Hecatoncheir: Scaling up and out spatial data management

Thanasis Georgiadis
University of Ioannina
Ioannina, Greece
ageorgiadis@cs.uoi.gr

Achilleas Michalopoulos
University of Ioannina
Ioannina, Greece
amichalopoulos@cs.uoi.gr

Dimitris Dimitropoulos
University of Ioannina &
Archimedes, Athena RC
Ioannina, Greece
ddimitropoulos@cs.uoi.gr

Dimitris Tsitsigkos
Archimedes, Athena RC
Athens, Greece
dtsitsigkos@athenarc.gr

Nikos Mamoulis
University of Ioannina &
Archimedes, Athena RC
Ioannina, Greece
nikos@cs.uoi.gr

Abstract

We present Hecatoncheir, a plug-and-play C/C++ library for distributed and parallel management of big spatial data, which does not depend on underlying engines such as Spark. Hecatoncheir uses state-of-the-art algorithms for in-memory index-based spatial query processing and the efficient C++ Boost Geometry for geometry comparisons in a distributed environment, achieving orders of magnitude faster performance than Apache Sedona.

CCS Concepts

• **Information systems → Geographic information systems**; **Parallel and distributed DBMSs**.

Keywords

spatial data, big data, distributed systems, mpi

ACM Reference Format:
Thanasis Georgiadis, Achilleas Michalopoulos, Dimitris Dimitropoulos, Dimitris Tsitsigkos, and Nikos Mamoulis. 2025. Hecatoncheir: Scaling up and out spatial data management. In *The 33rd ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '25), November 3–6, 2025, Minneapolis, MN, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3748636.3762796

1 Introduction

Existing distributed spatial libraries such as Apache Sedona (sedona.apache.org), SpatialHadoop [2], SIMBA [11], LocationSpark [9] and more, require Spark and Hadoop to be properly setup in the cluster in order to be deployed upon them. Additionally, the architectures of these frameworks were specifically designed to integrate seamlessly with their underlying engines. As a result, they inherit both the benefits of those engines and their limitations. Moreover, spatial libraries like JTS (github.com/locationtech/jts), GEOS (libgeos.org) and Google S2 are limited to providing APIs for geometry operations, but they do not constitute standalone systems for distributed indexing, data partitioning, and query evaluation. Several distributed spatial analytics frameworks have been compared against each other [6], with Apache Sedona being considered the most prominent and popular one. All these frameworks share the following drawbacks:

Setup Complexity They are based on an underlying engine (e.g., Spark) which needs to be installed beforehand. The engines come with their own dependencies; for example, cluster managers such as YARN or Kubernetes. On top of that, the frameworks have their own installation process as well and need to be configured to work along with their underlying engine. Hence, they cannot be considered as "plug-and-play" tools to the casual user.

Resource cost The spatial indexes provided by existing frameworks are becoming outdated, with high construction and usage costs. Their memory usage for large datasets often reaches tens of gigabytes, while query throughput hovers around a few hundred per minute [6]. In today's era of cloud services and pay-as-you-go pricing models, these limitations can significantly inflate operational costs for both users and enterprises.

No C/C++ support Currently, no distributed spatial data management framework offers C/C++ support for their API. C/C++ are usually the go-to language option for performance-focused implementations and, thus, there has been a huge gap in distributed spatial data management in C/C++ until now.

In this paper we introduce Hecatoncheir, the first plug-and-play C/C++ library for in-memory distributed and parallel spatial data management. Hecatoncheir's underlying layer is implemented using MPICH (mpich.org), a Message Passing Interface (MPI) standard. Hecatoncheir offers optimized spatial partitioning and indexing techniques to support scalable distributed spatial query processing, without depending on external process and resource managers or engines. It compiles and links using CMake, seamlessly integrating into the user's projects while encapsulating all distribution-related complexities within a black-box logic for a plug-and-play experience. Currently, Hecatoncheir uses Boost Geometry for the geometric operations, as it outperforms GEOS. To scale computationally intensive tasks, such as data partitioning and spatial query evaluation [4, 10], Hecatoncheir employs intra-node parallelism through OpenMP. By distributing and maintaining data in the main memory of each node, the system fully utilizes available resources for efficient, scalable query execution, where each machine operates *independently*, minimizing communication overhead.

Table 1: Hecatoncheir and Apache Sedona features list.

