



Postgres Analytics Accelerator - PGAA

Release 1.7.0

EnterpriseDB Corporation

Mar 23, 2026

1	Get started	3
1.1	Using PGAA	3
1.2	Replicating with PGD	3
1.3	Performance & optimization	4
1.4	Reference	4
2	Analytics Accelerator 1.7 release notes	5
2.1	Features	5
2.2	Bug fixes	5
2.3	Infrastructure & other changes	5
2.4	Deprecations	6
3	Analytics Accelerator	7
3.1	Analytics Accelerator quickstart guide	7
3.1.1	Prerequisites	8
3.1.2	Downloading and installing the package on your Postgres instance	8
3.1.3	Configuring a storage location	8
3.1.4	Creating analytical tables	9
3.1.5	Running an analytical query	9
3.1.6	Analyzing the query plan	10
3.1.7	Conclusion	11
3.2	Analytics Accelerator architecture	11
3.2.1	Architecture overview	11
3.2.2	Core components	11
3.2.3	Query execution modes	12
3.3	Concepts	12
3.3.1	Storage and table formats	12
3.3.2	Catalogs	13
3.3.3	Query execution	13
3.3.4	Caching	13
3.4	Known issues	14
3.5	Analytics Accelerator compatibility	14
3.5.1	Postgres compatibility	14
3.5.2	Platform compatibility	14
3.5.3	Spark	14
3.5.4	PGD	15
3.5.5	Hybrid Manager	15

3.5.6	Supported storage formats and providers	15
4	Installing Analytics Accelerator	17
4.1	Downloading and installing the package	17
4.2	Enabling the extension	18
4.3	Verifying the installation	18
5	Configuring storage locations	19
5.1	Creating a storage location	19
5.2	Types of buckets	19
5.3	Authentication	20
5.3.1	Static credentials	20
5.3.2	Environment variables	20
5.3.3	IAM roles	20
5.4	Testing storage access	21
6	Reading tables in object storage	23
6.1	Configuring Postgres to read from object storage	24
6.1.1	Creating a storage location	24
6.1.2	Mapping the table to Postgres	24
6.1.3	Querying and verifying acceleration	24
6.2	Limitations	25
7	Integrating with Iceberg catalogs	27
7.1	Adding a catalog connection	27
7.1.1	REST catalog	28
7.1.2	AWS S3 Tables	28
7.1.3	Verifying the catalog configuration	29
7.2	Registering catalog tables	29
7.2.1	Attaching a catalog	29
7.2.2	Importing a catalog	30
7.2.3	Creating catalog-managed tables	30
8	Writing to object storage	33
8.1	Using a standalone storage location	33
8.1.1	Creating a storage location	34
8.1.2	Creating the table with CTAS	34
8.2	Using an Iceberg catalog	34
8.2.1	Configuring a catalog connection	35
8.2.2	Creating the catalog-managed CTAS table	35
8.3	Limitations of CTAS	36
9	Replicating with PGD	37
9.1	Understanding table types	37
9.2	Tiered storage strategies	38
9.2.1	Implementing tiered tables	38
9.2.2	Replicating to analytics	38
9.2.3	Offloading to analytics	38
9.3	Metadata management	38
9.3.1	Getting started	39
9.3.2	Implementing tiered tables	40
9.3.3	Replicating to analytics	44
9.3.4	Offloading to analytics	46
10	Accelerating with Spark	49

10.1	Choosing your executor engine	49
10.2	Distributed execution options	49
10.2.1	Distributed Spark execution	50
10.2.2	GPU-accelerated Spark with NVIDIA RAPIDS	54
11	Monitoring and maintaining analytical tables	59
11.1	Analyzing table statistics	59
11.2	Optimizing table performance	60
11.2.1	Performing table maintenance with <code>pgaa.launch_task</code>	60
11.2.2	Advanced Spark-based maintenance	61
12	Optimizing query performance	63
12.1	Understanding scan types	63
12.2	Analyzing your query plan	63
12.3	Managing DirectScan fallbacks	64
12.4	Configuring compute pushdowns	64
13	Reference	65
13.1	Configuration parameters	65
13.1.1	Executor engine	65
13.1.2	Query behavior	66
13.1.3	Seafowl	68
13.1.4	Catalog synchronization	69
13.1.5	Caching	70
13.1.6	Postgres Distributed (PGD) integration	70
13.1.7	DataFusion-inherited configurations	70
13.1.8	Executor engine	71
13.1.9	Query behavior	71
13.1.10	Seafowl	72
13.1.11	Catalog synchronization	73
13.1.12	Caching	73
13.1.13	DataFusion-inherited configurations	73
13.2	Functions	73
13.2.1	Table functions	73
13.2.2	Catalog functions	75
13.2.3	Spark functions	81
13.2.4	Background task functions	83
13.2.5	Postgres Distributed (PGD) integration functions	86
13.2.6	Utility functions	90
13.3	Table options	91
13.3.1	PGAA table options	91
13.3.2	PGD table options	92
13.3.3	Managing access methods	93
13.4	Data types and definitions	93
13.4.1	Data types	93
13.4.2	Administrative enums and statuses	95
13.5	Benchmark datasets	95

Postgres Analytics Accelerator (PGAA) is a high-performance extension that enables Postgres to query large-scale data stored in open table formats like Delta Lake, Apache Iceberg, and Parquet. By offloading heavy analytical queries to a vectorized execution engine, PGAA bridges the gap between operational databases and data lakes.

- *Analytics Accelerator compatibility* : Check supported PostgreSQL versions, operating systems, and other requirements.
- *Analytics Accelerator architecture* : Understand the core architecture and how the vectorized engine works.
- *Concepts* : Understand the fundamental principles of vectorized execution, data lake integration, and DirectScan.
- *Analytics Accelerator quickstart guide* : Install PGAA, create a storage location and read table from our sample benchmark datasets.

1.1 Using PGAA

- *Installing Analytics Accelerator* : Step-by-step instructions for installing the extension and enabling the Seafoal background worker.
- *Configuring storage locations* : How to securely connect PGAA to AWS S3, GCS, and Azure Blob storage.
- *Reading tables in object storage* : Connect directly to S3, GCS, or Azure Blob Storage to query Parquet, Delta, or Iceberg files via a PGFS storage location.
- *Integrating with Iceberg catalogs* : Integrate with external Iceberg REST catalogs to manage table metadata.
- *Writing to object storage* : Use `CREATE TABLE AS SELECT (CTAS)` to export Postgres data into optimized lakehouse formats in your object store.

1.2 Replicating with PGD

- *Implementing tiered tables* : Combine PGD AutoPartition and PGAA to create an automated data lifecycle. Move older partitions to object storage while keeping recent data in Postgres tables.
- *Replicating to analytics* : Convert standard heap tables into HTAP tables. Use continuous logical replication to maintain a real-time analytical copy of your transactional data in the data lake.
- *Offloading to analytics* : Perform surgical storage management by manually moving entire HTAP tables to the cold tier, truncating local data to reclaim disk space immediately.

1.3 Performance & optimization

- *Accelerate with Spark* : Offload massive datasets and complex distributed joins to a remote Spark cluster via Spark Connect. PGAA offers two integration modes depending on your performance requirements:
- *Distributed Spark execution* Leverage a remote Spark cluster for high-concurrency analytical queries and distributed processing.
- *GPU-accelerated Spark with NVIDIA RAPIDS* : Integrate with the NVIDIA RAPIDS Accelerator for Apache Spark to leverage GPU acceleration.
- *Monitoring and maintaining analytical tables* : Audit storage utilization, monitor table health, and perform table maintenance tasks for PGAA-managed tables.
- *Optimizing query performance* : Maximize query speeds by managing DirectScan execution, configuring compute pushdowns, and troubleshooting path fallbacks.

1.4 Reference

- *Configuration parameters* : The behavior of the PGAA extension is governed by Grand Unified Configuration (GUC) variables. These parameters allow you to switch executors, enable performance optimizations, and manage security credentials.
- *Functions* : PGAA introduces a suite of SQL functions for administrative tasks, such as mapping new tables, monitoring storage health, and launching maintenance background jobs.
- *Table options* : When mapping or creating analytical tables, specific options allow you to define how data is read from or written to your object store.
- *Data types and definitions* : PGAA maps native Postgres data types to optimized columnar formats in the data lake.
- *Benchmark datasets* : Access pre-configured schemas and data loading instructions for analytical datasets to baseline your performance.

Analytics Accelerator 1.7 release notes

Released: 5 March 2026

Analytics Accelerator 1.7 includes the following enhancements and bug fixes:

2.1 Features

- **Iceberg compaction maintenance:** Added support for compaction maintenance operations on Iceberg tables via the `pgaa.launch_task()` function. This workflow is handled natively by the PGAA background worker, removing the previous requirement for an external Spark Connect integration to execute compaction. This applies to both native PGAA tables and standard heap tables replicated with PGD.
- **PGD replication performance:** Achieved a 2–2.5x increase in replication TPS to PGD by migrating the Seafoam-based sync writer with to the PGAA engine.
- **Memory management:** Introduced the `pgaa.max_in_memory_mb` configuration parameter to allow granular control over size-related writer flush criteria.
- **Storage authentication:** Enabled support for static credentials for storage backends that do not support automated credential vending.

2.2 Bug fixes

- Resolved an issue where lingering transaction locations prevented the promotion of durable LSNs.
- Fixed errors occurring during date and duration subtraction within the DirectScan engine.

2.3 Infrastructure & other changes

- Added a synthetic pgbench-like replication benchmark, accessible via the `pgaa._bench_replication` SQL function.
- Merge end-to-end benchmarking scripts into a single unified utility, `pgaabench.sh`.

2.4 Deprecations

- The function `pgaa.execute_compaction()` is now deprecated and will be removed in a future release. Use `pgaa.launch_task()` for performing table maintenance.

The Postgres Analytics Accelerator (PGAA) transforms Postgres into a high-performance analytical engine. By integrating vectorized execution and open table formats, PGAA allows you to run complex OLAP queries directly on your data lake without leaving the Postgres ecosystem.

Use the following resources to understand the foundation of the accelerator and prepare your environment for production workloads.

- [*Analytics Accelerator quickstart guide*](#) : A step-by-step guide to install PGAA, create a storage location and read a table from our sample benchmark datasets.
- [*Analytics Accelerator architecture*](#) : A deep dive into the decoupled compute and storage model. Learn how the PGAA extension interacts with Seafoam (the vectorized engine), and object storage providers like S3, GCS, and Azure Data Lake Storage.
- [*Concepts*](#) : Explore the fundamental principles of PGAA, including vectorized execution, columnar storage, and the difference between the DirectScan and CompatScan execution paths.
- [*Analytics Accelerator compatibility*](#) : A detailed matrix of supported Postgres versions, operating systems, and cloud provider availability.
- [*Known issues*](#) : A transparent list of current limitations, unsupported cases, and active bugs we are currently addressing.

3.1 Analytics Accelerator quickstart guide

In this quickstart guide, you will:

1. Install the PGAA extension on a standalone Community Postgres instance.
2. Create a storage location pointing to sample benchmark datasets in object storage.
3. Run a complex analytical query and verify that Seafoam engine is working.
4. Perform an analysis of the query plan.

3.1.1 Prerequisites

- An environment running a Postgres instance. See *Analytics Accelerator compatibility* for a full list of supported platforms.
- An EDB access token.

3.1.2 Downloading and installing the package on your Postgres instance

Download the package from the EDB repository:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/standard/
↳setup.deb.sh" | sudo -E bash
sudo apt-get install -y postgresql-<version>-pgaa
```

Where <version> is your Postgres version.

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/standard/
↳setup.rpm.sh" | sudo -E bash
sudo dnf install -y postgresql-<version>-pgaa
```

Where <version> is your Postgres version.

Edit your `postgresql.conf` file to add `pgaa` to to your `shared_preload_libraries`, and enable the automatic management and startup of the Seafoam query engine via the Postgres background worker:

```
shared_preload_libraries = pgaa
pgaa.autostart_seafoam = on
```

After saving the changes, restart your Postgres service. Once restarted, log in to your Postgres instance and create the PGAA extension:

```
CREATE EXTENSION IF NOT EXISTS pgaa CASCADE;
```

Using `CASCADE` ensures that any required dependencies (like `PGFS`) are also created. Use `\dx` to verify that the extensions have been installed and check their versions.

3.1.3 Configuring a storage location

Create a storage location pointing to our public S3 bucket containing TPC-H benchmark data:

```
SELECT pgfs.create_storage_location(
    quickstart-sample-data,
    s3://beacon-analytics-demo-data-us-east-1-prod,
    {"skip_signature": "true", "region": "us-east-1"}
);
```

Verify that the storage location was created:

```
SELECT pgfs.list_storage_locations();
```

3.1.4 Creating analytical tables

Create the following analytical tables. These tables map directly to Delta Lake files in our sample *Benchmark datasets*.

```
CREATE TABLE supplier () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
↳data, pgaa.path = tpch_sf_1/supplier, pgaa.format = delta);
CREATE TABLE lineitem () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
↳data, pgaa.path = tpch_sf_1/lineitem, pgaa.format = delta);
CREATE TABLE orders () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
↳data, pgaa.path = tpch_sf_1/orders, pgaa.format = delta);
CREATE TABLE nation () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
↳data, pgaa.path = tpch_sf_1/nation, pgaa.format = delta);
```

3.1.5 Running an analytical query

This query identifies suppliers in Saudi Arabia who were the cause of delays in multi-supplier orders. This is a heavy analytical task involving large joins and multiple subqueries.

```
SELECT
    s_name,
    COUNT(*) AS numwait
FROM
    supplier,
    lineitem l1,
    orders,
    nation
WHERE
    s_suppkey = l1.l_suppkey
    AND o_orderkey = l1.l_orderkey
    AND o_orderstatus = F
    AND l1.l_receiptdate > l1.l_commitdate
    AND EXISTS (
        SELECT
        - FROM
            lineitem l2
        WHERE
            l2.l_orderkey = l1.l_orderkey
            AND l2.l_suppkey <> l1.l_suppkey
        )
    AND NOT EXISTS (
        SELECT
        - FROM
            lineitem l3
        WHERE
            l3.l_orderkey = l1.l_orderkey
            AND l3.l_suppkey <> l1.l_suppkey
            AND l3.l_receiptdate > l3.l_commitdate
        )
    AND s_nationkey = n_nationkey
    AND n_name = SAUDI ARABIA
GROUP BY
    s_name
ORDER BY
```

(continues on next page)

(continued from previous page)

```

numwait DESC,
s_name
LIMIT 100;

```

3.1.6 Analyzing the query plan

To see how PGAA accelerates this query, prepend `EXPLAIN` to the statement above and inspect the output:

```

QUERY PLAN
-----
SeafowlDirectScan: Logical Plan
  Limit:  skip=0, fetch=100
    Sort:  numwait DESC NULLS FIRST, supplier.s_name ASC NULLS LAST
      Projection:  supplier.s_name, count(Int64(1)) AS count(*) AS numwait
        Aggregate:  groupBy=[[supplier.s_name]], aggr=[[count(Int64(1))]]
          Filter:  supplier.s_suppkey = l1.1_suppkey AND orders.o_orderkey = l1.1_
→orderkey AND orders.o_orderstatus = Utf8("F") AND l1.1_receiptdate > l1.1_
→commitdate AND EXISTS (<subquery>) AND NOT EXISTS (<subquery>) AND supplier.s_
→nationkey = nation.n_nationkey AND nation.n_name = Utf8("SAUDI ARABIA")
            Subquery:
              Projection:  l2.1_orderkey, l2.1_partkey, l2.1_suppkey, l2.1_
→linenumber, l2.1_quantity, l2.1_extendedprice, l2.1_discount, l2.1_tax, l2.1_
→returnflag, l2.1_linestatus, l2.1_shipdate, l2.1_commitdate, l2.1_receiptdate, l2.1_
→shipinstruct, l2.1_shipmode, l2.1_comment
                Filter:  l2.1_orderkey = outer_ref(l1.1_orderkey) AND l2.1_suppkey !
→= outer_ref(l1.1_suppkey)
                  SubqueryAlias:  l2
                    TableScan:  lineitem
              Subquery:
                Projection:  l3.1_orderkey, l3.1_partkey, l3.1_suppkey, l3.1_
→linenumber, l3.1_quantity, l3.1_extendedprice, l3.1_discount, l3.1_tax, l3.1_
→returnflag, l3.1_linestatus, l3.1_shipdate, l3.1_commitdate, l3.1_receiptdate, l3.1_
→shipinstruct, l3.1_shipmode, l3.1_comment
                  Filter:  l3.1_orderkey = outer_ref(l1.1_orderkey) AND l3.1_suppkey !
→= outer_ref(l1.1_suppkey) AND l3.1_receiptdate > l3.1_commitdate
                    SubqueryAlias:  l3
                      TableScan:  lineitem
          Cross Join:
            Cross Join:
              Cross Join:
                TableScan:  supplier
                SubqueryAlias:  l1
                  TableScan:  lineitem
                TableScan:  orders
              TableScan:  nation
(24 rows)

```

The presence of `SeafowlDirectScan` at the top of the plan indicates that PGAA has taken full control of the query execution, bypassing the standard Postgres executor to run the query directly against the data lake.

Instead of pulling billions of rows into Postgres to process them, the entire logical plan—including joins, filters, and aggregations—is pushed down into the optimized analytical engine.

- **Subquery pushdown:** Notice the `EXISTS` and `NOT EXISTS` subqueries on the `lineitem` table. These are not being executed one-by-one by Postgres. Instead, they are part of the global logical plan being processed in parallel against the Parquet files.
- **Vectorized joins:** The `Cross Join` nodes combined with the `Filter` higher up indicate that the engine is performing hash joins or nested loop joins at the storage layer.
- **Predicate pushdown:** Filters like `n_name = Utf8("SAUDI ARABIA")` and `o_orderstatus = Utf8("F")` are applied during the initial scan. This means only the relevant data is read from the lake, significantly reducing I/O.
- **Final aggregation and sort:** The `Aggregate` and the `Sort` happen at the end of the pipeline before the final 100 rows are handed back to the Postgres client.

3.1.7 Conclusion

You have successfully installed PGAA, configured it to reach cloud-based object storage, and verified that complex SQL logic—including subqueries and joins—is being natively offloaded.

By leveraging Seafoam DirectScan, your Postgres instance can now act as a gateway to petabyte-scale data lakes, providing analytical performance that scales independently of your transactional compute.

3.2 Analytics Accelerator architecture

This architecture page provides a high-level technical overview of how the Analytics Accelerator (PGAA) bridges the gap between transactional Postgres and the modern data lake.

3.2.1 Architecture overview

The EDB Analytics Accelerator extends Postgres’s capabilities to create a unified platform for transactional (OLTP) and analytical (OLAP) workloads. The system operates on three fundamental principles: separation of compute and storage for elastic scaling, transparent query access across hot and cold data tiers, and native compatibility with open table formats.

3.2.2 Core components

Postgres Analytics Accelerator (PGAA)

The PGAA extension hooks into the Postgres query planner to identify queries that can be offloaded to the analytical engine. It manages the metadata for analytical tables and coordinates communication between the Postgres process and the query executors.

PGFS storage abstraction

The Postgres File System (PGFS) acts as a unified abstraction layer for object storage, enabling Postgres to interact seamlessly with AWS S3, GCS, and Azure Blob Storage while masking the complexities of individual cloud provider protocols. It optimizes performance and reliability by managing secure authentication through IAM roles and HMAC keys, ensuring network resilience with automated retries and connection pooling, and reducing metadata overhead by caching remote file locations and schemas.

Executor engine

Seafowl is the default, DataFusion-based query engine for PGAA, utilizing a stateless architecture that allows lakehouse nodes to scale compute up or down dynamically without the overhead of local data storage or movement. As a vectorized execution engine designed specifically for analytical workloads, it maximizes CPU cache efficiency by processing data in columnar batches rather than the traditional row-by-row approach used by standard Postgres.

