PostgreSQL Sharding for the Gaia Use Case

Issue/Rev. No.: 1.1
Date: 7/1/2026
Author: Massimo Costantini
Affiliation: INAF - Istituto Nazionale di Astrofisica
Osservatorio Astronomico di Trieste
Gruppo IA2
Approved by: Cristina Knapic

Abstract

The Gaia space mission produces one of the largest astronomical catalogs ever created, with tens to hundreds of billions of rows distributed across multiple data releases. For evaluation purposes, a representative subset of the Gaia data was provided, consisting of three core tables (AstroElementary, CompleteSource and CrossMatch) totaling approximately 7 billion rows.

This Technical Report describes the design and implementation of a distributed database architecture based on PostgreSQL, the Citus extension and the pg_dog query router. The system aims to explore sharding strategies, optimize query performance and provide insights into scalability for massive scientific datasets.

The project was developed within Work Package 4 of Spoke 3 of the Italian National Center for High Performance Computing, Big Data and Quantum Computing (ICSC), established under the Italian National Recovery and Resilience Plan (PNRR), with the broader objective of strengthening national computational and data infrastructures to support multidisciplinary scientific innovation.

Contents

1 Introduction

The *Gaia* [1] mission, operated by *ESA* [2], is generating a detailed three-dimensional map of the *Milky Way*. Its data releases include hundreds of billions of measurements. Handling such a dataset requires careful database design, especially for interactive scientific queries.

This report describes the work carried out to ingest and shard a 7-billion-row subset of *Gaia data* using *PostgreSQL* [3], *Citus* [4] and *pg_dog* [5], exploring sharding strategies, optimizing query performance, and providing insights into scalability for massive scientific datasets.

2 Dataset Overview

The provided dataset consists of three tables:

- **AstroElementary** (~2.4 billion rows, 18 columns)
- **CompleteSource** (~150 million rows, 26 columns)
- **CrossMatch** (~5.5 billion rows, 16 columns)

All columns are potentially queryable. The data volume exceeds 7 billion rows in total and represents a realistic **Big Data** challenge for relational storage and querying.

3 Motivation for Sharding

A single *PostgreSQL* instance is not efficient for multi-billion row workloads:

- Sequential scans become prohibitive
- Index maintenance is costly
- Queries with joins and spatial conditions are slow to evaluate

Sharding distributes data into smaller logical units, reducing query scope and improving parallelism. In this setup, multiple sharding keys are exploited to optimize different classes of queries.

4 Architecture Design

Citus extends *PostgreSQL* with distributed table capabilities. In this configuration, each node runs *PostgreSQL* and *Citus* in single-node mode, but with different *sharding keys* applied to the same set of tables.

In addition to identifiers, spatial partitioning is supported through the *HEALPix*[1] *scheme*, which maps sky coordinates to equal-area cells on the sphere.

Sharding strategies by key:

- **dbs1**: sharded by *source_id* (optimizes queries keyed by source identifiers)
- **dbs2**: sharded by *transit_id* (optimizes queries keyed by transit identifiers)
- **dbs3**: sharded by *healpix_id* (spatial key for *CrossMatch*), while the other two tables use *source_id* and *transit_id*

A fourth server hosts *pg_dog*, a SQL router that inspects WHERE clauses and forwards queries to the most suitable node. Users remain unaware of sharding strategies: queries are written in standard SQL.

In this architecture, *Citus* handles logical sharding and distributed execution inside each database, whereas the complete dataset is intentionally replicated across multiple clusters, each optimized with a different distribution key to support distinct query patterns.

[1] *HEALPix (Hierarchical Equal Area isoLatitude Pixelization)* is an astronomical framework that partitions the sphere into equal-area cells, widely used for indexing and querying sky coordinates.

5 Data Ingestion and Indexing

Because ingestion runs for weeks, an artificial *ingest_id column* (bigserial) was introduced. This enables incremental dumps and reloads using filters such as: "*WHERE ingest_id BETWEEN ...".*

BRIN (Block Range Index)[2] *structures* were built on *ingest_id* to accelerate bulk exports. *BRIN indexes* store value ranges per block rather than row-by-row pointers, making them efficient and compact for sequential data.

In addition to distribution, tables can be indexed locally within each shard. *B-tree*[3] and *BRIN indexes* allow selective queries to benefit from both **partition pruning** and **fast lookup** within shards.

Spherical indexing using the *pg_sphere*[4] extension was introduced in preliminary tests to evaluate geometric query performance.

[2] *BRIN (Block Range Index)* is a *PostgreSQL index type* that stores summaries of block ranges, making it very space-efficient and suitable for large, sequentially ordered data such as time series or spatial coordinates.

[3] *B-tree* is the default *PostgreSQL index type*, optimized for fast lookups, equality and range queries, offering balanced performance across most workloads.

[4] *pg_sphere* is a *PostgreSQL* extension for spherical geometry.

6 Performance Considerations

The performance of a sharded *PostgreSQL* architecture depends critically on the alignment between data distribution, query patterns and routing logic. In the *Gaia* use case, query efficiency is primarily determined by the ability to localize execution to a minimal subset of shards, thereby reducing I/O, network traffic and distributed coordination overhead.