Feature	Apache Sedona	Hecatoncheir
Language	Java/Scala	C/C++
Index	RTree, QuadTree	Two-layered Grid
Queries	Range, (Distance) Join, kNN	Range, (Distance) Join, kNN

We demonstrate Hecatoncheir by highlighting its ease of installation, requiring minimal user effort. The system features a simple C++ API and an intuitive graphical user interface (GUI) that enable users to partition data and evaluate spatial join queries with just a few lines of C++ code or a few clicks. These operations can be seamlessly executed across a selected set of machines. Additionally, we demonstrate Hecatoncheir's scalability, performance, and memory efficiency, via a comparison against Apache Sedona. An overview of both frameworks' features can be seen in Table 1.

2 Architecture

Hecatoncheir's architecture is summarized in Figure 1. The MPI Layer is where all inter- and intra- process communication takes place. Internally, the Query Processor uses the Index Layer [10] for spatial queries, the results of which are potentially refined using Boost Geometry (boost.org). The API is the entry point to the system's functionalities for the user.

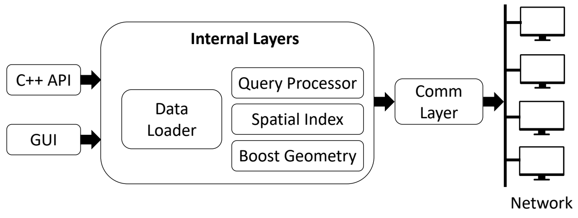


Figure 1: Hecatoncheir architecture overview.

2.1 Data Loading and Distribution

The Host partitions user data across Workers using a coarse-grained spatial grid, whose tiles are assigned to Workers in a round-robin manner [7]. An object is sent to the Worker responsible for any tile its minimum bounding rectangle (MBR) overlaps. This coarse grid acts as a global spatial index, and its granularity (e.g., 100×100 for 10 Workers) ensures more tiles than Workers, promoting load balancing and limiting fragmentation of nearby objects, which is important for distance-based queries. An illustration for a (global) dataset is shown in Figure 2 (bold lines).

Each Worker stores and indexes its assigned tiles using a much finer grid, with granularity set so that the number of cells is divisible by the number of threads, facilitating intra-node parallelism during query evaluation. Both grid granularities can be set manually or optimized automatically based on data distribution.

The Host is solely responsible for partitioning the data across Workers, which introduces considerable overhead. To mitigate this, if supported by the storage medium (e.g., SSD, NVMe), the Host spawns threads to parallelize both data reading and distribution. Each thread performs its own MPI calls, sending object batches to Workers based on a coarse partitioning grid. The batch size is configurable to (i) avoid exhausting Host memory and (ii) allow

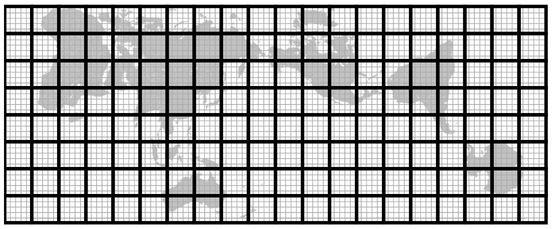


Figure 2: Spatial partitioning using a coarse grid.

Workers to begin processing before the entire dataset is read. Since each thread maintains a batch per Worker, the Host's memory usage is approximately $num_{threads} \times num_{workers} \times batch_{size}$, plus additional space for in-memory partitioned data. Hence, batch size must be chosen with memory limits and input size in mind.

2.2 MPI Layer

Communications in Hecatoncheir are illustrated in Figure 3. All machines in the cluster are connected within the same network by (passwordless) SSH. One of the machines takes the role of the Host. A user program, i.e. a Driver, communicates exclusively with the Host using MPI through the API. The Host is also a Worker, however it has the extra responsibility of communicating with the Driver. After receiving the Driver's requests, the Host propagates them to the Workers, returning any relevant messages or results to the Driver when necessary.

Each Worker (including the Host), spawns 2 processes: The *Controller* and the *Agent*. The Controller receives/sends messages and performs no CPU intensive tasks. The Agent performs all requested operations on the machine. Each Agent communicates only with its Controller (its parent process in the same machine). This way, Agents can work on tasks in parallel, while their Controllers can continue communicating with the Host or each other. Additionally, Agents utilize the available threads in the machine to parallelize their assigned tasks and operations.

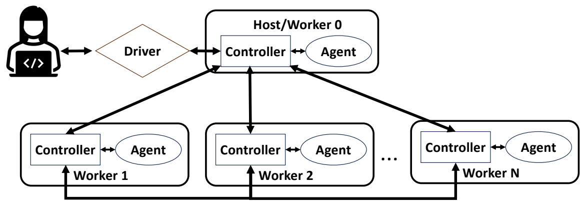


Figure 3: Communication setup.

