



Analytics Accelerator

Release 1.3

EnterpriseDB Corporation

Oct 20, 2025

1	Introduction to EDB Postgres® Analytics Accelerator	1
1.1	Navigating the analytics accelerator	1
1.2	Conceptual foundations	1
1.3	EDB core analytics technologies	2
1.4	Use Anywhere (Manual/Reference)	2
1.5	Use With PGD (Manual/Reference)	2
1.6	Use In Hybrid Manager (Manual/Reference)	2
1.7	How-Tos (Runbook-Aligned)	2
1.8	Where to start	3
1.8.1	Level 100	3
1.8.2	Level 200	3
1.8.3	Level 300	4
2	Getting Started	5
2.1	Analytics Generic Concepts	5
2.1.1	Architectures	5
2.1.2	Key technologies	6
2.1.3	Core concepts	7
2.1.4	Additional concepts	8
2.1.5	Explore next	9
2.2	Analytics Accelerator Concepts	9
2.2.1	EDB's vision for analytics on Postgres	9
2.2.2	Key EDB solutions and technologies	9
2.2.3	Analytics and AI/ML workloads	10
2.2.4	How EDB implements analytics across products	11
2.2.5	Explore next	11
2.3	Analytics Terminology	11
2.3.1	Apache Iceberg	12
2.3.2	Delta Lake	12
2.3.3	Data Lakehouse	12
2.3.4	EDB Postgres Lakehouse Cluster	12
2.3.5	EDB Postgres Distributed (PGD)	12
2.3.6	Tiered Tables	13
2.3.7	PGAA (Analytics Accelerator)	13
2.3.8	PGFS (Postgres File System)	13
2.3.9	AutoPartition	13

2.3.10	BDR Analytics Table	14
2.3.11	Iceberg Catalog	14
2.3.12	Lakekeeper	14
2.3.13	Open Table Formats	14
2.3.14	Vectorized Query Engine	15
2.3.15	Data Tiering	15
2.3.16	Separation of Storage and Compute	15
2.3.17	Next steps	15
2.4	Analytics Accelerator Architecture	15
2.4.1	Architecture Overview	15
2.4.2	Core Components	16
2.4.3	Architectural Layers	16
2.4.4	Data Flow Patterns	17
2.4.5	Performance Architecture	18
2.4.6	High Availability and Resilience	18
2.4.7	Security Architecture	19
2.4.8	Deployment Patterns	19
2.5	Analytics Accelerator Quick Start	19
2.5.1	Introduction	20
2.5.2	Getting started	20
2.5.3	Operating a Lakehouse node	23
2.5.4	Inspect the benchmark datasets	23
2.5.5	Query the benchmark datasets	24
2.6	Reference - EDB Postgres® AI Lakehouse	24
2.6.1	Benchmarking datasets	25
2.6.2	Queries	25
2.6.3	Delta Lake Table tools	26
2.6.4	DirectScan and fallback modes	26
2.6.5	User management	27
2.6.6	Loading data	27
2.6.7	PGAA functions reference	28
2.6.8	PGAA functions reference	28
2.6.9	Supported AWS instances	29
3	Use Anywhere	31
3.1	EDB Postgres Lakehouse Architecture	31
3.1.1	What is EDB Postgres Lakehouse	31
3.1.2	Who Benefits from Lakehouse Architecture	31
3.1.3	How to Deploy Lakehouse Architecture	32
3.1.4	Architecture Components	33
3.1.5	Performance Characteristics	33
3.1.6	Integration Patterns	34
3.1.7	Operational Considerations	35
3.2	Apache Iceberg Integration With Analytics Accelerator	35
3.2.1	What is Apache Iceberg	35
3.2.2	Who Uses Iceberg Integration	36
3.2.3	How to Use Iceberg with Analytics Accelerator	36
3.2.4	Understanding Iceberg Architecture	37
3.2.5	Schema Evolution Capabilities	38
3.2.6	Catalog Integration Patterns	38
3.2.7	Performance Optimization Strategies	39
3.2.8	Common Integration Patterns	39
3.2.9	Operational Considerations	40
3.3	Delta Lake Integration with Analytics Accelerator	41

3.3.1	Understanding Delta Lake in the Lakehouse Context	41
3.3.2	Architecture Overview	41
3.3.3	Delta Lake Table Format	41
3.3.4	Integration Capabilities	41
3.3.5	Practical Applications	42
3.3.6	Performance Optimization	42
3.3.7	Migration Considerations	43
3.3.8	Ecosystem Integration	43
4	Use With PGD	45
4.1	Tiered Tables With Analytics Accelerator	45
4.1.1	What are Tiered Tables	45
4.1.2	Who Benefits from Tiered Tables	45
4.1.3	How Tiered Tables Work	46
4.1.4	Architecture Overview	48
4.1.5	Performance Characteristics	48
4.1.6	Common Implementation Patterns	49
4.1.7	Operational Considerations	49
5	Learning	51
5.1	Analytics Accelerator learning resources	51
5.1.1	Documentation categories	51
5.1.2	Next steps	52
5.2	Analytics Accelerator Learning Paths	52
5.2.1	How to use these learning paths	52
5.2.2	Learning path levels	53
5.2.3	EDB training and certification	54
5.2.4	Additional guidance	54
5.2.5	Next steps	54
5.3	Analytics Accelerator 101 Foundational concepts	54
5.3.1	What you will learn	54
5.3.2	Before you begin	54
5.3.3	Suggested learning path	55
5.3.4	Next steps	55
5.4	Analytics Accelerator 201 Practical application and core solutions	55
5.4.1	What you will learn	55
5.4.2	Before you begin	55
5.4.3	Suggested learning path	55
5.4.4	Next steps	56
5.5	Analytics Accelerator 301 Advanced techniques and optimization	56
5.5.1	What you will learn	56
5.5.2	Before you begin	56
5.5.3	Suggested learning path	56
5.5.4	Next steps	57
5.6	Analytics Accelerator for your role a persona-based guide	57
5.6.1	Quickly find your role	57
5.6.2	Database administrator (DBA)	57
5.6.3	DevOps engineer / site reliability engineer (SRE)	58
5.6.4	Data scientist / data analyst	58
5.6.5	Application developer	59
6	How To Hybrid Manager	61
6.1	Create A Lakehouse Cluster	61
6.1.1	Why create a Lakehouse cluster?	61

6.1.2	Goals	61
6.1.3	Prerequisites	61
6.1.4	Steps	62
6.1.5	What you can do next	63
6.1.6	Related topics	63
7	How To Playbook	65
7.1	How To Get Setup for Lakehouse analytics	65
7.2	Getting setup for Lakehouse analytics	65
7.2.1	Goals	65
7.2.2	Requirements	65
7.2.3	Create a Lakekeeper catalog	67
7.2.4	Identify your bucket URL	67
7.2.5	Sanity checks	68
7.2.6	Next steps	68
7.2.7	Takeaways	68
7.3	How-To Read/Write Without Catalog	68
7.3.1	Prerequisites	69
7.3.2	Setup: Configure Analytics Infrastructure	69
7.3.3	Read-Only Analytics: Access Existing Data	71
7.3.4	Transactional Table Analytics Offload	71
7.3.5	Advanced Table Management Operations	73
7.3.6	AutoPartition with Analytics Offload	74
7.3.7	Troubleshooting and Validation	78
7.4	How-To Integrate with Third-Party Iceberg Catalogs	79
7.4.1	Overview	79
7.4.2	Context and Goals	79
7.4.3	Prerequisites	79
7.4.4	Practical Applications	80
7.4.5	Configuration Instructions	80
7.4.6	Verification and Testing	81
7.4.7	Known Limitations	82
7.4.8	Troubleshooting	82
7.4.9	Next Steps	82
7.5	How-To Lakehouse Read With/Without A Catalog	83
7.5.1	Overview	83
7.5.2	Prerequisites	83
7.5.3	Configuration Approaches	83
7.5.4	AWS S3 Tables Integration	83
7.5.5	Direct Delta Lake Access	84
7.5.6	Direct Iceberg Access	85
7.5.7	Catalog Attachment	86
7.5.8	Query Validation	87
7.5.9	Best Practices	87
7.5.10	Troubleshooting	88
7.5.11	Next Steps	88
7.6	Configure Analytics Storage and Data Tiering with PGAA and PGD	88
7.6.1	Table of Contents	88
7.6.2	Part 7: External Tool Integration	89
7.6.3	Steps Overview	89
7.6.4	Prerequisites	90
7.6.5	Setup: Configure Analytics Infrastructure	90
7.6.6	Read-Only Analytics: Access Existing Data	92
7.6.7	Transactional Table Analytics Offload	94

7.6.8 Advanced Table Management Operations 95

7.6.9 AutoPartition with Analytics Offload 97

7.6.10 Catalog-Based Operations 101

7.6.11 Troubleshooting and Validation 105

Introduction to EDB Postgres® Analytics Accelerator

Use the **Analytics Accelerator (PGAA)** to explore the analytical capabilities built on EDB Postgres®. This accelerator helps you understand core concepts, explore key technologies such as EDB Postgres® Lakehouse, and learn how to implement analytics with EDB Hybrid Manager (HM).

We integrate modern data architectures and open standards with the reliability and flexibility of Postgres to help you unlock valuable insights.

1.1 Navigating the analytics accelerator

The accelerator organizes content into four areas:

- **Conceptual foundations** Build your understanding of analytics principles and EDB’s approach.
- **EDB core analytics technologies** Learn about EDB solutions and technologies that power our analytics offerings.
- **Practical guidance and solutions** Find use cases, persona-based guides, how-to articles, and tutorials.
- **Product-specific implementations** Access documentation for how these analytics capabilities surface and are managed in EDB products, such as EDB Hybrid Manager.

1.2 Conceptual foundations

Understand the principles and strategies behind modern data analytics and EDB’s approach.

- *Analytics Generic Concepts*

Learn about data architectures (Data Warehouse, Data Lake, Lakehouse) and foundational technologies (columnar storage, vectorized engines, and others).

- *Analytics Accelerator Concepts*

Explore EDB’s vision for Postgres® analytics and how EDB leverages core technologies.

Review in-depth explanations of EDB analytical features, design choices, and advanced topics across the sections below.

1.3 EDB core analytics technologies

Learn about EDB's analytics technologies and how they extend Postgres®.

- *EDB Postgres Lakehouse Architecture*

Review the EDB Postgres® Lakehouse solution and its components for enabling analytics on object storage.

- *Apache Iceberg Integration With Analytics Accelerator*

Understand how EDB solutions use Apache Iceberg to manage large analytical datasets.

- *Delta Lake Integration with Analytics Accelerator*

Learn how EDB Postgres® interacts with Delta tables to enable reliable data lakes.

- *Tiered Tables With Analytics Accelerator*

Manage data across storage tiers using EDB Postgres Distributed (PGD) and Lakehouse capabilities to optimize cost and performance.

1.4 Use Anywhere (Manual/Reference)

- Lakehouse overview: *EDB Postgres Lakehouse Architecture*
- Open formats: *Apache Iceberg Integration With Analytics Accelerator* , *Delta Lake Integration with Analytics Accelerator*
- Storage locations: see *PGAA functions reference* for functions and configuration
- Reference: *PGAA functions reference* , *PGAA functions reference* , *Direct scan* , *Inspect the benchmark datasets*

1.5 Use With PGD (Manual/Reference)

- Concepts: *Tiered Tables With Analytics Accelerator*

1.6 Use In Hybrid Manager (Manual/Reference)

- Getting ready: *Getting setup for Lakehouse analytics*
- Provision: *Create A Lakehouse Cluster*
- Catalogs: *Configure Analytics Storage and Data Tiering with PGAA and PGD*
- Interop: *How-To Lakehouse Read With/Without A Catalog*

1.7 How-Tos (Runbook-Aligned)

These guides mirror the runbook flows and code examples.

— Core How-Tos

- How-to *How-To Lakehouse Read With/Without A Catalog*
- How-to *How-To Read/Write Without Catalog*
- How-to *Configure Analytics Storage and Data Tiering with PGAA and PGD*
- How-to *How-To Integrate with Third-Party Iceberg Catalogs*

1.8 Where to start

- Start with *Analytics Generic Concepts* and *EDB Postgres Lakehouse Architecture* to understand core ideas.
- If you're experimenting with external data, use the No Catalog how-tos.
- If you're integrating with PGD/Tiered Tables or catalogs, follow the PGD and Catalog how-tos.

Postgres Lakehouse is built using a number of technologies:

- PostgreSQL
- [Seafoam](#), an analytical database
- [Apache DataFusion](#), the query engine used by Seafoam
- [Delta Lake](#) (and specifically [delta-rs](#)), for implementing the storage and retrieval layer of Delta Tables

1.8.1 Level 100

The most important thing to understand about Postgres Lakehouse is that it separates storage from compute. This design allows you to scale them independently, which is ideal for analytical workloads where queries can be unpredictable and spiky. You wouldn't want to keep a machine mostly idle just to hold data on its attached hard drives. Instead, you can keep data in object storage (and also in highly compressible formats), and only provision the compute needed to query it when necessary.

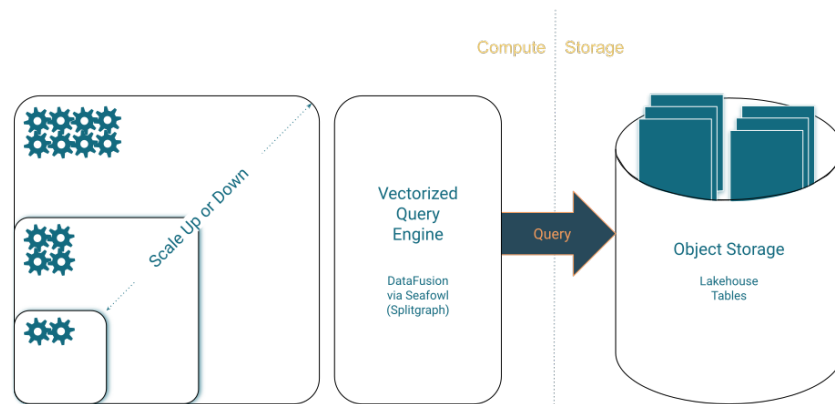


Fig. 1: Level 100 Architecture

On the compute side, a vectorized query engine is optimized to query Lakehouse tables but still fall back to Postgres for full compatibility.

On the storage side, Lakehouse tables are stored using highly compressible columnar storage formats optimized for analytics.

1.8.2 Level 200

Here's a slightly more comprehensive diagram of how these services fit together:

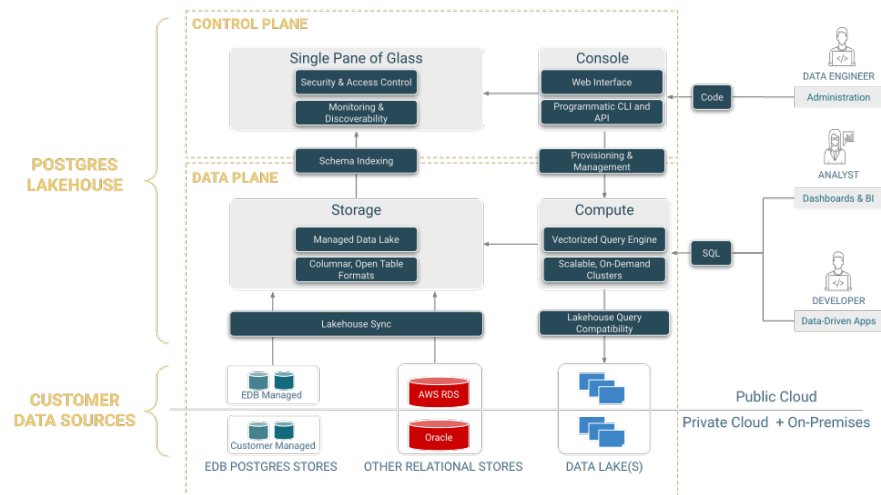


Fig. 2: Level 200 Architecture

1.8.3 Level 300

Here's the more detailed, zoomed-in view of "what's in the box":

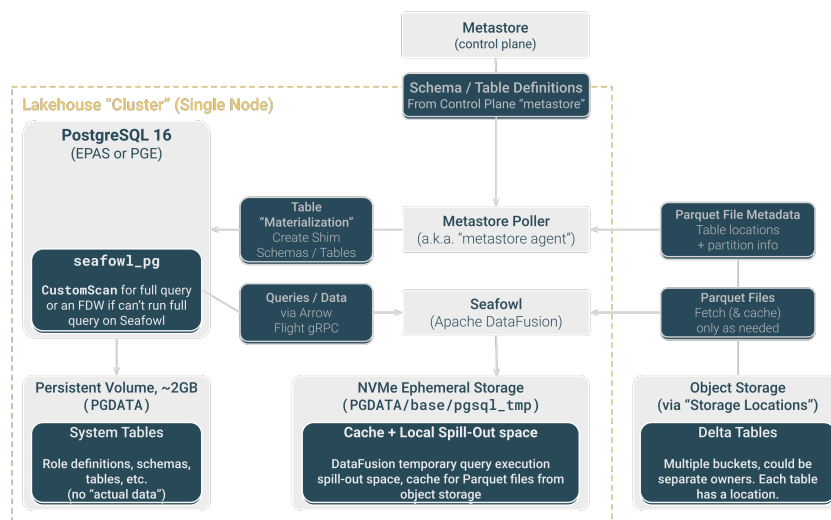


Fig. 3: Level 300 Architecture

2.1 Analytics Generic Concepts

This hub explains key industry concepts, architectures, and technologies that form the foundation of modern data analytics platforms.

This page is part of the Analytics Hub. For full navigation, visit: [Analytics Hub — *Analytics Accelerator Concepts* — *How-Tos \(Runbook-Aligned\)*](#)

Understanding these concepts helps you see how EDB's Analytics Accelerator builds on proven patterns. To learn how EDB implements these ideas, visit the [Analytics Accelerator Concepts](#).

2.1.1 Architectures

Data warehouses

What: Centralized repositories optimized for structured, SQL-based analytics. They use schema-on-write and are designed for fast analytical queries.

Why: Business intelligence teams need trusted, consistent data for reporting and historical analysis.

How: Data is loaded through ETL processes into relational tables with strong typing and constraints.

Why use it: When you need high performance for known, consistent reporting needs and strong governance over data quality.

Learn how this influences [Analytics Accelerator Concepts](#).

Data lakes

What: Flexible repositories for raw data (structured, semi-structured, unstructured). Schema-on-read.

Why: Data lakes let you store all your data at low cost, in its native format.

How: Data is loaded to object storage (e.g., S3, GCS) as files, often in Parquet or ORC format.

Why use it: When you need to capture all available data — even before knowing the full set of use cases — and run flexible processing later.

More on data lakes

Learn how this influences *Analytics Accelerator Concepts* .

Data lakehouse

What: A modern architecture that combines the flexibility of data lakes with the performance and reliability of data warehouses.

Why: Organizations want one system for both data science and BI, without moving large amounts of data.

How: Data is stored in the lake (object storage) using open table formats that add structure, transactions, and metadata.

