Cross-Platform Spatial Analytics with Apache Wayang: Geometries Have Boundaries, Your Analytics Shouldn't [Demo]

Felix Treykorn
felix.treykorn@student.hpi.de
HPI & Uni Potsdam
Germany

Anton Persitzky
anton.persitzky@student.hpi.de
HPI & Uni Potsdam
Germany

Maximilian Speer
maximilian.speer@student.hpi.de
HPI & Uni Potsdam
Germany

Zoi Kaoudi
zoka@itu.dk
IT University of Copenhagen
Denmark

Eleni Tzirita Zacharatou
eleni.tziritazacharatou@hpi.de
HPI & Uni Potsdam
Germany

Abstract

Various platforms support spatial operations, from lightweight geometry libraries to distributed frameworks and spatially extended relational databases, each with its own API and execution paradigm. No single platform is best for every workload, yet migrating between platforms means rewriting application code, locking developers into their initial choice. We present the first general-purpose, cross-platform framework for spatial analytics, extending Apache Wayang to treat spatial operators as first-class citizens. Our framework provides platform-agnostic spatial operators with execution mappings to three backends, enabling developers to build spatial analytics pipelines that run across backends without code rewrites.

We demonstrate our framework through spatial analytics on apartment listings in Berlin. Attendees compose query plans with an interactive drag-and-drop builder, assign operators to backends, and view results as maps and charts. By comparing the recorded runtimes of jobs run under different backend assignments, they observe how backend choice affects end-to-end performance, and that no single backend is best for every spatial workload.

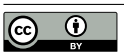
CCS Concepts

• **Information systems** → **Spatial-temporal systems**; *Middleware for databases.*

Keywords

unified spatial data analytics; cross-platform data processing

ACM Reference Format:
Felix Treykorn, Anton Persitzky, Maximilian Speer, Zoi Kaoudi, and Eleni Tzirita Zacharatou. 2026. Cross-Platform Spatial Analytics with Apache Wayang: Geometries Have Boundaries, Your Analytics Shouldn't [Demo]. In *The 34th ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '26), November 03–06, 2026, Riverside, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3841645.3843115



SIGSPATIAL '26, Riverside, CA, USA

ACM ISBN 979-8-4007-2950-8/2026/11
https://doi.org/10.1145/3841645.3843115

1 Introduction

Spatial data analytics is vital across various fields from urban planning and logistics to environmental monitoring and disaster response. Several platforms support spatial operations, from lightweight geometry libraries such as JTS [1], to distributed engines like Apache Sedona [11], Beast [7], and LocationSpark [9], to spatially extended relational databases such as PostGIS [2]. Each has its own API, ingestion model, and execution paradigm, and there is no one-size-fits-all solution. The right platform choice depends on factors such as data size, query selectivity, and deployment environment. However, switching platforms often requires rewriting code for a different API, which can lock developers into their initial choices and limit their ability to adapt the platform to the workload.

Cross-platform execution resolves this lock-in. It allows an application to run on or across multiple platforms without code rewriting, with each operator deployed on its best-suited backend [4]. However, cross-platform execution is challenging for spatial workloads. First, spatial systems differ in how they represent and encode geometries, using formats such as Well-Known Binary (WKB) or GeoJSON, so moving spatial records between platforms might require format conversions [10]. Second, spatial operators are highly sensitive to physical design, including indexing, partitioning, and data layout. Third, spatial systems differ in the coordinate reference systems they assume and in subtle aspects of geometry semantics. Together, these properties mean that a cross-platform framework must treat geometries and spatial operators as first-class citizens, so the optimizer can reason about placement and the runtime can move geometries safely. No existing system provides this.

Raven [5, 6] takes a first step toward spatial platform independence, but it targets only zonal statistics on raster and vector data. Apache Wayang [3, 4] is a cross-platform system for general analytics with a platform-agnostic operator model backed by a cost-based optimizer, but it lacks support for spatial data. We bridge this gap by extending Wayang, contributing the *first general-purpose, cross-platform spatial analytics capability*. Specifically:

- We introduce platform-agnostic spatial operators in Apache Wayang's operator abstraction, with execution mappings to Java/JTS, Apache Sedona, and PostGIS. The `SpatialGeometry` abstraction decouples operator logic from backend-specific geometry representations, and a concrete `WayangGeometry` implementation bridges heterogeneous geometry formats and data access patterns.