2.3 Internal Layers

Hecatoncheir's internal layers are hidden from the user. The MPI layer is not part of the internal layers. Even though the Driver communicates with the Host through MPI calls as well, this is done via specific API calls (i.e., the user does not do MPI calls directly).

2.3.1 Boost Geometry. Boost Geometry offers a wide variety of geometric operations and data structures, which can be used to post-process objects that pass the filter step of spatial queries on complex object geometries, such as range selections and intersection joins. Hence, for objects or object pairs that pass the MBR-based filter step of such queries, the Workers access their geometries and use the Boost Geometry library for the refinement step.

2.3.2 Index Layer. As explained, during data loading, each Worker re-partitions its local data by a fine-grained uniform grid. Non-point geometries are further indexed using Two-Layer partitioning [10]; in each fine cell, the objects are divided two four classes, depending on whether the bottom-left point of their MBR begins (A) inside that cell, (B) in a previous cell on Y axis, (C) in a previous cell on X axis, or (D) in a previous cell on both axes. Object MBRs that overlap with more than one cell are replicated and classified. Fine-grid cells indexed by a Worker w store only the objects assigned to w and are not visible to other workers. The local indices are stored in the main memories of the Workers. In addition, each Worker computes and stores a raster approximation [4] for each object assigned to it. The granularity of the fine grid is in the order of a thousand partitions for each dimension. In this demo, it is 800×800, i.e., each tile of the coarse grid includes 10×10 cells. Grid-based partitioning and indexing facilitates fast data updates.

2.3.3 Query Processing Layer. Query processing takes place after setting up Hecatoncheir and all data has been partitioned to and indexed by the Workers. Since our query processing algorithms do not require re-partitioning the input data, the preprocessing cost of data files is one-off. Queries are evaluated in parallel by each Worker using their local (fine-grid) index. The results are then collected by the Host and returned to the Driver. The following queries are currently supported by Hecatoncheir:

Range Queries (Polygonal or Box) The user (Driver) submits a range query or a batch of range queries to the Host. The Host Controller determines which partitions of the coarse grid overlap with the query to identify the Workers responsible for evaluating it and sends the query to these Workers. Range queries on non-point data are evaluated using the Two-Layer partitioning scheme [10] that identifies candidates fast using their MBRs and does not generate duplicate results. A refinement step is performed for each candidate object to verify whether the geometry is actually intersecting the query range, using Boost Geometry. To avoid refining all candidates the Worker first accesses their raster approximations [4] and apply the query there. Range queries on point data are easier to evaluate, as no refinement is necessary. The contents of cells that are fully covered by the query range are automatically returned as results, without any comparisons. For a query batch, the Host distributes the queries to Workers in sub-batches and each Worker evaluates its sub-batch in-parallel using its local threads.

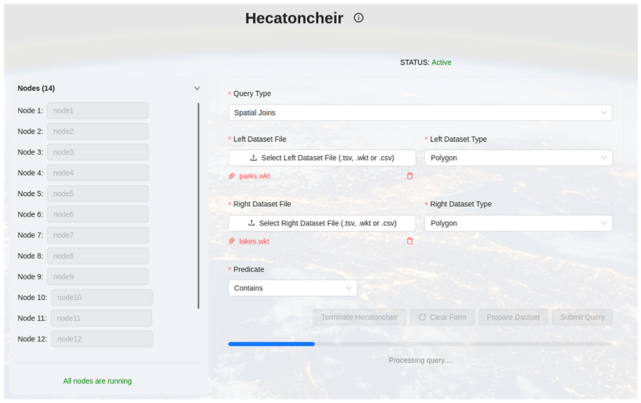
Topological Joins Spatial join queries between datasets containing non-point objects are first broadcast to all Workers. Based on the predicate, Workers use their local indexes to perform the join between the partitioned datasets. Supported spatial predicates for joins include: *Equals*, *Disjoint*, *Intersects*, *Inside*, *Contains*, *Meets*. All topology joins are enhanced through APRIL for better performance [3]. The system computes the result of *Disjoint* implicitly as the complement of *Intersects*, which is faster to derive. The partitioning of objects within cells to classes [10] accelerates the filter step of the spatial join. The pre-computed APRIL approximations of objects [4] further reduce the number of pairs for which the computationally expensive refinement step is applied (using Boost Geometry). Each worker performs the join for each of their own partitions independently and in parallel.