Why use it: When you want scalable, low-cost storage but also need ACID guarantees and high-performance queries.

Lakehouse architecture paper (Armbrust et al.)

Apache Iceberg

Delta Lake

Apache Hudi

Learn how this influences *Analytics Accelerator Concepts* .

2.1.2 Key technologies

Columnar storage formats

What: Data formats that store columns together rather than rows.

Why: Most analytical queries only access a subset of columns. Reading columns directly improves performance and reduces I/O.

How: Popular formats are [Apache Parquet](#) and [Apache ORC](#) , which support compression and column pruning.

Why use it: When you run large, read-heavy analytical queries and want maximum scan speed.

See how this is used in *Analytics Accelerator Concepts* .

Vectorized query engines

What: Query engines that process data in columnar batches (vectors) instead of row by row.

Why: Columnar batches match CPU cache behavior and SIMD instructions, making analytics much faster.

How: Engines like [Apache DataFusion](#) and [Velox](#) implement vectorized execution.

Why use it: When your analytical workload is CPU-bound and performance matters.

See how this is used in *Analytics Accelerator Concepts* .

Separation of storage and compute

What: Architectures where compute (query engines) and storage (object storage) are decoupled.

Why: This allows you to scale them independently, optimizing cost and performance.

How: Systems like the Analytics Accelerator use object storage (e.g., S3) for data and stateless compute nodes for queries.

Why use it: When your compute needs are bursty or unpredictable, and you want to avoid over-provisioning.

See how this is used in *Analytics Accelerator Concepts* .

Open table formats

What: Formats that bring database features (ACID transactions, schema evolution) to data lakes.

Why: Without an open table format, lake data is a set of loose files. You need structure and governance.

How:

- [Apache Iceberg](#) uses metadata layers and manifest files
- [Delta Lake](#) uses transaction logs (`delta-rs`)
- [Apache Hudi](#) focuses on stream-first updates

Why use it: When you want strong data management for lake-based data.

See how this is used in [Analytics Accelerator Concepts](#) .

2.1.3 Core concepts

OLAP / OLTP

What:

- OLAP: Analytical queries (aggregations, trends) on large datasets
- OLTP: High-volume, small transactions (updates, inserts)

Why: Different workloads have different performance profiles.

How: OLAP is optimized with columnar storage and parallel execution. OLTP uses row-based storage and transactional guarantees.

Why use it: To choose the right architecture for each type of workload.

See how this is used in [Analytics Accelerator Concepts](#) .

ETL / ELT

What:

- ETL: Extract, Transform, Load
- ELT: Extract, Load, Transform (in the lake or warehouse)

Why: ELT takes advantage of modern lakehouses' ability to process raw data at scale.

How: Data is staged in object storage, then transformed using query engines or pipelines.

Why use it: To simplify data ingestion and make pipelines more flexible.

See how this is used in [Analytics Accelerator Concepts](#) .

Data tiering

What: Classifying data into hot, warm, cold based on access patterns.

Why: Not all data needs fast storage — moving cold data to cheaper storage saves cost.

How: Data is physically separated across storage types (fast SSDs, object storage).

Why use it: To optimize storage cost vs. performance.

See how this is used in [Analytics Accelerator Concepts](#) .

Semantic search

What: Search that understands meaning, not just keywords.

Why: Users expect smarter search experiences.

How: Data is represented as vector embeddings and compared using similarity search.

Why use it: To support search applications in AI and modern apps.

See how this is used in *Analytics Accelerator Concepts* .

Vector embeddings

What: Numerical representations of data (text, images, audio) in high-dimensional space.

Why: Enable similarity comparisons that go beyond exact matches.

How: Libraries like [FAISS](#) , [Pinecone](#) , and many ML models generate and use embeddings.

Why use it: For AI/ML search and retrieval tasks.

See how this is used in *Analytics Accelerator Concepts* .

Retrieval augmented generation (RAG)

What: AI pattern where an LLM retrieves documents from an external source before answering.

Why: Makes LLM responses more accurate and grounded in your data.

How: Vector search retrieves relevant embeddings; LLM combines them with the prompt.

Why use it: To improve factual accuracy in AI-driven apps.

[Intro to RAG](#)

See how this is used in *Analytics Accelerator Concepts* .

2.1.4 Additional concepts

Multi-engine interoperability

What: Architecture where multiple query engines can access shared data in the lake.

Why: Organizations want flexibility — different tools for different use cases (SQL engines, data science notebooks, streaming engines).

How: Open table formats (Iceberg, Delta Lake) enable multi-engine access:

- Spark writes data → Trino, Postgres Lakehouse, and other engines can query it.
- PGD offloads data → Lakehouse clusters and external engines can analyze it.

When to use: Build flexible data architectures without vendor lock-in.

Data freshness and latency

What: Ability to query data as soon as it lands in the lake, with minimal delay.

Why: Business users increasingly expect near-real-time access to new data.

How: Lakehouse architectures support this:

- ELT pipelines can continuously update lake data.
- Open table formats support atomic commits and transaction visibility.

- Vectorized Lakehouse engines can query updated data immediately.

When to use: Analytics on operational or streaming data, without requiring full warehouse ETL cycles.

2.1.5 Explore next

- [Analytics Accelerator Concepts](#)
- Analytics How-To Guides
- [Analytics Terminology](#)

2.2 Analytics Accelerator Concepts

This page explains how EDB delivers modern analytics capabilities through the **Analytics Accelerator (PGAA)** and **Hybrid Manager (HM)**.

This page is part of the Analytics Hub. For full navigation, visit: Analytics Hub — [Analytics Accelerator Concepts](#) — [How-Tos \(Runbook-Aligned\)](#)

It builds on established industry patterns and technologies. If you're new to the space, start with:

- [Analytics Terminology](#) — learn the key terms used
- [Analytics Generic Concepts](#) — understand common architectures and patterns

2.2.1 EDB's vision for analytics on Postgres

EDB implements modern analytics patterns through the Analytics Accelerator (PGAA), enabling Postgres to serve as a unified platform for both operational and analytical workloads.

Key principles:

- Perform analytics **close to operational data**, reducing data movement and latency
- Use **open formats** and **vectorized query engines** for fast analytics on object storage
- Support **data tiering** to balance cost and performance
- Separate **compute and storage** to allow independent scaling
- Manage the entire stack via Hybrid Manager

Related concept: [Data lakehouse](#)

2.2.2 Key EDB solutions and technologies

EDB Postgres Lakehouse

Lakehouse architecture enables EDB Postgres to query data in object storage:

- Lakehouse nodes query data in **open table formats** (Iceberg, Delta Lake)
- Vectorized query execution powered by [Apache DataFusion](#)
- Columnar storage formats such as **Parquet**
- Separation of compute and storage

Related concepts:

- *Columnar storage formats*
- *Vectorized query engines*
- *Separation of storage and compute*
- *Open table formats*

Vectorized query optimization

Analytics Accelerator (PGAA) embeds [Apache DataFusion](#) in Lakehouse nodes:

- Processes data as **columnar batches**
- Uses **SIMD** instructions to accelerate performance
- Optimized for **Parquet-formatted data**

Related concept: *Vectorized query engines*

EDB Postgres Distributed (PGD) and Tiered Tables

EDB Postgres Distributed (PGD) powers Tiered Tables:

- **BDR AutoPartition** manages time-based partitioning
- Cold data is automatically **offloaded to object storage** (Iceberg tables)
- Hot data remains in transactional PGD nodes

This balances operational performance with cost-efficient historical data access.

Related concepts:

- *Data tiering*
- *ETL / ELT*

Advanced query optimization

Analytics Accelerator (PGAA) improves core Postgres capabilities for analytics:

- Enhanced parallel query processing
- Optimized join algorithms for analytical workloads
- Tight integration with vectorized engines for high-performance OLAP queries

Related concept: *OLAP / OLTP*

2.2.3 Analytics and AI/ML workloads

Analytics Accelerator supports traditional analytics and AI/ML data pipelines:

- Lakehouse nodes provide efficient access to large datasets used in model training
- Tiered data pipelines support cost-effective AI/ML staging patterns

- Open table formats enable integration with vector search and **retrieval augmented generation (RAG)** workflows

Related concepts:

- *Semantic search*
 - *Vector embeddings*
 - *Retrieval augmented generation (RAG)*
-

2.2.4 How EDB implements analytics across products

Hybrid Manager (HM)

Hybrid Manager acts as the control plane:

- Provision and manage **Lakehouse clusters**
- Configure and monitor **Tiered Tables**
- Manage **storage locations, catalog connections, and compute nodes**

EDB Postgres Advanced Server / Extended Server

EDB Postgres Advanced Server adds:

- Improved parallel query capabilities
- Advanced SQL features helpful for analytics
- Native support for PGAA features

EDB Postgres Distributed (PGD)

PGD provides the foundation for:

- High-availability transactional workloads
 - Data tiering and offloading (Tiered Tables)
 - Scalable architectures that integrate with Lakehouse clusters
-

2.2.5 Explore next

- *Analytics Terminology*
- *Analytics Generic Concepts*
- Analytics How-To Guides
- Analytics in Hybrid Manager

2.3 Analytics Terminology

This page defines key terms used in the Analytics Accelerator (PGAA) and Hybrid Manager (HM) analytics features. Use it as a quick reference when exploring concepts, how-tos, and feature guides.

2.3.1 Apache Iceberg

An open table format for large analytic datasets stored in object storage. Provides schema evolution, time travel, and interoperability with many analytics engines (Spark, Trino, Flink, and Postgres).

Apache Iceberg Integration With Analytics Accelerator

2.3.2 Delta Lake

An open table format and storage layer that adds ACID transactions and reliability to data lakes. Built on Parquet files with a `_delta_log` transaction log.

Delta Lake Integration with Analytics Accelerator

2.3.3 Data Lakehouse

A modern architecture that combines elements of data lakes and data warehouses:

- Stores data in object storage
- Supports open table formats (Iceberg, Delta Lake)
- Provides fast SQL analytics using vectorized query engines
- Decouples storage from compute

Related concept: *Data lakehouse*

2.3.4 EDB Postgres Lakehouse Cluster

A managed analytical compute cluster provisioned by Hybrid Manager to run fast SQL queries on data stored in object storage.

- Supports Iceberg and Delta Lake formats
- Uses a vectorized query engine (Apache DataFusion)
- Separates storage and compute for scalability and cost efficiency

EDB Postgres Lakehouse Architecture

2.3.5 EDB Postgres Distributed (PGD)

An advanced, distributed version of Postgres:

- Provides high availability and multi-master replication
- Enables advanced data tiering patterns (Tiered Tables)
- Works seamlessly with Lakehouse clusters for scalable analytics

Learn more

2.3.6 Tiered Tables

A feature that uses PGD **AutoPartition** to manage large time-based datasets:

- **Hot data** stays in the PGD transactional cluster
- **Cold data** is offloaded to object storage as Apache Iceberg tables
- Queries can transparently access both hot and cold data

Tiered Tables With Analytics Accelerator

2.3.7 PGAA (Analytics Accelerator)

An EDB capability that enables Postgres to:

- Query open table formats (Iceberg, Delta Lake)
- Define **external tables** that map to object storage
- Power Lakehouse cluster queries
- Enable PGD Tiered Table offloading and querying

See how-tos

2.3.8 PGFS (Postgres File System)

An extension that defines **storage locations** Postgres can use to access object storage.

- Used by PGAA and PGD for Iceberg and Delta Lake access
- Supports AWS S3, Google Cloud Storage, and compatible services
- Must be configured on each Lakehouse or PGD cluster that needs to query object storage

See how-tos

2.3.9 AutoPartition

A capability of PGD (Bi-Directional Replication):

- Automatically creates time-based partitions for a PGD table
- Automatically offloads older partitions to object storage when configured with an `analytics_offload_period`

See how-tos

2.3.10 BDR Analytics Table

An internal PGD concept:

- The **analytics_table** view tracks which PGD tables are marked for Tiered Table offload
- Tracks the state of each table and offload progress

Useful for monitoring and validating Tiered Table behavior.

See how-tos

2.3.11 Iceberg Catalog

A metadata service that tracks Iceberg table schemas, locations, and versions:

- Required for full interoperability across multiple engines (Spark, Trino, Postgres)
- Can be HM-managed (Lakekeeper) or external (AWS Glue, Nessie, Polaris, etc.)
- PGAA and PGD both support connecting to an Iceberg REST catalog

See how-tos

2.3.12 Lakekeeper

An HM-managed Iceberg catalog service:

- Provides a central catalog for Iceberg tables
- Supports PGD Tiered Table offload with catalog integration
- Supports Lakehouse cluster queries through catalog-based table discovery
- Uses Iceberg REST Catalog API

See how-tos

2.3.13 Open Table Formats

Standardized file layouts and metadata formats used for analytics:

- **Apache Iceberg**
- **Delta Lake**

Enable multi-engine access to the same data (Spark, Trino, Postgres Lakehouse, etc.).

[Learn more](#)

2.3.14 Vectorized Query Engine

An analytics engine that processes data in **columnar batches** (instead of row-by-row):

- Accelerates analytics queries
- Makes use of modern CPU features (SIMD)
- Powers Lakehouse cluster query performance

EDB Lakehouse clusters embed [Apache DataFusion](#) for this.

2.3.15 Data Tiering

The practice of storing **hot**, **warm**, and **cold** data on different storage tiers:

- Hot → fast transactional PGD nodes
- Cold → object storage (Iceberg tables)
- Tiered Tables automate this pattern in Postgres + HM

Related concept: [Data tiering](#)

2.3.16 Separation of Storage and Compute

A pattern used by Lakehouse architectures:

- Store data in **object storage** (independently scaled)
- Scale analytical compute (Lakehouse clusters) independently of storage

Enables cost-efficient, elastic analytics architectures.

Related concept: [Separation of storage and compute](#)

2.3.17 Next steps

Now that you understand key terminology:

- [Analytics Accelerator Concepts](#)
- [EDB Postgres Lakehouse Architecture](#)
- [Tiered Tables With Analytics Accelerator](#)
- See How-To Guides

2.4 Analytics Accelerator Architecture

2.4.1 Architecture Overview

Analytics Accelerator extends PostgreSQL's capabilities to create a unified platform for transactional and analytical workloads. The architecture combines operational PostgreSQL databases with lakehouse storage patterns, enabling organizations to query data across multiple storage tiers without complex ETL pipelines or data duplication.

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 compatibility with open table formats for ecosystem interoperability. These principles enable cost-effective analytics at petabyte scale while maintaining PostgreSQL's familiar SQL interface and operational characteristics.

2.4.2 Core Components

EDB Postgres Distributed Foundation

EDB Postgres Distributed (PGD) provides the transactional foundation with built-in high availability, horizontal scaling, and automated data lifecycle management. PGD nodes handle operational workloads with full ACID compliance while AutoPartition manages time-based partitioning strategies essential for data tiering.

The distributed architecture ensures zero downtime during maintenance operations and provides geographic distribution for disaster recovery. Write operations maintain consistency through multi-master replication while read operations scale horizontally across available nodes.

Lakehouse Query Engine

Dedicated Lakehouse nodes execute analytical queries against data in object storage. These stateless compute nodes leverage Apache DataFusion's vectorized execution engine to process columnar data formats efficiently. The separation from transactional nodes ensures analytical workloads don't impact operational performance.

The query engine implements sophisticated optimizations including partition pruning, predicate pushdown, and adaptive query execution. These optimizations reduce data scanning by orders of magnitude, enabling interactive query performance on massive datasets.

Storage Abstraction Layer

PostgreSQL File System (PGFS) provides unified access to diverse storage backends including AWS S3, Azure Blob Storage, Google Cloud Storage, and on-premises object stores. This abstraction handles authentication, network resilience, and performance optimization transparently.

The storage layer supports multiple simultaneous storage locations, enabling queries that span cloud providers or combine cloud and on-premises data. Intelligent caching reduces metadata operations and improves query latency for frequently accessed data.

Table Format Integration

Native support for Apache Iceberg and Delta Lake enables interoperability with the broader data ecosystem. Analytics Accelerator (PGAA) reads table metadata to understand schema, partitioning, and file locations without manual configuration.

For Iceberg, the system supports complete functionality including schema evolution, hidden partitioning, and time travel queries. Delta Lake support focuses on read operations, providing access to existing Spark-based data lakes. Both formats maintain compatibility with external tools ensuring data remains accessible across multiple compute engines.

2.4.3 Architectural Layers

Ingestion and Transaction Layer

Applications write data to PostgreSQL or PGD clusters using standard database connections. This layer handles real-time transactions, maintains referential integrity, and ensures data consistency through ACID guarantees.

Streaming ingestion systems deliver continuous data flows while batch loads handle periodic updates. The transaction layer maintains operational performance regardless of historical data volume through intelligent partitioning and tiering strategies.

Storage and Metadata Layer

Data resides in object storage using open table formats that provide structure and governance. Parquet files store actual data while metadata tracks schemas, partitions, and file locations. This separation enables schema evolution and time travel without data rewriting.

Optional catalog services provide centralized metadata management for multi-engine environments. Catalogs enable consistent table discovery, coordinate concurrent modifications, and maintain audit trails across different compute engines.

Query and Transformation Layer

Multiple query engines access the same underlying data based on workload requirements. Analytics Accelerator handles interactive SQL queries, Apache Spark processes complex transformations, and machine learning platforms read training data directly.

Query coordination ensures consistency when multiple engines operate concurrently. Transaction isolation prevents conflicts while snapshot semantics provide repeatable read guarantees essential for analytical accuracy.

Access and Integration Layer

Standard PostgreSQL wire protocol ensures compatibility with existing tools and applications. Business intelligence platforms, data science notebooks, and custom applications connect without specialized drivers or modifications.

REST APIs enable programmatic access for automation and integration scenarios. These APIs support catalog management, monitoring, and administrative operations essential for production deployments.

2.4.4 Data Flow Patterns

Operational to Analytical Pipeline

Data begins its lifecycle in PostgreSQL supporting transactional operations. As data ages and access patterns shift from operational to analytical, Tiered Tables automatically migrate partitions to object storage. This migration occurs transparently with no application changes required.

Recent data remains in PostgreSQL for sub-millisecond query latency while historical data moves to cost-effective object storage. Queries automatically span both tiers, providing complete visibility across all data regardless of storage location.

Direct Lakehouse Access

Organizations with existing data lakes connect Analytics Accelerator directly to Iceberg or Delta Lake tables. This pattern eliminates data movement, enabling immediate analytical capabilities on existing investments.

The system discovers table schemas automatically, translates PostgreSQL SQL to appropriate operations, and returns results through standard database protocols. This approach proves particularly valuable during migrations or when integrating with established data platforms.

Multi-Engine Collaboration

Different engines process data based on their strengths while sharing common storage. Spark handles complex ETL transformations writing results as Iceberg tables. Analytics Accelerator queries these tables for interactive analysis. Machine learning platforms read the same data for model training.

This collaborative approach eliminates data duplication and ensures consistency across analytical workflows. Changes made by one engine become immediately visible to others through shared metadata and snapshot isolation.