The Postgres instance communicates with the Seafowl engine using Apache Arrow Flight. This is a high-performance RPC framework designed for large-scale data transfer. By using Arrow, PGAA can move columnar data between the database and the query engine with minimal serialization overhead.

While Seafowl is the default executor for high-speed SQL, PGAA also supports **Spark Connect**. This allows organizations to leverage Apache Spark for extremely complex ETL transformations while still accessing the data through the familiar Postgres wire protocol.

3.2.3 Query execution modes

When a query is executed against an analytics table, PGAA automatically selects the most efficient scanning mode based on the query structure and the location of the data:

DirectScan: The query is offloaded entirely to the Seafowl engine. Seafowl reads the remote files directly from object storage, processes all logic (filtering, aggregations, and joins), and returns only the final result set to Postgres. This mode provides maximum performance by utilizing vectorized execution at the source.

CompatScan: This mode is used when a query requires features not yet supported by the vectorized engine or when joining remote lakehouse data with local Postgres tables. In this scenario, Seafowl performs initial filtering and projection on the remote data, then streams those optimized results back to Postgres to complete the final join or complex processing locally.

3.3 Concepts

This concepts guide provides a technical foundation for the Analytics Accelerator (PGAA) architecture, organized by functional layers.

3.3.1 Storage and table formats

PGFS storage abstraction: The **Postgres File System (PGFS)** is a cloud-native storage layer that provides a unified interface between Postgres and object storage (S3, GCS, Azure). It handles authentication, network resilience, and the mapping of remote objects to local table structures.

Delta Lake: An open-source storage framework that brings ACID transactions to data lakes.

Transaction log: A central metadata record — such as the Delta Log in Delta Lake or snapshots in Iceberg — that tracks every change to the table, ensuring data consistency.

Time travel: The ability to query previous versions of data by referencing specific timestamps or log versions.

Schema evolution: The capability to change a table's schema (e.g., adding columns) over time without rewriting the entire dataset.

Apache Iceberg: A high-performance table format for massive analytical datasets.

Metadata: A multi-layered manifest system that tracks file-level statistics, enabling efficient data skipping.

Snapshots: Versions of a table at a specific point in time; queries always run against a consistent snapshot.

Partition evolution: The ability to change a table's partitioning strategy (e.g., changing from daily to monthly) without needing to rewrite existing data.

Parquet columnar storage: The underlying file format used by PGAA. Unlike standard Postgres which stores data in rows, Parquet stores data in columns.

Compression and encoding: Columnar storage allows for aggressive compression (like Snappy or Zstandard) and encoding (like Delta or Dictionary encoding) because similar data types are stored together, significantly reducing the storage footprint and I/O overhead.

3.3.2 Catalogs

Catalogs act as the central source of truth for table metadata. They allow different compute engines (Postgres, Spark, Trino) to discover tables, resolve schemas, and maintain transactional integrity across a shared data lake.

Iceberg REST: A standardized API used by PGAA to communicate with external catalogs. It allows EDB to integrate with any catalog service that supports the REST specification, such as Lakekeeper, Tabular, or Snowflake Polaris.

AWS S3 Tables: A managed catalog and storage service from AWS. PGAA integrates with S3 Tables to provide a serverless metadata layer, simplifying the management of Iceberg tables within the AWS ecosystem.

3.3.3 Query execution

Table Access Method (TAM) architecture: The internal Postgres API that allows PGAA to plug in a custom storage engine. This allows the Postgres planner to interact with remote object storage as if it were a native local table.

DirectScan: A performance optimization where a query is fully offloaded to the analytical executor engine (Seafowl or Spark). This occurs when a query only references analytics tables, allowing the vectorized engine to handle the entire operation and return the final result to the user.

CompatScan: A hybrid execution mode used to join local Postgres tables with remote analytics tables, or to process functions only supported by Postgres.

Partial pushdown: PGAA optimizes the query by pushing filters and projections down to the object storage layer, streaming only the necessary rows back to Postgres to complete the join or complex local operation.

Cost-based planning: The process where the Postgres optimizer uses statistics (like row counts and data distribution) from the lakehouse metadata to decide the most efficient way to execute a query, such as choosing between a Hash Join or a Nested Loop.

Vectorized execution in Seafowl: The core processing technology of the Seafowl engine. Instead of processing data row-by-row, it processes data in batches (vectors). This leverages modern CPU instructions (SIMD) to perform mathematical operations on multiple data points simultaneously, dramatically accelerating analytical queries.

3.3.4 Caching

To minimize the latency inherent in network-based object storage, PGAA implements a multi-tiered caching hierarchy. This ensures that repeated queries can execute at speeds comparable to local storage.

Object store cache: This layer caches the actual data blocks (Parquet files) retrieved from the data lake on the local SSDs of the lakehouse nodes. By keeping frequently accessed data “close” to the compute engine, PGAA avoids the overhead of repeated GET requests to the remote object store for the same data segments.

Metadata caching: Maintaining a local, high-speed copy of the data lake’s structural information—such as file manifests and schema definitions. This way the engine can perform partition pruning and file-level filtering instantaneously, identifying exactly which data to fetch without the high latency of repeatedly querying the remote storage provider.

Metadata caching stores the structure of the data lake tables—including file manifests, schema definitions, and partition locations. Since Iceberg and Delta Lake tables can consist of thousands of individual files, caching this “map” allows the query planner to skip irrelevant data almost instantaneously without reaching out to the storage provider.

Table stats caching: Maintaining a local record of physical table sizes to support the `pgaa.lakehouse_table_stats()` function. While the query planner retrieves fresh metadata from the storage

provider at execution time for plan accuracy, this cache allows the function to provide immediate visibility into the scale of data lake objects without requiring a live remote scan.

3.4 Known issues

These are the currently known issues and limitations identified in this Analytics Accelerator release. Where applicable, we have included workarounds to help you mitigate the impact of these issues. These issues are actively tracked and are planned for resolution in a future release.

- While the function `pgaa.launch_task()` supports multiple maintenance operations for Delta Lake (such as `z-order`, `vacuum`, and `purge`), it currently only supports the `compaction` operation for Iceberg tables. For other Iceberg maintenance routines, use `pgaa.spark_sql()`.
- The function `pgaa.execute_compaction()` does not support the parameter `settings`. Use `pgaa.spark_sql()` or `pgaa.launch_task()` to run Spark procedures directly in your cluster or to execute compaction for Iceberg tables.
- Integration with Apache Spark is supported for:
 - Read-only queries on Parquet files in S3-compatible object storage or a shared POSIX filesystem.
 - Read-only queries for Iceberg tables in Iceberg REST catalogs.
- GPU acceleration with Apache Spark is only supported for read-only queries on Parquet files in S3-compatible object storage or a shared POSIX filesystem.
- There is a known issue in the Iceberg Spark Library version 1.9.2 when using Spark to read from Iceberg tables. Under certain conditions, equality deletes may occasionally be skipped during concurrent executions. The current workaround is to disable the Spark application setting `spark.sql.iceberg.executor-cache.enabled` on your `spark-defaults.conf` file. Disabling this cache ensures data consistency by correctly processing all deletes, but it may have performance implications for high-volume read workloads.
- Integration with Amazon S3 Tables is currently not supported for PGD-integrated features (tiered tables, replication, and offloading). For PGD-managed lifecycles, use either a storage location or an Iceberg REST-compliant catalog.

3.5 Analytics Accelerator compatibility

3.5.1 Postgres compatibility

Postgres distribution	Postgres version
Postgres (PG)	16, 17, 18
EDB Postgres Extended Server (PGE)	16, 17, 18
EDB Postgres Advanced Server (EPAS)	16, 17, 18

3.5.2 Platform compatibility

Debian 11/12 or Ubuntu 22/24 on amd64 architectures, and RHEL 8/9 on both amd64 and arm64.

3.5.3 Spark

Apache Spark: Version 3.4 or later.

3.5.4 PGD

PGD version 6.1 or later.

3.5.5 Hybrid Manager

PGAA version 1.7 is included with Hybrid Manager Innovation Release 2026.3.

3.5.6 Supported storage formats and providers

Data Format	Operation	Object storage	Engine support
Iceberg	Read	Local, S3, GCP, Azure (via storage locations)	Seafoal
Iceberg	Read	Iceberg REST catalog	Seafoal Spark
Delta Lake	Read	Local, S3, GCP, Azure (via storage locations)	Seafoal
Parquet	Read	Local, S3, GCP, Azure (via storage locations)	Seafoal
Parquet	Read	Local, S3 (via storage locations)	Seafoal Spark Spark + RAPIDS
Iceberg	Write (via <i>CTAS</i>)	Local, S3, GCP, Azure (via storage locations)	Postgres engine via CompatScan
Iceberg	Write (via <i>CTAS</i>)	Iceberg REST catalog	Postgres engine via CompatScan
Delta Lake	Write (via <i>CTAS</i>)	Local, S3, GCP, Azure (via storage locations)	Postgres engine via CompatScan
Iceberg	Write (using PGD)	Local, S3, GCP, Azure (via storage locations)	Seafoal
Delta Lake	Write (using PGD)	Local, S3, GCP, Azure (via storage locations)	Seafoal
Iceberg	Write (using PGD)	Iceberg REST catalog	Postgres engine via CompatScan

Installing Analytics Accelerator

To use the Postgres Analytics Accelerator (PGAA), you must install the extension on your Postgres instance and enable the background worker for the query engine. See *Analytics Accelerator compatibility* for a full list of supported platforms.

4.1 Downloading and installing the package

Download the package from the EDB repository:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↳SUBSCRIPTION_PLAN/setup.deb.sh" | sudo -E bash
sudo apt-get install -y <postgresql-distribution>-pgaa
```

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↳SUBSCRIPTION_PLAN/setup.rpm.sh" | sudo -E bash
sudo dnf install -y <postgresql-distribution>-pgaa
```

Where

- `$EDB_SUBSCRIPTION_TOKEN` is the token you received when you registered for the EDB subscription.
- `$EDB_SUBSCRIPTION_PLAN` is either `standard` (for PGE or Community Postgres) or `enterprise` (for EPAS).
- `<postgresql-distribution>` specifies your Postgres distribution (`edb-postgresextended`, `edb-as`, or `postgres`), and version.

4.2 Enabling the extension

To initialize PGAA, you must update your `postgresql.conf` file. Adding `pgaa` to `shared_preload_libraries` allows Postgres to manage the life cycle of the Seafoam query engine as a background worker.

1. Edit `postgresql.conf` :

```
shared_preload_libraries = pgaa
pgaa.autostart_seafoam = on
```

1. Restart your Postgres service to load the new library.
2. Log in to your Postgres instance. Depending on whether this is a new installation or an update, run the appropriate command:

- For new installations:

```
CREATE EXTENSION IF NOT EXISTS pgaa CASCADE;
```

Using `CASCADE` ensures that required dependencies, such as the Postgres File System (PGFS), are automatically installed.

- For upgrading an existing installation:

```
ALTER EXTENSION pgaa UPDATE;
```

4.3 Verifying the installation

Confirm that the extension is active using the `\dx` meta-command or by calling the version function:

```
SELECT pgaa.pgaa_version();
```

Once installed, you can [Configuring storage locations](#) to grant PGAA access to your data lake, or integrate Postgres [Integrating with Iceberg catalogs](#) .

Examples:

Point to parquet files:

```
SELECT pgaa.test_storage_location (my-storage-location, false);
```

The function returns `NULL` if successful.

Configuring storage locations

You can access data in object storage either through a storage location or via an integrated [Integrating with Iceberg catalogs](#). A storage location establishes a bridge between your Postgres instance and your data lake.

Postgres Analytics Accelerator (PGAA) leverages the Postgres File System extension (PGFS) to define target buckets and manage the authentication required to interact with remote files.

5.1 Creating a storage location

Storage locations act as aliases for your cloud buckets. You can create a storage location with the `pgfs.create_storage_location()` function using the following syntax:

```
select pgfs.create_storage_location(name=>storage_location_name,  
                                   url=>protocol://path,  
                                   options => {},  
                                   credentials => {});
```

For a comprehensive list of configuration parameters and supported storage providers, see [Creating a PGFS storage location](#).

5.2 Types of buckets

The primary difference between public and private buckets lies in how the engine handles request signing and authorization:

- **Private buckets:** Used for most production data. Accessing them requires valid credentials (IAM roles, static credentials, or environment variables) to sign every request sent by the PGFS extension. The data is only visible to the specific PGAA instance or authorized users.
- **Public buckets:** Accessible without authentication. These are commonly used for [Benchmark datasets](#) or public data repositories.

The following is an example of a storage location pointing to a public bucket:

```
SELECT pgfs.create_storage_location(  
    my_lake_data,  
    s3://your-bucket-name/path,  
    {"aws_skip_signature": "true"}  
);
```

5.3 Authentication

PGFS supports three primary methods for authorizing access to your object storage:

5.3.1 Static credentials

You can embed credentials directly within the storage location definition. By passing these as a second parameter to `pgfs.create_storage_location`, they are stored as a user mapping—a standard Postgres foreign data wrapper security feature. This ensures that the secrets are isolated and cannot be read by other database users.

While convenient for testing, ensure you manage these secrets securely. The parameters provided vary by storage providers. For example, for S3, you must provide either an access key id and a secret access key, or a session token:

```
SELECT pgfs.create_storage_location(  
    dev_lake,  
    s3://dev-bucket/,  
    {  
        "region": "us-east-1"  
    },  
    {  
        "access_key_id": "AKIA...",  
        "secret_access_key": "wJalr..."  
    }  
);
```

5.3.2 Environment variables

PGFS can inherit credentials directly from the operating system environment where Postgres is running. For example:

- Set the variable (OS level):

```
export GOOGLE_APPLICATION_CREDENTIALS=/var/run/gcs.json
```

- Create the storage location and omit the `credentials` parameter. PGFS will automatically check for search for the relevant environment keys to authorize the connection:

```
SELECT pgfs.create_storage_location(  
    edb_ai_example_images,  
    gs://my-company-ai-images);
```

5.3.3 IAM roles

If your Postgres instance is running on cloud infrastructure, use Instance Profiles or IAM Roles. This is the most secure method as it uses temporary, rotating credentials. Attach an IAM policy directly to the underlying virtual machine or container. PGFS automatically detects the instance metadata and uses these temporary, rotating credentials to sign requests.

```
SELECT pgfs.create_storage_location(  
    production_lake,  
    s3://my-analytics-bucket/,  
    {"region": "us-east-1"}  
);
```

5.4 Testing storage access

After creating a location, verify its existence with the following command:

```
SELECT pgfs.list_storage_locations();
```

Use the `pgaa.test_storage_location()` function to verify connectivity. The second parameter specifies whether to test for read access (`false`) or write access (`true`).

```
SELECT pgaa.test_storage_location (my-storage-location, false);
```

The function returns `NULL` if successful.

Reading tables in object storage

The Postgres Analytics Accelerator (PGAA) allows you to perform high-performance analytical queries from tables in object storage using Postgres’s SQL interface by establishing a direct connection to the data files. In this mode, Postgres interacts directly with the storage path to resolve the table structure, acting as the metadata coordinator by reading manifest files from your cloud bucket to determine how to scan and process the underlying data.

To query data at scale, PGAA supports both structured open table formats and raw file formats. These standards define how data is organized and how Postgres retrieves it:

- **Apache Iceberg:** A high-performance table format that uses a hierarchical metadata tree to track table state. It allows for “time travel” to previous snapshots and “hidden partitioning,” where Postgres can prune files without you needing to manually manage partition columns in the query.
- **Delta Lake:** An open-source storage layer that brings reliability to data lakes. It uses a transaction log (the `_delta_log` folder) to maintain a versioned history of the table, ensuring Postgres reads a consistent set of Parquet files even while other processes are writing to the same location.
- **Parquet:** The underlying columnar storage format used by Iceberg and Delta Lake data lakes. When used as a standalone format without a management layer, PGAA maps directly to the physical files or directories.

To query data lakes without a central catalog, the system establishes a link via a storage location, a set of pointers to the specific bucket or directory where your data files reside, supporting environments across AWS S3 (and S3-compatible API), Google Cloud Storage (GCS), Azure Blob Storage (including ADLS Gen2), and local filesystems.

The retrieval process follows these three steps:

- PGAA first scans the storage path for the latest metadata file or transaction log (in the case of Iceberg and Delta), or inspects the file headers to determine the schema and data types for raw Parquet.
- The system parses the metadata to create a virtual map of the relevant objects in S3, GCS, Azure, or local file system.
- The Seafoam analytical engine pulls the data directly from the object store, applying vectorized filters and aggregations at the source. The processed results are then streamed back into the Postgres executor, ensuring that only the relevant data is transferred.

6.1 Configuring Postgres to read from object storage

The process involves two steps: defining the physical storage access and mapping the remote metadata to a local definition.

6.1.1 Creating a storage location

You must create a PGFS storage location to tell Postgres where your bucket is and provide the necessary credentials. See *Configuring storage locations* for details. For example, with a public S3 bucket:

```
SELECT pgfs.create_storage_location(
    my-lake-data,
    s3://beacon-analytics-demo-data-us-east-1-prod,
    {"skip_signature": "true", "region": "us-east-1"}
);
```

6.1.2 Mapping the table to Postgres

Create a table using the PGAA access method that points to your storage location. For example:

```
CREATE TABLE iceberg_data ()
USING PGAA
WITH (
    pgaa.format = iceberg,
    pgaa.storage_location = my_lake_data,
    pgaa.path = path/to/data
);
```

Where:

- `pgaa.format` : Specifies the underlying table or file format. Supported values are 'iceberg', 'delta', or 'parquet'.
- `pgaa.storage_location` : The name of the storage location defined in the previous step.
- `pgaa.path` : The relative path from the root of the storage location to the directory containing the table's metadata and data files. For Iceberg, this should point to the folder containing the `/metadata` directory.

Note

If you leave the column definitions empty (`()`), PGAA will automatically discover the schema from the metadata files when you first query the table and will map columns by name to the fields found in the remote Iceberg/Delta metadata or Parquet headers.

6.1.3 Querying and verifying acceleration

Once the table is created, you can query it with standard SQL. To verify that PGAA's vectorized engine is actually handling the work, use `EXPLAIN`.

```
EXPLAIN (COSTS OFF) SELECT count(*), region FROM iceberg_data GROUP BY region;
```

If the engine is active, you will see a node called `SeafowlDirectScan`. This confirms the query has been offloaded to the vectorized accelerator rather than being processed by the standard Postgres executor.

6.2 Limitations

When using direct storage locations, you are responsible for managing the alignment between the local Postgres table definition and the remote data files. This manual approach introduces conflicts between the static schema stored in the Postgres system catalog and the actual, dynamic schema defined in the data lake.

If an external engine evolves the data lake schema, this can lead to query failures, silent data omission, or incorrect data mapping.

Working without a centralized catalog also limits the ability to discover new tables, manage concurrent writes from multiple engines, and affects query performance.

You must manually resolve these limitations as the data lake evolves:

- Use `ALTER TABLE` commands in Postgres to add, drop, or modify columns to match the remote schema.
- The most common fix is to drop and recreate the table using empty parentheses `()`. Upon the first query, PGAA performs a fresh discovery of the remote metadata.
- To prevent drift, explicitly define column names and types in your `CREATE TABLE` statement to match the object storage files exactly.

The most effective way to eliminate schema interference is to move from a standalone storage location to an external Iceberg catalog. By using a catalog, you shift the responsibility of schema management from the user to the system. See [Integrating with Iceberg catalogs](#) for details.

Integrating with Iceberg catalogs

A catalog service acts as the centralized “brain” of a data lake, decoupling physical storage from logical table definitions. By maintaining a single source of truth for metadata, schemas, and partitioning, it allows multiple independent engines—such as Postgres, Spark, and Trino—to share the same data without the risk of corruption or schema drift.