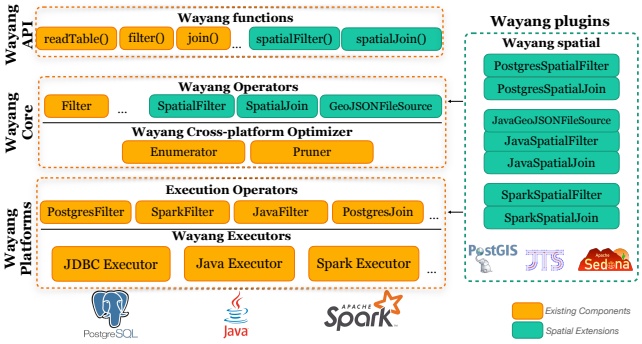


Figure 1: Architecture of Apache Wayang Spatial. Spatial operators are defined at the platform-agnostic level and mapped to execution backends via a plugin architecture.

- We implement the spatial extension as a plugin that encapsulates all spatial operator implementations and their backend dependencies, letting users opt into spatial functionality without affecting the core framework's behavior.
- We provide an interactive demonstration on apartment listings in Berlin, where attendees create query plans using a drag-and-drop builder, assign operators to backends, and view results as interactive maps and charts. By re-running plans with different backend assignments, attendees can see how operator placement impacts performance.

Our spatial extension is open-source and available as a plugin for Apache Wayang.[1]

2 Spatial Extension of Apache Wayang

After some background on Wayang, we describe our spatial extension, Wayang Spatial. Figure 1 shows an overview of its architecture.

2.1 Background: Apache Wayang

Apache Wayang [3, 4] is an open-source middleware framework that decouples data analytics applications from the underlying processing platforms. Its architecture consists of three main layers: (1) a *platform-agnostic API* through which users define analytics tasks using Wayang operations, (2) a *core layer* that encompasses the definitions of platform-agnostic Wayang operators and a cross-platform query optimizer that determines the most efficient execution plan potentially spanning multiple platforms, and (3) a *platforms layer* that contains platform-specific operator implementations with their mappings to Wayang operators and a set of executors that handle the execution on specific platforms.

Wayang's operator abstraction is central to its design. Each Wayang operator (e.g., `Map`, `Join`) is a platform-agnostic specification of a data transformation. For each supported backend, an *execution operator* implements the Wayang operator using platform-specific APIs. For example, a `Join` operator has separate execution operators for Spark (`SparkJoin`), Java (`JavaJoin`), and JDBC (`PostgresJoin`). Adding a new operator or platform requires providing these mappings, along with optional cost estimators for the optimizer. This extensible architecture makes Wayang a natural host for cross-platform spatial operations.

[1]Code: https://github.com/apache/wayang/tree/main/wayang-plugins/wayang-spatial

2.2 Spatial Wayang Operators

We extend Wayang's operator set with two families of spatial operators, spatial filters and spatial joins, that cover common spatial analytics operations. Our spatial operators act on records that contain at least one geometry attribute alongside conventional attributes. To support interoperability across backends, we introduce a `SpatialGeometry` abstraction in Wayang's core that the spatial operators are typed against, with a concrete `WayangGeometry` implementation that translates between multiple geometry representations (WKT, WKB, GeoJSON, JTS objects). Spatial data can be ingested from GeoJSON files via a new `GeoJSONFileSource` operator introduced by our extension, from CSV or other text formats using Wayang's existing file source operators, or directly from database tables via JDBC for PostGIS.

Spatial Filter. A spatial filter selects the records of a dataset whose geometry attribute satisfies a spatial predicate against a reference geometry, such as intersection or containment. The `SpatialFilter` operator takes a dataset, a predicate, and a reference geometry, and outputs the records that satisfy the predicate.

Spatial Join. A spatial join combines two datasets based on a spatial predicate over the geometry attributes of their records, such as intersection or containment. The `SpatialJoin` operator takes two spatial datasets and a join predicate, and outputs all pairs of records that satisfy it.

Composing Rich Analytics. Spatial filters and spatial joins integrate seamlessly with Wayang's existing platform-agnostic operators, such as `Map`, `Filter`, and `ReduceByKey`, enabling multi-step pipelines that combine spatial and non-spatial transformations for a broad range of analyses. For instance, given point records (e.g., apartment listings) and region polygons (e.g., postal districts), a spatial join pairs each record with the region that contains it. Then, a downstream `Map` can compute a new value for each point from the combined attributes of the point and its region, and a `ReduceByKey` can group records by region to compute aggregates, such as averages per region. Other spatial operators, such as k-nearest-neighbor, can be implemented either as compositions like the above or as new platform-agnostic operators with per-backend mappings, following the same pattern as `SpatialFilter` and `SpatialJoin`. Section 3 presents a multi-step pipeline over Berlin apartment listings that computes different rent statistics per district (cf. Figure 3 (left)).