2.4.5 Performance Architecture

Query Optimization Pipeline

The optimizer analyzes queries to identify optimization opportunities specific to lakehouse workloads. Partition pruning eliminates irrelevant data before query execution begins. Predicate pushdown moves filters to the storage layer minimizing data transfer.

Cost-based optimization considers network latency, data locality, and format characteristics when generating execution plans. The optimizer adapts strategies based on actual data distribution discovered during execution.

Caching Hierarchy

Multiple caching layers optimize repeated access patterns. Metadata caching stores table schemas, partition information, and file locations. Data caching retains frequently accessed Parquet files in local storage. Query result caching serves identical queries without recomputation.

Cache invalidation respects table format versioning ensuring consistency despite underlying changes. Adaptive cache management prioritizes retention based on access frequency, query cost, and available resources.

Parallel Execution

Queries parallelize across multiple dimensions for maximum throughput. Partition-level parallelism processes independent partitions simultaneously. File-level parallelism reads multiple files within partitions concurrently. Column-level parallelism leverages vectorized execution for columnar operations.

The execution framework dynamically adjusts parallelism based on available resources and query characteristics. This adaptation prevents resource exhaustion while maximizing hardware utilization.

2.4.6 High Availability and Resilience

Transactional Tier Availability

PGD provides continuous availability through multi-master replication. Node failures trigger automatic failover with zero data loss. Geographic distribution enables disaster recovery with configurable recovery objectives.

Maintenance operations including upgrades and schema changes occur without downtime through rolling deployment strategies. The system maintains full functionality during partial failures ensuring business continuity.

Analytical Tier Availability

Stateless Lakehouse nodes provide availability through redundancy rather than replication. Load balancers distribute queries across available nodes with automatic failure detection and rerouting.

Object storage durability guarantees ensure data availability independent of compute failures. Multiple availability zones and cross-region replication provide additional resilience for critical datasets.

Catalog Resilience

Catalog services implement their own availability strategies based on chosen implementation. REST catalogs leverage HTTP load balancing and backend redundancy. AWS S3 Tables provides serverless availability through managed infrastructure.

Catalog unavailability affects table discovery but not query execution for known tables. The architecture gracefully degrades maintaining core functionality during catalog maintenance or failures.

2.4.7 Security Architecture

Access Control Framework

PostgreSQL role-based access control extends to lakehouse resources. Table and column-level permissions apply transparently regardless of storage location. Row-level security policies filter data dynamically based on user context.

Integration with enterprise authentication systems provides single sign-on capabilities. LDAP, Active Directory, and OAuth implementations ensure consistent identity management across the platform.

Encryption Strategy

Comprehensive encryption protects data throughout its lifecycle. Transport Layer Security encrypts data in motion between clients, compute nodes, and storage. Server-side encryption protects data at rest using cloud provider or customer-managed keys.

Column-level encryption provides additional protection for sensitive fields. Encrypted columns remain queryable through specialized indexes while maintaining security compliance.

Audit and Compliance

Detailed audit logging tracks all data access for compliance requirements. Query logs capture user, timestamp, and accessed objects. Storage access logs record all object operations. Authentication logs track login attempts and privilege changes.

Compliance frameworks including GDPR, HIPAA, and PCI DSS guide security implementations. Data residency controls ensure geographic compliance while retention policies automate data lifecycle management.

2.4.8 Deployment Patterns

Managed Service Deployment

Hybrid Manager provides fully managed Analytics Accelerator deployment with automated provisioning, scaling, and maintenance. The platform handles infrastructure management, security updates, and performance optimization.

Self-Managed Deployment

Self-managed deployments provide complete control over infrastructure and configuration. Organizations deploy on-premises, in private clouds, or using infrastructure-as-code approaches.

Kubernetes operators simplify deployment and lifecycle management. Helm charts provide templated configurations for common scenarios. Terraform modules enable infrastructure automation across cloud providers.

Hybrid Deployment

Hybrid patterns combine managed control planes with self-managed data planes. This approach balances operational simplicity with data sovereignty requirements common in regulated industries.

The managed control plane handles orchestration, monitoring, and metadata management. Self-managed data planes maintain data within organizational boundaries while leveraging platform capabilities.

2.5 Analytics Accelerator Quick Start

In this quick start, you will:

1. Create a Lakehouse node.
2. Connect to the node with your preferred Postgres client.

- Query sample data (TPC-H, TPC-DS, Clickbench, or 1BRC) in object storage.

For more details and advanced use cases, see [PGAA functions reference](#).

2.5.1 Introduction

Postgres Lakehouse is a new type of Postgres cluster (it's really just one node) that you can provision with on the EDB Postgres® AI Platform. It includes a vectorized query engine based on Apache [DataFusion](#) for fast queries over columnar Lakehouse tables in object storage (using the [Delta Lake protocol](#)).

What to know

For more details, see [Analytics Accelerator Concepts](#) and [PGAA functions reference](#), but these are the main points to know:

- Postgres Lakehouse nodes are ephemeral.
- Postgres Lakehouse nodes are preconfigured to query benchmark datasets in object storage.
- Postgres Lakehouse is generally read-only but otherwise supports all valid Postgres queries.
- System tables are writable, so you can create users and roles. These tables are stored on the attached block storage, so they persist between instance restarts and are saved as part of backup/restore operations. Otherwise, Lakehouse tables aren't part of backups, since they're ultimately stored in object storage.

Basic architecture

The figure shows what's in the box of a Lakehouse node:

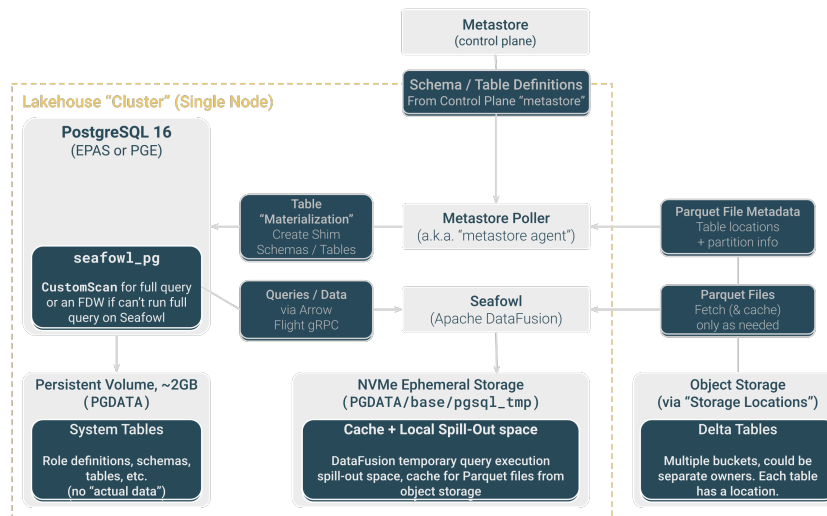


Fig. 1: Level 300 Architecture

2.5.2 Getting started

Once you've logged in to your HCP portal and created a project, you can create a cluster.

Create a Lakehouse node

On your project page, select **Create New > Lakehouse Analytics**:

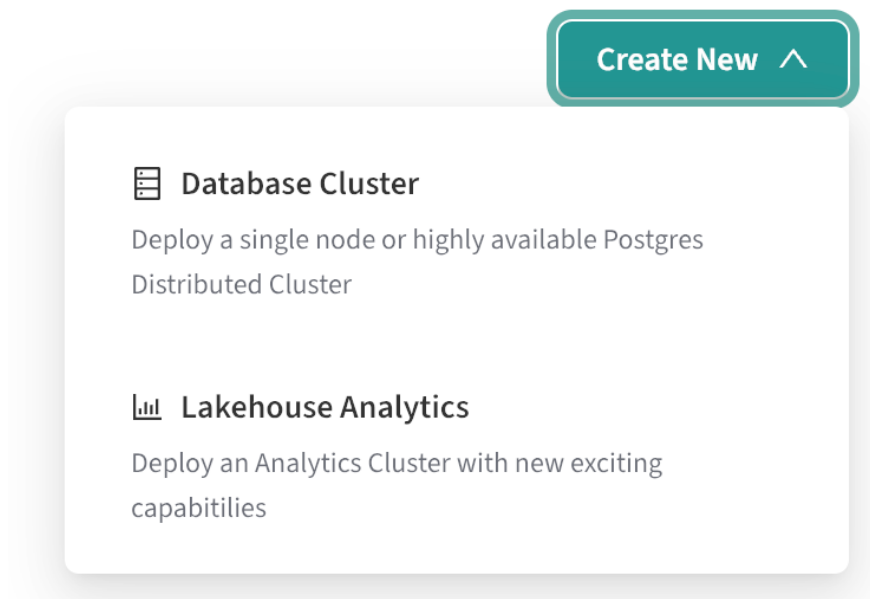


Fig. 2: Create Lakehouse Node Dropdown

The **Create Analytical Cluster** page opens, where you can choose between a templated build or a custom build:

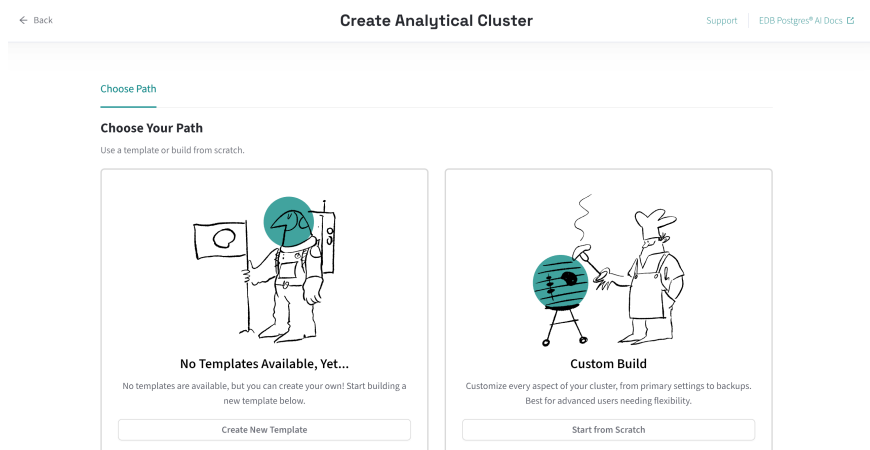


Fig. 3: Create Lakehouse Node Wizard Step 1

Select **Custom Build**, and then select **Start from Scratch**. The **Create Analytical Cluster** page opens, where you can choose the cluster size and configuration:

You then return to the **Clusters** page, where you can see the status of your cluster as it's created. Provisioning the cluster can take 10-15 minutes.

A few notes:

← Back

Create Analytical Cluster

Support | EDB PostgreSQL AI Docs

Cluster Settings

Additional Settings

Cluster Name

Enter a name for your cluster

Password

Passwords must be at least 12 characters.

Enter a password

We do not store your password and cannot retrieve it. If you forget your password, you can simply change it on the "Edit Cluster" page or from Clusters Action menu.

Tags

Add Tag

Deployment Location

Location

managed-devspatcher

Database Type

Please reference [this page](#) for more information on database types.

Postgres Type

PostgreSQL

EDB Postgres Extended Server

EDB Postgres Advanced Server

Postgres Version

Software Image

Select...

Select...

16 is the minimum supported version for Lakehouse clusters

Instance Size

CPU (0.1-15 Cores)

Memory (0.1-10 Gi)

Cores

Gi

Cluster Summary

Cluster Type

Single Node

Cluster Nodes

1 node

Cluster Name

--

Tags

--

Deployment Location

managed-devspatcher

Postgres Type

--

Postgres Version

--

CPU Cores

--

Memory

--

Networking

Private

Backups Retention

30 Days

Scheduled Backup Time

21:49

Fig. 4: Create Lakehouse Node Wizard Step 2

22

Chapter 2. Getting Started

- You must provide your own password. Create a secure password and save it to your password manager. HCP doesn't save passwords.
- You can choose EDB Postgres Advanced Server or EDB Postgres Extended Server.
- All data on the NVMe drives is lost on cluster shutdown.
- Persistent data in system tables (users, roles, and so on) is stored in an attached block storage device and will survive a restart or backup/restore cycle.
- Only Postgres 16 is supported.

2.5.3 Operating a Lakehouse node

Connect to the node

You can connect to the Lakehouse node with any Postgres client in the same way that you connect to any other cluster from EDB Postgres AI Platform. Navigate to the cluster detail page and copy its connection string.

For example, you might copy the `.pgpass` blob into `~/.pgpass` (making sure to replace `$YOUR_PASSWORD` with the password you provided when launching the cluster). Then you can copy the connection string and use it as an argument to `psql` or `pgcli`.

In general, you can connect to the database with any Postgres client. We expect all introspection queries to work, and if you find one that doesn't, then that's a bug.

Understand the constraints

- Every cluster uses EDB Postgres Advanced Server or EDB Postgres Extended Server. So expect to see boilerplate tables from those flavors in the installation when you connect.
- Queryable data (like the benchmarking datasets) is stored in object storage as Delta Tables. Every cluster comes preloaded to point to a storage bucket with benchmarking data inside (TPC-H, TPC-DS, Clickbench) at scale factors from 1 to 1000.
- Only AWS is currently supported. Bring-your-own-account (BYOA) isn't supported.
- You can deploy a cluster in any region that's activated in your EDB Postgres AI account. Each region has a bucket with a copy of the benchmarking data. So when you launch a cluster, it uses the benchmarking data in the location closest to it.
- The cluster is ephemeral. None of the data is stored on the hard drive, except for data in system tables, for example, roles and users and grants. If you restart the cluster or back up the cluster and then restore it, it restores these system tables. But the data in object storage remains untouched.
- The cluster supports read-only queries of the data in object storage, but it supports write queries to system tables for creating users and so on. You can't write directly to object storage. You can't create tables.
- If you want to load your own data into object storage, see [Reference - Bring your own data](#).

2.5.4 Inspect the benchmark datasets

Inspect the benchmark datasets. Every cluster has some benchmarking data available out of the box. If you're using `pgcli`, you can run `\dn` to see the available tables.

The available benchmarking datasets are:

- TPC-H, at scale factors 1, 10, 100, and 1000
- TPC-DS, at scale factors 1, 10, 100, and 1000
- ClickBench, 100 million rows

- 1 Billion Row Challenge

For more details on benchmark datasets, see [Reference - Available benchmarking datasets](#) .

2.5.5 Query the benchmark datasets

You can try running some basic queries:

```
edb_admin> select count(*) from clickbench.hits;
+-----+
| count |
+-----+
| 99997497 |
+-----+
SELECT 1
Time: 0.945s

edb_admin> select count(*) from brc_1b.measurements;
+-----+
| count |
+-----+
| 1000000000 |
+-----+
SELECT 1
Time: 0.651s
```

!!!note

See [PGAA functions reference](#) for more caveats and information about syntax/query compatibility. !!!

2.6 Reference - EDB Postgres® AI Lakehouse

HCP Lakehouse is a powerful tool for running analytical queries on your data. It's built on top of Postgres, so you can use your existing Postgres knowledge and tools.

To get the best experience with Lakehouse, follow the [Analytics Accelerator Quick Start](#) to query benchmarking data. Then you can try loading your own data and [How-To Lakehouse Read With/Without A Catalog](#) .

This section details some of the important things to know:

- [Benchmarking datasets](#)
- [Queries in Lakehouse](#)
- [Delta Table tools](#)
- [DirectScan and fallback modes](#)
- [Users](#)
- [Loading data](#)
- [PGAA functions reference](#)

For information on the PGFS functions for managing storage location, see the AI Factory PGFS documentation .

2.6.1 Benchmarking datasets

When you provision a Lakehouse node, it comes preconfigured to point to a public S3 bucket in AWS us-east-1 that contains sample benchmarking datasets.

Note

Accessing these datasets can incur costs for data transfer and be subject to cross-region latencies.

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 “Notes about ClickBench data”

Data columns (`EventData`) are integers, not dates.

You must quote ClickBench column names because they contain uppercase letters, but unquoted identifiers in Postgres are case-insensitive. For example:

```
❑ select "Title" from clickbench.hits;
❑ select Title from clickbench.hits; !!!
```

2.6.2 Queries

Read-only queries are supported. You can't write directly to object storage. This includes all Postgres built-in functions, statements, and types. It also includes any of those provided by EDB Postgres Advanced Server or EDB Postgres Extended Server, depending on the distribution you choose to deploy.

You can't insert, update, delete, or otherwise modify data. You can use `CREATE TABLE` but only to create a normal Postgres table on the node. The node is ephemeral and will be destroyed when you terminate it. You can also use `CREATE TABLE... USING PGAA` to create an external table that references Delta Tables in S3-compatible object storage.

You must load data into the bucket out-of-band, either with your own ETL scripts or using a third-party ETL provider that supports writing to Delta Tables or Iceberg tables.

See [Loading data](#) for more details.

One exception is Postgres system tables, such as those used for storing users, roles, and grants. These tables are stored on the local block device, which is included in backups and restores. So you can use `CREATE USER`, `CREATE ROLE`, or `GRANT USAGE`, and these users/roles/grants will survive restarts and restores.

2.6.3 Delta Lake Table tools

Creating Delta Lake Tables

Using the `deltalake` Python library

You can use the `deltalake` Python library to create Delta Tables and write to the bucket.

Using the `lakehouse-loader` utility

You can also use the `lakehouse-loader` utility that EDB created for this task to export data from an arbitrary Postgres instance to Lakehouse tables in a storage bucket.

For example, with the `lakehouse-loader` utility:

```
export PGPASSWORD="..."
export AWS_ACCESS_KEY_ID="..."
export AWS_SECRET_ACCESS_KEY="..."

# export other AWS envvars

./lakehouse-loader postgres-to-delta postgres://test-user@localhost:5432/test-db -q
↪ "SELECT * FROM some_table" s3://my-bucket/my_schema/my_table
```

This code exports the data from the `some_table` table in the `test-db` database to a Delta Table in the `my_schema/my_table` path in the `my-bucket` bucket.

You can then query this table in the Lakehouse node by creating an external table that references the Delta Table in the `my_schema/my_table` path. See *How-To Lakehouse Read With/Without A Catalog* for details on how to do this.

2.6.4 DirectScan and fallback modes

DirectScan

Postgres Lakehouse is fastest when it can “push down” your entire query to DataFusion, the vectorized query used for handling queries when possible. (In the future, you will have more fine-grained control as we add support for partial pushdowns.)

Postgres Lakehouse can execute your query in two modes. First, it attempts to run the entire query using Seafoal, a dedicated columnar database based on DataFusion. If Seafoal can’t run the entire query, for example, because it uses PostgreSQL-specific operations like JSON, then Postgres Lakehouse falls back to using the PostgreSQL executor, with Seafoal streaming full table contents to it.

If your query is extremely slow, it’s possible that’s what’s happening.