The Postgres Analytics Accelerator (PGAA) integrates with third-party catalogs to provide seamless data lake connectivity for analytical workloads within your Postgres deployment.

PGAA supports reading from and writing (using *Using an Iceberg catalog*) to Iceberg tables using the following catalog implementations:

- **REST catalogs:** Any service following the Apache Iceberg REST API, such as Lakekeeper, Tabular, Apache Polaris, or Project Nessie.
- **AWS S3 Tables:** A high-performance, AWS-native catalog designed for cloud-scale analytics.

Managing external Iceberg catalogs with PGAA involves two main phases:

1. *Adding a catalog connection* .
2. *Registering catalog tables* .

Note

When you use an external catalog, you don't need to create a storage location. A table is either linked to a storage location or a catalog.

7.1 Adding a catalog connection

Use the function `pgaa.add_catalog()` to register an external Iceberg catalog with your Postgres instance. The function creates a metadata link which tells Postgres where the catalog lives and how to log in. It also validates the catalog connection and credentials, and saves the configuration.

```
SELECT pgaa.add_catalog(
  catalog_name TEXT,
  catalog_type TEXT,
  catalog_options JSON
);
```

Specify one of the following catalog types:

- iceberg-rest : For Iceberg REST catalog (Lakekeeper, Snowflake Polaris, Tabular).
- iceberg-s3tables : For AWS S3 Tables.

7.1.1 REST catalog

For most REST catalog services, provide the endpoint and token details via a JSON object:

```
SELECT pgaa.add_catalog(
  my_iceberg_catalog,
  iceberg-rest,
  {
    "url": "https://your-catalog-endpoint.com",
    "warehouse_name": "your_warehouse_name",
    "warehouse": "your_warehouse_id",
    "token": "your_secret_auth_token"
  }
);
```

Some Iceberg REST Catalog providers, like Snowflake, require more advanced settings for authentication. PGAA supports full OAuth2 options to handle secure, rotating authentication. For example:

```
SELECT pgaa.add_catalog(
  my_iceberg_catalog,
  iceberg-rest,
  {
    "oauth2.grant_type": "client_credentials",
    "oauth2.client_id": "oauth-client-id",
    "oauth2.client_secret": "oauth-client-secret",
    "oauth2.token_uri": "https://EXAMPLE-polaristest.snowflakecomputing.com/
→polaris/api/catalog/v1/oauth/tokens",
    "oauth2.scope": "PRINCIPAL_ROLE:read_only"
  }
);
```

7.1.2 AWS S3 Tables

For a AWS S3 Tables catalog, specify the Amazon Resource Name (ARN) and the region:

```
SELECT pgaa.add_catalog(
  my_s3_catalog,
  iceberg-s3tables,
  {
    "arn": "arn:aws:s3tables:us-east-1:123456:bucket/analytics",
    "region": "us-east-1"
  }
);
```

(continues on next page)

(continued from previous page)

```
}  
);
```

7.1.3 Verifying the catalog configuration

After adding a catalog, use the following functions to verify the connection and explore available data:

- List registered catalogs:

```
SELECT * FROM pgaa.list_catalogs();
```

- Test connectivity and write permissions of your configuration. The function returns `NULL` if the connection is successful. For example:

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := FALSE);
```

- Browse the tables available within the external catalog before importing them:

```
SELECT pgaa.list_catalog_tables(my_iceberg_catalog);
```

For a complete list of functions supporting catalog registration and maintenance, see the [Catalog functions](#) reference section.

7.2 Registering catalog tables

Once you have established a catalog connection, you must register its metadata so Postgres can recognize and query the external data. PGAA provides three distinct paths for registration:

- *Attaching a catalog* for dynamic, automated synchronization.
- *Importing a catalog* for a static, one-time snapshot.
- *Creating catalog-managed tables* to register a single table with a high level of granularity.

7.2.1 Attaching a catalog

Attaching a catalog establishes a dynamic link between Postgres and the external Iceberg metadata service. Unlike a static import, this method ensures your Postgres environment stays synchronized with the data lake as it evolves.

When you attach a catalog, PGAA starts a background worker that continuously polls the catalog to detect new tables or schema evolution. PGAA honors the namespace hierarchy defined on the catalog side. If the attached catalog contains a table that shares the name with any existing tables on your database, the background worker will skip it to avoid overwriting it.

The synchronization process is managed by two primary configuration parameters:

- `pgaa.metastore_sync_poll_rate_s`: Controls the polling frequency.
- `pgaa.metastore_sync_stale_duration_s`: Defines the threshold for metadata freshness.

Use the `pgaa.attach_catalog()` function to attach a catalog:

```
SELECT pgaa.attach_catalog(my_iceberg_catalog);
```

Verify that the tables have been created on your Postgres environment. Once the catalog tables are registered in your Postgres instance, they are accessible just like standard tables.

You can use the `pgaa.detach_catalog()` function to detach the catalog:

```
SELECT pgaa.detach_catalog(my_iceberg_catalog);
```

The function stops the background synchronization worker and removes the dynamic link to the Iceberg metadata service. The catalog remains added (visible in `pgaa.list_catalogs()`), allowing you to re-attach it later without re-entering your credentials, but it will no longer poll for updates or automatically map new tables. If you use the option `CASCADE`, all tables managed by this catalog are also dropped.

7.2.2 Importing a catalog

Instead of continuously polling, you can perform a one-time manual synchronization. Postgres will pull the current state of the tables, but will not detect future changes until you run the import again. If the imported catalog contains a table that shares the name with any existing tables on your database, PGAA skips creating that table to avoid overwriting it. You can also import a catalog to force an update without waiting for the next background poll cycle.

Use the `pgaa.import_catalog()` function to import a catalog:

```
SELECT pgaa.import_catalog(my_iceberg_catalog);
```

You can optionally use the command specifying a namespace filter so only tables within this specific Iceberg namespace will be imported. For example:

```
SELECT pgaa.import_catalog(my_iceberg_catalog, tpeds_sf_1000);
```

Validate that the new tables appear in your Postgres environment. You can run standard SQL queries directly against your data lake without further configuration.

7.2.3 Creating catalog-managed tables

Catalog-managed tables allow you to define custom local table names, choose specific schema placements, and explicitly map columns while leveraging your existing catalog connection.

Manual registration provides more flexibility than attaching an entire catalog. You gain total control over your Postgres namespace, allowing you to place a table in any local schema regardless of its location in the Iceberg catalog.

The synchronization of a manual table depends on the type of change occurring in the data lake:

Change type	Synchronization	Behavior details
Data changes	Automatic	Every query resolves the latest Iceberg snapshot. New, updated, or deleted rows are reflected immediately.
Partition evolution	Automatic	Iceberg stores partitioning logic in metadata. If the partitioning strategy changes, PGAA handles the file-pruning adjustments behind the scenes.
Schema evolution	Manual	Structural changes (new or deleted columns) are ignored or will cause errors. You must update the Postgres table using <code>ALTER TABLE</code> to match the new remote schema.

Create catalog-managed tables using the `CREATE TABLE ... USING PGAA WITH` syntax. For example:

```
CREATE TABLE managed_sales_data (
  id BIGINT,
  sale_date TIMESTAMP,
  amount NUMERIC
)
```

(continues on next page)

(continued from previous page)

```
USING PGAA
WITH (
    pgaa.format = iceberg,
    pgaa.managed_by = my_iceberg_catalog,
    pgaa.catalog_namespace = public,
    pgaa.catalog_table = managed_sales_data
);
```

Where:

- `pgaa.format` : Sets the table format ('iceberg').
- `pgaa.managed_by` : Specifies the specific external catalog (previously defined by `pgaa.add_catalog`).
- `pgaa.catalog_namespace` : Defines the namespace where the table resides within your Iceberg catalog.
- `pgaa.catalog_table` : Specifies the name of the table as it appears in your data lake.

Note

If you leave the column definitions empty (`()`), PGAA will automatically discover and map the schema from the Iceberg catalog.

Writing to object storage

The Postgres Analytics Accelerator (PGAA) supports writing directly to your data lake using the `CREATE TABLE AS SELECT` (CTAS) syntax. This enables you to process and persist data from any source — whether transforming local Postgres tables, restructuring existing object storage files, or joining data across both — into high-performance columnar formats like Iceberg or Delta Lake.

Because PGAA integrates with the Postgres query executor, the source data for a CTAS operation can reside in any location accessible to your instance. This enables several advanced workflows:

- **Exporting:** Move data from a standard local Postgres table into your data lake to archive historical data.
- **Transforming:** Read from one object storage location, perform a join or aggregation, and write the result back to a different cloud directory.
- **Federated joins:** Join a local relational table with a remote object storage table and write the combined result back to the cloud.

By using the `PGAA` access method, you can stream the result set of any query directly to a specified storage location — including AWS S3 (and S3-compatible APIs), Google Cloud Storage (GCS), Azure Blob Storage (ADLS Gen2), and local filesystems, or directly into an integrated Iceberg catalog.

You can perform a CTAS operation using either a standalone storage location or an integrated Iceberg catalog.

Note

CTAS is designed for immutable, write-once operations. For more advanced write capabilities—such as continuous `INSERT`, `UPDATE`, or `DELETE` operations on object storage—refer to the [PGD integration](#).

8.1 Using a standalone storage location

This method is ideal for one-off exports or ad-hoc data movement. PGAA handles the conversion from Postgres rows to the target format, generates the necessary metadata or transaction logs (such as the Iceberg manifest or Delta transaction log), and commits the results in a single atomic operation.

Provisioning a table in object storage via CTAS requires a two-step process: first, you establish a physical connection to the remote storage environment via a storage location, and then you execute the SQL command that maps your query results to that location.

8.1.1 Creating a storage location

To write to object storage, you must [create a PGFS storage location](#). This defines the target bucket and provides the necessary credentials. For example:

```
SELECT pgfs.create_storage_location(  
    my_lake_data,  
    s3://your-bucket-name/path,  
    {  
        "aws_region": "us-east-1", "aws_access_key_id": "...", "aws_secret_access_key":  
        → "..."}  
);
```

Note

The credentials associated with your storage location must have both read and write permissions for the destination bucket. You can verify this by running the following function, which returns *NULL* if successful:

```
SELECT pgaa.test_storage_location (my_lake_data, true);
```

8.1.2 Creating the table with CTAS

Create a table with the `USING PGAA WITH` and `AS` clauses that exports its result set directly to a structured table format in object storage. For example:

```
CREATE TABLE s3_export_table  
USING PGAA  
WITH (  
    pgaa.storage_location = my_lake_data,  
    pgaa.path = path/in/s3/iceberg_data,  
    pgaa.format = iceberg  
) AS  
SELECT * FROM local_source_table;
```

Where:

- `pgaa.format`: Specifies the destination table format. Supported values are `iceberg` and `delta`.
- `pgaa.storage_location`: The name of the storage location defined in the previous step.
- `pgaa.path`: The destination directory. PGAA creates the metadata and log directories automatically within this path.

8.2 Using an Iceberg catalog

If you have integrated an Iceberg catalog, you can write to it using CTAS. In this mode, the catalog automatically manages the storage paths and table versioning for you.

Writing to a table in object storage via CTAS and using an Iceberg catalog comprises two steps: first, configure a catalog connection, and then execute the SQL command that maps your query results via that catalog.

8.2.1 Configuring a catalog connection

Use the function `pgaa.add_catalog()` to register an external Iceberg catalog with your Postgres instance. For example:

```
SELECT pgaa.add_catalog(
  my_iceberg_catalog,
  iceberg-rest,
  {
    "url": "https://your-catalog-endpoint.com",
    "warehouse_name": "your_warehouse_name",
    "warehouse": "your_warehouse_id",
    "token": "your_secret_auth_token"
    "danger_accept_invalid_certs": "false"
  }
);
```

See [Adding a catalog connection](#) for a full list of configuration options.

Note

You must have write permissions on your catalog. You can verify this by running the following function, which returns `NULL` if successful:

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := TRUE);
```

8.2.2 Creating the catalog-managed CTAS table

Create a table with the `USING PGAA WITH` and `AS` clauses that exports its result set directly to a configured Iceberg catalog. For example:

```
CREATE TABLE export_local_transactions
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = managed_sales_data
) AS
SELECT * FROM local_transactions;
```

Where:

- `pgaa.format`: Sets the destination table format. Currently, only 'iceberg' is supported.
- `pgaa.managed_by`: Specifies the catalog name.
- `pgaa.catalog_namespace`: Defines the namespace or database where the table will be located within the Iceberg catalog.
- `pgaa.catalog_table`: Sets the name of the table to be created in your data lake.

8.3 Limitations of CTAS

While CTAS is a powerful tool for data movement and transformation, it operates under a specific execution model with the following constraints:

- **Engine dependency:** When executing a CTAS operation, the source (inner) query does not utilize DirectScan. Instead, the data retrieval is processed by the standard Postgres engine via CompatScan. Consequently, complex joins or aggregations within the `SELECT` phase will not benefit from the full vectorized acceleration provided by the Seafoam engine.
- **No automatic overwrites:** To prevent accidental data loss, when using a standalone storage location, the command will fail by default if data already exists at the specified `pgaa.path`. If you intend to replace the existing data, you must explicitly set the `pgaa.purge_data_if_exists` parameter to `true` in the `WITH` clause of your CTAS statement.
- **Write-once:** CTAS is an atomic, write-once operation. To modify the data, you must drop the table and recreate it at a new path.

Replicating with PGD

Integrate Postgres Distributed (PGD) and Postgres Analytics Accelerator (PGAA) to manage the complete data lifecycle within a unified architecture. PGD provides the high-availability transactional backbone for hot data, while PGAA delivers the vectorized engine and cloud-native handlers for cold data in object storage.

This collaboration creates two distinct storage tiers:

- Hot tier (PGD): Local PostgreSQL storage for current operational data requiring sub-millisecond latency and full ACID compliance.
- Cold tier (PGAA): Cost-effective object storage using columnar formats like Apache Iceberg® or Delta Lake, optimized for high-speed analytical queries.

Together, they allow you to seamlessly transition data from high-performance transactional storage to a cost-effective data lake, all while maintaining transparent access through a unified Postgres interface. This architecture supports a variety of high-impact use cases:

- **Regulatory compliance:** Offload aged data and audit trails to maintain lean, high-performance systems while ensuring long-term retention.
- **Time-series & IoT:** Balance real-time dashboards using hot recent data with cost-effective cold historical analytics for sensor and event streams.
- **Financial services:** Manage massive volumes of required historical transaction records without inflating transactional storage costs.
- **Retail & seasonal business cycles:** Use adaptive policies to keep peak-season order data hot for customer service while tiering older records to cold storage.

9.1 Understanding table types

This architecture implements two distinct storage tiers: the hot tier (local SSD, sub-millisecond latency) and the cold tier (object storage, columnar format). Data exists in one of the following three distinct table types:

Table type	Storage location	Behavior
Heap	Local disk only	Standard transactional table in the hot tier.

continues on next page

Table 1 – continued from previous page

Table type	Storage location	Behavior
Hybrid Analytics and Transactional Processing (HTAP)	Local + object storage	A heap table mapped to an analytical copy. PGD replicates all changes in near real-time.
PGAA	Object storage only	A purely analytical cold table, available in the Delta or Iceberg format.

9.2 Tiered storage strategies

There are three available methods to transition data between tiers based on your operational needs.

9.2.1 Implementing tiered tables

This method leverages **PGD AutoPartition** to automate the lifecycle of high-volume data. A heap table is transformed into a partitioned structure where: - **Hot partitions**: Recent data remains as heap (or HTAP if `enable_replication` is enabled) for active processing. - **Cold partition**: Once partitions cross an age threshold, PGD bulk-copies them to object storage, converts them to PGAA tables, and reclaims local disk space, maintaining the data queryable.

9.2.2 Replicating to analytics

This method continuously synchronizes a table with an analytical copy in object storage. A heap table is converted into an HTAP table. To the user, it appears as a standard transactional table, but an analytical copy is maintained in the background.

To ensure near real-time synchronization without taxing your transactional throughput, PGAA uses Iceberg Merge-on-Read (MoR). This records changes in small “delete files” rather than rewriting entire data blocks, keeping the data lake fresh with minimal overhead.

9.2.3 Offloading to analytics

This method offers a one-time, surgical way to immediately reclaim local disk space for HTAP tables, leaving only the copy in object storage (PGAA table). The local heap copy is truncated, and the table access method is altered to `PGAA`, which now points to object storage.

If you need to bring your data back for transactional writes, you can restore the table back from object storage into the local Postgres heap.

Method	Purpose	Use case	Origin Table	Destination table
—	Tiered tables	Automated lifecycle management	High-volume time-series data where historical retention is critical and automated.	Heap Heap or HTAP (hot) PGAA (cold) Replication
—	Continuous synchronization	Replicating a transactional table in near real-time for general analytics.	Heap HTAP Offload	Selective archiving
—	One-time, surgical removal and archival of a specific table.	HTAP PGAA		

9.3 Metadata management

PGAA supports two methods for managing object store metadata when integrating with PGD:

- **No-catalog offload**: Store metadata alongside data files using *Configuring storage locations* to connect directly to object storage.
- **Catalog-managed offload**: Decouple data from the metadata *Integrating with Iceberg catalogs* like Iceberg REST.

9.3.1 Getting started

To implement tiered storage within a distributed environment, you must first establish a consistent link between your EDB Postgres Distributed (PGD) cluster and your object store.

The following sections guide you through the prerequisites and the configuration steps required to point your cluster to a storage provider. These steps define the external location where a version of your PGD data is persisted in a columnar format.

By configuring these settings at the node-group level, you ensure that every node in the cluster knows exactly where to write and read tiered data.

Prerequisites

- **Cluster:** PGD version 6.1 or later with PGAA and PGFS extensions installed.
- **Storage locations:** Local, S3, GCP, or Azure storage using Iceberg or Delta Lake formats.
- **Catalog:** If using an external catalog service, only Iceberg REST catalogs are supported.
- **Permissions:** The database user must have `CREATE`, `ALTER`, and `EXECUTE` privileges for the PGD and PGAA functions.

Note

The credentials associated with a storage location or a catalog service must have both read and write permissions for the destination bucket. You can verify permissions by running the following functions, which return `NULL` if successful. - For a storage location:

```
SELECT pgaa.test_storage_location (my_storage_location, true);
```

- For a catalog service:

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := FALSE);
```

Pointing to object storage

Before creating a tiered table, you must define the destination for the cold data. Because you are working within a distributed PGD cluster, you must wrap these configuration calls in `bdr.replicate_ddl_command` to ensure the settings are propagated to all nodes in the group.

Tip

To view your node group, use `SELECT node_group_id FROM bdr.node_group; . !!!`

1. Create a storage location using the [PGFS extension](#) and set it as the analytics storage for your node group.

