SHM-DB - A Shared Memory Database for FAIR Geospatial Algorithm Development

Martin Werner
martin.werner@tum.de
Professorship Big Geospatial Data Management, TUM School of Engineering and Design, Technical University of Munich
Munich, Germany

Katharina Anders
k.anders@tum.de
Professorship of Remote Sensing Applications, TUM School of Engineering and Design, Technical University of Munich
Munich, Germany

Abstract

This paper introduces the shared memory database SHM-DB and demonstrates how this can be used for developing point cloud processing algorithms on comparably large datasets with improved software quality. The SHM-DB acts as a component that completely disentangles all aspects of the computational process such that individual aspects can be implemented in a very reusable form. As a side-effect, the approach introduces a novel dimension of algorithmic transparency to the development system as the system visualizes intermediate results while an algorithm is running. Despite demonstrating the system on point cloud data, the system is generic and can be used beyond point clouds for other types of challenging data.

CCS Concepts

• **Information systems** → *Database management system engines*; • **Computing methodologies** → **Concurrent computing methodologies**; • **Software and its engineering** → **Integrated and visual development environments**.

Keywords

In-Memory Data Management, Modularization, Real-Time Visualization of Algorithms, Point Clouds

ACM Reference Format:
Martin Werner and Katharina Anders. 2025. SHM-DB - A Shared Memory Database for FAIR Geospatial Algorithm Development. In *The 33rd ACM International Conference on Advances in Geogrphic Information Systems (SIGSPATIAL '25), November 3–6, 2025, Minneapolis, MN, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3748636.3762794

1 Introduction

In the last decade, the amount of publicly available big geospatial data has been increasing significantly. Data infrastructures for geospatial applications, mainly built around concepts of relational databases or cloud computing, have been proposed and are ready to use, but there are geospatial datasets that are not well supported by these approaches and the adoption of relational data management systems for certain types of data is not (yet) common.

This category of geospatial data objects can be subsumed by its properties as follows: datasets in which an excessive number of comparably small objects need to be processed in both geospatial and attribute-driven relation with each other. For such data, spatiotemporally local processing schemes are often proposed as a full relation of each object with each other, which is both infeasible (n objects lead to $O(n^2)$ connections) and of limited value due to Tobler's first law of geography: everything is related to everything else, but near things are more related than distant things [7].

In this context, the decomposition of data into pieces (e.g., spatial grid cells, time slices, etc.) is a common external strategy allowing for efficient processing of such data in main memory. A lot of the current open source geospatial software is organized like this: assuming a fragmentation of the data into pieces manageable in main memory, multiple processing steps are implemented independently from each other and can be orchestrated into a data processing pipeline or geospatial processing workflow.

In software engineering, the Unix Philosophy is a set of rules originated around the rise of the Unix operating systems that find many formulations, the first one due to [3] with a few influential later simplifications. We want to stick to the following definition due to Peter H. Salus in [6]:

(1) Write programs that do one thing and do it well.
(2) Write programs to work together.
(3) Write programs to handle text streams, because that is a universal interface.

We want to constate here that most open source developments in the area of geospatial computing adhere to the first two of these fundamental principles of reusable software engineering, but not to the third as text streams are not widely perceived as a reasonable interface for complex multimedia data. In this paper, we want to update the third point of the Unix philosophy as follows:

3. Write programs to handle in-memory buffer objects, because buffers are a universal interface for geospatial and multimedia data.

There is a certain tradition of in-memory buffer representation standardization both from the hardware community (e.g., OpenGL data formats for textures and geometries, [2]) as well as in the OGC (e.g., the Well-Known Binary definitions, [4]), but even recently in the cloud computing community around projects such as Apache Arrow [1].