Using EXPLAIN to check the query plan

You can check which mode is being used by running `EXPLAIN` on the query and making sure that the top-most query node is `DirectScan`. For example:

```
explain select count from (select count(*) from tpch_sf_1.lineitem);

↪ QUERY PLAN
-----
↪
↪
↪
↪
```

(continues on next page)

(continued from previous page)

```

Aggregate  (cost=167.52..167.55 rows=1 width=8)
  -> Append  (cost=0.00..165.01 rows=1001 width=0)
    -> CompatScan on "16529" lineitem_2  (cost=100.00..150.00 rows=1000 width=0)
        SeafowlPlan: logical_plan
          TableScan:  tpch_sf_1.lineitem projection=[l_orderkey, l_partkey, l_
->suppkey, l_linenummer, l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag,
->l_linestatus, l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode,
->l_comment]
(6 rows)

```

In this case, the query is executed by PostgreSQL, and Seafowl is involved only when scanning the table (see `CompatScan` at the bottom). The fix in this case is to explicitly name the inner `COUNT(*)` column, since Seafowl gives it an implicit name `count(*)`, whereas PostgreSQL calls it `count`:

```

edb_admin=> explain select count from (select count(*) as count from tpch_sf_1.
->lineitem);

               QUERY PLAN

-----
DirectScan: logical_plan
  Projection:  COUNT(*) AS count
    Aggregate:  groupBy=[[]], aggr=[[COUNT(UInt8(1)) AS COUNT(*)]]
      TableScan:  tpch_sf_1.lineitem projection=[]
(4 rows)

```

Here, you can see the `DirectScan` at the top, which means that Seafowl is running the entire query.

If you're having trouble rewording your query to make it run fully on Seafowl, open a support ticket.

2.6.5 User management

When you provision a Lakehouse node, you must provide a password. You need the password to log in as the `edb_admin` user. The `edb_admin` user is a superuser account that lets you create users, roles, and grants.

HCP doesn't save this password.

You can either share the credentials for `edb_admin` or use it to create other users and distribute those credentials.

2.6.6 Loading data

Bringing your own data

It's possible to point your Lakehouse node at an arbitrary S3 bucket with Delta Tables inside of it. However, there are some major caveats to doing so. (These will be resolved in future releases.)

Caveats

- The tables must be stored as [Delta Lake Tables](#) within the location.
- A *Delta Lake Table* (or *Delta Table*) is a folder of Parquet files along with some JSON metadata.

Loading data into your bucket

You can use the `lakehouse-loader` utility to export data from an arbitrary Postgres instance to Delta Tables in a storage bucket. See [Delta Lake Table tools](#) for more information on how to get and use that utility.

For more details, see [How-To Lakehouse Read With/Without A Catalog](#).

2.6.7 PGAA functions reference

Synopsis

Lists all PGAA tables on the instance, their storage location, and path.

Parameters

None.

Returns

Column	Type	Description
<i>schema_name</i>	<i>text</i>	The schema of the table
<i>table_name</i>	<i>text</i>	The name of the table
<i>storage_location_name</i>	<i>text</i>	The name of the storage location for the table
<i>path</i>	<i>text</i>	The path to the table in the storage location
<i>managed_by</i>	<i>text</i>	The name of the user who created the table
<i>format</i>	<i>text</i>	The format of the table (currently <i>delta</i> or <i>iceberg</i>)

`pgaa.lakehouse_table_stats()`

Synopsis

Returns the size in bytes of the latest version of a Lakehouse table and the total size of all versions.

Parameters

Parameter	Type	Description
<i>relation</i>	<i>regclass</i>	The name of the table to get the size of

Returns

Column	Type	Description
<i>latest_snapshot_size</i>	<i>bigint</i>	The size in bytes of the latest snapshot of the table
<i>total_size</i>	<i>bigint</i>	The total size in bytes of all versions of the table

2.6.8 PGAA functions reference

`pgaa.create_storage_location`

Synopsis

Creates a new storage location that references an S3-compatible object storage location.

Parameters

Parameter	Type	Description
<i>name</i>	<i>text</i>	The name of the storage location
<i>uri</i>	<i>text</i>	The URI of the S3-compatible object storage location, for example, <i>s3://bucket-name</i> or <i>s3://bucket-name/prefix</i>
<i>options</i>	<i>json</i>	Optional settings for the storage location

Options

Option	Type	Description
<i>access_key_id</i>	<i>text</i>	The access key ID for the storage location
<i>secret_access_key</i>	<i>text</i>	The secret access key for the storage location
<i>session_token</i>	<i>text</i>	The session token for the storage location
<i>region</i>	<i>text</i>	The region for the storage location
<i>endpoint</i>	<i>text</i>	The endpoint for the storage location
<i>bucket</i>	<i>text</i>	The bucket for the storage location
<i>use_http</i>	<i>boolean</i>	Use HTTP instead of HTTPS for the storage location
<i>skip_signature</i>	<i>boolean</i>	Skip signature verification for the storage location

2.6.9 Supported AWS instances

When deploying a Lakehouse node, you must choose an instance type from the `m6id` family of instances. Importantly, these instances come with NVMe drives attached to them.

Instances are ephemeral. These NVMe drives are used only for spill-out space *while processing queries, and for caching Delta Tables on disk. All data on the NVMe drives will be lost when the cluster is shutdown.

System tables are persisted. Persistent data in system tables (users, roles, *etc.) is stored in an attached block storage device, and will survive a pause/resume cycle.

Supported instances

API Name	Memory	vCPUs	Cores	Storage
<i>m6id.large</i>	8.0 GiB	2 vCPUs	1	118 GB NVMe SSD
<i>m6id.xlarge</i>	16.0 GiB	4 vCPUs	2	237 GB NVMe SSD
<i>m6id.2xlarge</i>	32.0 GiB	8 vCPUs	4	474 GB NVMe SSD
<i>m6id.4xlarge</i>	64.0 GiB	16 vCPUs	8	950 GB NVMe SSD
<i>m6id.8xlarge</i>	128.0 GiB	32 vCPUs	16	1900 GB NVMe SSD
<i>m6id.12xlarge</i>	192.0 GiB	48 vCPUs	24	2850 GB (2 * 1425 GB NVMe SSD)
<i>m6id.16xlarge</i>	256.0 GiB	64 vCPUs	32	3800 GB (2 * 1900 GB NVMe SSD)
<i>m6id.24xlarge</i>	384.0 GiB	96 vCPUs	48	5700 GB (4 * 1425 GB NVMe SSD)
<i>m6id.32xlarge</i>	512.0 GiB	128 vCPUs	64	7600 GB (4 * 1900 GB NVMe SSD)

3.1 EDB Postgres Lakehouse Architecture

3.1.1 What is EDB Postgres Lakehouse

EDB Postgres Lakehouse extends PostgreSQL’s analytical capabilities to operate directly on data lake storage, combining the reliability of traditional databases with the scalability and cost-efficiency of modern object storage. This architecture enables organizations to run complex SQL analytics against petabyte-scale datasets stored in S3, Azure Blob Storage, or Google Cloud Storage without moving data into expensive analytical databases.

This page is part of the Analytics Hub. For full navigation, visit: Analytics Hub — [Analytics Accelerator Concepts](#) — [How-Tos \(Runbook-Aligned\)](#)

The lakehouse approach eliminates the traditional separation between data warehouses and data lakes. Instead of maintaining duplicate data across systems, Analytics Accelerator (PGAA) queries data directly where it resides, using open table formats like Apache Iceberg and Delta Lake to provide ACID transactions, schema enforcement, and time travel capabilities on object storage.

3.1.2 Who Benefits from Lakehouse Architecture

Database Administrators leverage lakehouse capabilities to implement cost-effective tiered storage strategies, automatically offloading historical data from expensive PostgreSQL storage to object storage while maintaining full query access.

Data Engineers build simplified data pipelines that write once to object storage, eliminating complex ETL processes between operational and analytical systems. The lakehouse architecture supports both batch and streaming workloads through unified table formats.

Data Analysts access all organizational data through familiar SQL interfaces, querying both operational PostgreSQL data and historical lakehouse tables without understanding underlying storage complexity.

Platform Teams implement unified analytical architectures supporting multiple compute engines. Spark processes complex transformations while Analytics Accelerator serves interactive queries, all accessing the same underlying data.

3.1.3 How to Deploy Lakehouse Architecture

Basic Deployment Pattern

Deploy dedicated Lakehouse nodes through Hybrid Manager or self-managed installations:

```
-- Configure storage location for lakehouse data
SELECT pgfs.create_storage_location(
  lakehouse_storage,
  s3://analytics-bucket/lakehouse,
  {"region": "us-east-1"}
);

-- Create external table over Iceberg data
CREATE TABLE sales_history () USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.storage_location = lakehouse_storage,
  pgaa.path = warehouse/sales.db/transactions
);

-- Query seamlessly across storage tiers
SELECT
  DATE_TRUNC(month, sale_date) as month,
  SUM(amount) as revenue
FROM sales_history
WHERE sale_date >= 2024-01-01
GROUP BY 1;
```

Unified OLTP + OLAP Architecture

Combine operational and analytical workloads in a single platform:

```
-- Hot data in PostgreSQL
CREATE TABLE sales_current (
  sale_id BIGINT PRIMARY KEY,
  sale_date DATE,
  amount DECIMAL(10,2)
);

-- Cold data in lakehouse
CREATE TABLE sales_archive () USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = lakehouse_catalog,
  pgaa.catalog_table = sales_historical
);

-- Unified view across tiers
CREATE VIEW sales_complete AS
SELECT * FROM sales_current
UNION ALL
SELECT * FROM sales_archive;
```

3.1.4 Architecture Components

Stateless Lakehouse Nodes

Lakehouse nodes provide dedicated computational resources for analytical workloads without storing data locally. This stateless design enables elastic scaling based on query demand rather than data volume.

Each node runs the full PostgreSQL stack enhanced with Analytics Accelerator extensions. The familiar PostgreSQL interface ensures compatibility with existing tools and applications while the underlying vectorized engine delivers performance optimized for analytical queries.

Nodes automatically coordinate through shared metadata, enabling parallel query execution across multiple instances. Load balancers distribute queries across available nodes, providing both high availability and horizontal scalability.

Vectorized Query Engine

Apache DataFusion powers the vectorized execution engine, processing columnar data in batches rather than row-by-row. This approach maximizes CPU cache utilization and enables SIMD optimizations that dramatically accelerate analytical operations.

The engine automatically pushes filters and projections to the storage layer, minimizing data transfer. Partition pruning eliminates entire data files from consideration before reading begins. These optimizations combine to deliver query performance approaching specialized analytical databases while maintaining PostgreSQL compatibility.

Adaptive query execution adjusts processing strategies based on actual data characteristics discovered during execution. Initial statistics guide planning, but the engine dynamically optimizes join orders and aggregation strategies as queries progress.

Storage Abstraction Layer

PostgreSQL File System (PGFS) provides unified access to diverse object storage systems. This abstraction layer handles authentication, protocol differences, and performance optimizations transparently.

The system supports multiple storage backends simultaneously, enabling queries that span cloud providers or combine cloud and on-premises storage. Intelligent caching reduces repeated metadata operations while respecting consistency requirements of underlying table formats.

Connection pooling and retry logic handle transient network issues common in cloud environments. The storage layer automatically adjusts request patterns based on observed latency and throughput characteristics.

Table Format Integration

Native support for Apache Iceberg and Delta Lake enables interoperability with the broader data ecosystem. Analytics Accelerator reads table metadata to understand schema, partitioning, and file locations without requiring manual configuration.

For Iceberg tables, the system supports the complete feature set including schema evolution, hidden partitioning, and time travel queries. Catalog integration enables automatic table discovery and metadata synchronization across multiple compute engines.

Delta Lake support currently focuses on read operations, providing access to existing data lake investments. The transaction log parsing occurs transparently, presenting Delta tables as standard PostgreSQL relations.

3.1.5 Performance Characteristics

Query Optimization Strategies

The query optimizer understands lakehouse-specific characteristics when generating execution plans. Statistics from table format metadata guide join ordering and aggregation strategies. The optimizer balances partition pruning effectiveness against metadata operation overhead.

Predicate pushdown extends beyond simple filters to include complex expressions and joins. The system rewrites queries to maximize operations performed at the storage layer, reducing data movement and computational requirements.

Cost-based optimization considers network latency, data locality, and format-specific characteristics. Plans adapt to whether data resides in cache, regional storage, or requires cross-region transfer.

Caching Architecture

Multiple caching layers optimize repeated access patterns common in analytical workloads. Metadata caching eliminates repeated catalog lookups and schema discovery operations. File-level caching stores frequently accessed Parquet files in local SSDs.

Query result caching serves dashboard and reporting workloads where identical queries execute repeatedly. The cache invalidation mechanism respects table format versioning, ensuring consistency despite underlying data changes.

Adaptive cache management prioritizes retention based on access patterns, query cost, and available resources. Hot data remains cached while cold data ages out, maintaining optimal resource utilization.

Scalability Considerations

Horizontal scaling adds nodes to handle increased query concurrency without disrupting running workloads. The stateless architecture ensures new nodes immediately contribute to query processing capacity.

Vertical scaling adjusts individual node resources for complex analytical queries requiring more memory or CPU. The system automatically adjusts parallelism based on available resources, maximizing utilization without causing resource exhaustion.

Storage scaling occurs independently through object storage, supporting virtually unlimited data growth. The separation of compute and storage enables organizations to retain years of historical data without proportional compute costs.

3.1.6 Integration Patterns

Tiered Storage Implementation

Organizations implement tiered storage by combining PostgreSQL operational tables with lakehouse historical tables. Recent data requiring transactional consistency remains in PostgreSQL while aged data migrates to object storage.

Automated migration policies move data based on age, access patterns, or business rules. The migration process preserves data integrity while transforming row-oriented PostgreSQL data into columnar Parquet files optimized for analytical queries.

Transparent query routing directs queries to appropriate tiers based on predicates. Time-based queries automatically union results from both tiers, providing complete analytical views without manual query modification.

Multi-Engine Analytics

The lakehouse architecture supports specialized compute engines for different workload types. Apache Spark handles complex ETL transformations, Analytics Accelerator serves interactive SQL queries, and machine learning platforms access training data directly.

Shared table formats ensure consistency regardless of which engine performs modifications. Transaction isolation prevents conflicts between concurrent operations from different engines. This specialization enables organizations to use the best tool for each job while maintaining a single source of truth.

Streaming Analytics

Modern streaming platforms write continuously to lakehouse tables using micro-batch commits. Analytics Accelerator provides near real-time visibility through snapshot isolation that ensures consistent query results despite ongoing updates.

Event streaming systems like Apache Kafka or Amazon Kinesis deliver data to lakehouse tables for durable storage and analytical processing. The combination of streaming ingestion and SQL analytics enables use cases like real-time fraud detection, operational monitoring, and customer behavior analysis.

3.1.7 Operational Considerations

Deployment Models

Managed Deployments through Hybrid Manager provide automated provisioning, scaling, and maintenance. The platform handles node lifecycle, security updates, and performance optimization transparently.

Self-Managed Deployments offer complete control over infrastructure and configuration. Organizations with specific compliance requirements or existing Kubernetes infrastructure often choose self-managed deployment.

Hybrid Deployments combine managed control plane with self-managed data plane. This approach balances operational simplicity with data sovereignty requirements common in regulated industries.

Monitoring Requirements

Effective monitoring tracks query performance, resource utilization, and storage access patterns. Query execution time, data scanned, and partition pruning effectiveness indicate optimization opportunities.

Storage metrics reveal access patterns that inform partitioning strategies and caching policies. Hot spots in data access suggest candidates for materialized views or alternative partition schemes.

Resource utilization metrics guide capacity planning and scaling decisions. CPU, memory, and network utilization patterns indicate whether workloads are compute, memory, or I/O bound.

Security Architecture

The lakehouse security model implements defense in depth across multiple layers. PostgreSQL role-based access control manages user permissions while object storage IAM policies control data access.

Encryption protects data at rest using cloud provider encryption services or customer-managed keys. Transport layer security encrypts data in motion between compute nodes and storage systems.

Audit logging tracks all data access for compliance and security analysis. Query logs, storage access logs, and authentication events provide complete visibility into system usage.

3.2 Apache Iceberg Integration With Analytics Accelerator

Terminology: Hybrid Manager (HM) is EDB's control plane for provisioning and operating Postgres, analytics, and AI across environments. Analytics Accelerator (PGAA) refers to EDB's analytics capabilities that extend Postgres with lakehouse, tiering, and vectorized query features.

3.2.1 What is Apache Iceberg

Apache Iceberg is an open table format that brings database-like reliability to data lakes built on object storage. Unlike traditional file-based approaches, Iceberg provides ACID transactions, schema evolution, and time travel capabilities while maintaining compatibility with standard formats like Parquet and ORC. Analytics Accelerator (PGAA) leverages Iceberg to enable PostgreSQL queries against petabyte-scale data lakes without moving data from object storage.

This page is part of the Analytics Hub. For full navigation, visit: [Analytics Hub](#) — [Analytics Accelerator Concepts](#) — [How-Tos \(Runbook-Aligned\)](#)

3.2.2 Who Uses Iceberg Integration

Data Engineers implement Iceberg tables for reliable data pipelines that require schema evolution and concurrent read-write operations. They value Iceberg's ability to handle streaming ingestion while maintaining query consistency.

Data Analysts query Iceberg tables through Analytics Accelerator using familiar SQL syntax, accessing historical data through time travel queries without understanding underlying storage complexity.

Platform Teams deploy Iceberg as the foundation for multi-engine analytical environments where Spark, Trino, and PostgreSQL must access the same datasets consistently.

Data Scientists leverage Iceberg's versioning capabilities to reproduce experiments and access training data at specific points in time, ensuring model reproducibility.

3.2.3 How to Use Iceberg with Analytics Accelerator

Basic Setup

Connect Analytics Accelerator to an Iceberg catalog to discover and query existing tables:

```
- - Add Iceberg catalog
SELECT pgaa.add_catalog(
  data_lake,
  iceberg-rest,
  {
    "url": "https://catalog.company.com",
    "warehouse": "analytics",
    "token": "auth_token"
  }
);

- - Attach catalog for querying
SELECT pgaa.attach_catalog(data_lake);

- - Query Iceberg tables directly
SELECT * FROM data_lake.sales.transactions
WHERE transaction_date >= 2024-01-01;
```

Creating External Tables

Define PostgreSQL tables that reference Iceberg data:

```
CREATE TABLE customer_events () USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = data_lake,
  pgaa.catalog_namespace = events,
  pgaa.catalog_table = customer_activity
);
```

Time Travel Queries

Access historical data without maintaining separate copies:

```
- - Query data as of specific timestamp
SELECT COUNT(*) FROM sales_facts
FOR SYSTEM_TIME AS OF 2024-12-31 23:59:59;

- - Compare current with historical state
SELECT
    current.product_id,
    current.price as current_price,
    historical.price as year_end_price
FROM sales_facts current
JOIN sales_facts FOR SYSTEM_TIME AS OF 2024-12-31 historical
    ON current.product_id = historical.product_id
WHERE current.price != historical.price;
```

3.2.4 Understanding Iceberg Architecture

Metadata Structure

Iceberg implements a three-layer metadata architecture that enables its advanced capabilities while maintaining performance at scale.