Distance queries Hecatoncheir also supports distance spatial queries, including ϵ-range queries (given a query point q, retrieve all points with distance at most ϵ to q), kNN queries (find the k nearest neighbors to query point q), and ϵ-distance joins (given two input datasets, find the pairs of objects in them having distance at most ϵ to each other). For ϵ-range queries and kNN queries, processing of each query is done independently at each Worker which may include query results and the results are combined at the Host. For instance if the query is at a boundary of the coarse grid, two Workers compute and report their kNN results (i.e., $2k$ objects), which are then refined at the Host. Count and distribution statistics for the coarse grid are kept and used by the Host to determine the relevant Workers for each query. For distance joins, Workers coordinate with each other and exchange points at the borders of the coarse grid, to facilitate correct and duplicate-free join computation [5] whilst minimizing the communication overhead.

2.4 C++ API & Interface

Hecatoncheir's API (header-based) facilitates access to the system's features, without burdening the user with low-level implementation details. The Driver initializes Hecatoncheir by calling the appropriate initialization method, specifying the desired number of Workers and their host IP addresses. Passwordless SSH connection between the Driver's machine and the Workers must be properly setup. The user then can develop their own program to load and query their datasets with just a few method calls.

Additionally, we provide a web-based GUI that enables users to interact with Hecatoncheir without writing any code. The interface abstracts the underlying communication with Hecatoncheir's API, allowing users to configure and execute spatial queries through an easy-to-use point-and-click workflow. A snapshot of a spatial query execution through Hecatoncheir's GUI is shown in Figure 4.



STATUS: Active

Nodes (14)
Node 1: node1
Node 2: node2
Node 3: node3
Node 4: node4
Node 5: node5
Node 6: node6
Node 7: node7
Node 8: node8
Node 9: node9
Node 10: node10
Node 11: node11
Node 12: node12
All nodes are running

* Query Type: Spatial Joins
* Left Dataset File: Select Left Dataset File (.tsv, .wkt or .csv) — parks.wkt
* Left Dataset Type: Polygon
* Right Dataset File: Select Right Dataset File (.tsv, .wkt or .csv) — lakes.wkt
* Right Dataset Type: Polygon
* Predicate: Contains

Processing query....

Figure 4: Snapshot of spatial query execution through Hecatoncheir's GUI.

2.5 In-Memory Management

Hecatoncheir is designed to perform all operations in memory without requiring all data to be memory-resident. For instance, non-point geometries, which can have a significant memory footprint, can be stored on disk and loaded on-demand during query processing, ensuring that only the required objects are brought into memory when needed. This is attributed to Hecatoncheir's filter-and-refine approach, that uses space-efficient approximations such as MBRs [1] and APRIL [4] as effective filters.

3 Demonstration

The functionality of Hecatoncheir is demonstrated through a scenario that simulates the process that casual user would follow to download, install, and utilize the system. Additionally, Apache Sedona will run side-by-side with Hecatoncheir during the demonstration, to further showcase the latter's superior performance.

3.1 Setup & Usage

To showcase Hecatoncheir's plug-and-play design, we demonstrate its deployment by downloading it directly from its GitHub repository onto a cluster of virtual machines. Each node in the cluster operates on a fresh installation of Ubuntu 22.04, configured with only the minimum requirements necessary for Hecatoncheir: (i) host set up with Boost Geometry 1.73, CMake 3.22, and C/C++17; (ii) MPICH installed on all nodes; and (iii) passwordless SSH configured between the nodes. Once downloaded, we build Hecatoncheir using the installation script and we execute the provided test suite to validate inter-node connectivity, communication, and the system's deployment in the cluster. We demonstrate Hecatoncheir's easy-to-use API by live-coding and executing queries in real time, with just a few lines of code. At the same time, we will monitor the virtual machines' resources consumption and load balancing using our cluster manager's overview platform.

3.2 Performance and Scalability

We showcase Hecatoncheir's scaling-out ability by running a spatial intersection join scenario between the Lakes (8.4M polygons) and Parks (10M polygons) OSM datasets [8], using an increasing number of VMs on a cluster that runs on 5 host computers. Each VM runs on an Intel CPU i9 (4 cores), clocked at 3.70GHz and 12GB RAM. The virtual machines run Ubuntu 22.04 LTS and have C++17 and MPICH 4.0 installed. Table 2 shows the scalability of Hecatoncheir's indexing and join evaluation as the number of Workers increases. We will analyze to the attendees the reasons behind scalability. Namely, Hecatoncheir's preparation cost remains unaffected by the number of Workers, with the only added cost being a few extra inter-node communications. At the same time indexing and query time drop due to the use of two-level parallelism, with operations executed concurrently by Workers and their threads.