SIGSPATIAL '25, Minneapolis, MN, USA

ACM ISBN 979-8-4007-2086-4/2025/11
https://doi.org/10.1145/3748636.3762794

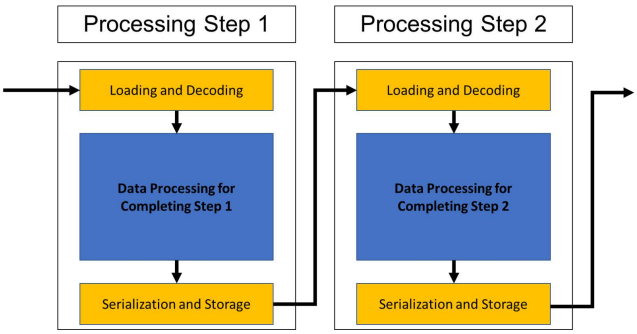


Figure 1: Marshalling Overhead of classical step-based processing systems with file-based data exchange between steps

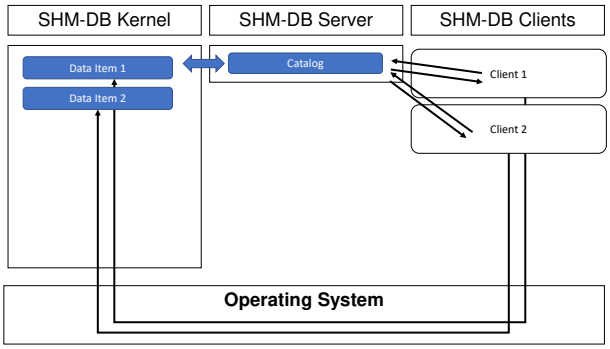


Figure 2: The SHM-DB Architecture

The remainder of this paper is structured as follows: In the following Section 2, we introduce the challenge of efficient parallel geospatial processing and how we solve it for datasets that are big but can still be held in main memory at least once. Then, we explain the demonstration of the system with real-world data in Section 3. Section 4 concludes the paper.

2 The Challenge and Its Solution

In this paper, we demonstrate a straightforward approach to work on very large datasets in main memory from different programming systems and processes by exploiting the long-existing, but not widely used mechanism of shared memory.

As depicted in Figure 1, classical processing pipelines implementing sequential tasks suffer from significant overheads when using the disk as the main communication medium between tasks. In many cases, a file is written just in order to be re-read immediately afterwards by a different computational step.

On the positive side, this pattern enforces a full specification of the communication between steps in the form of files, and allows for reusing steps as they only depend on input files and output files. The whole GNU coreutils collection of programs facilitates reuse essentially in this way. In scientific computing, however, and especially during development, such pipelines have their downsides as well. For example, when visualization of geospatial data is required, it is often implemented as just a step implying that the first glance of the outcome is available only after completing all previous steps and (maybe) the visualization step itself.

There is a common solution to avoiding the overheads introduced by frequent serialization and deserialization of intermediate results: practitioners keep the data in main memory in a long-running service such as a Jupyter Notebook.

From a software engineering perspective, however, this enables high coupling and easily leads to cross-step cohesion, mixing otherwise isolated aspects such as processing and visualization. As a consequence, only snippets of the source code - if at all - can be made to work in a different pipeline easily, and the overall aim to generate research software that maximizes reusability is at risk.

Even worse, this approach constrains us to the use of a single environment and programming language which might be far from optimal for some of the steps. For example, visualization in the Python ecosystem is often suboptimal ranking from users experiencing it as slow (when using matplotlib in a suboptimal way) or a quite specific dependency (e.g., open3d for pointclouds) requiring specialized knowledge. In addition, the lifetime of data, auxiliary data, and process is tightly coupled.

In this context, we propose SHM-DB as a solution. It allows us to handover ownership for any data item to SHM-DB decoupling processing steps entirely in terms of programming language, runtime, lifetime and effecitvely supports low cohesion in an efficient way allowing to avoiding all unneccessary serialization and deserialization steps.