Metadata Files track the current table state including schema, partitioning, and snapshot history. Each table modification creates a new metadata file, preserving previous versions for time travel.

Manifest Lists organize collections of data files with statistics for partition pruning. Analytics Accelerator uses these statistics to eliminate unnecessary file scanning before query execution begins.

Manifest Files contain column-level statistics for individual data files. These statistics enable predicate pushdown, allowing Analytics Accelerator to skip files that cannot contain matching rows.

Hidden Partitioning

Iceberg abstracts partition complexity from users through hidden partitioning. Queries use natural column values while Iceberg automatically translates these into partition filters:

```
- - User writes simple query
SELECT * FROM orders WHERE order_date = 2024-03-15;

- - Iceberg automatically prunes to day=2024-03-15 partition
- - without user knowing partitioning scheme
```

This abstraction enables partition evolution without query modifications. Organizations can refine partitioning strategies as data volumes grow without breaking existing applications.

Transaction Model

Iceberg implements optimistic concurrency control using snapshot isolation. Writers create new snapshots that become visible atomically, ensuring readers observe consistent table states despite concurrent modifications.

Analytics Accelerator leverages this model to provide repeatable query results. Long-running analytical queries continue accessing their snapshot even as streaming systems write new data, eliminating read-write conflicts.

3.2.5 Schema Evolution Capabilities

Iceberg supports comprehensive schema modifications without rewriting data, exposed through Analytics Accelerator's standard SQL commands.

Safe Schema Changes

Add, rename, or drop columns without touching data files:

```
-- Add column (instant operation)
ALTER TABLE customer_events ADD COLUMN region VARCHAR;

-- Rename column (metadata only)
ALTER TABLE customer_events RENAME COLUMN usr_id TO user_id;

-- Drop column (marks as deleted, data retained)
ALTER TABLE customer_events DROP COLUMN deprecated_field;
```

Type Promotions

Widen data types safely as requirements evolve:

```
-- Promote int to bigint
ALTER TABLE transactions ALTER COLUMN amount TYPE BIGINT;

-- Promote float to double
ALTER TABLE metrics ALTER COLUMN value TYPE DOUBLE PRECISION;
```

3.2.6 Catalog Integration Patterns

Analytics Accelerator supports multiple Iceberg catalog implementations, each suited to different operational requirements.

REST Catalogs

REST catalogs provide vendor-neutral integration through HTTP protocols. AWS Glue, Tabular, and custom implementations expose Iceberg tables through standardized REST APIs:

```
SELECT pgaa.add_catalog(
  aws_catalog,
  iceberg-rest,
  { "url": "https://glue.us-east-1.amazonaws.com",
    "warehouse": "production",
    "credential": "aws_iam" }
);
```

AWS S3 Tables

Native integration with AWS S3 Tables provides serverless Iceberg catalog functionality:

```
SELECT pgaa.add_catalog(
  s3_tables,
  iceberg-s3tables,
  { "arn": "arn:aws:s3tables:us-east-1:123456:bucket/analytics",
```

(continues on next page)

(continued from previous page)

```
"region": "us-east-1"}
);
```

Project Nessie

Nessie adds Git-like branching to Iceberg tables, enabling experimental analyses:

```
- - Query production branch
SELECT * FROM catalog.main.sales_facts;

- - Query experimental branch
SELECT * FROM catalog.experiment_branch.sales_facts;
```

3.2.7 Performance Optimization Strategies

File Organization

Optimal file sizing balances parallelism with I/O efficiency. Target 128-512 MB Parquet files through regular compaction:

```
- - Compact small files for better performance
CALL pgaa.compact_table(catalog.schema.table,
    target_file_size => 256MB);
```

Sort Optimization

Sorting data within files improves compression and enables efficient skipping:

```
- - Z-order by frequently filtered columns
CALL pgaa.zorder_table(catalog.schema.table,
    columns => ARRAY[customer_id, order_date]);
```

Metadata Management

Regular maintenance keeps metadata efficient:

```
- - Expire old snapshots
CALL pgaa.expire_snapshots(catalog.schema.table,
    older_than => CURRENT_TIMESTAMP - INTERVAL 30 days);

- - Remove orphan files
CALL pgaa.remove_orphan_files(catalog.schema.table);
```

3.2.8 Common Integration Patterns

Tiered Storage Architecture

Combine PostgreSQL operational data with Iceberg historical data:

```
- - Unified view across tiers
CREATE VIEW sales_unified AS
SELECT * FROM postgres.sales_current
```

(continues on next page)

(continued from previous page)

```
UNION ALL
SELECT * FROM iceberg.sales_historical;
```

Multi-Engine Processing

Iceberg's open format enables tool specialization:

- **Apache Spark:** Complex ETL transformations
- **Analytics Accelerator:** Interactive SQL queries
- **Trino:** Federated queries across systems
- **Flink:** Stream processing and real-time analytics

Streaming Data Integration

Process streaming data with consistent snapshots:

```
- - Query latest streaming data
SELECT COUNT(*) FROM events
WHERE event_time >= CURRENT_TIMESTAMP - INTERVAL 1 hour;

- - Streaming platforms write continuously
- - Analytics Accelerator queries see consistent snapshots
```

3.2.9 Operational Considerations

Monitoring Requirements

Track key metrics for optimal performance:

- **Snapshot count:** Indicates metadata growth
- **File count per partition:** Identifies compaction needs
- **Metadata operation latency:** Reveals catalog performance
- **Query pruning effectiveness:** Validates partitioning strategy

Capacity Planning

Iceberg separates storage from compute, enabling independent scaling:

- **Storage:** Grows with data volume and snapshot retention
- **Compute:** Scales with concurrent query load
- **Metadata:** Requires periodic maintenance as tables evolve

Migration Strategy

Adopt Iceberg incrementally to minimize risk:

1. **Pilot Phase:** Convert non-critical datasets for validation
2. **Production Migration:** Move high-value analytical tables
3. **Full Adoption:** Standardize on Iceberg for all analytical data

3.3 Delta Lake Integration with Analytics Accelerator

3.3.1 Understanding Delta Lake in the Lakehouse Context

Delta Lake serves as a foundational table format within Analytics Accelerator (PGAA)'s lakehouse architecture, providing ACID transactions and time travel capabilities for large-scale analytical workloads. The integration enables PostgreSQL to directly query Delta Lake tables stored in object storage systems, eliminating the traditional ETL pipeline between operational databases and analytical systems.

This page is part of the Analytics Hub. For full navigation, visit: [Analytics Hub — *Analytics Accelerator Concepts* — *How-Tos \(Runbook-Aligned\)*](#)

Analytics Accelerator's Lakehouse Connector implements native Delta Lake reading capabilities, allowing organizations to leverage existing Delta Lake investments without data migration. This approach maintains compatibility with the broader Spark ecosystem while providing PostgreSQL's familiar SQL interface for analytical queries.

3.3.2 Architecture Overview

The Analytics Accelerator architecture positions Delta Lake tables within the open lakehouse ecosystem, accessible through the Lakehouse Connector component. This connector translates PostgreSQL queries into optimized operations against Delta Lake's Parquet files and transaction logs.

The system architecture separates compute from storage, enabling independent scaling based on workload requirements. Query processing occurs within the Analytics Engine, which implements columnar data optimization specifically designed for Delta Lake's storage format. This separation delivers significant query performance improvements compared to traditional row-based processing while maintaining full SQL compatibility.

3.3.3 Delta Lake Table Format

Delta Lake structures data as Parquet files with an accompanying transaction log stored in the `_delta_log` directory. This transaction log maintains a complete history of all table modifications, enabling point-in-time queries and audit capabilities essential for regulatory compliance.

The format implements optimistic concurrency control, allowing multiple readers to access data while writers perform modifications. This design ensures consistent query results without blocking analytical workloads during data updates. Analytics Accelerator leverages these consistency guarantees to provide reliable query results even as underlying data changes.

File organization within Delta Lake tables follows configurable partitioning schemes that align with query patterns. Common partitioning strategies include date-based partitioning for time-series data or geographic partitioning for regional analytics. Analytics Accelerator's query optimizer automatically identifies relevant partitions, reducing data scanning by orders of magnitude.

3.3.4 Integration Capabilities

Analytics Accelerator provides bidirectional integration with Delta Lake tables, though current production deployments primarily focus on read operations. The Lakehouse Connector handles metadata synchronization, schema discovery, and query translation transparently.

Read Operations

Query execution against Delta Lake tables occurs through standard PostgreSQL SQL syntax. The system automatically discovers table schemas from Delta Lake metadata, eliminating manual schema definition requirements. Column statistics and partition information guide query optimization, ensuring efficient execution plans.

Time travel queries access historical table versions using Delta Lake's transaction log. This capability proves invaluable for regulatory reporting, data recovery, and change analysis. Organizations can query data as it existed at any point in time without maintaining separate historical copies.

Write Operations

While Analytics Accelerator's primary focus remains read optimization, the architecture supports future write capabilities to Delta Lake tables. Current implementations typically use EDB Postgres Distributed (PGD) for operational writes with periodic synchronization to Delta Lake for analytical processing.

3.3.5 Practical Applications

Unified Analytics Platform

Organizations implement Analytics Accelerator with Delta Lake to create unified analytics platforms spanning operational and historical data. Real-time operational data resides in PostgreSQL while historical data lives in cost-effective object storage as Delta Lake tables. Queries transparently access both data sources, providing complete analytical views without data movement.

Financial services organizations use this pattern for risk analytics, combining current trading positions in PostgreSQL with years of historical market data in Delta Lake. The unified query interface enables complex risk calculations that span multiple time horizons without the complexity of federated query systems.

Data Lake Modernization

Existing data lakes built on raw file formats benefit from Delta Lake's transaction consistency and schema enforcement. Analytics Accelerator provides the query layer for these modernized data lakes, enabling SQL access for business analysts familiar with traditional databases.

Retail organizations modernize their customer analytics by converting raw event logs to Delta Lake format. Analytics Accelerator then enables marketing teams to query customer behavior patterns using familiar SQL rather than requiring specialized big data skills.

Streaming Analytics

Delta Lake's support for streaming ingestion combined with Analytics Accelerator's query capabilities enables near real-time analytics on streaming data. Event streams write continuously to Delta Lake while Analytics Accelerator provides consistent query results despite ongoing updates.

Manufacturing companies implement this pattern for production line monitoring, where sensor data streams into Delta Lake tables. Quality control teams query recent production metrics through Analytics Accelerator while historical analysis identifies long-term trends and patterns.

3.3.6 Performance Optimization

Effective Delta Lake integration requires understanding performance characteristics and optimization strategies specific to the format.

Z-Order Optimization

Delta Lake supports Z-ordering, a technique that co-locates related data within Parquet files. This optimization dramatically improves query performance for filters on Z-ordered columns. Analytics Accelerator's query planner recognizes Z-ordered columns and generates execution plans that maximize data locality benefits.

Organizations typically Z-order on high-cardinality columns frequently used in WHERE clauses. Customer ID in transaction tables or timestamp fields in event logs represent common Z-ordering candidates that can reduce query times by 50-80%.

File Compaction

Small files created by streaming ingestion create metadata overhead and reduce read efficiency. Regular compaction consolidates these files into optimally-sized units. Analytics Accelerator's performance improves significantly when Delta Lake tables maintain files between 128-512 MB.

The compaction process runs as a background maintenance task, typically scheduled during low-usage periods. Organizations balance compaction frequency against write workload requirements, with daily compaction suitable for most analytical workloads.

Caching Strategies

Analytics Accelerator implements multiple caching layers for Delta Lake queries. Metadata caching eliminates repeated transaction log parsing, while data caching stores frequently accessed Parquet files locally. These caches operate transparently, automatically invalidating when Delta Lake tables update.

Materialized views provide an additional caching layer for complex aggregations. Rather than repeatedly scanning large Delta Lake tables, materialized views store pre-computed results in PostgreSQL. This approach works particularly well for dashboard queries that aggregate millions of rows into summary statistics.

3.3.7 Migration Considerations

Organizations migrating to Delta Lake from existing data warehouse or data lake implementations should understand key architectural differences and migration strategies.

From Traditional Data Warehouses

Data warehouse migrations typically begin with historical data archival to Delta Lake while maintaining current operational data in existing systems. Analytics Accelerator queries span both systems during the transition period, providing unified access without disrupting existing processes.

The migration process preserves existing SQL queries through Analytics Accelerator's compatibility layer. Query rewriting occurs automatically for most standard SQL operations, though complex warehouse-specific functions may require modification.

From Raw Data Lakes

Raw file formats lacking transaction consistency benefit significantly from Delta Lake conversion. The conversion process adds transaction logs and schema enforcement while preserving existing Parquet data where possible. Analytics Accelerator immediately enables SQL access to converted tables, democratizing data access across the organization.

Organizations typically convert high-value datasets first, validating query performance and data quality before proceeding with broader migration. This incremental approach minimizes risk while delivering immediate value from improved data consistency and query capabilities.

3.3.8 Ecosystem Integration

Analytics Accelerator's Delta Lake support maintains compatibility with the broader lakehouse ecosystem, enabling multi-engine access patterns essential for modern data architectures.

Apache Spark continues to serve data engineering workloads, writing transformed data to Delta Lake tables that Analytics Accelerator queries for analytical processing. This division of responsibilities leverages each engine's strengths while maintaining a single source of truth in Delta Lake.

Business intelligence tools connect to Analytics Accelerator through standard PostgreSQL drivers, accessing Delta Lake data without specialized connectors. This compatibility preserves existing reporting infrastructure investments while extending analytical capabilities to lakehouse data.

Machine learning platforms read training data directly from Delta Lake tables, ensuring consistency between model training and analytical reporting. Analytics Accelerator provides data validation and quality checks through SQL queries, identifying data issues before model training begins.

4.1 Tiered Tables With Analytics Accelerator

4.1.1 What are Tiered Tables

Tiered Tables automatically manage data lifecycle by moving historical partitions from expensive transactional storage to cost-effective object storage while maintaining transparent query access across all data. This capability combines EDB Postgres Distributed (PGD) partitioning with Analytics Accelerator (PGAA) lakehouse integration to optimize both performance and cost for growing datasets.

This page is part of the Analytics Hub. For full navigation, visit: [Analytics Hub](#) — [Analytics Accelerator Concepts](#) — [How-Tos \(Runbook-Aligned\)](#)

The system distinguishes between “hot” data requiring transactional performance and “cold” data accessed primarily for analytical or compliance purposes. Hot data remains in PostgreSQL for optimal OLTP performance while cold data moves to Apache Iceberg or Delta Lake formats in object storage, reducing transactional database size by 70-90% for typical time-series workloads.

4.1.2 Who Benefits from Tiered Tables

Database Administrators implement tiered storage to control database growth without sacrificing data accessibility. They maintain lean, performant transactional systems while ensuring regulatory compliance through complete data retention.

Financial Services Teams use tiering for transaction histories that must be retained for 7-10 years per regulatory requirements but are rarely accessed after initial settlement periods.

IoT Platform Operators manage sensor data that arrives continuously but loses operational relevance after days or weeks, transitioning naturally to analytical workloads.

Retail Organizations tier order histories where recent orders require real-time access for customer service while historical data serves trend analysis and forecasting.

4.1.3 How Tiered Tables Work

Basic Configuration

Enable tiered storage on a time-partitioned table using BDR AutoPartition, then configure analytics offload once partitions exist. This mirrors the runbook flow.

Create a partitioned table and initialize AutoPartition (no offload yet):

```
-- Create a time-partitioned table
CREATE TABLE partitioned_table (
    key BIGINT,
    inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    revenue DOUBLE PRECISION,
    comment TEXT,
    PRIMARY KEY (key, inserted_at)
) PARTITION BY RANGE(inserted_at);

-- Create yearly partitions (no offload yet)
SELECT bdr.autopartition (
    relation := partitioned_table,
    partition_increment := 1 year,
    partition_initial_lowerbound := 2010-01-01 00:00:00,
    managed_locally := TRUE
);

-- Wait for partitions to be created across nodes
SELECT bdr.autopartition_wait_for_partitions_on_all_nodes(partitioned_table::regclass,
    ↪ CURRENT_TIMESTAMP::text);
```

Populate data and then enable offload for cold partitions:

```
-- Insert sample data
INSERT INTO partitioned_table (
    SELECT
        g,
        2010-01-01 00:00:00::timestamp + INTERVAL 1 second *
            (EXTRACT (EPOCH FROM (CURRENT_TIMESTAMP - 2010-01-01 00:00:00::timestamp)) /
    ↪ 1000000 * g),
        g * 0.001,
        encode(sha256(g::text::bytea), hex)
    FROM generate_series(1, 1000000) g
);

-- Reconfigure AutoPartition to offload partitions older than 1 year
SELECT bdr.autopartition (
    relation := partitioned_table,
    partition_increment := 1 year,
    partition_initial_lowerbound := 2010-01-01 00:00:00,
    managed_locally := TRUE,
    analytics_offload_period := 1 year
);
```

Offload Patterns

AutoPartition handles time-based tiering automatically. For manual/no-catalog offload of a whole table, use analytics replication:

```
-- Example: offload an entire transactional table (no catalog)
CREATE TABLE transactional_table (
  key BIGINT,
  inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  revenue DOUBLE PRECISION,
  comment TEXT,
  PRIMARY KEY (key, inserted_at)
);

-- Enable replication to analytics storage
ALTER TABLE transactional_table SET (pgd.replicate_to_analytics = TRUE);

-- Optionally convert to PGAA to read directly from Iceberg
ALTER TABLE transactional_table
SET ACCESS METHOD PGAA,
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.storage_location = appliance-bucket,
  pgaa.path = public.transactional_table
);
```

Partial offload can be achieved by splitting data into a separate table and offloading only that table:

```
BEGIN;
CREATE TABLE transactional_table_before_2020 AS
  SELECT * FROM transactional_table
  WHERE inserted_at < 2020-01-01;
DELETE FROM transactional_table WHERE inserted_at < 2020-01-01;
COMMIT;

ALTER TABLE transactional_table_before_2020 ADD PRIMARY KEY (key, inserted_at);
ALTER TABLE transactional_table_before_2020 SET (pgd.replicate_to_analytics = TRUE);

-- (Optional) convert to PGAA for direct Iceberg reads
ALTER TABLE transactional_table_before_2020
SET ACCESS METHOD PGAA,
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.storage_location = appliance-bucket,
  pgaa.path = public.transactional_table_before_2020
);
```

Restore Operations

Switch a table back to transactional storage and disable analytics replication:

```
ALTER TABLE transactional_table
SET ACCESS METHOD heap,
SET (pgd.replicate_to_analytics = FALSE);
```

4.1.4 Architecture Overview

Storage Tiers

The architecture implements three distinct storage tiers optimized for different access patterns:

Hot Tier (PostgreSQL) stores current operational data requiring sub-millisecond latency and full transactional capabilities. This tier typically contains days to months of recent data depending on workload characteristics.