```
SELECT bdr.replicate_ddl_command($$
  SELECT pgfs.create_storage_location(
    my-storage-location,
    protocol://my-bucket-name/path/,
    {"region": "region-name"},
  )
```

(continues on next page)

(continued from previous page)

```
{
  "access_key_id": "...", "secret_access_key": "..."
}
$$);
```

1. Clear `analytics_write_catalog`. This must be `NULL` before an `analytics_storage_location` can be assigned:

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_write_catalog, NULL);
```

1. Set the analytics storage location for your node group:

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_storage_location, my-
→storage-location);
```

1. Configure a connection to an *Integrating with Iceberg catalogs* :

```
SELECT bdr.replicate_ddl_command($$
SELECT pgaa.add_catalog(
  my-catalog-name,
  iceberg-rest,
  {
    "url": "https://my-catalog-rest-endpoint.com",
    "token": "MY_AUTH_TOKEN",
  }
)
);
```

1. Set the catalog as the default analytics write location:

```
SELECT bdr.replicate_ddl_command($$
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_write_catalog, my-
→catalog-name)
);
```

Next steps

Now that you have defined the storage target for your PGD cluster, you can determine how your tables interact with that storage based on your requirements for performance and capacity:

- *Implementing tiered tables* Establish a zero-touch data lifecycle by automatically transitioning partitions from local heap storage to your analytics storage targeted based on an age threshold.
- *Replicating to analytics* Maintain a local transactional heap table while simultaneously synchronizing a columnar copy to your storage target for heavy analytical processing.
- *Offloading to analytics* Reclaim heap disk space from existing tables which are being replicated to analytics.

9.3.2 Implementing tiered tables

Tiered tables provide a unified storage architecture by combining Postgres Distributed (PGD) AutoPartition with Postgres Analytics Accelerator (PGAA) to create an automated data lifecycle. When a standard heap table is converted to a tiered structure, the system manages the transition of data from transactional to analytical storage without manual intervention, providing zero-touch data lifecycle management that optimizes storage costs.

In this architecture, queries span both hot and cold data tiers seamlessly, providing a single point of access for the entire dataset regardless of its physical location. Recent data remains in local heap partitions for high-performance

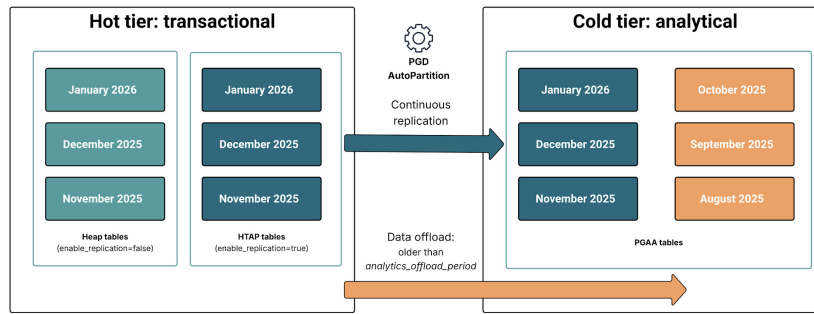


Fig. 1: tiered_tables

transactional processing, while older cold partitions are offloaded to object storage transparently once they cross a defined age threshold.

By calling `pgaa.convert_to_tiered_table()`, a standard heap table is transformed into a hybrid partitioned structure:

- **Hot tier:** Recent data stays in local heap partitions, or Hybrid Analytics and Transactional Processing (HTAP) tables if replication is enabled. For HTAP, PGAA uses Iceberg Merge-on-Read (MoR) to capture updates and deletes as lightweight “delete files.” This keeps the cloud tier synchronized without the overhead of rewriting large data blocks.
- **Cold tier:** Once partitions cross the age threshold defined by `analytics_offload_period`, they are automatically converted to PGAA tables in the data lake. Local storage is reclaimed, but the data remains transparently queryable through the same table interface.

Process overview

To convert a heap table into a tiered table, follow these steps:

1. *Getting started* by choosing either a storage location or a catalog service.
2. Use the `pgaa.convert_to_tiered_table()` function to *Converting a heap table to a tiered table*.
3. *Monitoring tiered tables* the automated offloading.
4. *Modifying and disabling tiered tables* settings as your storage needs evolve.

Converting a heap table to a tiered table

The `pgaa.convert_to_tiered_table()` function initiates the automated offload process. If `enable_replication` is `true`, it also converts active partitions into HTAP tables for continuous replication. For example:

```
CALL pgaa.convert_to_tiered_table(
  relation := my-transactional-table::regclass,
  range_partition_column := date,
  partition_increment := 1 month,
  analytics_offload_period := 1 year,
  initial_lower_bound := 2010-01-01,
  retention_period := 5 years,
  enable_replication := true,
  minimum_advance_partitions := 2,
  maximum_advance_partitions := 5,
```

(continues on next page)

(continued from previous page)

```
drop_after_retention_period := true,
purge_analytics_target := true
);
```

Where:

- `relation`: The name of the heap table to convert to tiered table.
- `range_partition_column`: The column used for range partitioning. It must be `DATE` or `TIMESTAMP`, and it must also be included in the primary key.
- `partition_increment`: The time interval defining the size of each partition.
- `analytics_offload_period`: The threshold age at which a partition is moved from local storage to the cold tier in the data lake.
- `initial_lower_bound`: The starting value for the first partition's lower bound.
- `retention_period`: The period after which data is purged from the cold tier.
- `enable_replication`: If `true`, enables real-time replication to object storage for active partitions (as an HTAP table). If `false`, the hot partitions remain as heap tables.
- `minimum_advance_partitions`: Minimum number of future partitions to maintain.
- `maximum_advance_partitions`: Maximum number of future partitions to maintain.
- `drop_after_retention_period`: Whether to drop the partition locally after the retention period expires.
- `purge_analytics_target`: If `true`, any pre-existing tables in the object store directory or catalog entry are deleted. If set to `false` and pre-existing tables are detected, the command will fail with an error (Supported in PGD 6.3 and later).

Note

The column used for range partitioning must be a primary key.

Tables with sequences or foreign key constraints are not supported with tiered tables. You must drop the sequence or constraint before converting to a tiered table.

When used in conjunction with a catalog service, tiered tables create a unified view in the catalog that performs a `UNION` of all partitions. You can enable the `bdr.prefer_analytics_engine` parameter to route queries on the entire tiered table to the PGAA engine (Seafowl or Spark):

```
SET bdr.prefer_analytics_engine = TRUE;
```

If `enable_replication` is `true`, this setting enables DirectScan for the entire partitioned table. Because the catalog maintains a view of both the offloaded cold data and the replicated hot data, the analytics engine can execute the query across the complete lifecycle of the data.

Monitoring tiered tables

The `pgaa.list_tiered_tables()` function provides a high-level overview of all tables currently undergoing automated lifecycle management. Its output is useful for identifying which heap tables have been successfully converted to tiered tables, what partitioning logic is being applied, and the current balance between local storage and object storage (tiered and untiered data size).

```
SELECT * FROM pgaa.list_tiered_tables();
```

Query the `bdr.analytics_table` view to see the low-level mapping between your Postgres schema and the underlying analytical engine. This table tracks the relationship between your local node group and the remote analytics target.

```
SELECT * FROM bdr.analytics_table;
```

Use the following query to distinguish between hot and cold partitions:

```
SELECT
    relname AS partition_name,
    amname AS access_method
FROM pg_class c
JOIN pg_am am ON c.relam = am.oid
WHERE relname LIKE my-transactional-table%
    AND c.relkind = r
ORDER BY relname;
```

Modifying and disabling tiered tables

You can modify their lifecycle policies or revert them to standard partitioned tables using a combination of PGD and PGAA commands.

Adjusting lifecycle settings

You can update the partitioning and offloading logic for an existing tiered table running the `bdr.autopartition()` function.

This is particularly useful if you need to shorten or extend the offload threshold. For example, to change `analytics_offload_period` from 1 month to 3 months, run:

```
SELECT bdr.replicate_ddl_command($$
SELECT bdr.autopartition(
    relation := public.my-transactional-table,
    partition_increment := 1 month,
    partition_initial_lowerbound := 2020-01-01,
    analytics_offload_period := 3 months
)
$$);
```

See [AutoPartition](#) in PGD for details.

Warning

Modifying these settings only affects new partitions and does not reorganize existing data. If you increase the partition size, the system will not merge your current small partitions; it will only begin creating larger partitions once the current set is exhausted. While new offload timing applies to all partitions, the physical structure created under previous settings remains unchanged. !!!

Modifying replication status

If you wish to change the current setting for `enable_replication`, you must manage the replication status of the partitions.

Use the `pgaa.enable_analytics_replication()` or `pgaa.disable_analytics_replication()` functions against the parent table to ensure consistency across the hierarchy:

```
CALL pgaa.enable_analytics_replication(my-transactional-table::regclass);
```

Using `ALTER TABLE` against the parent table only sets the value of `enable_replication` for future partitions; it does not retroactively update existing child partitions. You must manually run the `ALTER TABLE` command on every existing partition to ensure consistency across your data.

Disabling tiered tables

To revert to a standard partitioned heap table:

1. If `enable_replication` was active, stop replication on the parent table:

```
CALL pgaa.disable_analytics_replication(my-transactional-table::regclass);
```

1. Restore cold partitions. For each cold partition, run:

```
pgaa.restore_from_analytics(partition_name);
```

Or use `ALTER TABLE partition_name SET ACCESS METHOD heap;`

1. Remove the tiered table flag from the parent table:

```
ALTER TABLE my-transactional-table SET (pgaa.tiered_table = false);
```

1. Stop AutoPartition (optional). To stop the table from creating new partitions entirely, run:

```
bdr.drop_autopartition(my-transactional-table);
```

1. Verify that the table is not listed as a tiered table anymore with `SELECT * FROM pgaa.list_tiered_tables()`.
2. Verify that all partitions list `heap` as their access method:

```
SELECT
  c.relname AS partition_name,
  a.amname AS access_method
FROM pg_class c
JOIN pg_am a ON c.relam = a.oid
WHERE c.relname LIKE my-transactional-table%
AND c.relkind = r;
```

9.3.3 Replicating to analytics

Use Postgres Analytics Accelerator (PGAA) within a Postgres Distributed (PGD) cluster to synchronize local heap tables with an analytical data lake by converting them to Hybrid Analytics and Transactional Processing (HTAP) tables.

In this state, every `INSERT`, `UPDATE`, and `DELETE` operation is captured and streamed in near real-time to object storage. While the table remains a standard transactional endpoint for applications, PGD maintains a high-fidelity analytical replica in the background, making data immediately available for BI tools and query engines without impacting production performance.

The key benefits of this process are:

- **Near real-time analytics:** Access the latest transactional data in your data lake without waiting for batch ETL processes.

- **Dual-state accessibility:** Maintain local high-performance storage for transactions while providing a columnar version for heavy analytical queries.
- **Unified management:** Manage your data lifecycle and replication settings directly through standard SQL and PGAA functions.
- **Efficiency with Merge-on-Read:** PGAA uses Iceberg Merge-on-Read (MoR) to synchronize data. Instead of rewriting large files every time a row changes, it simply records the change in a delete file. This keeps replication fast and significantly reduces the load on your object storage.

Process overview

To enable continuous replication for a table, follow these steps:

1. *Getting started* by choosing either a storage location or a catalog service.
2. *Enabling replication* to start the data stream.
3. *Monitoring replication* status and health.
4. *Disabling replication* if you no longer require synchronization.

Enabling replication

Use the `pgaa.enable_analytics_replication()` function to configure an existing heap table to stream data to object storage. The function converts the heap table to an analytical HTAP table and starts the logical replication worker.

```
CALL pgaa.enable_analytics_replication(my_table::regclass, purge_analytics_target := TRUE);
```

Set `purge_analytics_target := TRUE` to purge existing data at the target location before replication begins (requires PGD 6.3 or later).

Alternatively, switch the replication state directly on the table object:

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = TRUE);
```

You can enable the `bdr.prefer_analytics_engine` parameter to route queries to the PGAA engine (Seafoam or Spark):

```
SET bdr.prefer_analytics_engine = TRUE;
```

If disabled, queries are executed by the standard Postgres engine against the local heap.

Monitoring replication

Once replication is enabled, you can verify the HTAP status by checking that the table appears in both the local relation list and the analytics metadata.

View mappings between local tables and analytics targets:

```
SELECT * FROM bdr.analytics_table;
```

List all tables currently being managed by PGAA:

```
SELECT * FROM pgaa.list_analytics_tables();
```

Disabling replication

Disabling replication stops the logical worker from capturing changes. The local table reverts to a standard heap table, and the remote copy in object storage becomes a static snapshot of the data.

Use the `pgaa.disable_analytics_replication()` function to stop the synchronization:

```
CALL pgaa.disable_analytics_replication(my_table::regclass);
```

Alternatively, use `ALTER TABLE` :

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = FALSE);
```

```
SET bdr.prefer_analytics_engine= FALSE;
```

Note

If you re-enable replication on a table where it was previously disabled, the system performs a full reload. This process effectively removes the existing analytics data in object storage, re-uploads the current local heap data in its entirety, and starts continuous synchronization. !!!

9.3.4 Offloading to analytics

An Hybrid Analytics and Transactional Processing (HTAP) table uses Postgres Analytics Accelerator (PGAA) to continuously synchronize a Postgres Distributed (PGD) cluster with an analytical copy in object storage, enabling heavy analytics without impacting transaction performance at the cost of redundant storage in both the local heap and the data lake.

Offloading an HTAP table to analytics is a manual, surgical operation used to remove the local heap copy and reclaim disk space. When you offload a table, the system:

- Truncates the local heap table, immediately freeing up disk space on the PGD nodes.
- Stops the real-time replication to the data lake.
- Changes the table access method to PGAA, pointing the Postgres table definition directly to the existing data in object storage.

The result is a cold table that remains fully queryable through Postgres but consumes no local storage. If you require transactional writes again in the future, you can restore the table and stream the data back from the lake into the local heap.

Process overview

To surgically offload a table to the data lake, follow these steps:

1. *Getting started* by choosing either a storage location or a catalog service.
2. *Offloading a table* .
3. *Restoring from analytics* back to heap if you require transactional access again.

Offloading a table

Use the `pgaa.convert_to_analytics()` function to convert a synchronized HTAP table into a cold, analytical table and remove the data from local disk.

```
CALL pgaa.convert_to_analytics(my-table::regclass);
```

The function removes the local data and redefines the table's access method to PGAA to point solely to the external Iceberg or Delta files.

Alternatively, switch the access method with the `ALTER TABLE` command. Use the `pgaa.auto_truncate` option to ensure local disk space is freed immediately.

For storage locations:

```
ALTER TABLE public.my_table
SET ACCESS METHOD pgaa,
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.storage_location = my-storage-location,
  pgaa.path = public.my_table
);
```

For a catalog-managed table:

```
ALTER TABLE public.my_catalog_managed_table
SET ACCESS METHOD pgaa,
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = my_remote_table_name
);
```

Restoring from analytics

If you need to resume transactional writes or bring the data back to local high-performance transactional storage, use the `pgaa.restore_from_analytics()` function:

```
CALL pgaa.restore_from_analytics(public.transactional_table);
```

The function checks the PGAA table's configuration, identifies its external Iceberg/Delta path in object storage, and rewrites the data back into the table's local storage structure. Once the data is locally present, the function sets the table's access method back to the local transactional endpoint again.

You can also use the `ALTER TABLE` command to restore a table from its cold state in the data lake back into a local transactional heap table. This manual method provides granular control over the transition back to local storage.

```
ALTER TABLE public.my_table
SET ACCESS METHOD heap;
```

Warning

Restoring a table to the heap access method does not remove the files from object storage. Manually delete the data via your object storage provider, or use a tool like PyIceberg to drop the table from the catalog. !!!

Accelerating with Spark

By default, the Postgres Analytics Accelerator (PGAA) utilizes Seafoam, an embedded analytical engine, to accelerate queries. However, for large-scale data processing that exceeds the resources of a single Postgres instance, you can offload execution to a remote Apache Spark cluster via Spark Connect.

Spark Connect is a thin client-server protocol for Apache Spark that decouples the application from the Spark driver. It acts as a high-speed bridge, allowing Postgres to send query instructions to a remote, distributed Spark cluster. This enables you to leverage the massive compute power of an external cluster without requiring Spark to run on the same machine as your database.

10.1 Choosing your executor engine

The `pgaa.executor_engine` configuration parameter determines where the heavy lifting of your analytical queries happens. When moving beyond the default Seafoam engine, you have two primary options for distributed execution:

10.2 Distributed execution options

To leverage Spark, you can configure your environment in one of two ways:

- ***Distributed Spark execution*** : Use standard Apache Spark clusters to process massive datasets using distributed CPU cores. This is the standard approach for large-scale data processing and heavy maintenance tasks.
- ***GPU-accelerated Spark with NVIDIA RAPIDS*** : Integrate the NVIDIA RAPIDS Accelerator for Apache Spark. This option offloads query execution to GPUs, significantly reducing processing time and infrastructure costs for compute-intensive joins and aggregations.

Feature	Seafoam	Spark (CPU)	Connect	Spark + RAPIDS (GPU)
Architecture	Process alongside Postgres.	Remote Spark cluster.		Remote Spark with NVIDIA GPUs.

continues on next page

Table 1 – continued from previous page

Feature	Seafowl	Spark (CPU)	Connect	Spark + RAPIDS (GPU)
Best for	Small/medium datasets.	Petabyte-scale, complex ETL.		Massive joins & aggregations.
Scalability	Single host limits.	Horizontal (Multi-node).		Horizontal + GPU Acceleration.
Complexity	Zero-config.	Requires Spark endpoint.		Requires GPU-enabled nodes.
Cost efficiency	Best for datasets < 1 TB.	Best for datasets between 1 TB and 3 TB.		Best for datasets 3 TB or higher.

While Seafowl is highly optimized for single-node performance, consider switching to a Spark-based executor if:

- **Memory constraints:** Your queries exceed the `pgaa.autostart_seafowl_max_memory_mb` limit.
- **Maintenance heavy:** You are performing resource-intensive operations like large-scale Compaction or Z-Ordering on Iceberg tables.
- **Extreme scale:** You need to process joins and aggregations across datasets that require the distributed compute of an external cluster.

10.2.1 Distributed Spark execution

Integrate the Postgres Analytics Accelerator (PGAA) with Apache Spark to transform Postgres into a high-performance platform that overcomes the CPU bottlenecks of multi-terabyte datasets.

Offloading query execution to Apache Spark via PGAA provides immediate architectural advantages:

- **Intentional hand-off:** You maintain full control by explicitly selecting the execution engine; PGAA then packages the SQL query and routes it to a Spark Connect endpoint.
- **Distributed processing:** By moving workloads to a Spark cluster, you leverage distributed computing power to handle massive datasets that would otherwise bottleneck a single database instance.
- **Zero indexing:** Achieve peak analytical performance without the need for manual index creation or constant maintenance, significantly reducing operational overhead.

Note

Currently, integration with Apache Spark supports: - Read-only queries on Parquet files in S3-compatible object storage or a shared POSIX filesystem. - Read-only queries for Iceberg tables in Iceberg REST catalogs (OAuth2 authentication is not supported). !!!

The configuration of PGAA with Spark consists of the following steps:

1. *Configuring your Spark environment*
2. *Connecting Postgres to Spark*
3. *Configuring the source data*
4. *Running analytical queries*
5. *Inspecting the query plan*

Configuring your Spark environment

You can leverage or install a Spark cluster installation running 3.4 or later.

You must have a running [Spark Connect](#) server configured with the following dependencies:

```
org.apache.spark:spark-connect_2.12:3.5.6,\
io.delta:delta-spark_2.12:3.3.1,\
org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.9.2,\
org.apache.iceberg:iceberg-aws-bundle:1.9.2,\
org.apache.hadoop:hadoop-aws:3.3.4
```

Important

There is a known issue in the Iceberg Spark Library version 1.9.2 when using Spark to read from Iceberg tables. Under certain conditions, equality deletes may occasionally be skipped during concurrent executions. The current workaround is to disable the Spark application setting `spark.sql.iceberg.executor-cache.enabled` on your `spark-defaults.conf` file.

Disabling this cache ensures data consistency by correctly processing all deletes, but it may have performance implications for high-volume read workloads.

1. Set the execution engine and define your Spark Connect URL in your Postgres session:

```
SET pgaa.executor_engine = spark_connect;
SET pgaa.spark_connect_url = sc://spark-connect:15002;
```

Where `spark-connect` points to your Spark Connect service address.

1. Validate the connection by running a simple version check through the PGAA interface:

```
SELECT pgaa.spark_sql(SELECT version());
```

If successful, the command returns the version string of your Spark cluster.

Configuring the source data

PGAA supports either reading Parquet files from S3-compatible storage and local file system as data storage locations, or reading Iceberg tables in Iceberg REST catalogs.

Option 1: Reading from Parquet files

1. Create a [PGFS storage location](#) in your Postgres database that points to the bucket containing your analytical data. For example, with a public bucket:

```
SELECT pgfs.create_storage_location(
my-sample-data,
s3://beacon-analytics-demo-data-us-east-1-prod,
{"skip_signature": "true", "region": "us-east-1"}
);
```

1. Create tables using the PGAA access method. Specify the storage location created in the previous step, the path to either a single Parquet file or a directory containing multiple Parquet files, and the format (currently Parquet). For example:

```
CREATE TABLE customer () USING PGAA WITH (pgaa.storage_location = my-sample-data, ↵
↵pgaa.path = tpch_sf_1/customer, pgaa.format = parquet);
CREATE TABLE orders () USING PGAA WITH (pgaa.storage_location = my-sample-data, pgaa.
↵path = tpch_sf_1/orders, pgaa.format = parquet);
CREATE TABLE lineitem () USING PGAA WITH (pgaa.storage_location = my-sample-data, ↵
↵pgaa.path = tpch_sf_1/lineitem, pgaa.format = parquet);
CREATE TABLE nation () USING PGAA WITH (pgaa.storage_location = my-sample-data, pgaa.
↵path = tpch_sf_1/nation, pgaa.format = parquet);
```

If you specify a directory containing multiple Parquet files, PGAA automatically unions all files into a single table for processing.

Option 2: Reading from Iceberg tables

1. Configure a catalog connection. For example:

```
SELECT pgaa.add_catalog(
  my_iceberg_catalog,
  iceberg-rest,
  {
    "url": "https://iceberg.example.com/api/v1",
    "warehouse": "a1b2c3d4-e5f6-g7h8",
    "token": "secret_token_abc123"
  }
);
```

1. Create the necessary catalog-managed tables. Alternatively, attach or import the catalog.