2.3 Platform Integration

For each spatial operator, we provide implementations for three backends, each with distinct execution strategies and performance characteristics.

Java/JTS. The Java executor uses the JTS Topology Suite [1] for geometry operations. Spatial filters iterate over records and evaluate JTS predicates (e.g., `Geometry.intersects()`). Spatial joins use a nested-loop strategy with an optional in-memory JTS `STRtree` index. The Java backend is suitable for small to medium datasets and is useful for prototyping and debugging.

Apache Sedona. Sedona [11] extends Apache Spark with spatial data types, indexes, and query operators. We map Wayang's spatial operators to Sedona's distributed spatial join and range query APIs. Sedona partitions data spatially and builds local indexes within each partition, enabling scalable processing across a cluster. The Sedona backend is the natural choice for large-scale workloads.

PostGIS. PostGIS [2] extends PostgreSQL with spatial types and indexing (GiST-based R-trees). We map spatial operators to SQL queries with spatial predicates (e.g., `ST_Intersects`, `ST_Contains`). A key difference from the other backends is the data access model: PostGIS operates on data already stored in the database, while the Java and Sedona backends read data from external files (e.g., GeoJSON, CSV). PostGIS is therefore well suited to workloads of many queries over the same data, where the one-time loading and indexing cost is amortized.

Geometry Interoperability and Data Movement. Platform-agnosticism rests on the principle that spatial operators are typed against the abstract `SpatialGeometry` interface instead of backend-specific geometry types. The concrete `WayangGeometry` implementation holds whichever representation a geometry arrives in (e.g., WKT, WKB, GeoJSON, or a JTS object) and converts between them on demand, using caching to minimize recomputation, so each execution operator can obtain the geometry in the form its backend expects. When a pipeline crosses a backend boundary, the geometry is transferred in a portable representation and reconstructed on the other side, allowing the same Wayang plan to run on, or across, multiple backends. Coordinate reference systems (CRSs) add another interoperability layer, as backends may use different CRSs. `WayangGeometry` is designed to track each geometry's CRS and reproject it to a common CRS at the boundary, so spatial predicates remain accurate across heterogeneous stores.

2.4 Plugin Architecture and Extensibility

Apache Wayang is extensible by design, facilitating the addition of new operators and backends via abstract interfaces. However, a key challenge in adding spatial support is managing heterogeneous dependencies across backends. The Java executor relies on JTS for geometry operations, the Spark executor on Apache Sedona, and the JDBC executor on PostGIS-specific SQL extensions. Bundling these libraries into the framework would impose unnecessary dependencies on users who do not need spatial functionality.

To address this, we implement the spatial extension as a self-contained plugin that encapsulates all spatial operator implementations and their backend-specific dependencies. The plugin connects to the framework's core through abstract interfaces, keeping the concrete spatial geometry types and libraries fully isolated. As illustrated in Figure 1, the Sedona and JTS dependencies remain entirely within the spatial plugin. Users opt in with a single declaration: `.withPlugin(Spatial.plugin())`.

2.5 Performance Comparison

We run three Wayang jobs varying the dataset, indexing, and cluster size to show that the best spatial backend shifts with the workload.

Setup. All experiments run on AMD EPYC ROME nodes (8 cores, 8 GB RAM, NVMe SSD) with Sedona 1.6.1 on a 4-node cluster for Jobs 1–2 (scaled to 8 nodes for Job 3) and PostGIS 3.4 on PostgreSQL 16. We use two spatial join workloads. The first joins OpenStreetMap Parks (~44k polygons) with Lakes (~140k polygons) using the `ST_Contains` predicate. The second joins synthetically generated bounding boxes (100k × 1M) using `ST_Intersects`.

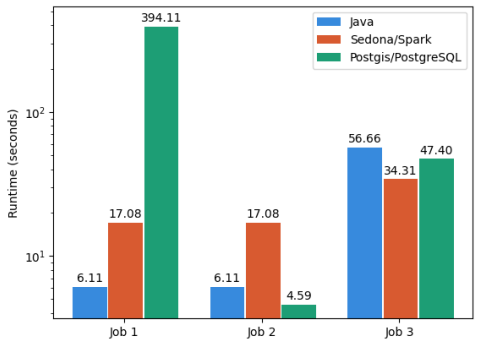


Figure 2: Execution time (log scale) across backends for three jobs varying in dataset, indexing, and cluster size. A different backend is the fastest in each job.