Cold Tier (Object Storage) holds historical data in columnar formats optimized for analytical queries. Data remains fully queryable but with higher latency acceptable for compliance and analytical use cases.

Query Processing

Tiered Tables leverage PostgreSQL's partition-wise query planning to optimize cross-tier queries. The planner analyzes predicates to determine which partitions require access, automatically pruning irrelevant hot and cold partitions.

When queries span tiers, the system executes parallel subqueries against each storage layer. Hot tier queries execute through standard PostgreSQL processing while cold tier queries utilize Analytics Accelerator's vectorized engine. Results merge transparently, presenting unified datasets to applications.

Time-based predicates prove particularly efficient as they naturally align with partition boundaries. A query for last quarter's data might access one hot partition and two cold partitions, each processed optimally for its storage characteristics.

Metadata Management

The system maintains comprehensive metadata tracking partition locations, offload timestamps, and storage statistics. This metadata enables intelligent query routing and provides visibility into storage utilization.

PGD tracks partition lifecycle states including active, offloading, offloaded, and restoring. State transitions occur atomically, ensuring consistent visibility across cluster nodes. Failed offload operations automatically retry or rollback, maintaining data integrity.

Storage statistics track size reduction achieved through offloading, helping quantify cost savings. Compression ratios typically range from 3:1 to 10:1 depending on data characteristics and chosen columnar format.

4.1.5 Performance Characteristics

Hot Tier Optimization

Removing historical data from transactional storage delivers immediate performance benefits. Smaller tables mean more efficient indexes, faster vacuum operations, and improved cache hit rates. Organizations typically observe 20-40% improvement in transactional query performance after implementing tiering.

The reduced dataset size particularly benefits backup and recovery operations. Smaller databases backup faster, require less storage, and restore more quickly during disaster recovery scenarios.

Cold Tier Performance

Analytical queries against cold data leverage columnar storage optimizations including compression, dictionary encoding, and vectorized processing. While individual row lookups perform slower than PostgreSQL, analytical aggregations often execute faster due to columnar efficiency.

Partition merging addresses metadata overhead when dealing with thousands of historical partitions. Multiple daily partitions can merge into monthly or yearly partitions in object storage, reducing metadata operations while preserving query capabilities.

Cross-Tier Queries

Queries spanning hot and cold tiers exhibit performance characteristics based on data distribution. Queries primarily accessing hot data with minimal cold data filtering maintain near-PostgreSQL performance. Analytical queries aggregating years of cold data with recent hot data benefit from parallel processing across tiers.

The system implements intelligent caching for frequently accessed cold partitions. Metadata caching eliminates repeated discovery operations while data caching stores commonly accessed Parquet files locally.

4.1.6 Common Implementation Patterns

Regulatory Compliance

Financial services organizations implement tiering to meet regulatory retention requirements cost-effectively. For example, transaction data can remain in PostgreSQL for 90 days supporting operational queries, then automatically offload to Iceberg for 7-year retention.

The implementation maintains complete audit trails with timestamp tracking for all tier transitions. Compliance queries access historical data transparently, generating reports spanning entire retention periods without manual data assembly.

Time-Series Analytics

IoT platforms receiving millions of sensor readings daily implement aggressive tiering policies. Raw sensor data older than 7 days offloads to object storage while aggregated metrics remain in PostgreSQL indefinitely.

This pattern optimizes for common query patterns where operational dashboards display recent raw data while historical analytics work with aggregated summaries. The tiering boundary aligns with natural access patterns, minimizing cross-tier queries.

Seasonal Business Cycles

Retail organizations with strong seasonal patterns implement dynamic tiering policies. Order data from peak seasons remains hot longer to support extended return periods and customer service needs.

Off-season data aggressively offloads after 30 days while peak season data retains 90-day hot retention. This adaptive approach balances storage costs with business requirements throughout the year.

4.1.7 Operational Considerations

Monitoring Requirements

Effective tiering requires monitoring partition growth rates, offload success rates, and query patterns across tiers. Dashboards track storage utilization trends, identifying tables benefiting from tiering and validating cost savings.

Query performance monitoring distinguishes between tier-local and cross-tier queries. Increasing cross-tier query frequency might indicate need for adjusted retention policies or materialized views bridging tiers.

Capacity Planning

Tiered architectures require capacity planning for both storage tiers. Hot tier capacity depends on data ingestion rates and retention periods. Cold tier capacity grows continuously but at predictable rates based on offload policies.

Network bandwidth between tiers affects offload and restore operations. Organizations typically schedule offload operations during maintenance windows to minimize impact on production workloads.

Maintenance Operations

Regular maintenance includes partition merging in cold storage, orphan file cleanup, and metadata optimization. These operations typically run as scheduled jobs during low-activity periods.

Backup strategies must account for tiered data distribution. Hot tier backups occur frequently for operational recovery while cold tier backups might rely on object storage durability guarantees.

Reversibility Guarantees

All offload operations support reversal through restore operations. This reversibility provides confidence when implementing tiering strategies, knowing that operational requirements changes can be accommodated.

The system maintains offload history enabling point-in-time restoration. Organizations can restore data as it existed at specific dates, supporting scenarios where historical data temporarily requires transactional access.

5.1 Analytics Accelerator learning resources

Use this guide to navigate all documentation for the Analytics Accelerator (PGAA), built on Postgres®. Whether you want to understand core concepts, follow step-by-step tutorials, solve specific problems, or explore how analytics applies to your role, this guide helps you find the right resources.

Hub quick links: Analytics Hub — *Analytics Accelerator Concepts* — *How-Tos (Runbook-Aligned)*

The documentation is structured to help you learn effectively and find information efficiently.

5.1.1 Documentation categories

Structured learning paths

Follow structured learning paths to progress from foundational concepts to advanced implementations.

Explore learning paths: *Analytics Accelerator Learning Paths*

- *Analytics Accelerator 101 Foundational concepts*
- *Analytics Accelerator 201 Practical application and core solutions*
- *Analytics Accelerator 301 Advanced techniques and optimization*

Conceptual explanations

Build foundational knowledge of analytics principles, modern data architectures, and Analytics Accelerator technologies and strategies.

Explore: *Analytics Generic Concepts* and *Analytics Accelerator Concepts*

Covers generic analytics concepts, the Analytics Accelerator (PGAA) vision, and deep dives into core technologies such as EDB Postgres Lakehouse, Apache Iceberg, Delta Lake, Tiered Tables, and architecture patterns.

How-to guides

Find concise, step-by-step instructions for common tasks related to the Analytics Accelerator.

Explore: [Analytics Accelerator how-to guides index](#)

Use cases and solutions

Discover how the Analytics Accelerator addresses business problems and supports architecture patterns across industries.

Explore: [Analytics in Hybrid Manager](#)

Persona-based guidance

Access curated information and learning paths tailored to your technical role.

Explore: [Analytics Accelerator for your role a persona-based guide](#)

Analytics Accelerator in Hybrid Manager (HM)

For users of EDB's managed services, this section provides documentation on how Analytics Accelerator concepts and technologies are implemented, provisioned, and managed in Hybrid Manager.

Explore: [Analytics in Hybrid Manager](#)

5.1.2 Next steps

- Follow a [Analytics Accelerator Learning Paths](#) to build your expertise
 - Build foundational knowledge with [Analytics Generic Concepts](#) and [Analytics Accelerator Concepts](#)
 - Apply what you've learned with [How-to guides](#)
 - Explore practical examples via [Analytics in Hybrid Manager](#)
 - Get guidance tailored to your role in [Persona-based guides](#)
 - For HM users: dive into [Analytics in Hybrid Manager](#)
-

Use this learning guide to chart your path through the Analytics Accelerator documentation — whether you're starting with core concepts or going deep on advanced architectures.

5.2 Analytics Accelerator Learning Paths

Use these learning paths to build knowledge progressively — from foundational concepts to advanced implementations and optimization techniques.

5.2.1 How to use these learning paths

Select the level that matches your current knowledge and goals. Each level links to curated guides with conceptual explanations, how-to articles, and practical tutorials.

For advanced topics and certification, explore official EDB training programs.

5.2.2 Learning path levels

Level 101: Foundations of the Analytics Accelerator

Target audience: Users new to the Analytics Accelerator, those new to modern data analytics concepts, or users seeking a refresher.

Focus: Core ideas, key terminology, the Analytics Accelerator landscape, and reasons behind analytical approaches.

What you will learn:

- Fundamental concepts of data warehouses, data lakes, and the lakehouse paradigm
- Key terms such as columnar storage, vectorized query engines, and open table formats (Apache Iceberg, Delta Lake)
- Overview of the Analytics Accelerator vision with Postgres and core solutions such as EDB Postgres Lakehouse
- Benefits of separating storage and compute for analytics

Start here: [Analytics Generic Concepts](#)

Level 201: Applying the Analytics Accelerator — core technologies and use cases

Target audience: Users with foundational knowledge ready to apply core technologies practically. Suitable for DBAs, data engineers, and developers.

Focus: Hands-on use of EDB Postgres Lakehouse, PGD analytics (including Tiered Tables), and Iceberg/Delta Lake integration. Basic use cases and configurations.

What you will learn:

- Set up and query EDB Postgres Lakehouse clusters
- Work with Apache Iceberg and Delta Lake tables
- Implement basic Tiered Table configurations using EDB Postgres Distributed (PGD)
- Connect BI tools to the Analytics Accelerator
- Apply analytics to BI reporting, historical data analysis, and operational analytics

Continue your journey: [Analytics Accelerator Concepts](#) + [How-to guides](#)

Level 301: Advanced Analytics Accelerator, optimization, and architecture

Target audience: Experienced users, architects, and senior engineers.

Focus: Advanced features, performance tuning, complex architectures, scalability, security, and troubleshooting.

What you will learn:

- Advanced configuration of EDB Postgres Lakehouse, PGAA, and PGFS
- Performance tuning for vectorized queries and object storage access
- Design and implement complex Tiered Table strategies with PGD for optimal cost and performance
- Manage advanced Iceberg catalogs and integrate third-party catalogs
- Architect solutions combining the Analytics Accelerator with other data frameworks (Spark, streaming platforms)
- Troubleshoot complex analytical query performance and data pipelines

- Apply security best practices for analytical environments

Advance your expertise: See *Analytics Accelerator Architecture* and *PGAA functions reference*

5.2.3 EDB training and certification

For expert-led instruction and official certification, explore EDB's training programs and certifications.

- *Analytics Accelerator Learning Paths (Coming soon)*
- *Analytics Accelerator Learning Paths (Coming soon)*

5.2.4 Additional guidance

- **Role-based guidance:** See *Analytics Accelerator for your role a persona-based guide* for resources tailored to your job function.
- **Solution-oriented examples:** Visit Analytics in Hybrid Manager for real-world examples.

5.2.5 Next steps

- Build your foundation with *Analytics Generic Concepts*
- Learn EDB's approach in *Analytics Accelerator Concepts*
- Apply your knowledge with *How-to guides*
- Explore industry applications via Analytics in Hybrid Manager

Use these learning paths to progress your Analytics Accelerator expertise — from foundational knowledge to advanced optimization and architecture.

5.3 Analytics Accelerator 101 Foundational concepts

Start here to learn the core ideas that underpin the Analytics Accelerator. This level is designed for those new to modern analytics concepts, or those seeking a refresher.

5.3.1 What you will learn

- Fundamental concepts of data warehouses, data lakes, and lakehouse architectures
- Columnar storage and vectorized query engines
- Open table formats: Apache Iceberg and Delta Lake
- The EDB Analytics Accelerator vision and core architecture
- The value of separating storage and compute for analytics

5.3.2 Before you begin

- *Analytics Generic Concepts*
- *Analytics Accelerator Concepts*

5.3.3 Suggested learning path

1. *Analytics Generic Concepts*
2. *Analytics Accelerator Concepts*
3. *Apache Iceberg Integration With Analytics Accelerator*
4. *Delta Lake Integration with Analytics Accelerator*
 - *EDB Postgres Lakehouse Architecture*

5.3.4 Next steps

Once you've completed 101, continue with:

- *Analytics Accelerator 201 Practical application and core solutions*
- *Analytics Accelerator Learning Paths*

5.4 Analytics Accelerator 201 Practical application and core solutions

Build on foundational knowledge and begin applying key Analytics Accelerator technologies in your environment.

5.4.1 What you will learn

- How to set up and query EDB Postgres Lakehouse clusters
- How to work with Apache Iceberg and Delta Lake tables
- How to implement tiered tables with PGD
- How to connect BI tools to the Analytics Accelerator
- How to apply these technologies to BI reporting, historical data analysis, and operational analytics

5.4.2 Before you begin

- *Analytics Accelerator 101 Foundational concepts*
- *Analytics Generic Concepts*
- *Analytics Accelerator Concepts*

5.4.3 Suggested learning path

1. *EDB Postgres Lakehouse Architecture*
2. *Apache Iceberg Integration With Analytics Accelerator*
3. *Delta Lake Integration with Analytics Accelerator*
4. *Tiered Tables With Analytics Accelerator*
5. *Create A Lakehouse Cluster*
6. *Configure Analytics Storage and Data Tiering with PGAA and PGD*
7. *How-To Lakehouse Read With/Without A Catalog*
8. *How-To Read/Write Without Catalog*

9. *How-To Integrate with Third-Party Iceberg Catalogs*

5.4.4 Next steps

Once you've completed 201, continue with:

- *Analytics Accelerator 301 Advanced techniques and optimization*
- Reference index

5.5 Analytics Accelerator 301 Advanced techniques and optimization

For experienced users, architects, and engineers ready to deepen their expertise with advanced optimization and architecture patterns.

5.5.1 What you will learn

- Advanced configuration of EDB Postgres Lakehouse, PGAA, and PGFS
- Performance tuning for vectorized queries and object storage access
- Complex tiered table strategies for optimized cost and performance
- Managing advanced Iceberg catalogs and integrating third-party catalogs
- Architecting solutions that combine Analytics Accelerator with other data frameworks (Spark, streaming platforms)
- Troubleshooting complex analytical queries and data pipelines
- Security best practices for analytics environments

5.5.2 Before you begin

- *Analytics Accelerator 201 Practical application and core solutions*
- *Analytics Accelerator Concepts*
- *Tiered Tables With Analytics Accelerator*

5.5.3 Suggested learning path

1. *Analytics Accelerator Architecture*
2. *Apache Iceberg Integration With Analytics Accelerator*
3. *Delta Lake Integration with Analytics Accelerator*
4. *DirectScan and fallback modes*
5. *PGAA functions reference*
6. *PGAA functions reference*
7. *Benchmarking datasets*
8. *User management*
9. *Analytics Accelerator for your role a persona-based guide*

5.5.4 Next steps

Consider official training and certification:

- *EDB training and certification*

5.6 Analytics Accelerator for your role a persona-based guide

Use this guide to understand how to leverage EDB Postgres® AI and analytics capabilities for your specific role. EDB Postgres AI provides a powerful platform for a wide range of analytical needs. This guide helps you focus on the capabilities that align with your goals.

5.6.1 Quickly find your role

- *Database administrator (DBA)*
- *DevOps engineer / site reliability engineer (SRE)*
- *Data scientist / data analyst*
- *Application developer*

For details on how to implement and manage these solutions within EDB Hybrid Manager (HM), see [Analytics in Hybrid Manager](#).

5.6.2 Database administrator (DBA)

Focus on performance, reliability, cost efficiency, and manageability of your database infrastructure.

Key objectives and Analytics Accelerator solutions

Performance monitoring and tuning

Identify slow queries, bottlenecks, and inefficient resource usage. Use HM dashboards to monitor performance of EDB Postgres Lakehouse Nodes and PGD clusters.

Storage management and cost optimization

Manage large databases and historical data cost effectively. Use tiered tables with PGD and EDB Postgres Lakehouse to offload cold data to object storage (as Iceberg tables). [Tiered Tables With Analytics Accelerator](#). Manage tiered tables via HM.

Capacity planning

Forecast storage and compute needs. Analyze historical growth patterns and current utilization via Lakehouse and HM dashboards. Leverage the separation of compute and storage in Lakehouse for flexible scaling.

Backup and recovery of analytical data

Protect data in object storage and associated metadata. Follow Iceberg catalog backup strategies and HM backup options.

Ensuring data accessibility for analytics

Provide performant and secure access to data analysts and data scientists. Use Lakehouse Nodes and PGD offloading to minimize impact on transactional systems.

Recommended resources

- EDB Postgres Distributed (PGD) documentation
- *EDB Postgres Lakehouse Architecture*
- Analytics in Hybrid Manager

5.6.3 DevOps engineer / site reliability engineer (SRE)

Focus on automation, infrastructure management, scalability, reliability, and observability.

Key objectives and Analytics Accelerator solutions

Automated provisioning and management

Use Hybrid Manager to provision Lakehouse clusters and related resources via the console, API, or CLI. *Create A Lakehouse Cluster*.

Scalability and elasticity

Independently scale Lakehouse compute resources to meet demand.

Monitoring and observability

Gain visibility into performance, health, and cost of analytical systems. Use HM dashboards and logging.

CI/CD for data pipelines and analytical applications

Integrate schema changes (Iceberg evolution) and data offloading via PGD into CI/CD pipelines.

Cost management for cloud resources

Track and optimize costs for object storage and analytical compute. Use HM resource insights.

Recommended resources

- Analytics in Hybrid Manager
- *EDB Postgres Lakehouse Architecture*
- EDB API and CLI documentation

5.6.4 Data scientist / data analyst

Focus on extracting insights, building models, and answering business questions.

Key objectives and Analytics Accelerator solutions

Accessing large and diverse datasets

Query large datasets stored as Iceberg or Delta Lake tables via Lakehouse Nodes. Use Python, R, SQL clients, and BI tools.

Data preparation and exploration

Perform data filtering, aggregation, and transformation with SQL via Lakehouse Nodes.

Working with open table formats

Leverage schema evolution, time travel, and ACID transactions in Iceberg and Delta Lake tables. *Apache Iceberg Integration With Analytics Accelerator* . *Delta Lake Integration with Analytics Accelerator* .

Integration with data science tools and frameworks

Load Lakehouse query results into pandas, R data frames, or use with Spark, TensorFlow, or PyTorch.

Query performance

Accelerate analytical SQL queries with vectorized query engines in Lakehouse Nodes.

Recommended resources

- *EDB Postgres Lakehouse Architecture*
- *How-To Lakehouse Read With/Without A Catalog*
- Connecting BI tools

5.6.5 Application developer

Focus on building data-driven applications and embedding insights.

Key objectives and Analytics Accelerator solutions

Building applications with analytical features

Use Lakehouse Nodes or PGD read replicas to provide reporting, dashboards, and data exploration in applications.

Efficiently handling large reporting queries

Offload large aggregation and historical data queries to Lakehouse Nodes. Use tiered tables to direct historical queries to an analytical tier.

Accessing data from a unified platform

Provide consistent SQL access to both operational and analytical data.

Developing AI/ML-enabled applications

Store and access ML model data using Lakehouse Nodes.