```
CREATE TABLE customer () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
↵iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = customer);
CREATE TABLE orders () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
↵iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = orders);
CREATE TABLE lineitem () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
↵iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = lineitem);
CREATE TABLE nation () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
↵iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = nation);
```

See [Integrating with Iceberg catalogs](#) for details on how to configure access to tables with an Iceberg catalog.

Running analytical queries

This example query generates a Top 20 Revenue Report by identifying customers who achieved the highest revenue within a specific quarter. By offloading this to Spark, PGAA parallelizes complex conditional aggregations and utilizes predicate pushdown to minimize data transfer from object storage.

```
SELECT
  c_custkey,
  c_name,
  SUM(l_extendedprice * (1 - l_discount)) AS revenue,
  c_acctbal,
  n_name,
  c_address,
  c_phone,
```

(continues on next page)

(continued from previous page)

```

    c_comment
FROM
    customer,
    orders,
    lineitem,
    nation
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate >= 1993-10-01
    AND o_orderdate < 1994-01-01
    AND l_returnflag = R
    AND c_nationkey = n_nationkey
GROUP BY
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
ORDER BY
    revenue DESC,
    c_custkey
LIMIT 20;

```

Inspecting the query plan

When Spark is not enabled, PGAA defaults to the Seafowl engine for local vectorized execution. Prepend `EXPLAIN` to your statement to see the transition from local, single-node execution to a remote, distributed processing model via Spark Connect.

Comparing these two query plans highlights the transition from local, single-node execution to a remote, distributed processing model via Spark Connect:

- **Join strategy:** Seafowl uses a “nested-loop” approach (`Cross Join + Filter`), which forces the database to compare rows sequentially. Spark upgrades this to a `BroadcastHashJoin`, where small tables are sent to every worker node to process the large fact table in parallel.
- **Aggregation logic:** Instead of a single-node sequential `aggregate`, Spark uses a two-stage `HashAggregate`. This calculates a `partial_sum` on each individual worker node and then performs an `Exchange` (shuffle) to combine the results, providing horizontal scalability for large-scale data.
- **Data access:** Seafowl performs a standard `TableScan` on all available data. Spark uses a metadata-aware `BatchScan` via the Iceberg REST Catalog, which reads Iceberg manifest files to skip irrelevant data blocks entirely before they are even pulled from S3.

The Spark plan is a sophisticated, distributed blueprint designed to scale horizontally across a cluster for massive datasets. Conversely, the Seafowl plan is a high-speed, localized blueprint optimized for smaller datasets where the overhead of network latency and cluster coordination would outweigh the actual processing time.

10.2.2 GPU-accelerated Spark with NVIDIA RAPIDS

Integrate NVIDIA RAPIDS with your Spark-connected Postgres Analytics Accelerator (PGAA) environment to achieve hardware-level acceleration for your data lakehouse. While Spark orchestrates distributed workloads across the cluster, RAPIDS offloads intensive computational tasks to the GPU, delivering massive throughput gains for complex queries on large-scale datasets.

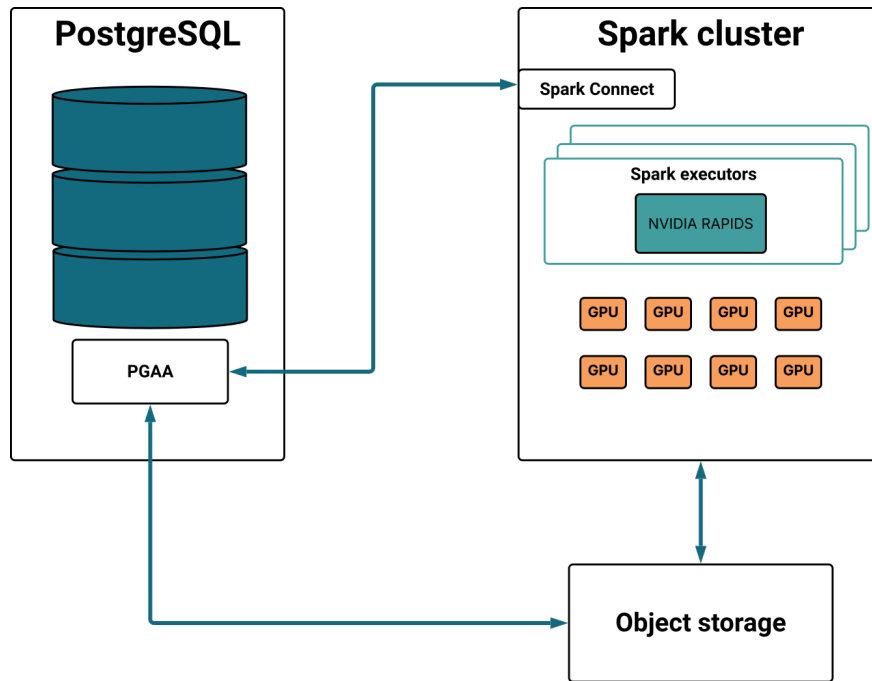


Fig. 1: rapids_arch

This GPU-accelerated approach allows PGAA to transcend standard distributed processing through the following core capabilities:

- **Massively parallel execution:** Thousands of NVIDIA GPU cores process columnar data simultaneously, completing complex queries up to 100x faster than standard Postgres tables.
- **Hardware-accelerated scaling:** While Spark manages distribution at 100 GB, RAPIDS becomes the primary performance driver at 3 TB and beyond, widening the hardware efficiency gap as data grows.
- **Zero-overhead performance:** Operates directly on columnar data in memory, delivering peak performance without manual indexing, vacuuming, or traditional database tuning.
- **Intelligent fallback:** Ensures query completion by automatically shifting unsupported operations back to the Spark CPU or the Postgres kernel without user intervention.

Note

Currently, GPU acceleration with Apache Spark is optimized for read-only queries on Parquet files in S3-compatible object storage or a shared POSIX filesystem. Support for Iceberg (read/write) is coming soon. !!!

Configuring GPU acceleration with NVIDIA RAPIDS

The core architecture consists of a Postgres server with the PGAA extension, an Apache Spark cluster configured with NVIDIA RAPIDS, and compatible object storage containing Parquet-formatted data.

To implement this environment, you can leverage existing infrastructure, perform a manual installation, or use our [ready-to-run Docker Compose stack](#) to quickly deploy all necessary components. This stack includes pre-configured examples optimized for the NVIDIA Brev platform, allowing you to easily deploy self-serve GPU instances in the cloud. If compatible GPUs are detected, NVIDIA RAPIDS will automatically provide hardware acceleration; otherwise, the system transparently falls back to CPU-based processing.

The configuration of PGAA with NVIDIA RAPIDS consists of the following steps:

1. Verify the *Prerequisites* .
2. Confirm that *Enabling NVIDIA RAPIDS* is properly installed and active within your Spark cluster.
3. *Connecting Postgres to Spark* .
4. *Configuring the source data* .
5. *Running analytical queries* and verify that the workload is utilizing GPU acceleration.

Prerequisites

- **Postgres:** Version 16 or later with PGAA and PGFS extensions installed.
- **Storage locations:** Currently, only S3-compatible storage and local file system are supported as data storage locations.
- **Apache Spark:** Version 3.4 or later.
- **Spark Connect:** A running [Spark Connect](#) server configured with the following dependencies:

```
org.apache.spark:spark-connect_2.12:3.5.6,\
io.delta:delta-spark_2.12:3.3.1,\
org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.9.2,\
org.apache.iceberg:iceberg-aws-bundle:1.9.2,\
org.apache.hadoop:hadoop-aws:3.3.4
```

- **NVIDIA RAPIDS:** Version 2025.10 or above installed on your Spark cluster. See *Enabling NVIDIA RAPIDS* for details.
- **Hardware:** Access to a machine equipped with NVIDIA GPUs.

Enabling NVIDIA RAPIDS

To utilize GPU acceleration, you must correctly install and activate NVIDIA RAPIDS within your Apache Spark cluster. Configure your Spark environment to support hardware acceleration by following these integration guidelines:

1. **Download RAPIDS JARs:** Ensure your Spark environment downloads the necessary

RAPIDS JAR files . Specify the plugin version

`com.rapids-4-spark_2.12:25.10.0` when submitting a job.

2. **GPU discovery:** Install the [GPU discovery script](#) that allows Spark to identify the available GPUs.
3. **Apply Spark configuration:** Enable the following parameters as a minimum. For a full list of optimization settings, see the

[RAPIDS configuration and Tuning guide](#) .

```
spark.rapids.sql.enabled=true
spark.rapids.filecache.enabled=true
spark.executor.resource.gpu.amount=1
spark.plugins=com.nvidia.spark.SQLPlugin
spark.shuffle.manager=com.nvidia.spark.rapids.spark356.RapidsShuffleManager
```

1. **Container provisioning:** If deploying Spark via Docker, modify your container manifest to grant Spark access to the host GPUs.

For a complete example that implements the above steps—including volume mounting the discovery script, setting environment-level Spark configurations, and configuring the NVIDIA container runtime—refer to our sample [docker-compose.yaml](#).

Connecting Postgres to Spark

1. From your Postgres's terminal, set PGAA to use Spark as the execution engine and define your Spark Connect endpoint:

```
SET pgaa.executor_engine = spark_connect;
SET pgaa.spark_connect_url = sc://spark-connect:15002;
```

Where `spark-connect` points to your Spark Connect service address.

1. Confirm that Postgres can communicate with the Spark cluster by executing a simple version check through the PGAA interface:

```
SELECT pgaa.spark_sql(SELECT version());
```

If successful, the command returns the version string of your Spark cluster.

Configuring the source data

1. Create a [PGFS storage location](#) in your Postgres database that points to the bucket containing your analytical data. For example, with a public bucket:

```
SELECT pgfs.create_storage_location(
my-sample-data,
s3://beacon-analytics-demo-data-us-east-1-prod,
{"skip_signature": "true", "region": "us-east-1"}
);
```

1. Create tables using the PGAA access method. Specify the storage location created in the previous step, the path to either a single Parquet file or a directory containing multiple Parquet files, and the format (currently Parquet). For example:

```
CREATE TABLE store () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/store, pgaa.
↪format = parquet);
CREATE TABLE customer () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/customer, ↪
↪pgaa.format = parquet);
CREATE TABLE date_dim () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/date_dim, ↪
↪pgaa.format = parquet);
```

(continues on next page)

(continued from previous page)

```
CREATE TABLE store_returns () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/store_
  ↳returns, pgaa.format = parquet);
```

If you specify a directory containing multiple Parquet files, PGAA automatically unions all files into a single table for processing.

Running analytical queries

Run heavy analytical queries against your defined tables to verify that they are successfully leveraging GPU acceleration.

The query is offloaded to the Spark cluster, and if NVIDIA RAPIDS is active, the join and aggregation operations are performed in parallel across thousands of GPU cores.

The following example runs a TPC-DS inspired analytical query that identifies high-return customers by comparing their total store returns against the regional average. This involves complex Common Table Expressions (CTEs), correlated subqueries, and multi-stage aggregations—precisely the type of CPU-intensive workload that PGAA accelerates through vectorized execution:

```
WITH customer_total_return AS (
  SELECT
    sr_customer_sk AS ctr_customer_sk,
    sr_store_sk AS ctr_store_sk,
    SUM(sr_return_amt) AS ctr_total_return
  FROM store_returns, date_dim
  WHERE sr_returned_date_sk = d_date_sk
    AND d_year = 2000
  GROUP BY sr_customer_sk, sr_store_sk
)
SELECT c_customer_id
FROM customer_total_return ctr1, store, customer
WHERE ctr1.ctr_total_return > (
  SELECT AVG(ctr_total_return) * 1.2
  FROM customer_total_return ctr2
  WHERE ctr1.ctr_store_sk = ctr2.ctr_store_sk
)
AND s_store_sk = ctr1.ctr_store_sk
AND s_state = TN
AND ctr1.ctr_customer_sk = c_customer_sk
ORDER BY c_customer_id
LIMIT 100;
```

To confirm that NVIDIA RAPIDS is actively accelerating your queries, monitor the execution plan through the Spark Connect UI:

1. Access the Spark UI at `http://<spark_driver_host>:4040`.
2. Navigate to the SQL tab and select your recently executed query.
3. Inspect the plan. If RAPIDS is active, the query nodes will be prefixed with **Gpu**. For example, `GpuFilter`, `GpuBatchScan`, `GpuUnion`, etcetera.

Beyond the query plan, you can analyze GPU-specific performance metrics directly within the Spark UI to identify potential bottlenecks. For a detailed breakdown of these metrics and how to optimize them, refer to the [NVIDIA tuning guide](#).

Monitoring and maintaining analytical tables

Postgres Analytics Accelerator (PGAA) provides built-in functions to help you audit storage utilization, table formats, and metadata health of your data lake tables directly from the Postgres prompt.

11.1 Analyzing table statistics

In environments with hundreds of standard Postgres tables, identifying which ones are managed by PGAA can be challenging. The `pgaa.list_analytics_tables()` function provides a global overview of all lakehouse-integrated tables.

```
SELECT * FROM pgaa.list_analytics_tables();
```

The function output provides the following insights:

- **Identity & format:** Displays the schema, table name, and storage format (e.g., Iceberg, Delta, or Parquet).
- **Storage location:** The exact path to where the data files reside.
- **Catalog metadata:** If using an external catalog, this function output shows the external catalog service name, namespace, and remote table name.
- **Replication status:** Displays the current state of data movement (only for PGD replication scenarios).
- **Storage metrics:**
 - `object_storage_snapshot_size_bytes` : The size of the latest live snapshot. This represents the actual volume of data scanned during a `SELECT *`.
 - `object_storage_total_size_bytes` : The total footprint in the bucket, including metadata, manifests, and historical data files not used by the current version.
 - `object_storage_total_size_bytes` : The total footprint in the bucket, including metadata, manifests, and historical data files not used.

The function `pgaa.lakehouse_table_stats()` provides specific information about table's physical characteristics. This is essential for capacity planning and fine-tuning query performance.

```
SELECT * FROM pgaa.lakehouse_table_stats(my_schema.my_table::regclass);
```

This function queries the underlying object storage to verify what is happening “under the hood” of a specific mapping. It helps you:

- **Distinguish live vs. dead data:** Identify exactly how much storage is being “wasted” by old transaction logs or uncompact small files.
- **Monitor growth:** Track how snapshot sizes evolve over time as new data is ingested from external sources.
- **Cache awareness:** The results of this function are governed by the `pgaa.lakehouse_table_stats_cache_ttl_s` configuration parameter. If you need the absolute latest stats and aren’t seeing them, ensure your cache TTL is set to 0.

Comparing `latest_snapshot_size` and `total_size` is essential for balancing performance and cost. The snapshot size represents “live” data—the actual volume scanned during a query—while the total size includes the entire bucket footprint, such as historical versions, transaction logs, and uncompact rows.

In formats like Iceberg or Delta Lake, the total size exceeds the snapshot size to support time-travel and rollbacks. A significant gap between these values indicates a buildup of historical overhead and serves as a signal to perform a `VACUUM` or compaction. Monitoring these metrics helps you maintain a lean, high-performing data lake.

11.2 Optimizing table performance

Analytical performance is heavily impacted by the “small file problem”, where frequent data updates create thousands of tiny files that increase metadata overhead.

11.2.1 Performing table maintenance with `pgaa.launch_task`

Maintenance for both Delta and Iceberg tables is handled asynchronously by a native background worker using the `pgaa.launch_task()` function. This allows you to trigger heavy operations without blocking your active SQL session.

```
SELECT pgaa.launch_task(
  table_name => my_delta_table::regclass,
  task_type => compaction, -- or z-order, vacuum, purge
  options => {"target_size": "128MB"}
);
```

Note

While Delta Lake supports all available task types, Iceberg tables currently only support the *compaction* operation. Furthermore, Iceberg compaction does not support the granular configuration keys available for Delta Lake, except for *dry_run*.

Supported task types include:

- **Compaction:** Merges small files into larger, optimized blocks (defined by `target_size`) to speed up analytical scans.
- **Z-Order (Delta only):** Reorganizes data across multiple columns so related values are stored physically close together. This enables data skipping, where the engine ignores entire files that don’t match query filters.
- **Vacuum (Delta only):** Deletes data files no longer associated with active snapshots and older than a specified retention period. This reclaims object storage space.

- **Purge (Delta only):** An administrative tool to explicitly remove data from a specific storage path, useful for manual cleanup of failed exports.

Examples

- Use compaction against a Delta Lake table to merge small files into larger, 256MB blocks to speed up scan performance:

```
SELECT pgaa.launch_task(
  table_name => lakehouse.sales_data::regclass,
  task_type  => compaction,
  options    => {"target_size": "256MB"}
);
```

- Use z-order to physically reorganize data so that rows with similar values in `customer_id` and `region` are stored together, allowing the engine to skip irrelevant files:

```
SELECT pgaa.launch_task(
  table_name => lakehouse.sales_data::regclass,
  task_type  => z-order,
  options    => {"columns": ["customer_id", "region"]}
);
```

- Use vacuum to delete files that are no longer part of the active table state and are older than 7 days to reclaim storage space:

```
SELECT pgaa.launch_task(
  table_name => lakehouse.sales_data::regclass,
  task_type  => vacuum,
  options    => {"retention_period": "7 days"}
);
```

See the function reference section for [pgaa.launch_task\(\)](#) for a complete list of supported options and examples.

11.2.2 Advanced Spark-based maintenance

For complex Iceberg maintenance operations not yet supported by the native background worker, use the following functions.

Note

To use these functions, you must set the `pgaa.spark_connect_url` configuration parameter to point to an available Spark Connect service.

`pgaa.spark_sql()`

Run Iceberg compaction routines or Spark functions that aren't available in Postgres via the configured Spark Connect endpoint. See [Spark procedures](#) for a list of the available procedures.