Jobs and Results. Figure 2 shows the runtime for each backend across three jobs. Job 1 executes the Parks–Lakes containment join without a spatial index on the PostgreSQL tables, simulating a cold start where a user queries freshly ingested data. Without an index, PostGIS is by far the slowest (394.11 s), while Java/JTS completes in just 6.11 s. This makes the Java backend ideal for quick, local exploration during early development, allowing users to prioritize immediate insights over the overhead of infrastructure setup or index tuning. Job 2 repeats the same join *with* a GiST index: PostGIS drops to 4.59 s, now the fastest, while Java and Sedona remain unchanged, showing that once an analysis matures, PostGIS with proper indexing is the strongest single-node choice. Job 3 scales up to the 100k × 1M box intersection join with an 8-node Spark cluster. Here, Sedona's distributed execution pays off (34.31 s), outperforming both PostGIS (47.40 s) and Java (56.66 s), making it the preferred choice when scaling to large datasets. These results reflect a typical data science workflow (from laptop prototyping to indexed single-node analysis to cluster deployment), where Wayang Spatial enables each transition without a code rewrite.

3 Demonstration

We demonstrate Wayang Spatial through an application that performs spatial analytics over apartment listings in Berlin. The data spans different geometry types, e.g., points for exact apartment locations and polygons for postal-code boundaries. Beyond these listings, we provide real spatial datasets from UCR-STAR [8], and attendees are welcome to load their own.

Attendees interact through a web-based GUI (Figure 3) with three views: a drag-and-drop *Query Builder* for composing pipelines and assigning operators to backends, a *Query Results* view for inspecting the output, and a *Job Inspector* for performance comparisons. The GUI submits plans to a remote cluster where Wayang Spatial executes them across the three backends.

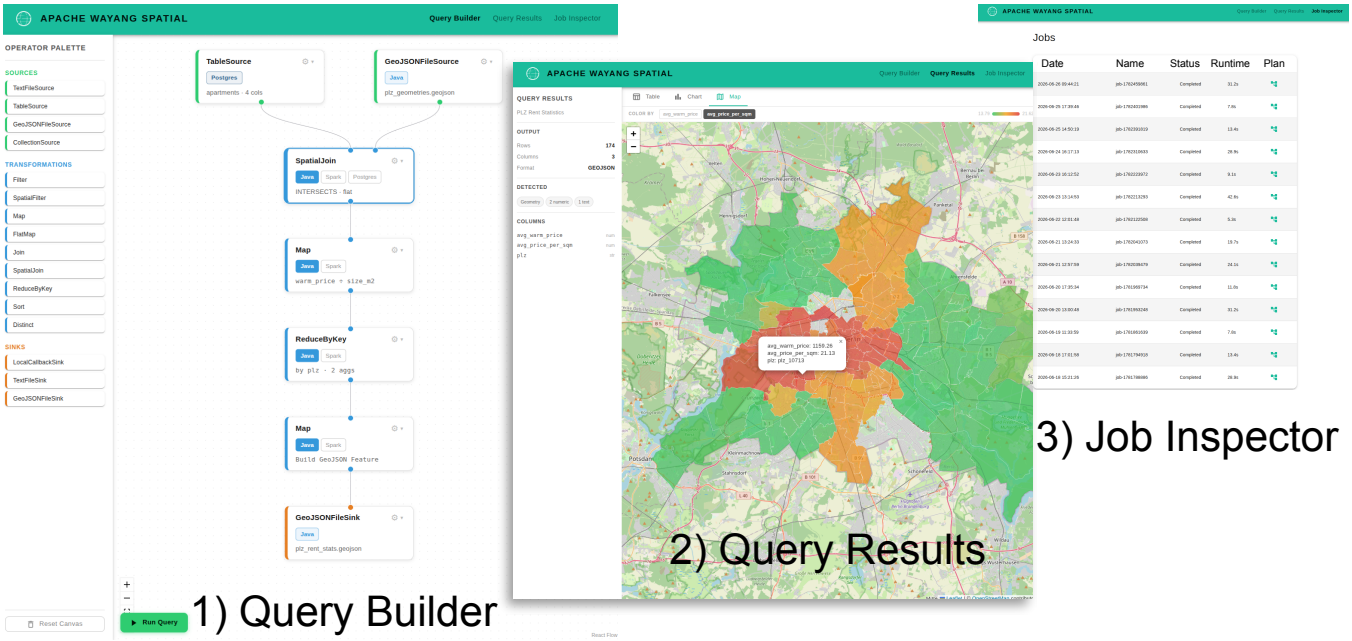


Figure 3: The demonstration GUI: (1) the drag-and-drop Query Builder, where each operator is assigned to a backend; (2) the map-based Query Results view (© OpenStreetMap contributors); and (3) the Job Inspector, listing executed jobs with runtimes.