This section analyzes the main factors influencing query performance in the proposed architecture, with particular attention to sharding key selection, *data colocation*[5] and the impact of spatial filtering through *HEALPix-based partitioning*.

The discussion highlights both the conditions under which the system achieves near–single-node efficiency and the scenarios that introduce additional costs due to redistribution or remote access:

- Queries using the sharding key benefit from *data colocation* and execute efficiently on a single shard
- Spatial queries mapped to *healpix ranges* can be routed to dbs3, avoiding full-table scans
- Join queries are efficient only when joins align with sharding keys, otherwise they require re-partitioning or remote lookups
- Incremental ingestion with *BRIN indexes* reduces downtime and accelerates export/import cycles

Detailed performance measurements for spherical queries and the evaluation of logical sharding with *Citus*, are presented.

[5] *Data colocation* is the property whereby all data relevant to a query are stored on the same shard, allowing local execution without cross-shard joins or remote lookups.

7 Challenges and Trade-offs

While the proposed architecture demonstrates significant performance benefits for targeted query workloads, it also introduces a set of structural trade-offs that must be carefully evaluated. The decision to replicate the full dataset across multiple clusters, each optimized for a specific sharding key, shifts complexity from query execution to system design, storage management and operational maintenance.

This section discusses the principal challenges encountered during implementation, including storage overhead, query routing complexity and operational scalability.

Rather than representing implementation shortcomings, these issues reflect inherent trade-offs in designing a flexible, high-performance sharded system for multi-billion-row scientific datasets:

- **Data duplication:** all nodes host the full dataset, tripling storage requirements
- **Routing complexity:** *pg_dog* must parse queries reliably to detect the relevant sharding key
- **Load balance:** some queries may still fall back to less optimal nodes if multiple predicates are used
- **Maintenance:** index creation and statistics gathering are non-trivial at multi-billion row scale

8 Architectural Choices

The retrieval experiments were conducted on the final sharded deployment of the **gaia.completesource** table, using a logically distributed *PostgreSQL* deployment based on *Citus*.

In this configuration, the dataset is horizontally partitioned using a hierarchical *HEALPix-based strategy*: a coarse *HEALPix resolution* (norder = 6) is employed as the distribution key to shard the table across workers, while a finer resolution (norder = 12) is used for spatial indexing within each shard, 64 shards in total.

Unlike the preliminary tests performed using *pg_sphere* on a non-sharded database, all spatial queries are expressed in terms of precomputed *HEALPix* pixel identifiers. Before query execution, the target sky region is converted outside the database into a list of *HEALPix* pixels using the *healpy*[6] library:

- hp.query_polygon()
- hp.query_disc()
- hp.query_strip()

These computations are extremely fast (milliseconds).

Depending on the query geometry, this conversion relies on polygon, disc or declination-strip operators, and is performed at both *HEALPix resolutions*. The resulting pixel lists are then passed to SQL queries as array parameters.

The coarse-resolution *HEALPix identifiers* (hpx6) enable *Citus* to restrict execution to the subset of shards intersecting the sky region of interest, while the fine-resolution identifiers (hpx12) are used for precise filtering within each shard through a B-tree index.

This approach avoids on-the-fly trigonometric evaluations and allows the *PostgreSQL* planner to rely exclusively on integer comparisons and indexed lookups.

Three main classes of spatial queries were benchmarked:

- Area searches over arbitrary sky regions (searching a sky region defined by a list of *HEALPix* pixels):

```
SELECT count(*)

FROM gaia.completesource

WHERE hpx6 = ANY(ARRAY[pix6])

AND hpx12 = ANY(ARRAY[pix12]);
```

[6] *healpy* is a *Python* package to handle pixelated data on the sphere.

Precalculated values:

```
pix6 = hp.query_polygon(64, vertices, nest=True);

pix12 = hp.query_polygon(4096, vertices, nest=True);
```

- Cone searches within a given angular radius (searching within an angular radius):

```
SELECT count(*)

FROM gaia.completesource

WHERE hpx6 = ANY(ARRAY[pix6])

AND hpx12 = ANY(ARRAY[pix12]);
```

Precalculated values:

```
vec = hp.ang2vec(theta, phi);

pix6 = hp.query_disc(64, vec, radius, nest=True);

pix12 = hp.query_disc(4096, vec, radius, nest=True);
```

- Rectangular selections in right ascension and declination (searching inside a rectangular region, for example: alpha ∈ [100°,150°], delta ∈ [–20°, +20°]):

```
SELECT count(*)

FROM gaia.completesource

WHERE hpx6 = ANY(ARRAY[pix6])

AND alpha BETWEEN 100 AND 150 AND delta BETWEEN -20 AND +20;
```

Precalculated value:

```
pix6 = hp.query_strip(64, theta_min, theta_max, nest=True);
```

The nest=True option guarantees the hierarchical structure of *HEALPix*, ensuring that all order-12 child of an order-6 parent are stored in the same *Citus* shard.

In all cases, query execution times were reduced to the order of a few minutes, despite operating on a multi-billion-row astrometric catalogue.