```
SELECT pgaa.spark_sql(query, catalog_name);
```

`pgaa.execute_compaction()` (deprecated)

The `pgaa.execute_compaction()` function consolidates fragmented data files into larger, more efficient Parquet files, improving query performance and optimizing metadata:

```
SELECT pgaa.execute_compaction(my_iceberg_table::regclass);
```

Note

The function `pgaa.execute_compaction()` is deprecated in favor of `pgaa.launch_task()`.

Examples

- Use standard table compaction to merge fragmented data files into larger files and updates the Iceberg manifest:

```
SELECT pgaa.execute_compaction(lakehouse.inventory_iceberg::regclass);
```

- Use `pgaa.spark_sql()` to reduce metadata overhead via the `rewrite_data_files` Spark task:

```
SELECT pgaa.spark_sql($$  
    CALL preexisting.system.rewrite_data_files(  
        table => "preexisting"."ns-1"."table-1",  
        strategy => sort,  
        sort_order => value DESC,  
        options => map(rewrite-all, true)  
    )  
$$);
```

See the [Functions](#) for a complete list of supported options and examples.

Optimizing query performance

The Postgres Analytics Accelerator (PGAA) achieves its high performance by offloading analytical workloads to Seafohl and Apache Spark, a vectorized execution engine designed for massive scale. By understanding how PGAA selects execution paths, you can ensure your queries stay on the “fast path” for maximum acceleration.

12.1 Understanding scan types

When you query an analytical table, PGAA manages data retrieval and processing via two distinct execution paths:

DirectScan: Seafohl reads data directly from object storage, applies all filters and aggregations at the source, and returns only the final result set to Postgres. This maximizes performance by fully utilizing vectorized acceleration and minimizing network traffic.

CompatScan: If a query contains SQL features or custom functions that Seafohl cannot process natively, the system falls back to a hybrid mode. In this scenario, Seafohl still acts as the data source—streaming data from object storage and performing basic filtering—while Postgres acts as the orchestrator for final complex processing.

In CompatScan, the workload is divided:

- Seafohl reads files and handles authorized pushdowns (like joins or group-bys).
- Postgres receives the optimized stream from Seafohl to execute specialized functions, final sorting, or complex logic that the vectorized engine doesn’t yet support.

While DirectScan is the goal for maximum speed, CompatScan serves as a reliable fallback, ensuring full Postgres feature compatibility at the cost of some performance.

12.2 Analyzing your query plan

To verify how PGAA is executing your query, use the `EXPLAIN` command. The output will reveal which engine is doing the heavy lifting:

- **DirectScan:** Look for a `SeafohlDirectScan` or a `SparkDirectScan` node. This indicates that Seafohl is handling the entire operation (filtering, joining, and aggregating) and returning only the final result to Postgres.

- **CompatScan:** If you see a `SeafowlCompatScan` or a `SparkCompatScan` node, the query is running in “hybrid” mode. While Postgres orchestrates the final execution to ensure full feature compatibility, the remote engine can still accelerate parts of the query if pushdown settings are enabled.

12.3 Managing DirectScan fallbacks

If your query falls back to the slower `CompatScan` path, follow these steps to troubleshoot:

1. Ensure that `DirectScan` is permitted in your current session:

```
SHOW pgaa.enable_direct_scan;  
-- If off, enable it:  
SET pgaa.enable_direct_scan=on;
```

2. Force Error reporting: Change the failure behavior You can to reveal the cause of the fallback:

```
SET pgaa.direct_scan_fail_behavior = error;
```

Postgres will throw an explicit error identifying the incompatible SQL syntax or data type.

3. If the query is complex, try removing clauses one by one to isolate the specific “compatibility trigger” forcing the hybrid mode.

12.4 Configuring compute pushdowns

Even in hybrid mode, you can maintain high performance by offloading resource-intensive operations. The following settings are enabled by default and act as a performance safety net:

pgaa.enable_join_pushdown Authorizes the system to attempt to offload joins to the remote executor (Seafowl or Spark). By joining data where it resides, you avoid pulling massive raw datasets into Postgres memory, significantly reducing network traffic.

pgaa.enable_groupby_pushdown Authorizes the offloading of aggregations (`SUM`, `COUNT`, `AVG`) and `GROUP BY` clauses. Aggregating 1 billion rows into 10 groups at the source reduces data transfer by several orders of magnitude, saving both bandwidth and Postgres CPU.

The reference section provides detailed technical specifications for the Postgres Analytics Accelerator (PGAA). Use these resources to fine-tune your configuration, understand supported data types, and manage your analytical tables.

For information on the PGFS functions for managing storage location, see the [AI Factory PGFS documentation](#).

- *Configuration parameters* : The behavior of the PGAA extension is governed by Grand Unified Configuration (GUC) variables. These parameters allow you to switch executors, enable performance optimizations, and manage security credentials.
- *Functions* : PGAA introduces a suite of SQL functions for administrative tasks, such as mapping new tables, monitoring storage health, and launching maintenance background jobs.
- *Table options* : When mapping or creating analytical tables, specific options allow you to define how data is read from or written to your object store.
- *Data types and definitions* : PGAA maps native Postgres data types to optimized columnar formats in the data lake.
- *Benchmark datasets* : Access pre-configured schemas and data loading instructions for analytical datasets to baseline your performance.

13.1 Configuration parameters

This reference guide describes the configuration parameters provided by the Postgres Analytics Accelerator (PGAA) extension, allowing you to tune the execution engine, manage remote connectivity, and optimize query performance across your data lake.

13.1.1 Executor engine

pgaa.executor_engine

Determines which compute backend is responsible for executing analytical operations.

Supported values are `seafowl`, and `spark_connect`.

Default is `seafowl`.

pgaa.spark_connect_url

Defines the connection endpoint for the Spark Connect service. Specifies the URL of the remote Spark Connect service. This setting is mandatory when `pgaa.executor_engine` is set to `spark_connect`.

The value must follow the specific Spark Connect URI format: `sc://<host>:<port>`.

Default is `NULL`.

pgaa.spark_connect_extra_config

When using `pgaa.executor_engine= 'spark_connect'`, provides fine-grained tuning parameters and runtime settings directly to the Spark session as a JSON object. See [Spark Connect configuration](#) for a list of options.

Example

```
SET pgaa.spark_connect_extra_config = {  
  "spark.sql.execution.arrow.maxRecordsPerBatch": "65536"  
};
```

Default is `NULL`.

13.1.2 Query behavior

pgaa.enable_direct_scan

Controls the query execution strategy for analytical tables. It determines whether PGAA attempts to offload the entire query to the optimized Seafowl engine or processes it using standard Postgres compatibility layers.

Default is `on`, which means PGAA will attempt a `DirectScan` first before failing over to `CompatScan`.

pgaa.direct_scan_fail_behavior

Defines the system's reaction when a `Direct Scan` is attempted but cannot be completed due to unsupported SQL features or syntax.

Supported values are:

- `ignore`: The system silently falls back to `CompatScan`.
- `warn` (default): The query executes using the compatibility fallback, but Postgres issues a warning notice indicating that `DirectScan` failed.
- `error`: The query execution is aborted entirely if it cannot use `DirectScan`.

Behavior if `DirectScan` was requested but failed, one of `ignore`, `warn`, `error`. Default is `warn`.

pgaa.enable_join_pushdown

In `CompatScan`, controls the optimization strategy for join operations between analytical tables. It determines whether joins are processed locally within Postgres or offloaded to the remote execution engine (Seafowl or Spark).

Default is `on`.

pgaa.enable_groupby_pushdown

In `CompatScan`, attempt to offload aggregation operations (like `SUM`, `COUNT`, `AVG`) and (`GROUP BY`) to the remote executor engine (Seafowl or Spark).

Default is `on`.

pgaa.ctas_intermediate_flush_size

Controls the batching behavior during large `INSERT` or `CREATE TABLE AS` (CTAS) operations into PGAA-managed tables.

Default is 8192.

pgaa.flush_task_interval_s

Specifies the frequency, in seconds, at which the PGAA background process checks for and executes pending flushes of analytical data to persistent storage.

Default is 5.

pgaa.max_in_memory_mb

Sets the maximum amount of memory, in megabytes, that the writer can utilize to buffer data before a mandatory flush to storage is triggered.

Default is 3072 (MB).

pgaa.scan_aggregation_cost_factor

A multiplier used by the Postgres query planner to weight the cost of performing aggregations on the remote backend (Spark or Seafowl) compared to local Postgres execution.

Default is 0.005.

pgaa.scan_per_tuple_cost

A multiplier used by the Postgres query planner to weight the cost of scanning a single row (tuple) on the remote backend (Spark or Seafowl) compared to local Postgres execution.

Default is 0.02.

pgaa.scan_startup_cost

A multiplier used by the Postgres query planner to weight the fixed “startup” cost of initiating a scan on the remote backend (Spark or Seafowl) compared to local Postgres execution.

Default is 25.

pgaa.use_seafowl_cost_estimates

Determines whether the Postgres query planner should request real-time cost estimates directly from the Seafowl engine instead of relying on the statistics collected internally by Postgres.

Default is `on`.

pgaa.write_lock_timeout_s

Specifies the maximum duration, in seconds, that a process will wait to acquire a write lock before timing out and aborting the operation.

Default is 3.

13.1.3 Seafoal

pgaa.autostart_seafoal

Controls automatic start of the Seafoal process managed by the background worker.

Default is `on`.

pgaa.autostart_seafoal_max_memory_mb

Sets the upper limit in megabytes on the amount of RAM the embedded Seafoal engine is permitted to consume, ensuring that the analytical engine does not starve the main Postgres process of resources.

Default is 2147483647.

pgaa.autostart_seafoal_enable_metrics

Enables Prometheus metrics on the Seafoal process by opening a dedicated HTTP endpoint that serves real-time performance metrics to monitor the health, query throughput, and resource utilization of the analytical engine using standard observability stacks.

Default value is `on`.

pgaa.autostart_seafoal_metrics_host

Metrics endpoint bind host.

When `pgaa.autostart_seafoal_enable_metrics= on`, specifies the hostname or IP address where the Prometheus metrics service will be reachable.

Default is `127.0.0.1`, which points to the local loopback for security, ensuring metrics aren't exposed publicly by default.

pgaa.autostart_seafoal_metrics_port

Defines the specific network port used for the Prometheus metrics service when `pgaa.autostart_seafoal_enable_metrics= on`.

Supported values range from 1024 to 65535. Default is 9090.

pgaa.autostart_seafoal_enable_object_store_cache

Determines whether to enable the Seafoal object store cache for external storage locations on S3, Azure, or GCP.

Default is `on`.

pgaa.autostart_seafoal_log_level

Specifies the logging verbosity for the background Seafoal process.

Supported values include `error`, `warn`, `info`, `debug`, and `trace`.

Default is `info`.

pgaa.autostart_seafoal_path

Specifies the absolute file path to the Seafoal executable. This tells the PGAA supervisor exactly which binary to launch when starting the background analytical engine. Modification of this parameter is only required for custom configurations where you are running the Seafoal service on a separate machine or container.

Default is an empty string.

pgaa.autostart_seafowl_port

Specifies the TCP port that the background Seafowl process listens on. The PGAA extension uses this port to communicate with the local execution engine.

Default is 47470.

pgaa.autostart_seafowl_tmp_dir

Specifies the directory where the background Seafowl process stores temporary files during query execution.

Default is an empty string.

pgaa.enable_maintenance_worker

Controls whether the PGAA background supervisor launches and manages a dedicated maintenance worker process.

Default is `off`.

pgaa.maintenance_worker_sleep_interval

Defines the duration, in seconds, that the maintenance worker process waits before checking for new tasks when there is currently no pending work to perform.

Default is 30 seconds.

pgaa.seafowl_client_retry_base_ms

Specifies the initial delay, in milliseconds, that the Postgres client waits before retrying a failed request to the Seafowl process. For every subsequent failure, the wait time doubles.

Default is 100.

pgaa.seafowl_url

Specifies the endpoint URL used by the PGAA extension to communicate with the Seafowl service. By default, PGAA manages the Seafowl process automatically on the same host as the database. Modification of this parameter is only required for custom configurations where you are running the Seafowl service on a separate machine or container.

Default is `http://localhost:47470`.

13.1.4 Catalog synchronization

pgaa.metastore_sync_poll_rate_s

Sets the interval, in seconds, for polling the state of the current attached catalog. Setting this to 0 completely disables the automatic background synchronization.

Default is 60 seconds.

pgaa.metastore_sync_stale_duration_s

Defines the threshold at which the local metadata for an attached catalog is considered stale. If the metadata in the local cache is older than this duration, the system prioritizes a refresh to ensure consistency with the data lake.

Default is 60 seconds.

pgaa.enable_metastore_sync_worker

Determines whether PGAA spawns a dedicated background worker to continuously synchronize metadata from a remote provider, such as an Iceberg REST catalog.

Default is `off`.

13.1.5 Caching**pgaa.lakehouse_table_stats_cache_ttl_s**

Controls the duration (in seconds) that physical table sizes are cached to support the `pgaa.lakehouse_table_stats()` function. This prevents frequent requests to the object store for metadata. Setting this to 0 disables statistics caching and forces the function to retrieve data directly from the storage provider.

Default is 300 seconds.

pgaa.autostart_seafowl_object_store_cache_capacity_mb

Specifies the maximum amount of local disk space, in megabytes, that the Seafoal process can use for caching data from storage locations.

Default is 1024.

pgaa.autostart_seafowl_object_store_cache_min_fetch_size_kb

Defines the minimum size of an outbound request sent to the object store when `pgaa.autostart_seafowl_enable_object_store_cache` is enabled.

Default is 1024.

pgaa.autostart_seafowl_object_store_cache_ttl_s

Specifies the Time To Live (TTL) for objects stored in the Seafoal object store cache. This parameter determines how long a cached data chunk remains valid before it is eligible for eviction.

Default is 3600.

13.1.6 Postgres Distributed (PGD) integration**pgaa.max_replication_lag_s**

Maximum allowed lag (seconds) before a flush is forced.

Default is 5.

pgaa.flush_task_interval_s

Frequency (seconds) at which the system checks for flush requirements.

Default is 5.

13.1.7 DataFusion-inherited configurations

Since PGAA uses Apache DataFusion as its underlying execution engine for the Seafoal query engine, PGAA provides several `pgaa.datafusion.*` configuration parameters that allow you to tune the query execution engine.

These settings control low-level behaviors like memory management, execution parallelism, and optimizer rules.

You can check the full list of DataFusion configuration parameters that PGAA incorporates by running:

```
SELECT name, setting, short_desc, context FROM pg_settings WHERE name LIKE %pgaa.data
-->;
```

Refer to the [DataFusion documentation](#) for a comprehensive list of keys, default values, and performance impacts. When referring to the DataFusion documentation, you can translate their documentation keys to the PGAA configuration parameters by prefixing them with `pgaa.`.

13.1.8 Executor engine

`pgaa.executor_engine`

Determines which compute backend is responsible for executing analytical operations.

Supported values are `seafowl`, and `spark_connect`.

Default is `seafowl`.

`pgaa.spark_connect_url`

Defines the connection endpoint for the Spark Connect service. Specifies the URL of the remote Spark Connect service. This setting is mandatory when `pgaa.executor_engine` is set to `spark_connect`.

The value must follow the specific Spark Connect URI format: `sc://<host>:<port>`.

Default value is `NULL`.

`pgaa.spark_connect_extra_config`

When using `pgaa.executor_engine = 'spark_connect'`, provides fine-grained tuning parameters and runtime settings directly to the Spark session as a JSON object. See [Spark Connect configuration](#) for a list of options.

Example

```
SET pgaa.spark_connect_extra_config = {
  "spark.sql.execution.arrow.maxRecordsPerBatch": "65536"
};
```

Default value is `NULL`.

13.1.9 Query behavior

`pgaa.enable_direct_scan`

Controls the query execution strategy for analytical tables. It determines whether PGAA attempts to offload the entire query to the optimized Seafowl engine or processes it using standard PostgreSQL compatibility layers.

Default value is `on`, which means PGAA will attempt a `DirectScan` first before failing over to `CompatScan`.

`pgaa.direct_scan_fail_behavior`

Defines the system's reaction when a `Direct Scan` is attempted but cannot be completed due to unsupported SQL features or syntax.

Supported values are:

- `ignore` (default): The system silently falls back to `CompatScan`.
- `warn`: The query executes using the compatibility fallback, but PostgreSQL issues a warning notice indicating that `DirectScan` failed.

- `error` : The query execution is aborted entirely if it cannot use DirectScan.

Behavior if DirectScan was requested but failed, one of `ignore` , `warn` , `error` . Default is `warn` .

pgaa.enable_join_pushdown

In CompatScan, controls the optimization strategy for join operations between analytical tables. It determines whether joins are processed locally within PostgreSQL or offloaded to the remote execution engine (Seafowl or Spark).

Default value is `on` .

pgaa.enable_groupby_pushdown

In CompatScan, attempt to offload aggregation operations (like `SUM` , `COUNT` , `AVG`) and `(GROUP BY)` to the remote executor engine (Seafowl or Spark).

Default value is `on` .

13.1.10 Seafowl

pgaa.autostart_seafowl

Controls automatic start of the Seafowl process managed by the background worker.

Default value is `on` .

pgaa.autostart_seafowl_max_memory_mb

Sets the upper limit in megabytes on the amount of RAM the embedded Seafowl engine is permitted to consume, ensuring that the analytical engine does not starve the main PostgreSQL process of resources.

Default value is 2147483647.

pgaa.autostart_seafowl_enable_metrics

Enables Prometheus metrics on the Seafowl process by opening a dedicated HTTP endpoint that serves real-time performance metrics to monitor the health, query throughput, and resource utilization of the analytical engine using standard observability stacks.

Default value is `on` .

pgaa.autostart_seafowl_metrics_host

Metrics endpoint bind host.

When `pgaa.autostart_seafowl_enable_metrics= on` , specifies the hostname or IP address where the Prometheus metrics service will be reachable.

Default value is `127.0.0.1` , which points to the local loopback for security, ensuring metrics aren't exposed publicly by default.

pgaa.autostart_seafowl_metrics_port

Defines the specific network port used for the Prometheus metrics service when `pgaa.autostart_seafowl_enable_metrics= on` .

Supported values range from 1024 to 65535 . Default value is 9090 .

13.1.11 Catalog synchronization

`pgaa.metastore_sync_poll_rate_s`

Determines the interval, in seconds, for polling the state of the current attached for catalog refreshes.

Sets the interval, in seconds, for polling the state of attached catalogs. Setting this to 0 completely disables the automatic background synchronization.

Default value is 60 seconds.

`pgaa.metastore_sync_stale_duration_s`

Defines the threshold at which the local metadata for an attached catalog is considered stale. If the metadata in the local cache is older than this duration, the system prioritizes a refresh to ensure consistency with the data lake.

Default value is 60 seconds.

13.1.12 Caching

`pgaa.lakehouse_table_stats_cache_ttl_s`

Controls how long (in seconds) the results of a table statistics lookup are kept in memory. When a query is planned, PostgreSQL needs these stats to choose the most efficient execution path. By caching them, the system avoids making high-latency network requests to the object store for every single query. Setting this to 0 completely disables caching statistics.

Default value is 300 seconds.

13.1.13 DataFusion-inherited configurations

Since PGAA uses Apache DataFusion as its underlying execution engine for the Seafoam query engine, PGAA provides several `pgaa.datafusion.*` configuration parameters that allow you to tune the query execution engine.

These settings control low-level behaviors like memory management, execution parallelism, and optimizer rules.

You can check the full list of DataFusion configuration parameters that PGAA incorporates by running:

```
SELECT name, setting, short_desc, context FROM pg_settings WHERE name LIKE %pgaa.data
↪ %;
```

Refer to the [DataFusion documentation](#) for a comprehensive list of keys, default values, and performance impacts. When referring to the DataFusion documentation, you can translate their documentation keys to the PGAA configuration parameters by prefixing them with `pgaa.`.

13.2 Functions

The reference section is a list of functions available with Analytics Accelerator.

Refer to [PGFS functions for Pipelines](#) for details on how to create storage locations for PGAA.

13.2.1 Table functions

`pgaa.list_analytics_tables()`

Returns a list of all analytics tables managed by PGAA, including their configuration, storage statistics, and current replication status.

Synopsis

