reduce or remove this requirement, for example by distributing the indexing and routing process across multiple machines.

6.5 Final Remarks

This thesis demonstrates that the main clustering principles of FISHDBC can be adapted to a distributed execution model without requiring the Workers to produce independent final clusterings. By separating local neighborhood discovery from global density and connectivity reconstruction, the proposed architecture allows a substantial part of the FISHDBC workload to be executed across independent machines.

The experiments show that this approach can preserve the expected clustering structure on the evaluated synthetic datasets while improving scalability relative to centralized FISHDBC. At the same time, the results make clear that distribution does not remove all scalability limitations: global routing and reconstruction remain important centralized costs. The prototype therefore provides a practical starting point for distributed flexible hierarchical density-based clustering and identifies the main architectural components that should be addressed in future work.

Bibliography

- [Ank+99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. “OPTICS: Ordering Points To Identify the Clustering Structure”. In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’99. New York, NY, USA: Association for Computing Machinery, 1999, pp. 49–60. DOI: [10.1145/304182.304187](https://doi.org/10.1145/304182.304187).
- [CMS13] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. “Density-Based Clustering Based on Hierarchical Density Estimates”. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–172. ISBN: 978-3-642-37456-2.
- [CGL10] Min Chen, Xuedong Gao, and Huifei Li. “Parallel DBSCAN with Priority R-tree”. In: *Proceedings of the 2010 2nd IEEE International Conference on Information Management and Engineering (ICIME 2010)*. IEEE, 2010, pp. 508–511. DOI: [10.1109/ICIME.2010.5477926](https://doi.org/10.1109/ICIME.2010.5477926).
- [Cou+11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. 5th. Addison-Wesley / Pearson, 2011. ISBN: 978-0132143011.
- [Del19] Matteo Dell’Amico. *FISHDBC: Flexible, Incremental, Scalable, Hierarchical Density-Based Clustering for Arbitrary Data and Distance*. 2019. arXiv: [1910.07283](https://arxiv.org/abs/1910.07283) [cs.LG]. URL: <https://arxiv.org/abs/1910.07283>.
- [Del] Matteo Dell’Amico. *Flexible Clustering: FISHDBC Implementation*. <https://github.com/matteodellamico/flexible-clustering>. Source code repository, accessed 2026.
- [Den+19] Shiyuan Deng, Xiao Yan, Kelvin K. W. Ng, Chenyu Jiang, and James Cheng. *Pyramid: A General Framework for Distributed Similarity Search*. 2019. arXiv: [1906.10602](https://arxiv.org/abs/1906.10602) [cs.DC]. URL: <https://arxiv.org/abs/1906.10602>.
- [Doca] Docker, Inc. *Deploy Services to a Swarm*. <https://docs.docker.com/engine/swarm/services/>. Accessed: 2026.

- [Docb] Docker, Inc. *How Services Work*. <https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/>. Accessed: 2026.
- [Docc] Docker, Inc. *Manage Swarm Service Networks*. <https://docs.docker.com/engine/swarm/networking/>. Accessed: 2026.
- [Docd] Docker, Inc. *Swarm Mode Key Concepts*. <https://docs.docker.com/engine/swarm/key-concepts/>. Accessed: 2026.
- [Epp94] David Eppstein. “Offline Algorithms for Dynamic Minimum Spanning Tree Problems”. In: *Journal of Algorithms* 17.2 (1994), pp. 237–250. DOI: [10.1006/jagm.1994.1033](https://doi.org/10.1006/jagm.1994.1033).
- [Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. 1996, pp. 226–231.
- [He+10] Qing He, Fuzhen Zhuang, Jincheng Li, and Zhongzhi Shi. “Parallel Implementation of Classification Algorithms Based on MapReduce”. In: *Rough Set and Knowledge Technology*. Ed. by Jian Yu, Salvatore Greco, Pawan Lingras, Guoyin Wang, and Andrzej Skowron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 655–662. ISBN: 978-3-642-16248-0.
- [Hua+17] Fang Huang, Qiang Zhu, Ji Zhou, Jian Tao, Xiaocheng Zhou, Du Jin, Xicheng Tan, and Lizhe Wang. “Research on the Parallelization of the DBSCAN Clustering Algorithm for Spatial Data Mining Based on the Spark Platform”. In: *Remote Sensing* 9.12 (2017). ISSN: 2072-4292. DOI: [10.3390/rs9121301](https://doi.org/10.3390/rs9121301). URL: <https://www.mdpi.com/2072-4292/9/12/1301>.
- [Jai10] Anil K. Jain. “Data clustering: 50 years beyond K-means”. In: *Pattern Recognition Letters* 31.8 (2010). Award winning papers from the 19th International Conference on Pattern Recognition (ICPR), pp. 651–666. ISSN: 0167-8655. DOI: <https://doi.org/10.1016/j.patrec.2009.09.011>. URL: <https://www.sciencedirect.com/science/article/pii/S0167865509002323>.
- [Kru56] Joseph B. Kruskal. “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem”. In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50. DOI: [10.1090/S0002-9939-1956-0078686-7](https://doi.org/10.1090/S0002-9939-1956-0078686-7).
- [Lul+16] Alessandro Lulli, Matteo Dell’Amico, Pietro Michiardi, and Laura Ricci. “NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data”. In: *Proceedings of the VLDB Endowment* 10.3 (2016), pp. 157–168. DOI: [10.14778/3021924.3021932](https://doi.org/10.14778/3021924.3021932).

- [MY20] Yu A Malkov and Dmitry A Yashunin. “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2020), pp. 824–836. DOI: [10.1109/TPAMI.2018.2889473](https://doi.org/10.1109/TPAMI.2018.2889473).
- [MHA17] Leland McInnes, John Healy, and Steve Astels. “hdbscan: Hierarchical density based clustering”. In: *Journal of Open Source Software* 2.11 (2017), p. 205. DOI: [10.21105/joss.00205](https://doi.org/10.21105/joss.00205). URL: <https://doi.org/10.21105/joss.00205>.
- [Met26] MetaBrainz Foundation. *MusicBrainz Database Download*. 2026. URL: https://musicbrainz.org/doc/MusicBrainz_Database/Download (visited on 09/29/2026).
- [Pas23] Edoardo Pastorino. “Parallel Flexible Clustering”. Master’s thesis. Università degli Studi di Genova, Dec. 2023. URL: <https://unire.unige.it/handle/123456789/7200>.
- [Pat+12] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. “A new scalable parallel DBSCAN algorithm using the disjoint-set data structure”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC ’12)*. IEEE Computer Society, 2012, pp. 1–11. DOI: [10.1109/SC.2012.9](https://doi.org/10.1109/SC.2012.9).
- [SS13] Peter Sanders and Christian Schulz. “Think Locally, Act Globally: Highly Balanced Graph Partitioning”. In: *Experimental Algorithms (SEA 2013)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 164–175. DOI: [10.1007/978-3-642-38527-8_16](https://doi.org/10.1007/978-3-642-38527-8_16).
- [TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. Prentice Hall, 2007. ISBN: 978-0132392273.
- [Wu+22] Guoqing Wu, Liqiang Cao, Hongyun Tian, and Wei Wang. “HY-DBSCAN: A hybrid parallel DBSCAN clustering algorithm scalable on distributed-memory computers”. In: *Journal of Parallel and Distributed Computing* 168 (2022), pp. 57–69. DOI: [10.1016/j.jpdc.2022.06.005](https://doi.org/10.1016/j.jpdc.2022.06.005).