Attendees will be able to interact with Hecatoncheir through either its GUI or its C++ API to execute spatial query scenarios and observe its performance, compared against Apache Sedona side-by-side. Table 3 shows a comparison with 2 executor-cores per VM, 10GB of memory per executor, QuadTree partitioning with sampling enabled, RTree indexing per Spark partition and underlying Hadoop for data distribution. In Sedona, to avoid expensive index rebuild for each query, we simulated batch range query processing as a spatial join between the queries and the dataset. Hecatoncheir is at least two orders of magnitude faster than Sedona on range queries. Additionally, Apache Sedona took 1425 seconds to run a Lakes-Parks spatial join scenario of Table 2 on 10 nodes while Hecatoncheir finished in 6 seconds. The memory requirements of Hecatoncheir increase with the number of machines, due to a finer grid used with more machines that results in more replication [10]. As shown in Table 3, Hecatoncheir requires less than half the memory occupied by Sedona for the exact same query and both frameworks achieve a balanced memory requirement across the nodes, based on their Mean Absolute Deviation of memory usage across nodes. Overall, Hecatoncheir's superior performance is attributed to its use of advanced spatial data management techniques.

Table 2: Hecatoncheir's scalability.

Hecatoncheir/Nodes	2	4	6	8	10
Partitioning Time (sec)	64.03	64.09	64.19	64.25	64.35
Indexing Time (sec)	12.35	8.16	6.12	5.10	4.16
Join Time (sec)	19.30	12.95	10.50	7.05	5.86
Total Memory (GB)	13.9	14.8	15.9	17.7	18.2
Mean Abs. Dev. (GB)	0.47	0.34	0.40	0.77	0.73

Table 3: Hecatoncheir and Apache Sedona comparison on a cluster with 10 nodes.

10K Range Queries on 2.25M points		
Selectivity %	**Sedona time (s)**	**Hecatoncheir time (s)**
0.01	24	0.1
0.1	32	0.13
1	209	0.27

8.7M and 10M spatial join for polygon datasets		
	Sedona	**Hecatoncheir**
Time (s)	1425	6
Total Memory (GB)	42.65	18.2
Mean Abs. Dev. (GB)	0.77	0.73

Acknowledgments

Work supported by (a) project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program, and (b) the Hellenic Foundation for Research and Innovation (HFRI) under the "2nd Call for HFRI Research Projects to support Faculty Members & Researchers" (Project No. 2757).

References

[1] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1994. Multi-Step Processing of Spatial Joins. In *SIGMOD*.
[2] Ahmed Eldawy and Mohamed F. Mokbel. 2013. A Demonstration of Spatial-Hadoop: An Efficient MapReduce Framework for Spatial Data. *Proc. VLDB Endow.* 6, 12 (2013), 1230–1233.
[3] Thanasis Georgiadis and Nikos Mamoulis. 2026. Scalable Spatial Topology Joins. In *Proceedings 29th International Conference on Extending Database Technology, EDBT 2026, Tampere, Finland, March 24-27, 2026.* OpenProceedings.org, 110–116.
[4] Thanasis Georgiadis, Eleni Tzirita Zacharatou, and Nikos Mamoulis. 2025. Raster interval object approximations for spatial intersection joins. *VLDBJ* 34, 1 (2025), 8.
[5] Nikolaos Koutroumanis, Christos Doulkeridis, and Akrivi Vlachou. 2025. Parallel Spatial Join Processing with Adaptive Replication. In *EDBT*.
[6] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *PVLDB* 11, 11 (2018), 1661–1673.
[7] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *SIGMOD*.
[8] SpatialHadoop. 2015. *Datasets.* http://spatialhadoop.cs.umn.edu/datasets.html
[9] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *Proc. VLDB Endow.* 9, 13 (2016), 1565–1568.
[10] Dimitrios Tsitsigkos, Panagiotis Bouros, Konstantinos Lampropoulos, Nikos Mamoulis, and Manolis Terrovitis. 2024. Two-Layer Space-Oriented Partitioning for Non-Point Data. *IEEE Trans. Knowl. Data Eng.* 36, 3 (2024), 1341–1355.
[11] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*.