Cone searches, implemented by converting the spherical cap into *HEALPix* discs, benefit from accurate shard pruning and precise boundary filtering. Similarly, extended sky regions defined by polygons or declination bands are efficiently mapped to *HEALPix* pixel lists without requiring any geometric computation at query time.

9 Retrieval Results

Overall, the experiments confirm that the combination of hierarchical *HEALPix* indexing and distributed query execution enables efficient and scalable retrieval on large astrometric datasets. Compared to earlier non-sharded implementations based on dynamic spherical predicates, the optimized approach achieves several orders of magnitude improvement while preserving correctness and flexibility.

These results demonstrate that *HEALPix* based spatial queries naturally scale with sharding and provide robust performance even for complex sky selections:

Query type	Predicate / Description	Spatial method	Rows returned	Execution time
Single object lookup	`WHERE sourceId = ...`	Primary key lookup	1	<1 ms
Rectangular search	`alpha BETWEEN ... AND delta BETWEEN ...` with shard pruning	HEALPix strip (hp6)	~10	~3 min
Area search	`hpx6 = ANY(pix6) AND hpx12 = ANY(pix12)`	HEALPix polygon (hp6 + hp12)	10^6–10^7	~2 min
Cone search	HEALPix disc selection around target coordinates	HEALPix disc (hp6 + hp12)	~10	~2 min
Wide declination band	`hpx6 = ANY(pix6)`	HEALPix strip (hp6)	$> 10^6$	~3 min

Table 1: Summary of retrieval performance on the sharded gaia.completesource table. Spatial queries are expressed exclusively through precomputed *HEALPix* pixel identifiers.
The coarse *HEALPix* level (norder = 6) is used for shard selection in *Citus*, while the fine level (norder = 12) enables indexed filtering within each shard.
All *HEALPix* pixel lists are generated outside the database using the *healpy* library.

10 Conclusion and Future Work

By combining *PostgreSQL*, *Citus* and *pg_dog*, an experimental environment was established to explore sharding strategies for the *Gaia dataset*. This architecture demonstrates that sharding by different keys (*source_id*, *transit_id* and *healpix_id*) can accelerate distinct query patterns.

Future work will focus on consolidating the proposed architecture and extending it toward operational scientific use.

A first area of investigation concerns **incremental updates** and new *Gaia* data releases, including strategies for efficient ingestion, index maintenance and statistics refresh in a multi-billion-row distributed environment, while minimizing downtime and rebalancing costs.

Further developments will address **query routing and optimization**, extending *pg_dog*'s ability to reason about complex predicates involving multiple filtering dimensions (e.g. spatial and identifier-based constraints) and to select the most appropriate shard layout accordingly.

From an infrastructure perspective, alternative deployment models will be explored, including **selective replication and hybrid sharding strategies**, in order to reduce storage overhead while preserving query specialization and performance.

Finally, the system will be evaluated using real scientific workloads expressed in ADQL (Astronomical Data Query Language), with the goal of assessing usability, performance and scalability under realistic access patterns representative of the *Gaia* user community.

11 References

[1] *Gaia* satellite has made more than three trillion observations of two billion stars and other objects throughout our *Milky Way* galaxy and beyond:
https://www.esa.int/Science_Exploration/Space_Science/Gaia

[2] *ESA* is the European Space Agency, Europe's gateway to space. Its mission is to shape the development of Europe's space capability and ensure that investment in space continues to deliver benefits to the citizens of Europe and the world:
https://www.esa.int

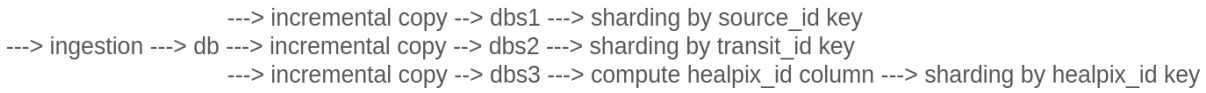
[3] *PostgreSQL* is a powerful, open source object-relational database system:
https://www.postgresql.org

[4] *Citus* is a *PostgreSQL* extension that transforms *PostgreSQL* into a distributed database, so you can achieve high performance at any scale:
https://github.com/citusdata/citus

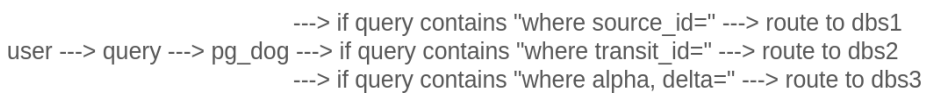
[5] *pg_dog* is the missing routing layer for *PostgreSQL*. It moves data between nodes and directs query traffic, transparently to the application:
https://pgdog.dev

12 Appendices (Schema Overview)

The following schemas provide an overview of the data ingestion and query-routing architecture. They illustrate how ingested data is replicated across multiple databases using different sharding keys and how incoming queries are routed to the appropriate database based on their filtering predicates:



Schema 1: Ingestion is copied into three databases, each sharded by a different key.



Schema 2: Queries are routed by *pg_dog* to different databases, each based on the filtering key.