As depicted in Figure 2, SHM-DB consists of a kernel component holding data items, a network server for interacting with the SHM-DB system, and a set of client libraries for accessing data in SHM-DB through the operating system's memory mapping subsystem.

In this way, data is never communicated or written to disk, but directly mapped into the address space of clients. For Python, it is made available as a numpy array through the Python buffer protocol [5]. From a general perspective, this allows for access to the data by isolated processing steps one after another and even concurrently.

By handing over data ownership to SHM-DB during development, we implicitly define what part of the data is important across steps. With little overhead, one can then implement a generic data persistency operation enumerating all datasets that have been given to SHM-DB and storing them in an open file format such as HDF5.

3 A Selected Point Cloud Demo

For demonstrating SHM-DB, we consider an environmental science scenario.

The Isar river near Munich is an important alpine river that contributes to one of the major river catchments in Europe. The study site where we collected data represents one of the last remaining natural stretches of the river and is a valuable area of natural heritage and ecological conservation.

However, as this river crosses the city of Munich, the water discharge within this river has been managed since decades. In the wild river area, however, questions of sediment transport and destruction and regeneration of natural habitats remain largely unanswered. We are currently acquiring aerial images in this area

Aspect	Ecosystem	Depends on	Lines
Loading	Python	laspy	20
Visualization	C	glfw	240
Shuffle	Python	(fire)	28
Dump	Python	h5py, (fire)	38

Table 1: Computational Aspects including Dependecies and number of lines (including comments)

on a regular basis (bi-annually), and use photogrammetric algorithms to reconstruct the 3D environment at centimeter quality in order to assess changes of the topography over months and years.

The latest point cloud of the study area contains about 90 million points. Representing a point cloud like this in a convenient analysis-ready data representation using floating points for real-world coordinates leads to an in-memory size of 2.26 GB. Figure 4a depicts the point cloud.

While this number does not seem too big, one should keep in mind that we are aiming for time series of such data, with three acquisitions already available and many environmental monitoring applications producing high-frequency acquisitions of thousands of epochs. Furthermore, such an in-memory size is actually quite big when parallel processing is applied and each parallel process reads its own copy of the data.

With this point cloud in mind, we instantiate a very simple example of a scientific processing chain that loads this point cloud, performs an operation on it, and visualizes the outcome.

As the research software engineering pattern, we adopt the well-known Model View Controller pattern, but not in an object-oriented sense, but rather by making the model, the view, and the controller independent processing modules that collaborate through SHM-DB.

The Data Model For the data model, we define a schema in which the point cloud is represented as two numpy arrays of the same number of rows where each row represents a 3D point. The first table `vertices` is storing the three geographic coordinates X, Y, and Z in a double precision floating point and the second table `colors` holds point cloud attributes, here RGB color information as three columns of bytes.

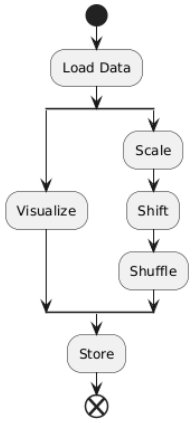


Figure 3: Flow Diagram of the Demo

Computational Aspects: The processing of the point cloud is composed of five isolated modules: Loading, Visualization, Shift&Scale, Shuffle, Dump. A flow diagram in Figure 3 illustrates the execution of those aspects.

The **Loading** module is responsible for reading a point cloud file and handing it over to SHM-DB. Therefore, it creates two numpy arrays named vertices and colors of three columns each, one holding geometry, the other one holding color information. It connects the color table with the point cloud table through a metadata record held in the SHM-DB catalog.

This step is chosen to be implemented in Python and introduces software dependencies related to file formats (e.g., laspy for reading a file in LAS file format, the de-facto standard for point cloud data) and additionally has a risk of creating machine-dependent artifacts by relating to filenames (e.g., local paths). However, this file does not depend on any visualization routines or library beyond numpy. This allows to keep it tremendously concise. Through using the SHM-DB catalog to model the relation between points and colors, algorithms that use only one of them (e.g., color filters or geometry operations) can do so and find the data in the appropriate form.