```
SELECT * FROM pgaa.list_analytics_tables();
```

Parameters

None.

Returns

Column	Type	Description
<i>nspoid</i>	<i>OID</i>	The OID of the namespace (schema) containing the table.
<i>reloid</i>	<i>OID</i>	The OID of the table relation.
<i>schema_name</i>	<i>TEXT</i>	Name of the Postgres schema.
<i>table_name</i>	<i>TEXT</i>	Name of the analytics table.
<i>format</i>	<i>pgaa.table_format</i>	The storage format (<i>delta</i> , <i>iceberg</i> , or <i>parquet</i>).
<i>object_storage_snapshot_size_bytes</i>	<i>BIGINT</i>	The size in bytes of the latest table snapshot in object storage.
<i>object_storage_total_size_bytes</i>	<i>BIGINT</i>	The total size in bytes of all versions/files for this table in object storage.
<i>replication_status</i>	<i>pgaa.replication_status</i>	The current state of data movement (<i>enabled</i> , <i>disabled</i> , or <i>initial_offload</i>). Requires PGD integration.
<i>storage_location_name</i>	<i>TEXT</i>	The name of the PGFS storage location associated with the table.
<i>storage_location_path</i>	<i>TEXT</i>	The path within the storage location where the data resides.
<i>catalog_name</i>	<i>TEXT</i>	Name of the external catalog, if applicable.
<i>catalog_namespace</i>	<i>TEXT</i>	The namespace within the external catalog.
<i>catalog_table</i>	<i>TEXT</i>	The table name as defined in the external catalog.

A table can either have a storage location or a catalog. It cannot have both.

The catalog details refer to either the replication target, if performing a catalog-managed offload or replication to object storage, or to the source catalog location, if [Integrating with Iceberg catalogs](#).

pgaa.lakehouse_table_stats()

Returns storage statistics for a specific analytical table, including the size of the latest active snapshot and the total cumulative size of all data versions (historical data and logs) stored in object storage.

Synopsis

```
SELECT * FROM pgaa.lakehouse_table_stats(table_name::regclass);
```

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the analytical table to investigate.

Returns

Column	Type	Description
<i>latest_snapshot_size</i>	<i>BIGINT</i>	The size in bytes of the latest active snapshot of the table.
<i>total_size</i>	<i>BIGINT</i>	The total size in bytes of all files associated with the table in object storage, including metadata, transaction logs, and historical snapshots.

pgaa.test_storage_location()

Tests the connectivity and configuration of a defined storage location, and verifies read or write permissions.

Synopsis

```
SELECT pgaa.test_storage_location(location_name, test_writes:= true);
```

Parameters

Parameter	Type	Description
<i>name</i>	<i>TEXT</i>	The name of the storage location to test.
<i>test_writes</i>	<i>BOOLEAN</i>	If <i>true</i> , the function attempts a write operation to verify permissions. If <i>false</i> , only reads are tested.

Returns

Returns `NULL` if the test is successful. Returns a descriptive error message if the test fails.

pgaa.pgaa_version()

Returns the version information for the EDB Postgres Analytics Accelerator, including the specific build version, build date, and Git commit hash.

pgaa.add_catalog

```
SELECT pgaa.pgaa_version();
```

Parameters

None.

Returns

A string containing the semantic version, the build timestamp, and the unique Git identifier for the current installation.

13.2.2 Catalog functions**pgaa.add_catalog()**

Registers a new Iceberg catalog with PGAA. This function performs an automated connection check to validate credentials and accessibility before the catalog is registered in the system.

Synopsis

```
SELECT pgaa.add_catalog(catalog_name, catalog_type, catalog_options::json);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	A unique name for the catalog within PGAA.
<i>catalog_type</i>	<i>pgaa.catalog_type</i>	The catalog type. Supported values are <i>iceberg-rest</i> (Iceberg REST catalog), and <i>iceberg-s3tables</i> (AWS S3 Tables).
<i>catalog_options</i>	<i>JSON</i>	A JSON object containing the the connection and authentication parameters.

Depending on which storage you use, your JSON file for the `catalog_options` must specify different options.

- For REST catalogs

```
{
  "url": "https://your-catalog-endpoint.com",
  "warehouse_name": "your_warehouse_name",
  "warehouse": "your_warehouse_id",
  "token": "your_secret_auth_token"
  "danger_accept_invalid_certs": "false"
}
```

Where:

```
- `url` : The base HTTP(S) endpoint of the REST catalog service. - `warehouse_name` :
  ↳ A human-readable alternative to the warehouse ID, supported by some REST providers
  ↳ for easier configuration. - `warehouse` : The unique identifier for the specific
  ↳ warehouse within the catalog service. - `danger_accept_invalid_certs` : If set to
  ↳ `true`, Postgres skips SSL certificate validation. Use this only for internal
  ↳ testing or with self-signed certificates; never use it for sensitive public
  ↳ connections.
```

Some Iceberg REST Catalog providers, like Snowflake Open Catalog, might expect different parameters. Specify the following OAuth options:

```
{
  "oauth2.grant_type": "client_credentials",
  "oauth2.client_id": "oauth-client-id",
  "oauth2.client_secret": "oauth-client-secret",
  "oauth2.token_uri": "https://EXAMPLE-polaristest.snowflakecomputing.com/polaris/api/
  ↳ catalog/v1/oauth/tokens",
  "oauth2.scope": "PRINCIPAL_ROLE:read_only"
}
```

Where:

```
- `oauth2.grant_type` : Specifies the OAuth2 flow. The most common value for service-
  ↳ to-service communication is `client_credentials`. - `oauth2.client_id` : The
  ↳ unique identifier for your application registered with the identity provider. -
  ↳ `oauth2.client_secret` : The secret key used to authenticate the client ID. -
  ↳ `oauth2.token_uri` : The full HTTP(S) endpoint where PGAA sends the credentials to
  ↳ request an access token. - `oauth2.scope` : (Optional) A space-delimited list of
  ↳ permissions or resources the client is requesting access to.
```

- For AWS S3 Tables:

```
{
  "arn": "arn:aws:s3tables:us-east-1:1234567890:bucket/my-bucket",
  "region": "us-east-1"
}
```

Where:

```
- `arn` : Specifies the Amazon Resource Name (ARN), the unique identifier for your S3
  ↳ Table bucket. - `region` : Specifies the physical AWS data center location where
  ↳ your S3 Table bucket resides.
```

Returns

Returns the name of the catalog on success.

pgaa.update_catalog()

Updates the configuration options (the JSON object) for an existing Iceberg catalog. Like `pgaa.add_catalog()`, this function performs a validation check to ensure the new connection parameters are functional before applying the changes.

Synopsis

```
SELECT * FROM pgaa.update_catalog(catalog_name, new_options::json);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	A unique name for the catalog within PGAA.
<i>new_options</i>	<i>JSON</i>	A JSON object containing the updated connection and authentication parameters.

See `pgaa.add_catalog()` for a detailed breakdown of the required JSON fields for each catalog type.

Returns

Returns the name of the catalog upon successful update.

Example

Rotate an OAuth2 token:

```
SELECT * FROM pgaa.lakehouse.update_catalog(
  marketing_lakehouse,
  {
    "url": "https://iceberg.example.com",
    "oauth2.client_id": "pgaa_service_user",
    "oauth2.client_secret": "new_secure_secret_2026",
    "oauth2.token_uri": "https://auth.example.com/token"
  }::json
);
```

pgaa.delete_catalog()

Removes a registered catalog from the database. To prevent accidental data loss, this function requires a cascade parameter to be explicitly set to true if there are still tables managed by the catalog.

Synopsis

```
SELECT * FROM pgaa.delete_catalog(catalog_name, cascade := true);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	The name of the catalog to be removed.
<i>cascade</i>	<i>BOOLEAN</i>	If <i>true</i> , PGAA automatically drops all local table definitions managed by this catalog before deleting the catalog entry. Default is <i>false</i> .

Returns

Returns the name of the deleted catalog upon successful completion.

pgaa.list_catalogs()

Returns a list of all registered catalogs in the system, including their connection configuration, metadata synchronization timestamps, and current operational status.

Synopsis

```
SELECT * FROM pgaa.list_catalogs();
```

Parameters

None.

Returns

Column	Type	Description
<i>name</i>	<i>TEXT</i>	The name of the catalog.
<i>type</i>	<i>pgaa.catalog_type</i>	The catalog type. Supported values are <i>iceberg-rest</i> (Iceberg REST catalog), and <i>iceberg-s3tables</i> (AWS S3 Tables).
<i>options</i>	<i>JSON</i>	The connection parameters (URL, ARN, etc.) used for this catalog.
<i>status</i>	<i>pgaa.catalog_status</i>	The current health of the catalog (<i>detached</i> , <i>attached</i> , <i>refresh_retry</i> , or <i>refresh_failed</i>).
<i>created_at</i>	<i>TIMESTAMPTZ</i>	The timestamp when the catalog was first registered.
<i>last_refreshed_at</i>	<i>TIMESTAMPTZ</i>	The last time PGAA successfully synced metadata from this catalog.

pgaa.import_catalog()

Performs a one-time scan and import of table definitions from a registered Iceberg catalog into Postgres. This function creates the local metadata required for PGAA to query the remote tables. This is a manual, once-off import and does not enable automatic, continuous synchronization.

Synopsis

```
SELECT pgaa.import_catalog(catalog_name, namespace_filter);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	The name of the previously registered catalog to import from.
<i>namespace_filter</i>	<i>VARCHAR</i>	If provided, only tables within this specific Iceberg namespace will be imported. If omitted or <i>NULL</i> , all tables in all namespaces are imported. Default is <i>NULL</i> .

Returns

None.

pgaa.attach_catalog()

Enables continuous metadata synchronization for a previously registered Iceberg catalog (using [pgaa.add_catalog\(\)](#)). Once attached, PGAA automatically monitors the remote catalog for changes and updates the local Postgres metadata accordingly. See [Catalog synchronization](#) for polling rate configuration.

Synopsis

```
SELECT pgaa.attach_catalog(catalog_name);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	The name of the registered catalog to start synchronizing.

Returns

None.

pgaa.detach_catalog()

Stops continuous metadata synchronization for a registered Iceberg catalog and moves it to a `detached` state. If `cascade` is `true`, it drops all tables managed by the catalog before detaching. If `cascade` is `false`, the tables remain in the database but synchronization is halted.

Synopsis

```
SELECT * FROM pgaa.detach_catalog(catalog_name, cascade := true);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	The name of the registered catalog to detach.

continues on next page

Table 11 – continued from previous page

Parameter	Type	Description
<i>cascade</i>	<i>BOOLEAN</i>	Optional. If <i>true</i> , PGAA automatically drops all local table definitions associated with this catalog before detaching. Default is <i>false</i> .

Returns

The function returns the row from the `pgaa.catalog` system table for the catalog being detached.

Column	Type	Description
<i>name</i>	<i>TEXT</i>	The name of detached catalog.
<i>type</i>	<i>pgaa.catalog_type</i>	The catalog type.
<i>status</i>	<i>pgaa.catalog_status</i>	The new status, which will be <i>detached</i> .

pgaa.test_catalog()

Tests the connectivity and configuration of a registered Iceberg catalog. This function verifies that the Postgres instance can communicate with the remote catalog endpoint and, optionally, validates that the provided credentials have write permissions.

Synopsis

```
SELECT pgaa.test_catalog(catalog_name, test_writes:=true);
```

Parameters

Parameter	Type	Description
<i>name</i>	<i>TEXT</i>	The name of the registered catalog to test.
<i>test_writes</i>	<i>BOOLEAN</i>	If <i>true</i> , the function attempts a write operation to the catalog metadata service to verify permissions. If <i>false</i> , only read permissions are tested.

Returns

Returns `NULL` if the test is successful. Returns a descriptive error message if the test fails.

pgaa.list_catalog_tables()

Returns a list of all tables and views available in a registered Iceberg catalog. This function allows you to explore the contents of a remote catalog without having to import or attach the tables to your local database first.

Synopsis

```
SELECT * FROM pgaa.list_catalog_tables(catalog_name, namespace_filter);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>TEXT</i>	The name of the registered catalog to explore.

continues on next page

Table 14 – continued from previous page

Parameter	Type	Description
<i>namespace_filter</i>	<i>TEXT</i>	If provided, specifies the Iceberg namespace (schema) to filter by. If omitted or <i>NULL</i> , all tables in the catalog are returned. Default is <i>NULL</i> .

Returns

Column	Type	Description
<i>schema_name</i>	<i>TEXT</i>	The remote Iceberg namespace/schema.
<i>table_name</i>	<i>TEXT</i>	The name of the table or view.

pgaa.drop_catalog_tables()

Removes all local Postgres table and view definitions that are managed by a specific Iceberg catalog, including any dependent objects if *cascade* is *true*. It does not affect tables in the remote Iceberg catalog.

Synopsis

```
SELECT * FROM pgaa.drop_catalog_tables(catalog_name, cascade := true);
```

Parameters

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	The name of the registered catalog to whose managed tables will be removed.
<i>cascade</i>	<i>BOOLEAN</i>	If <i>true</i> , the function automatically drops objects that depend on these tables. Default is <i>false</i> .

Returns

Column	Type	Description
<i>schema_name</i>	<i>TEXT</i>	The local Postgres schema where the table resided.
<i>table_name</i>	<i>TEXT</i>	The name of the local table that was dropped.

13.2.3 Spark functions**pgaa.spark_sql()**

Executes a Spark SQL query directly on your Postgres cluster via the configured Spark Connect endpoint. This allows you to run Iceberg compaction routines or Spark functions that aren't available in Postgres.

See [Spark procedures](#) for a list of the available procedures.

To run this function, you must set the configuration parameter `pgaa.spark_connect_url` to point to an available Spark Connect service.

Synopsis

For a single catalog:

```
SELECT pgaa.spark_sql(query, catalog_name);
```

For multiple catalogs:

```
SELECT pgaa.spark_sql(query, ARRAY[catalog1, catalog2]);
```

Parameters

Parameter	Type	Description
<i>query</i>	<i>TEXT</i>	The Spark SQL statement to execute.
<i>catalog</i>	<i>TEXT</i> or <i>TEXT[]</i>	A single catalog name, or an array of catalog names to use for the query.

Returns

The result set of the Spark query formatted as a JSON object.

Example

Reduce metadata overhead via the `rewrite_data_files` Spark task:

```
SELECT pgaa.spark_sql($$
    CALL preexisting.system.rewrite_data_files(
        table => "preexisting"."ns-1"."table-1",
        strategy => sort,
        sort_order => value DESC,
        options => map(rewrite-all, true)
    )
$$);
```

pgaa.execute_compaction()

Performs compaction to improve the performance and storage efficiency of an analytical Iceberg table. Requires Spark Connect to be configured via the `pgaa.spark_connect_url` configuration parameter.

Note

The function `pgaa.execute_compaction()` is deprecated in favor of `pgaa.launch_task()`.

Synopsis

```
SELECT pgaa.execute_compaction(table_name::regclass, settings::json);
```

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the analytical table to compact.
<i>settings</i>	<i>JSON</i>	Currently not supported. An optional JSON object defining compaction behavior.

Note

The parameter *settings* is currently not supported. Instead, use *pgaa.spark_sql()* to run Spark queries directly on your cluster.

Returns

None.

Example

```
SELECT pgaa.execute_compaction(lakehouse.inventory_iceberg::regclass);
```

13.2.4 Background task functions

pgaa.launch_task()

Schedules a background maintenance task for an analytical Delta or Iceberg table.

Note

While Delta Lake supports all available task types, Iceberg tables currently only support the *compaction* operation. Furthermore, Iceberg compaction does not support the granular configuration keys available for Delta Lake, except for *dry_run*.

Synopsis

```
SELECT pgaa.launch_task(
    table_name::regclass,
    task_type,
    task_options::jsonb,
    scheduled_at::timestamp
);
```

Parameters

Parameter	Type	Description
<i>table_name</i>	<i>REGCLASS</i>	The name or OID of the analytical table to run the task on.
<i>task_type</i>	<i>TEXT</i>	The maintenance operation: <i>compaction</i> , <i>zorder</i> , <i>vacuum</i> , or <i>purge</i> .
<i>task_options</i>	<i>JSONB</i>	Configuration specific to the task type (see below).
<i>scheduled_at</i>	<i>TIMESTAMP</i>	If provided, the task will wait until this time to execute. Default is <i>NULL</i> .

The values for the JSONB *task_options* depend on each *task_type*. All options are optional except *zorder*, which requires *columns*, and *purge*, which requires both *storage_location* and *path*.

- *compaction*: Merges small files into larger ones to speed up analytical scans. The available *task_options* are:

```
{
  "target_size": 536870912,
  "preserve_insertion_order": true,
  "max_concurrent_tasks": 10,
  "max_spill_size": 2147483648,
  "min_commit_interval": 60,
  "dry_run": false
}
```

Where:

```
- `target_size` (Delta only): Specifies the size of the output files in bytes. -
  ↳ `preserve_insertion_order` (Delta only): Whether to maintain the existing sort
  ↳ order of rows. - `max_concurrent_tasks` (Delta only): Limits the number of
  ↳ parallel tasks the executor can run. - `max_spill_size` (Delta only): Sets the
  ↳ maximum data size in bytes allowed to spill to disk during the process. - `min_
  ↳ commit_interval` (Delta only): Sets the minimum wait time in seconds between
  ↳ committing updates to the Delta log. - `dry_run` : If `true` , calculates and logs
  ↳ the operation, but does not perform compaction.
```

- **zorder** (Delta only): A clustering technique that reorganizes data across multiple columns to improve “data skipping” for queries with filters on those columns. The available `task_options` are:

```
{
  "columns": ["customer_id", "transaction_date"],
  "target_size": 1073741824,
  "preserve_insertion_order": false,
  "max_concurrent_tasks": 4,
  "max_spill_size": 2147483648,
  "min_commit_interval": 30
}
```

Where:

```
- `columns` : (Required) An array of strings representing the columns to be used for
  ↳ Z-ordering. - `target_size` : Specifies the size of the output files in bytes. -
  ↳ `preserve_insertion_order` : Whether to maintain the existing sort order of rows. -
  ↳ `max_concurrent_tasks` : Limits the number of parallel tasks the executor can run. -
  ↳ `max_spill_size` : Sets the maximum data size in bytes allowed to spill to disk
  ↳ during the process. - `min_commit_interval` : Sets the minimum wait time in seconds
  ↳ between committing updates to the Delta log.
```

- **vacuum** (Delta only): Deletes old data files that are no longer referenced by the Delta transaction logs, freeing up space in object storage. The available `task_options` are:

```
{
  "retention_period": "168 hours",
  "dry_run": false,
  "enforce_retention_duration": true
}
```

Where:

```
- `retention_period` : Defines the age at which unreferenced files become eligible
↳for deletion. - `dry_run` : If `true` , calculates and logs the files that would be
↳deleted, but performs no deletions. - `enforce_retention_duration` : If `true` ,
↳the task validates the `retention_period` against the system's global minimum
↳safety limit.
```

- **purge (Delta only):** Explicitly removes data from a specific storage path. The available `task_options` are:

```
{
  "storage_location": "s3_main",
  "path": "archive/2023/temp/"
}
```

Where:

```
- `storage_location` : Required. The name of the storage location. - `path` : The
↳relative directory path or file prefix within the storage location that should be
↳permanently deleted.
```

Returns

A unique task ID for the task.

You can check the task status by querying the `pgaa.background_task` table and the provided task ID.

Examples

- Delete old data files prior to the last 7 days:

```
SELECT pgaa.launch_task(
  sales.transactions,
  vacuum,
  {"retention_period": "7 days", "dry_run": false}::jsonb
);
```

- Perform compaction:

```
SELECT pgaa.launch_task(
  telemetry.logs,
  compaction,
  {
    "target_size": 536870912,
    "max_concurrent_tasks": 2
  }::jsonb
);
```

- Reorganize data rows by clustering on columns `region` and `customer_id`:

```
SELECT pgaa.launch_task(
  crm.customers,
  zorder,
  {
    "columns": ["region", "customer_id"],
    "target_size": 1073741824
  }::jsonb
);
```

13.2.5 Postgres Distributed (PGD) integration functions

`pgaa.convert_to_analytics()`

Converts a PGD-replicated table (HTAP) to the PGAA access method. This function points the table directly to the analytics data residing in object storage (Iceberg or Delta format) that has been synchronized via replication.