Recommended resources

- Postgres SQL documentation and client libraries
- *EDB Postgres Lakehouse Architecture*
- *Tiered Tables With Analytics Accelerator*

Explore these resources for more detailed guidance. For HM-specific implementation steps, see Analytics in Hybrid Manager .

6.1 Create A Lakehouse Cluster

This guide explains how to create a **Lakehouse cluster** in Hybrid Manager (HM). Lakehouse clusters provide **scalable analytical compute** for fast queries on **Apache Iceberg** or **Delta Lake** tables stored in object storage.

For architectural background, see *[Lakehouse clusters in Hybrid Manager](#)* .

6.1.1 Why create a Lakehouse cluster?

By creating a Lakehouse cluster:

- You unlock the ability to **run fast Postgres SQL queries on large datasets** in object storage.
- You can query **Tiered Tables offloaded from PGD** or **external Iceberg/Delta Lake tables**.
- You provide compute for **BI tools and ad-hoc analytics** on your lakehouse data.

This is a foundational step for building your analytics architecture in Hybrid Manager.

6.1.2 Goals

After completing this How-To, you will be able to:

- Deploy a Lakehouse cluster.
- Connect to it using standard Postgres clients.
- Configure storage and catalogs for querying Iceberg and Delta Lake tables.
- Use the cluster to query existing or newly defined lakehouse tables.

6.1.3 Prerequisites

Before you begin:

- You have access to a Hybrid Manager environment with permissions to create Lakehouse clusters.

- You have object storage available (S3, GCS, MinIO, etc.).
- You have Iceberg or Delta Lake tables available — or plan to define them after cluster creation.
- You have an Iceberg catalog prepared if working with Iceberg tables:
- *Configure Analytics Storage and Data Tiering with PGAA and PGD*

6.1.4 Steps

Step 1: Initiate cluster creation

- In the HM dashboard, click **Create New**.
- Select **Lakehouse Analytics** or **Analytical Cluster**.
- Confirm that you are creating an Analytics Cluster for querying open lakehouse formats.

Step 2: Choose your path

You will be prompted to choose:

- **Templates** — Use a pre-configured template (if available).
- **Custom build** — Click **Start from Scratch** for full configuration control (recommended for most cases).

Step 3: Configure cluster settings

Cluster Settings

- **Cluster name** — Enter a unique name (e.g. `sales_analytics_lakehouse`).
- **Password** — Enter a strong superuser password.
- **Tags** — Optionally add tags (e.g. `environment:production,project:q3-reporting`).
- **Deployment location** — Select cloud provider and region.
- **Database type** — Choose EDB Postgres Extended Server or Advanced Server.
- **Postgres version** — Select supported version with Lakehouse extensions.
- **Instance size** — Select compute size appropriate for your workload.

Additional Settings

- **Networking** — Configure VPC, subnet, security groups, IP whitelisting.
- **Backups** — Configure backup settings if applicable.

Step 4: Review and create

- Review the **Cluster Summary**.
- Confirm all settings.
- Click **Create Cluster**.
- Monitor progress in the HM dashboard.

6.1.5 What you can do next

Now that your Lakehouse cluster is provisioned, you can:

- **Connect to the cluster** to run SQL queries: Use standard Postgres clients (psql, DBeaver, pgAdmin) with HM-provided connection details.
- **Configure storage and catalogs to access data:**
 - Configure a PGFS storage location — if querying Delta Lake or filesystem-based Iceberg.
 - *Configure Analytics Storage and Data Tiering with PGAA and PGD* — if querying catalog-managed Iceberg tables or using PGD Tiered Tables.
- **Define and query Lakehouse tables:**
 - *How-To Lakehouse Read With/Without A Catalog*
- **Query Tiered Tables offloaded from PGD:**
 - Learn concepts for *Tiered Tables With Analytics Accelerator*

6.1.6 Related topics

- *Lakehouse clusters in Hybrid Manager*
- Analytics in Hybrid Manager

7.1 How To Get Setup for Lakehouse analytics

7.2 Getting setup for Lakehouse analytics

This document explains the prerequisites and setup steps needed before running read-only analytics on Delta Lake and Iceberg tables.

It covers the required image versions, environment variables, CA certificates, catalog configuration, and how to identify your object storage bucket.

7.2.1 Goals

- Verify required operand and extension versions
 - Establish environment variables and credentials
 - Retrieve appliance CA certificates
 - Identify the appliance-provisioned object storage bucket
-

7.2.2 Requirements

You need an operand image with:

- **PGD 6.0 Expanded** snapshot version
- **PGAA 1.2+**

Check your extension version:

```
SELECT name, installed_version, default_version
FROM pg_available_extensions
WHERE name = pgaa;
\`\`\`
```

The `default\_version` must be `1.2+` .

> Note: **If** your environment requires snapshot repositories **or** pre-built snapshot_↵
↵operand images, consult your internal release notes **and** deployment runbooks.

- - -

Assumptions **and** variables

This guide assumes a PGD6 **cluster** has already been provisioned **in** Hybrid Manager.↵
↵Provisioning **is out of scope**.

You should use a `~pgdx` image **version with** EPAS **or** PGE. Vanilla PostgreSQL **is**↵
↵**not** supported.

We will define the following variables **for** later use:

```
.. csv-table::
:header: Variable,Meaning,Example
:widths: 12,25,15
:align: left
:class: longtable

`PROJECT_ID`,Hybrid Manager Project ID,`prj_RAi8WE2xivmeK4qm`
`APPL_URL`,Hybrid Manager base URL,`https://portal.eks-04250637-main.edbaipatform.↵
↵enterisedb.network`
`API_PROV`,Hybrid Manager token with "editor" project permissions,`baak_abcdef`↵
`API_WRITE`,Hybrid Manager token with "write_catalog" project permissions,`baak_↵
↵ghijkl`
`PARAMS_URI`,Managed catalog API URL,`https://portal.eks-04250637-main.../catalog/`
`PARAMS_WAREHOUSE`,Managed catalog warehouse name (PyIceberg),`prj_RAi8WE2xivmeK4qm-↵
↵catalog`
`PARAMS_WAREHOUSE_ID`,Managed catalog warehouse UUID (PGAA),`cd8c29f6-20fa-11f0-97cd-↵
↵cfc1662c48a8`
`PARAMS_CONFIG`,Managed catalog config URL (PySpark),`https://portal.eks.../config?↵
↵warehouse=prj_RAi8WE2xivmeK4qm-catalog`
`BUCKET_URL`,URL of the bucket used by Hybrid Manager for offloads,`s3://eks-04220339-↵
↵main-edb-postgres`
```

- - -

Retrieve the appliance CA certificate

This certificate **is** required **for** the sample **data** loading script.

```
```bash
```

```
Using kube_context
```

(continues on next page)

(continued from previous page)

```
kubectl --context "$kube_context" get secret -n cert-manager global-ca-secret -o-
↪yaml | yq .data."ca.crt" | base64 -d > ca.crt

On RHOS or avoiding kube_context

kubectl get secret -n cert-manager global-ca-secret -o yaml | yq .data."ca.crt" |-
↪base64 -d > ca.crt

On RHOS GCP where cert-manager may not exist

kubectl get secret -n edbpgai-bootstrap beaconator-ca-bundle -o json | jq -r .data.
↪"public.crt" | base64 --decode > ca.crt
```

**Explanation** The CA cert is needed to establish trust when connecting to the Hybrid Manager appliance APIs and services.

## 7.2.3 Create a Lakekeeper catalog

Use your API\_PROV token:

1. In the Hybrid Manager UI, go to **Catalogs** → **Add Project Catalog**.
2. Copy the JSON with the catalog settings, for example:

```
{
 "uri": "https://portal-uat-rke.rke.hcp.enterprisedb.network/api/iceberg/prj_
↪efpRstCQ1l6m2lGm",
 "warehouse": "prj_efpRstCQ1l6m2lGm-catalog-iXFkMR",
 "warehouseId": "d1d1d4ce-850a-11f0-88fb-af8509011340",
 "config": "https://portal-uat-rke.rke.hcp.enterprisedb.network/api/iceberg/prj_
↪efpRstCQ1l6m2lGm/v1/config?warehouse=prj_efpRstCQ1l6m2lGm-catalog-iXFkMR"
}
```

**Explanation** These parameters are used later when linking PGAA to managed catalogs or external tools like PyIceberg and PySpark.

On RHOS, use the **internal Kubernetes service URL** instead of the public URL for in-cluster SQL statements like `pgaa.add_catalog`.

## 7.2.4 Identify your bucket URL

The appliance provisions its own object storage bucket. Construct the URL based on your CSP:

- **AWS** Use the `eks-XXXX` part of the portal URL and append `-main-edb-postgres`. Example:

```
https://portal.eks-03242048-v11x.edbaplatform.enterprisedb.network
→ s3://eks-03242048-main-edb-postgres
```

- **GCP** Use the `gke-XXXX` part of the portal URL and append `-main-edb-object-storage`. Example:

```
https://portal.gke-05151425-v12x.edbaipatform2.enterprisedb.network
→ gs://gke-05151425-main-edb-object-storage
```

- **RHOS** Inspect a Barman backup or the edb-object-storage secret:

```
kubectl get secret edb-object-storage -o yaml | yq .data.bucket_name | base64 -d
```

**Explanation** You will need the `BUCKET_URL` value to reference storage in PGAA commands and sample data scripts.

---

## 7.2.5 Sanity checks

### Check public bucket access (optional)

```
aws s3 ls s3://beacon-analytics-demo-data-us-east-1-dev/iceberg-example/default.db --
↪recursive --no-sign-request
```

You should see files like:

```
2025-03-18 08:44:38 902 iceberg-example/default.db/iceberg_table/data/00000-0-.
↪..
2025-03-18 08:44:38 4340 iceberg-example/default.db/iceberg_table/metadata/4ef9.
↪..
2025-03-18 08:44:38 1823 iceberg-example/default.db/iceberg_table/metadata/snap.
↪..
```

**Explanation** This confirms that the sample Iceberg dataset exists and is accessible for testing.

---

## 7.2.6 Next steps

- Continue to *How-To Lakehouse Read With/Without A Catalog* .
  - Or, try *How-To Read/Write Without Catalog* .
- 

## 7.2.7 Takeaways

- Prepared environment accelerates subsequent how-tos (catalogs, readers, offloads)
- CA bundles and bucket URLs are foundational for data access and tooling

## 7.3 How-To Read/Write Without Catalog

**Purpose:** Configure EDB PGAA with PGD High Availability to implement automated data tiering, analytics offload, and lakehouse architectures for cost optimization and performance scaling.

**Scope:** Complete workflow from initial setup through advanced partition management, including Delta Lake and Iceberg integration, automated tiering strategies, and storage optimization analysis.

### 7.3.1 Prerequisites

#### Technical Requirements

PostgreSQL with PGD High Availability must be operational with administrative privileges for catalog management, partition operations, and storage configuration. PGAA extension requires installation with network connectivity to external storage systems including S3-compatible endpoints. Sufficient bandwidth supports data transfer operations between storage tiers during offload processes.

#### Knowledge Requirements

Understanding of PostgreSQL partitioning concepts, system catalog structures, and distributed database management enables effective implementation. Familiarity with object storage systems, lakehouse architectures, and analytics workload patterns helps optimize configuration decisions. Basic knowledge of SQL performance analysis supports troubleshooting and optimization activities.

#### Environment Setup

Active PGD cluster with multiple nodes provides high availability foundation while PGAA extension enables analytics storage integration. Network security configurations must permit access to external storage endpoints with appropriate firewall and authentication settings. Monitoring infrastructure should track background task processing and storage utilization patterns.

### 7.3.2 Setup: Configure Analytics Infrastructure

#### Step 1: Initialize PGAA Extension and Performance Settings

Configure the analytics infrastructure with optimized settings for testing and production deployment scenarios.

```
-- Setup PGD configuration for efficient testing cycles
SET bdr.taskmgr_nap_time = 10000; -- AutoPartition worker runs every 10s vs default
-- 290s
SET pgaa.max_replication_lag_s = 5; -- Analytics replication lag threshold
SET pgaa.flush_task_interval_s = 5; -- Frequent storage flush for testing

-- Initialize PGAA extension
CREATE EXTENSION IF NOT EXISTS pgaa CASCADE;
```

The performance settings accelerate background task processing during development and testing phases. Production environments should evaluate these settings against system resources and operational requirements. The PGAA extension provides the foundation for analytics storage integration and lakehouse table management.

#### Step 2: Configure Storage Locations

Establish storage locations for both read-only analytics access and write operations with appropriate security configurations. Please see

```
-- Create read-only storage location for existing Delta Lake data
SELECT bdr.replicate_ddl_command($$
 SELECT pgfs.create_storage_location(
 biganimal-sample-data,
 s3://beacon-analytics-demo-data-us-east-1-prod,
 {"aws_skip_signature": "true"}
);
$$);
```

(continues on next page)

(continued from previous page)

```

- - Create development storage location for Iceberg tables
SELECT bdr.replicate_ddl_command($$
 SELECT pgfs.create_storage_location(
 biganimal-sample-data-dev,
 s3://beacon-analytics-demo-data-us-east-1-dev,
 {"aws_skip_signature": "true"}
);
$$);

```

*PGFS (Postgres File System)* . PGFS Storage location configuration establishes the foundation for both read-only analytics access and write operations. The `aws_skip_signature` setting applies to public buckets but production environments require proper AWS credential configuration through IAM roles or credential files.

### Step 3: Configure Analytics Write Location

Set up the primary analytics storage location for automated data offload operations. See *Getting setup for Lakehouse analytics* .

```

- - Create appliance storage location with dynamic path construction
SELECT bdr.replicate_ddl_command($$
 SELECT pgfs.create_storage_location(
 appliance-bucket,
 ${BUCKET_URL} || / ||
 (SELECT node_group_name FROM bdr.node_group WHERE node_group_name != world LIMIT
↪1) ||
 /analytics-offload
)
 $$);

- - Validate storage location accessibility
CREATE TABLE sentinel_sl_read_test () USING PGAA
WITH (pgaa.format = iceberg, pgaa.storage_location = appliance-bucket, pgaa.path =
↪nonexistent);
- - Should return ERROR: No valid Iceberg metadata file found (confirms bucket access)

- - Set as primary analytics write location
SELECT bdr.alter_node_group_option(
 (SELECT node_group_name FROM bdr.node_group WHERE node_group_name != world LIMIT 1),
 analytics_storage_location,
 appliance-bucket
);

```

The dynamic path construction includes the PGD node group name to prevent conflicts in multi-tenant environments. Storage location validation through deliberate error testing confirms proper authentication and network connectivity. Setting the analytics storage location activates the Seafoal query engine for lakehouse operations.

### Step 4: Verify Seafoal Query Engine Activation

Confirm that the analytics query engine has started and is ready for lakehouse operations.

```

- - Verify Seafoal query engine is running
SELECT application_name FROM pg_stat_activity WHERE application_name LIKE %seafoal%;

```

(continues on next page)

(continued from previous page)

```
-- Must show pgd seafowl_manager for proper operation
```

The Seafowl query engine provides optimized execution for analytics workloads across columnar storage formats.

### 7.3.3 Read-Only Analytics: Access Existing Data

Iceberg format specification enables access to Apache Iceberg tables with full metadata support including schema evolution, partition management, and time travel capabilities.

#### Step 5: Analyze Storage Characteristics

Use analytics functions to understand storage utilization and performance characteristics of lakehouse tables.

```
-- Analyze storage statistics for Delta Lake tables
SELECT * FROM pgaa.lakehouse_table_stats(lineitem::regclass);

-- Analyze storage statistics for Iceberg tables
SELECT * FROM pgaa.lakehouse_table_stats(iceberg_table::regclass);
```

Storage statistics provide insights into compression ratios, file organization, and query performance characteristics that inform optimization decisions. These metrics support capacity planning and performance tuning for analytics workloads.

### 7.3.4 Transactional Table Analytics Offload

#### Step 9: Create and Populate Transactional Table

Establish a transactional table with representative data for offload testing and validation.

```
-- Create transactional table for offload testing
CREATE TABLE transactional_table (
 key BIGINT,
 inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 revenue DOUBLE PRECISION,
 comment TEXT,
 PRIMARY KEY (key, inserted_at)
);

-- Insert test data with temporal distribution
INSERT INTO transactional_table (
 SELECT
 g,
 2010-01-01 00:00:00::timestamp + INTERVAL 1 second *
 (EXTRACT (EPOCH FROM (CURRENT_TIMESTAMP - 2010-01-01 00:00:00::timestamp)) /
↪1000000 * g),
 g * 0.001,
 encode(sha256(g::text::bytea), hex)
 FROM generate_series(1, 1000000) g
);
```

The data generation creates realistic temporal distribution patterns while providing sufficient volume for performance testing. SHA256 hash generation simulates variable-length text content found in production systems.

## Step 10: Configure Analytics Offload

Enable automatic replication of transactional data to analytics storage for cost optimization and performance scaling.

```
- - Enable analytics replication for transactional table
ALTER TABLE transactional_table SET (pgd.replicate_to_analytics = TRUE);

- - Monitor replication progress
SELECT * FROM bdr.analytics_table;

- - Wait until table appears in results (~120s for initial sync)
```

Analytics replication creates an Iceberg representation of transactional data in the configured storage location. The initial synchronization process handles format conversion and establishes ongoing replication streams for data consistency.

## Step 11: Create Analytics Reader Interface

Establish direct access to the analytics-tier data through dedicated reader tables.

```
- - Create reader interface for analytics data
CREATE TABLE transactional_table_analytics_reader ()
USING PGAA
WITH (
 pgaa.format = iceberg,
 pgaa.storage_location = appliance-bucket,
 pgaa.path = public.transactional_table
);

- - Verify data integrity across storage tiers
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue
FROM transactional_table_analytics_reader;

- - Should return: count=1000000, total_revenue=500000500
```

The analytics reader provides direct access to cost-effective storage while maintaining query transparency. Data integrity validation confirms successful format conversion and replication processes.

## Step 12: Convert Transactional Table to Analytics-Only

Transform the transactional table to use analytics storage directly while discarding local data.

```
- - Convert to analytics-only access
ALTER TABLE transactional_table
SET ACCESS METHOD PGAA,
SET (
 pgaa.auto_truncate = true,
 pgaa.format = iceberg,
 pgaa.storage_location = appliance-bucket,
 pgaa.path = public.transactional_table
);

- - Verify conversion maintained data access
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM transactional_
↪table;
```

The conversion process eliminates local storage overhead while maintaining transparent SQL access through the analytics tier. Auto-truncate removes local data after confirming analytics tier availability.

### 7.3.5 Advanced Table Management Operations

#### Step 13: Restore Table from Analytics to Transactional Storage

Demonstrate the reversibility of analytics offload for scenarios requiring local high-performance access.