The **Visualization** module takes care of real-time interactive visualization. We chose to implement it natively in C++ based on the GLFW context management library, the OpenGL Extension Wrangler library (GLEW) and modern OpenGL. Additional dependencies include only poco and the platform-independent boost::interprocess module for shared memory. This has been kept rather simple as it does only orbit a camera around the origin. Through the SHM-DB concept we are able to externalize the navigation to a shift, scale and rotation of the point cloud and can avoid the implementation of an interactive user interface in the viewer. Furthermore, the viewer renders only the first ten million points of the point cloud, allowing us to keep rendering performance real-time. Again, the other clients decide what is seen by shuffling the important points to the beginning of the array.

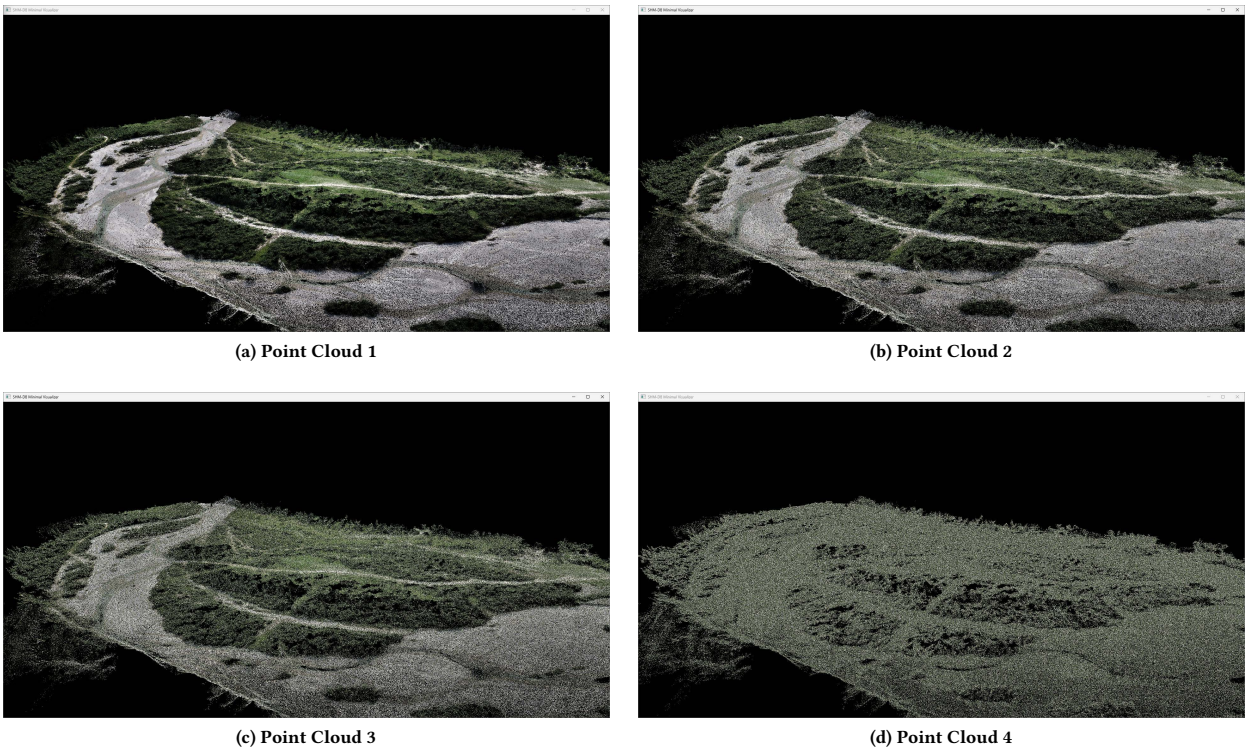
By using shared memory for accessing the points, we do not create copies of the data for visualization which is commonly the case when using off-the-shelf visualization libraries as part of the marshalling between the client's data representation and the internal representation of the scene in the library.