Synopsis

```
CALL pgaa.convert_to_analytics(relation_name::regclass);
```

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the PGD-replicated table (HTAP) to convert to analytics.

Returns

None.

Note

To convert a heap analytics table to a PGAA table, you can also run an *ALTER TABLE* command and *SET ACCESS METHOD PGAA*. See [Table options](#) for details.

`pgaa.convert_to_tiered_table()`

Converts a heap table into a PGD-managed tiered table with automatic partitioning and analytics offloading. This function automates the creation of a partitioned staging table, migrates existing data, and configures the lifecycle policies for both hot and cold tiers. See [Implementing tiered tables](#) for details.

Synopsis

```
CALL pgaa.convert_to_tiered_table(
  relation := relation_name::regclass,
  range_partition_column := column_name,
  partition_increment := interval_string,
  analytics_offload_period := interval,
  initial_lower_bound := start_date_string,
  retention_period := interval DEFAULT NULL,
  enable_replication := boolean DEFAULT false,
  minimum_advance_partitions := integer DEFAULT 2,
  maximum_advance_partitions := integer DEFAULT 5,
  drop_after_retention_period := boolean DEFAULT true,
  purge_analytics_target := boolean DEFAULT false
);
```

Note

The column used for range partitioning must be a primary key.

Tables with sequences or foreign key constraints are not supported with tiered tables. You must drop the sequence or constraint before converting to tiered table.

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the heap table to convert to tiered table.
<i>range_partition_column</i>	<i>TEXT</i>	The column used for range partitioning. It must be <i>DATE</i> or <i>TIMESTAMP</i> , and it must also be included in the primary key.
<i>partition_increment</i>	<i>TEXT</i>	The time interval defining the size of each partition, such as '1 month'.
<i>analytics_offload_period</i>	<i>INTERVAL</i>	The threshold age at which a partition is moved from local storage to the cold tier in the data lake.
<i>initial_lower_bound</i>	<i>TEXT</i>	The starting value for the first partition's lower bound.
<i>retention_period</i>	<i>INTERVAL</i>	(Optional) The period after which data is purged from the cold tier. Default is <i>NULL</i> .
<i>enable_replication</i>	<i>BOOLEAN</i>	(Optional) If <i>true</i> , enables real-time replication to object storage for active partitions (as an HTAP table). Default is <i>false</i> .
<i>minimum_advance_partitions</i>	<i>INTEGER</i>	(Optional) Minimum number of future partitions to maintain. Default is 2.
<i>maximum_advance_partitions</i>	<i>INTEGER</i>	(Optional) Maximum number of future partitions to maintain. Default is 5.
<i>drop_after_retention_period</i>	<i>BOOLEAN</i>	(Optional) Whether to drop the partition locally after the retention period expires. Defaults is <i>true</i> .
<i>purge_analytics_target</i>	<i>BOOLEAN</i>	(Optional) If <i>true</i> , any pre-existing tables in the object store directory or catalog entry are deleted. If set to <i>false</i> and pre-existing tables are detected, the command will fail with an error. Supported in PGD 6.3+. Default is <i>false</i> .

Returns

None.

Example

Converts a heap table into a tiered structure where data is partitioned by month, replicated to the cloud for real-time analytics, and automatically offloaded to cold storage once it is older than three months.

```
CALL pgaa.convert_to_tiered_table(
    relation           := public.sensor_readings::regclass,
    range_partition_column := reading_time,
    partition_increment := 1 month,
    analytics_offload_period := 3 months,
    initial_lower_bound  := 2026-01-01,
    retention_period     := 2 years,
    enable_replication   := true,
    minimum_advance_partitions := 3,
    maximum_advance_partitions := 6,
    drop_after_retention_period := true,
```

(continues on next page)

(continued from previous page)

```

    purge_analytics_target      := true
);

```

pgaa.disable_analytics_replication()

Disables PGD replication to object storage for an HTAP table. This function reverts the table from an HTAP state back to a Heap state by stopping the continuous replication stream to object storage. See [Replicating to analytics](#) for details.

Synopsis

```
CALL pgaa.disable_analytics_replication(relation_name::regclass);
```

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the HTAP table for which to disable analytics replication.

Returns

None.

Note

To disable replication for an HTAP table, you can also run an *ALTER TABLE* command and *SET pgd.replicate_to_analytics = FALSE*. See [Table options](#) for details.

pgaa.enable_analytics_replication()

Enables PGD replication for a heap table to object storage for analytics. This function transitions a standard Heap table to an HTAP table by initiating a continuous replication stream to the analytics engine. See [Replicating to analytics](#) for details.

Synopsis

```
CALL pgaa.enable_analytics_replication(
    relation regclass,
    purge_analytics_target boolean DEFAULT false
);
```

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the heap table for which to enable analytics replication.
<i>purge_analytics_target</i>	<i>BOOLEAN</i>	(Optional) If <i>true</i> , purges existing data in the analytics target before starting replication. Requires PGD 6.3 or later. Default is <i>false</i> .

Returns

None.

Note

To enable replication for a heap table, you can also run an *ALTER TABLE* command and *SET pgd.replicate_to_analytics = TRUE*. See *Table options* for details.

pgaa.htap_table_stats_uncached()

Returns statistics for HTAP tables in PGD environments.

Synopsis

```
SELECT * FROM pgaa.htap_table_stats_uncached(
    htap_catalog text,
    htap_catalog_namespace text,
    htap_catalog_table text,
    htap_storage_location text,
    htap_storage_path text,
    htap_format text
);
```

Parameters

Parameter	Type	Description
<i>htap_catalog</i>	<i>TEXT</i>	The name of the HTAP catalog.
<i>htap_catalog_namespace</i>	<i>TEXT</i>	The namespace/schema within the HTAP catalog.
<i>htap_catalog_table</i>	<i>TEXT</i>	The specific table name in the HTAP catalog.
<i>htap_storage_location</i>	<i>TEXT</i>	The URI or identifier for the object storage location.
<i>htap_storage_path</i>	<i>TEXT</i>	The specific path within the storage location where data resides.
<i>htap_format</i>	<i>TEXT</i>	The data format used in storage.

Returns

Column	Type	Description
<i>latest_snapshot_size</i>	<i>BIGINT</i>	The size of the most recent data snapshot in bytes.
<i>total_size</i>	<i>BIGINT</i>	The total cumulative size of all data associated with the table in bytes.

pgaa.list_tiered_tables()

Returns detailed information about PGD-managed tiered tables, including their partition settings, data retention policies, analytics offloading configurations, and storage size metrics.

Synopsis

```
SELECT * FROM pgaa.list_tiered_tables();
```

Parameters

None.

Returns

Column	Type	Description
<i>schema_oid</i>	<i>OID</i>	The internal object identifier for the schema.
<i>table_oid</i>	<i>OID</i>	The internal object identifier for the table.
<i>schema_name</i>	<i>TEXT</i>	Name of the schema containing the tiered table.
<i>table_name</i>	<i>TEXT</i>	Name of the tiered table.
<i>partition_increment</i>	<i>INTERVAL</i>	The interval defined for automatic partitioning.
<i>retention</i>	<i>INTERVAL</i>	The duration for which data is kept before being dropped.
<i>analytics_offload</i>	<i>INTERVAL</i>	The period after which data is offloaded to the analytics engine.
<i>replication_enabled</i>	<i>BOOLEAN</i>	Indicates if replication to analytics is active.
<i>tiered_data_size</i>	<i>BIGINT</i>	Total size of data stored in object storage in bytes.
<i>untiered_data_size</i>	<i>BIGINT</i>	Total size of data remaining in local storage in bytes.

pgaa.restore_from_analytics()

Converts a PGAA analytics table residing in object storage back to the standard PostgreSQL heap access method.

Synopsis

```
CALL pgaa.restore_from_analytics(relation_name::regclass);
```

Parameters

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	The name or OID of the PGAA analytics table to restore to a heap table.

Returns

None.

Note

To convert a PGAA analytics table to heap, you can also run an *ALTER TABLE* command and *SET ACCESS METHOD heap*. See [Table options](#) for details.

13.2.6 Utility functions

pgaa._bench_replication()

Runs a synthetic, pgbench-like replication benchmark to measure the performance of data ingestion into a specified storage location. It calculates the initial data load speed and the sustained transaction throughput.

Synopsis

```
SELECT * FROM pgaa._bench_replication(  
    storage_location text,
```

(continues on next page)

(continued from previous page)

```
format text DEFAULT iceberg,
scale integer DEFAULT 100,
transactions integer DEFAULT 100000,
dry_run boolean DEFAULT false
);
```

Parameters

Parameter	Type	Description
<i>storage_location</i>	<i>TEXT</i>	The name of the PGFS destination storage location where the benchmark data will be written.
<i>format</i>	<i>TEXT</i>	The table format to use for the benchmark. Default is 'iceberg'.
<i>scale</i>	<i>INTEGER</i>	The scale factor for the benchmark, determining the size of the initial dataset. Default is 100.
<i>transactions</i>	<i>INTEGER</i>	The total number of transactions to simulate during the throughput test. Default is 100000.
<i>dry_run</i>	<i>BOOLEAN</i>	If <i>true</i> , the function validates the parameters and connection without performing the actual benchmark. Default is <i>false</i> .

Returns

Column	Type	Description
<i>initial load speed (MB/s)</i>	<i>DOUBLE PRECISION</i>	The rate at which the initial dataset was written to storage.
<i>throughput (TPS)</i>	<i>DOUBLE PRECISION</i>	The sustained number of transactions per second achieved during the benchmark.

13.3 Table options

13.3.1 PGAA table options

You can run `CREATE TABLE`, `ALTER TABLE` statements with the `USING PGAA WITH ...` clause and the following options:

Option	Type	Description
<i>pgaa.storage_location</i>	text	PGFS storage location name.
<i>pgaa.path</i>	text	Path within storage location.
<i>pgaa.format</i>	enum	Table format. Valid values are <i>delta</i> , <i>iceberg</i> , and <i>parquet</i> . (Some might not be supported depending on the feature).
<i>pgaa.managed_by</i>	text	Catalog name for catalog-managed tables.
<i>pgaa.catalog_namespace</i>	text	Namespace/schema in catalog.
<i>pgaa.catalog_table</i>	text	Table name in catalog.

continues on next page

Table 31 – continued from previous page

Option	Type	Description
<code>pgaa.auto_truncate</code>	boolean	Truncate heap data on access method switch to PGAA. Requires PGD integration. See Offloading to analytics for details.
<code>pgaa.purge_data_if_exists</code>	boolean	Purge existing analytics data when running <code>CREATE TABLE AS</code> .
<code>pgaa.tiered_table</code>	boolean	Mark as tiered table. Requires PGD integration. See Implementing tiered tables for details.

Examples

- Creating a catalog-managed table:

```
CREATE TABLE managed_sales_data (
  id BIGINT,
  sale_date TIMESTAMPTZ,
  amount NUMERIC
)
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = managed_sales_data
);
```

Note

If you leave the column definitions empty (`()`), PGAA will automatically discover the schema from the metadata files when you first query the table and will map columns by name to the fields found in the remote Iceberg/Delta metadata or Parquet headers.

- Altering the table from the previous example to point at a PGFS storage location:

```
ALTER TABLE managed_sales_data
SET ACCESS METHOD PGAA,
SET (
  pgaa.format = iceberg,
  pgaa.storage_location = my_lake_data,
  pgaa.path = warehouse_data
);
```

13.3.2 PGD table options

You can run `CREATE TABLE`, `ALTER TABLE` statements for heap or HTAP tables and specify the following PGD options:

Option	Type	Description
<code>pgd.replicate_to_analytics</code>	boolean	Enable or disable continuous replication to object storage. See Replicating to analytics for details.
<code>pgd.purge_analytics_target</code>	boolean	Delete any pre-existing tables from the object store directory or catalog entry. See Implementing tiered tables for details.

Examples

- Alter an existing table to enable continuous replication to object storage:

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = TRUE);
```

13.3.3 Managing access methods

Transition tables between transactional and analytical storage by altering the table's access method. This provides a direct way to manually move data between the local Postgres disk and object storage. See [Offloading to analytics](#) for details.

Switching from HTAP to PGAA

To offload an HTAP table to object storage and reclaim local disk space, change the access method to PGAA. The local data will be removed, leaving the data exclusively in the data lake. This is a manual alternative to the automated lifecycle offloading.

```
ALTER TABLE sales SET ACCESS METHOD PGAA;
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.storage_location = my_lake_data,
  pgaa.path = analytical_data
);
```

Switching from PGAA to heap

To restore an analytical table to a local transactional table, change the access method from PGAA to heap. Postgres will pull the data back from object storage and reconstruct it into local storage pages. Use this when you need to perform high-frequency updates on data that was previously offloaded.

```
ALTER TABLE sales_history SET ACCESS METHOD HEAP;
```

13.4 Data types and definitions

The EDB Analytics Accelerator (PGAA) maps native Postgres data types to optimized columnar formats in the data lake. This page lists the supported types and defines the status codes found in PGAA administrative views.

13.4.1 Data types

PGAA independently manages how data is read into Postgres and how it is written to the data lake (using PGD integration). This architecture maximizes performance while ensuring broad compatibility for complex or unsupported formats.

Reading from analytics

When you query a PGAA table, the engine retrieves data from the lakehouse and attempts to coerce it into the target Postgres type defined in your schema. Native binary data is mapped directly to Postgres types for maximum efficiency, while other types are coerced back into its target type using internal Postgres conversion functions.

Natively Supported	Coerced
<i>BOOLOID</i>	<i>CSTRINGOID</i>
<i>BYTEAOID</i>	<i>TEXTOID</i>
<i>CHAROID</i>	<i>VARCHAROID</i>
<i>INT2OID</i>	<i>BPCHAROID</i>
<i>INT4OID</i>	<i>JSONOID</i>
<i>INT8OID</i>	<i>JSONBOID</i>
<i>OIDOID</i>	<i>UUIDOID</i>
<i>FLOAT4OID</i>	<i>ANYENUMOID</i>
<i>FLOAT8OID</i>	<i>CSTRINGARRAYOID</i>
<i>NUMERICOID</i>	<i>TEXTARRAYOID</i>
<i>DATEOID</i>	<i>VARCHARARRAYOID</i>
<i>TIMESTAMPOID</i>	<i>BPCHARARRAYOID</i>
<i>TIMESTAMPTZOID</i>	<i>UUIDARRAYOID</i>
<i>INTERVALOID</i>	Any other type
<i>BOOLARRAYOID</i>	
<i>INT2ARRAYOID</i>	
<i>INT4ARRAYOID</i>	
<i>INT8ARRAYOID</i>	
<i>FLOAT4ARRAYOID</i>	
<i>FLOAT8ARRAYOID</i>	

Writing to analytics

When replicating data to the data lake (using PGD integration), PGAA utilizes native binary builders for maximum efficiency. If a column type is encountered that is not natively supported by the underlying storage format, PGAA automatically converts it to a UTF-8 text representation to ensure a consistent and reliable write.

Natively supported	Converted to UTF-8 text
<i>BOOLOID</i>	<i>CSTRINGOID</i>
<i>BYTEAOID</i>	<i>TEXTOID</i>
<i>CHAROID</i>	<i>VARCHAROID</i>
<i>INT2OID</i>	<i>BPCHAROID</i>
<i>INT4OID</i>	<i>JSONOID</i>
<i>INT8OID</i>	<i>JSONBOID</i>
<i>OIDOID</i>	<i>UUIDOID</i>
<i>FLOAT4OID</i>	<i>ANYENUMOID</i>
<i>FLOAT8OID</i>	<i>CSTRINGARRAYOID</i>
<i>TIMESTAMPOID</i>	<i>TEXTARRAYOID</i>
<i>TIMESTAMPTZOID</i>	<i>VARCHARARRAYOID</i>
<i>DATEOID</i>	<i>BPCHARARRAYOID</i>
<i>INTERVALOID</i>	<i>UUIDARRAYOID</i>
<i>NUMERICOID</i>	All other types
<i>BOOLARRAYOID</i>	
<i>INT2ARRAYOID</i>	
<i>INT4ARRAYOID</i>	
<i>INT8ARRAYOID</i>	

continues on next page

Table 34 – continued from previous page

Natively supported	Converted to UTF-8 text
<i>FLOAT4ARRAYOID</i>	
<i>FLOAT8ARRAYOID</i>	

13.4.2 Administrative enums and statuses

The following parameters and values are primarily found in PGAA administrative views. The table lists their description and supported values.

Type	Supported values	Description
<i>pgaa.table_format</i>	<i>delta</i> , <i>iceberg</i> , and <i>parquet</i> .	The storage format of the table in object storage.
<i>pgaa.catalog_type</i>	<i>iceberg-rest</i> (Iceberg REST catalog), and <i>iceberg-s3tables</i> (AWS S3 Tables).	The type of catalog service.
<i>pgaa.catalog_status</i>	<i>detached</i> , <i>attached</i> , <i>refresh_retry</i> , and <i>refresh_failed</i> .	The current synchronization health between Postgres and the catalog.
<i>pgaa.replication_status</i>	<i>disabled</i> , <i>initial_offload</i> , and <i>enabled</i> .	The state of data movement for Tiered Tables.
<i>pgaa.task_status</i>	<i>pending</i> , <i>running</i> , <i>success</i> , and <i>failure</i> .	The lifecycle state of background worker tasks.

13.5 Benchmark datasets

You can use our public S3 buckets in AWS that contains sample benchmarking datasets. They are available through the following endpoints:

- `s3://beacon-analytics-demo-data-ap-east-1-prod`
- `s3://beacon-analytics-demo-data-ap-northeast-1-prod`
- `s3://beacon-analytics-demo-data-ap-northeast-2-prod`
- `s3://beacon-analytics-demo-data-ap-south-1-prod`
- `s3://beacon-analytics-demo-data-ap-south-2-prod`
- `s3://beacon-analytics-demo-data-ap-southeast-1-prod`
- `s3://beacon-analytics-demo-data-ap-southeast-2-prod`
- `s3://beacon-analytics-demo-data-ap-southeast-3-prod`
- `s3://beacon-analytics-demo-data-ca-central-1-prod`
- `s3://beacon-analytics-demo-data-eu-central-1-prod`
- `s3://beacon-analytics-demo-data-eu-south-2-prod`
- `s3://beacon-analytics-demo-data-eu-west-1-prod`
- `s3://beacon-analytics-demo-data-eu-west-2-prod`
- `s3://beacon-analytics-demo-data-us-east-1-prod`
- `s3://beacon-analytics-demo-data-us-east-2-prod`
- `s3://beacon-analytics-demo-data-us-west-2-prod`

You can query tables in these datasets by referencing them with their schema name.

Schema name	Dataset
<i>tpcds_sf_1</i>	TPC-DS, Scale Factor 1
<i>tpcds_sf_10</i>	TPC-DS, Scale Factor 10
<i>tpcds_sf_100</i>	TPC-DS, Scale Factor 100
<i>tpcds_sf_1000</i>	TPC-DS, Scale Factor 1000
<i>tpch_sf_1</i>	TPC-H, Scale Factor 1
<i>tpch_sf_10</i>	TPC-H, Scale Factor 10
<i>tpch_sf_100</i>	TPC-H, Scale Factor 100
<i>tpch_sf_1000</i>	TPC-H, Scale Factor 1000
<i>clickbench</i>	ClickBench, 100 million rows
<i>brc_1b</i>	Billion row challenge

** Note**

The data column *EventData* for the ClickBench dataset is an integer, not dates. You must quote ClickBench column names because they contain uppercase letters, but unquoted identifiers in Postgres are case-insensitive.