Query Builder. The Query Builder (Figure 3, left) is a visual, code-free interface. An operator palette groups operators into sources, transformations, and sinks, including the spatial operators `SpatialFilter` and `SpatialJoin` alongside Wayang's existing operators, such as `Map` and `ReduceByKey`. Attendees drag operators onto the canvas, connect them into a dataflow, and assign nodes to backends with one click. Since nodes are assigned independently, the pipeline can run on a single backend or be split across several. Wayang's optimizer already supports the new spatial operators with default cost estimates. However, the demo exposes operator-to-platform assignment decisions to attendees, letting them explore performance trade-offs. Refining the optimizer's spatial cost models is future work. Every node expands to reveal its controls, allowing attendees to select a predicate for spatial joins and filters (e.g., `INTERSECTS`) and a predefined function for `Map` and `ReduceByKey` (e.g., an average). The default pipeline computes the average rent and rent per square meter per postal district in Berlin across two backends. A `TableSource` on PostGIS and a `GeoJSONFileSource` on Java load the listings and postal-code boundaries. A `SpatialJoin` on Java assigns each listing to its enclosing district, so Wayang transfers the listings from PostGIS to Java. A `Map` then derives each listing's rent per square meter, a `ReduceByKey` groups by district and computes the two averages, a second `Map` builds a GeoJSON feature per district with its geometry, and a `GeoJSONFileSink` writes the result. Pressing *Run Query* submits the plan to the cluster.

Query Results. The Query Results view (Figure 3, center) presents the output as a table, a chart, or a map. For the default pipeline, the map renders a choropleth of average rent per square meter over Berlin postal districts, while a control at the top allows attendees to color the districts by the average rent. Districts with no listings are grayed out. Clicking a district opens a pop-up with its aggregates, and a side panel reports the schema and size of the result.

Job Inspector. The Job Inspector (Figure 3, right) keeps a time-ordered history of executed jobs with their status, end-to-end runtime, and corresponding plan. By re-running a pipeline under different backend assignments and comparing the resulting runtimes, attendees see how operator placement affects end-to-end cost and confirm that no single backend is best for every spatial workload.

References

[1] 2024. JTS Topology Suite. https://github.com/locationtech/jts.
[2] 2024. PostGIS. https://postgis.net/.
[3] Kaustubh Beedkar et al. 2025. Apache Wayang in Action: Enabling Data Systems Integration via a Unified Data Analytics Framework. In *Proc. SIGMOD-Companion*. 35–38. doi:10.1145/3722212.3725081
[4] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavriilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. 2023. Apache Wayang: A Unified Data Analytics Framework. *SIGMOD Record* 52, 3 (2023), 30–35. doi:10.1145/3631504.3631510
[5] Gereon Dusella, Haralampos Gavriilidis, Volker Markl, and Eleni Tzirita Zacharatou. 2026. Benchmarking Spatial Operations over Heterogeneous Data: The Case of Zonal Statistics. In *Proc. DBTest@SIGMOD*. 16–22. doi:10.1145/3810991.3811631
[6] Gereon Dusella, Haralampos Gavriilidis, Laert Nuhu, Volker Markl, and Eleni Tzirita Zacharatou. 2024. Multi-Backend Zonal Statistics Execution with Raven. In *Proc. SIGMOD-Companion*. 532–535. doi:10.1145/3626246.3654730
[7] Ahmed Eldawy et al. 2021. Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In *CIKM*. 3796–3807. doi:10.1145/3459637.3481897
[8] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. 2019. UCR-STAR: the UCR spatio-temporal active repository. *SIGSPATIAL Special* 11, 2 (2019), 34–40.
[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] Aske Wachs and Eleni Tzirita Zacharatou. 2024. Analysis of Geospatial Data Loading. In *Proc. DBTest@SIGMOD*. 36–42. doi:10.1145/3662165.3662761
[11] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial Data Management in Apache Spark: The GeoSpark Perspective and Beyond. *GeoInformatica* 23, 1 (2019), 37–78.