In the third module **Scale & Shift**, we augment the minimalistic viewer with full navigation capabilities for showing the point cloud centered to arbitrary points and scales by manipulating the coordinate array. This step is chosen to be implemented in Python as well and relies on numpy and the awesome broadcasting code to avoid loops making it concise as well.

One might wonder why we put Scale & Shift into its own module and do not integrate it with data loading. The reason is that point clouds often appear in geographic coordinate reference frames leading to very large numbers and need to be converted into concise numbers for visualization anyway (shifted towards the origin) in order to avoid roundoff artifacts in the multiplications of the projections during rendering. Data integration with GIS, however, needs the original coordinates.

It is worth noting that this step has no dependencies beyond numpy and only accesses the vertices and not the colors.

The fourth module **Shuffle** implements our toy algorithm. In fact, we only use the numpy shuffle algorithm first on vertices and



(a) Point Cloud 1

(b) Point Cloud 2

(c) Point Cloud 3

(d) Point Cloud 4

Figure 4: Beginning of the Numpy shuffle algorithm acting on a SHM-DB pointcloud while being visualized.

then on colors using the same permutation through fixing some random state.

In the viewer, we can observe how the point cloud remains geometrically consistent but wrong colors are rendered while the first shuffle on vertices is executed. Note that by shuffling vertices while not yet shuffling colors, points become associated with wrong colors. As the gray of the river banks dominates the dataset, it is not surprising that gray dominates this phase.

Then, the colors are shuffled and brought right into the correct place again leading to a reconstruction of the visual appearance of the point cloud despite that it shows a different set of points now.

The last module **Dump** is implemented in Python to store the state of the SHM-DB database in a file. We use the HDF5 container to store the numpy arrays held in the SHM-DB database. The associated metadata is encoded as a JSON string and written as an attribute in the HDF5 file. For certain objects, we store them in the native format of HDF5 such that the vertices are stored as a three-column table in HDF5. Data for which a specific mapping to HDF5 is not known will just be stored as an array of bytes.

Beyond numpy, this step depends only on the h5py library.

4 Discussion and Conclusion

In this paper, we demonstrate how the concept of Shared Memory can be used as a tool for improving research software engineering in geosciences for those data types where a database as a central exchange platform is inefficient. We showed how the Python buffer protocol works together with modern operating systems to build components for computational systems for which the ecosystem, programming language, and dependencies are individual.

Given the comment on practitioners holding alive Jupyter notebooks for very long times in order to avoid read-write delays, we aim to extend SHM-DB to become a plugin for Jupyter Lab such that even those users can gain from SHM-DB without getting out of their comfort zone.

Acknowledgments

With respect to this work, Martin Werner has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Grant No. 460036893 (NFDI4Earth).

References

[1] Apache Foundation. 2016. Apache Arrow. available at: https://arrow.apache.org/. last access: 14 January 2025.
[2] Khronos Group. 2022. OpenGL 4.6 (Core Profile). available at: https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf. last access: 14 January 2025.
[3] M McIlroy, EN Pinson, and BA Tague. 1978. UNIX time-sharing system. *The Bell system technical journal* 57, 6 (1978), 1899–1904.
[4] Open Geospatial Consortium. 2011. OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture . available at: https://www.ogc.org/de/publications/standard/sfa/. last access: 14 January 2025.
[5] Python Developer Team. 2024. Python Buffer Protocol. available at: https://docs.python.org/3/c-api/buffer.html. last access: 12 December 2024.
[6] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.
[7] Waldo R Tobler. 1970. A computer movie simulating urban growth in the Detroit region. *Economic geography* 46, sup1 (1970), 234–240.