```
-- Restore table to transactional storage
ALTER TABLE transactional_table SET ACCESS METHOD heap, SET (pgd.replicate_to_
↳analytics = FALSE);

-- Verify restoration with query plan analysis
EXPLAIN SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM
↳transactional_table;

-- Should show Parallel Seq Scan indicating local heap access
```

Restoration capabilities provide flexibility for changing access patterns or performance requirements. Query plan analysis confirms the transition back to local PostgreSQL heap storage with parallel processing capabilities.

#### Step 14: Handle Data Updates with Analytics Synchronization

Demonstrate data modification workflows that maintain consistency across storage tiers.

```
-- Update data in transactional table
UPDATE transactional_table SET revenue = revenue * 2;

-- Re-enable analytics replication to propagate changes
ALTER TABLE transactional_table SET (pgd.replicate_to_analytics = TRUE);

-- Wait for synchronization (~120s) then verify changes propagated
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue
FROM transactional_table_analytics_reader;

-- Should show doubled revenue: total_revenue=1000001000
```

Update propagation maintains consistency between transactional and analytics tiers while handling data modifications efficiently. The synchronization process recreates the analytics representation with current data.

#### Step 15: Implement Partial Table Offload

Execute selective data archival where only historical portions of tables move to analytics storage.

```
-- Perform partial offload with consistent transaction boundary
BEGIN;
CREATE TABLE transactional_table_before_2020 AS (
 SELECT * FROM transactional_table
 WHERE DATE_PART(year, inserted_at) < 2020
);
DELETE FROM transactional_table WHERE DATE_PART(year, inserted_at) < 2020;
COMMIT;

-- Configure partial table for analytics offload
ALTER TABLE transactional_table_before_2020 ADD PRIMARY KEY (key, inserted_at);
ALTER TABLE transactional_table_before_2020 SET (pgd.replicate_to_analytics = TRUE);

-- Convert to analytics-only access
ALTER TABLE transactional_table_before_2020
```

(continues on next page)

(continued from previous page)

```

SET ACCESS METHOD PGAA,
SET (
 pgaa.auto_truncate = true,
 pgaa.format = iceberg,
 pgaa.storage_location = appliance-bucket,
 pgaa.path = public.transactional_table_before_2020
);

-- Verify temporal data distribution
SELECT
 COUNT(*) FILTER (WHERE inserted_at < 2020-01-01) AS before_2020,
 COUNT(*) FILTER (WHERE inserted_at >= 2020-01-01) AS after_2020
FROM transactional_table_before_2020;

```

Partial offload enables fine-grained data lifecycle management where different time periods have different storage tier requirements. Transaction boundaries ensure data consistency during the archival process.

### 7.3.6 AutoPartition with Analytics Offload

#### Step 16: Configure Automated Partitioning

Establish time-based partitioning with automated lifecycle management for scalable data organization.

```

-- Create partitioned table structure
CREATE TABLE partitioned_table (
 key BIGINT,
 inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 revenue DOUBLE PRECISION,
 comment TEXT,
 PRIMARY KEY (key, inserted_at)
) PARTITION BY RANGE(inserted_at);

-- Configure AutoPartition without immediate offload
SELECT bdr.autopartition (
 relation := partitioned_table,
 partition_increment := 1 year,
 partition_initial_lowerbound := 2010-01-01 00:00:00,
 managed_locally := TRUE
);

-- Wait for partition creation across cluster
SELECT bdr.autopartition_wait_for_partitions_on_all_nodes(partitioned_table::regclass,
 → CURRENT_TIMESTAMP::text);

```

AutoPartition configuration creates annual partitions with boundaries that align with typical data access patterns. The initial configuration excludes offload settings to prevent premature data migration during setup and data loading phases.

#### Step 17: Load Test Data and Analyze Baseline Storage

Populate partitions with representative data and establish baseline storage metrics for optimization analysis.

```

-- Load test data across temporal partitions
INSERT INTO partitioned_table (

```

(continues on next page)

(continued from previous page)

```

SELECT
 g,
 2010-01-01 00:00:00::timestamp + INTERVAL 1 second *
 (EXTRACT (EPOCH FROM (CURRENT_TIMESTAMP - 2010-01-01 00:00:00::timestamp)) /
↪1000000 * g),
 g * 0.001,
 encode(sha256(g::text::bytea), hex)
FROM generate_series(1, 1000000) g
);

```

Data loading creates approximately 65,000 records per annual partition with consistent storage characteristics that enable predictable cost modeling and performance analysis.

### Step 18: Baseline Storage Analysis

Create comprehensive storage analysis capabilities and capture pre-offload metrics for comparison purposes.

```

-- Create optimized row counting function
CREATE OR REPLACE FUNCTION temp_get_table_rowcount(schema_name text, table_name text)
RETURNS bigint AS $$
DECLARE count bigint;
BEGIN
 -- Uses DirectScan for PGAA tables (1.8s vs 20s without optimization)
 EXECUTE FORMAT(SELECT COUNT(*) FROM %I.%I, schema_name, table_name) INTO count;
 RETURN count;
END $$ LANGUAGE plpgsql;

-- Comprehensive partition storage analysis
WITH partitioned_table_info AS (
 SELECT
 nspc.nspname AS schema_name,
 child.relname AS table_name,
 pg_catalog.pg_get_expr(child.relpartbound, child.oid) AS partition_constraint,
 pg_get_partition_constraintdef(child.oid) AS partition_constraint_full
 FROM pg_inherits inh
 JOIN pg_class parent ON inh.inhparent = parent.oid
 JOIN pg_class child ON inh.inhrelid = child.oid
 JOIN pg_namespace nspp ON parent.relnamespace = nspp.oid
 JOIN pg_namespace nspc ON child.relnamespace = nspc.oid
 WHERE parent.relname = partitioned_table AND nspp.nspname = public
),

pgaa_tables AS (
 SELECT
 m.schema_name, m.table_name, m.format,
 CASE
 WHEN m.storage_location_name IS NOT NULL THEN FORMAT(pgfs://%s/%s, m.storage_
↪location_name, m.path)
 WHEN (m.managed_by IS NOT NULL AND m.catalog_namespace IS NOT NULL AND m.
↪catalog_table IS NOT NULL)
 THEN FORMAT(%s.%s.%s, m.managed_by, m.catalog_namespace, m.catalog_table)
 END AS path,

```

(continues on next page)

(continued from previous page)

```

 temp_get_table_rowcount(m.schema_name, m.table_name) AS num_rows,
 s.total_size AS total_size,
 pti.partition_constraint
FROM pgaa.get_all_analytics_table_settings() m
LEFT JOIN LATERAL pgaa.lakehouse_table_stats(FORMAT('%I.%I', m.schema_name, m.table_
↪name)::regclass) s ON TRUE
JOIN partitioned_table_info pti ON m.schema_name = pti.schema_name AND m.table_name_
↪= pti.table_name
WHERE m.storage_location_name IS NOT NULL OR (m.managed_by IS NOT NULL AND m.
↪catalog_namespace IS NOT NULL AND m.catalog_table IS NOT NULL)
),

local_tables AS (
 SELECT
 pn.nspname AS schema_name, pc.relname AS table_name,
 temp_get_table_rowcount(pn.nspname, pc.relname) AS num_rows,
 pg_total_relation_size(FORMAT('%I.%I', pn.nspname, pc.relname)::regclass) AS total_
↪size,
 pti.partition_constraint
FROM pg_class pc
 JOIN pg_namespace pn ON pc.relnamespace = pn.oid
 JOIN pg_am am ON am.oid = pc.relam
 JOIN partitioned_table_info pti ON pn.nspname = pti.schema_name AND pc.relname =_
↪pti.table_name
WHERE pn.nspname NOT IN (pg_catalog, information_schema, pg_toast, pg_internal,_
↪pgaa, pgfs, bdr)
 AND pc.relkind = r AND am.amname != pgaa
),

all_leaf_tables AS (
 SELECT schema_name, table_name, format AS pgaa_format, path AS pgaa_path, num_rows,_
↪total_size, partition_constraint FROM pgaa_tables
 UNION
 SELECT schema_name, table_name, NULL AS pgaa_format, NULL AS pgaa_path, num_rows,_
↪total_size, partition_constraint FROM local_tables
),

final_table_info AS (
 SELECT
 schema_name, table_name,
 CASE
 WHEN pgaa_format = iceberg THEN iceberg
 WHEN pgaa_format = delta THEN delta
 ELSE heap
 END AS type,
 COALESCE(pgaa_path, local) AS location,
 num_rows, total_size AS total_size_bytes,
 SUBSTRING(partition_constraint, FROM '[^]+') AS start_bound,
 SUBSTRING(partition_constraint, TO '[^]+') AS end_bound
FROM all_leaf_tables
ORDER BY partition_constraint ASC
)

```

(continues on next page)

(continued from previous page)

```
SELECT * FROM final_table_info;
```

**Baseline Results:** All partitions show type = 'heap' and location = 'local' with approximately 10MB per historical partition, totaling ~154MB for complete dataset in expensive transactional storage.

### Step 19: Configure Automated Analytics Offload

Enable automated partition tiering based on data age thresholds while maintaining query transparency.

```
-- Configure analytics offload for partitions older than one year
SELECT bdr.autopartition (
 relation := partitioned_table,
 partition_increment := 1 year,
 partition_initial_lowerbound := 2010-01-01 00:00:00,
 managed_locally := TRUE,
 analytics_offload_period := 1 year
);
```

The one-year threshold balances operational access requirements with storage cost optimization by keeping recent data in high-performance storage while migrating historical partitions to cost-effective analytics tiers.

### Step 20: Monitor Automated Offload Operations

Track background migration progress and identify completion status for storage tier transitions.

```
-- Monitor offload work queue until completion
SELECT
 wqs.workid,
 wis.ap_wi_started_at,
 wis.ap_wi_finished_at,
 wis.ap_wi_status,
 "partition",
 sql
FROM bdr.taskmgr_local_workitem_status wis
 RIGHT JOIN bdr.taskmgr_local_work_queue_status wqs ON wis.ap_wi_workid = wqs.workid
WHERE relname = public.partitioned_table
AND sql NOT LIKE %bdr%.autopartition_create_partition%
ORDER BY ap_wi_finished_at DESC;
```

Monitor until all ap\_wi\_finished\_at values are NOT NULL, indicating successful completion. Processing typically requires 10+ minutes for 14 historical partitions (2010-2023 when running in 2025).

### Step 21: Validate Storage Tier Distribution

Confirm successful migration through storage tier assignment analysis and local storage reduction verification.

```
-- Examine storage tier assignments
dt+
-- Historical partitions should show "pgaa" with "0 bytes"
-- Recent partitions should show "heap" with actual storage allocation
```

Expected results show historical partitions with “pgaa” access method and zero local storage, confirming successful

migration to analytics tier. Recent partitions maintain “heap” access with local storage for optimal operational performance.

Step 22: Post-Migration Storage Analysis and ROI Calculation

Execute comprehensive storage analysis to quantify optimization results and calculate return on investment.

```
- - Run same storage analysis query as Step 18 to capture post-migration metrics
- - Verify data integrity across storage tiers
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM partitioned_table;
- - Should maintain: count=1000000, total_revenue=500000500
```

Storage Optimization Results:

Metric		Before Migration	After Migration	Savings
Local Storage (Historical)		144MB	16MB	128MB (89% reduction)
Storage Cost Profile		100% premium storage	11% premium, 89% standard	60-80% cost reduction
Query Performance		Uniform fast access	Recent fast, historical acceptable	Optimized by access pattern
Administrative Over-head		Manual management	Automated lifecycle	Eliminated manual intervention

The automated tiering delivers substantial infrastructure cost reductions while maintaining transparent query access across storage tiers. Recent data remains in high-performance storage for operational requirements while historical data optimizes for cost-effective analytics storage.

7.3.7 Troubleshooting and Validation

Common Issues and Resolution

**Storage Location Access:** Verify credential configuration and network connectivity to external storage endpoints. Use deliberate error testing to confirm bucket accessibility and authentication.

**Offload Processing Delays:** Monitor background task queues and adjust `bdr.taskmgr_nap_time` for development environments. Production systems should maintain default settings for resource optimization.

**Data Integrity Concerns:** Use row counts and aggregate calculations to verify data consistency across storage tier transitions. Analytics storage maintains complete data fidelity through format conversion processes.

**Query Performance Variations:** Expect different performance characteristics between storage tiers, with analytics storage optimized for throughput over latency. Recent data in local storage maintains optimal response times.

Performance Monitoring

Track storage utilization trends, query response times across tiers, and background task completion rates. Monitor network connectivity to external storage systems and resource utilization during offload operations.

Regular analysis of access patterns may reveal opportunities to optimize offload thresholds or identify partitions requiring different lifecycle policies based on business requirements.

The comprehensive configuration provides automated data tiering capabilities that balance operational performance requirements with storage cost optimization while maintaining transparent SQL access across hybrid storage architectures.

## 7.4 How-To Integrate with Third-Party Iceberg Catalogs

### 7.4.1 Overview

This guide provides instructions for configuring PostgreSQL Advanced Analytics (PGAA) to work with third-party Apache Iceberg catalogs, enabling seamless data lake integration for analytical workloads. PGAA supports reading from and writing to Iceberg tables through external catalog services, allowing you to leverage existing data lake infrastructure alongside your PostgreSQL deployments.

### 7.4.2 Context and Goals

#### What This Guide Covers

This documentation demonstrates how to:

- Configure PGAA to connect with third-party Iceberg catalogs
- Enable bidirectional data flow between PostgreSQL and Iceberg tables
- Set up catalog integration for PGD (Postgres Distributed) offloads
- Ensure compatibility with external analytical tools

#### Supported Catalog Types

PGAA currently supports integration with:

- **Snowflake Open Data Catalog (Apache Polaris):** Verified compatibility with manual OAuth token management
- **AWS S3 Tables:** Native integration using table buckets

### 7.4.3 Prerequisites

Before beginning this integration, ensure you have:

#### Required Components

- Active PGAA installation with appropriate licensing
- PostgreSQL cluster with BDR (Bi-Directional Replication) configured
- Administrative access to PGAA catalog management functions

#### For Snowflake Polaris Integration

- Snowflake trial or production account
- AWS S3 bucket configured in your organization's AWS account
- IT approval for Snowflake to access your S3 bucket
- OAuth client credentials from Snowflake

#### For AWS S3 Tables Integration

- AWS account with S3 Tables enabled
- Appropriate IAM permissions for S3 Tables management
- S3 bucket configured as a table bucket

## 7.4.4 Practical Applications

This integration enables several key use cases:

1. **Analytical Offloading:** Move historical data from operational PostgreSQL databases to cost-effective object storage
2. **Data Lake Integration:** Query Iceberg tables directly from PostgreSQL without data movement
3. **Hybrid Analytics:** Combine real-time PostgreSQL data with historical Iceberg data in unified queries
4. **Cross-Platform Analytics:** Share data between PostgreSQL and other analytical engines through a common Iceberg catalog

## 7.4.5 Configuration Instructions

### Setting Up Snowflake Polaris Integration

#### Step 1: Obtain OAuth Token

Due to current limitations, OAuth tokens must be obtained manually and have a 1-hour lifetime. Use the following Python script to retrieve your token:

```
from pyiceberg.catalog import load_catalog
from pyiceberg.catalog.rest import RestCatalog

Replace with your Snowflake OAuth credentials

CLIENT_ID = "your_client_id"
CLIENT_SECRET = "your_client_secret"
TOKEN = f"{CLIENT_ID}:{CLIENT_SECRET}"

Replace with your Snowflake Polaris endpoint

URI = "https://your-account.snowflakecomputing.com/polaris/api/catalog"

catalog = RestCatalog(
 name="PGAA Test",
 warehouse="pgaa_test",
 uri=URI,
 credential=TOKEN,
 scope="PRINCIPAL_ROLE:read_write_all"
)

This will print your temporary token

print(catalog.properties["token"])
```

**Important:** Save the token output - you'll need it for the next step.

#### Step 2: Configure PGAA Catalog

Execute the following SQL commands to add and configure the Snowflake Polaris catalog in PGAA:

```
-- Remove existing catalog if present
SELECT bdr.replicate_ddl_command($$
SELECT pgaa.delete_catalog(snowflake_polaris);
```

(continues on next page)

(continued from previous page)

```

$$);

-- Add new catalog configuration
SELECT bdr.replicate_ddl_command($$
 SELECT pgaa.add_catalog(
 snowflake_polaris,
 iceberg-rest,
 {"url": "https://your-account.snowflakecomputing.com/polaris/api/catalog",
 "token": "YOUR_OAUTH_TOKEN_HERE",
 "warehouse": "pgaa_test"}
);
$$);

-- Import catalog metadata
SELECT bdr.replicate_ddl_command($$
 SELECT pgaa.import_catalog(snowflake_polaris);
$$);

```

### Step 3: Configure Write Permissions

Enable write operations to the catalog by configuring your BDR node group:

```

SELECT bdr.alter_node_group_option(
 (SELECT node_group_name FROM bdr.node_group
 WHERE node_group_name != world LIMIT 1),
 analytics_write_catalog,
 snowflake_polaris
);

```

### Setting Up AWS S3 Tables Integration

For S3 Tables integration, follow the [AWS S3 Tables Getting Started Guide](#) to create your table bucket, then use iceberg-s3tables as the catalog type when configuring PGAA:

```

SELECT bdr.replicate_ddl_command($$
 SELECT pgaa.add_catalog(
 s3_tables_catalog,
 iceberg-s3tables,
 {"bucket": "your-table-bucket",
 "region": "us-east-1",
 "access_key": "YOUR_ACCESS_KEY",
 "secret_key": "YOUR_SECRET_KEY"}
);
$$);

```

## 7.4.6 Verification and Testing

After configuration, verify your integration by testing these operations:

## Reading from Catalog

```
-- List available tables
SELECT * FROM pgaa.catalog_tables(your_catalog_name);

-- Query an Iceberg table
SELECT * FROM your_catalog_name.schema_name.table_name LIMIT 10;
```

## Writing to Catalog

```
-- Create a new Iceberg table
CREATE TABLE your_catalog_name.schema_name.new_table AS
SELECT * FROM your_postgresql_table;
```

## PGD Offloads

Verify that PGD can successfully offload data to the configured catalog according to your retention policies.

## External Tool Compatibility

Test that third-party analytical tools can read both tables and views created through PGAA.

### 7.4.7 Known Limitations

- **Token Management:** Snowflake Polaris tokens expire after 1 hour and must be manually refreshed
- **Naming Restrictions:** Catalogs with spaces in their names may cause query errors
- **OAuth Flow:** The OAuth client credentials flow must be performed manually outside of PGAA

### 7.4.8 Troubleshooting

#### Common Issues

##### Token Expiration Error

- **Symptom:** Queries fail with authentication errors after approximately 1 hour
- **Solution:** Regenerate the OAuth token and update the catalog configuration

##### Catalog Import Failures

- **Symptom:** `pgaa.import_catalog()` returns errors
- **Solution:** Verify network connectivity and ensure the catalog URL is accessible from all PostgreSQL nodes

##### Query Performance Issues

- **Symptom:** Slow query performance when accessing Iceberg tables
- **Solution:** Ensure proper partition pruning and consider creating materialized views for frequently accessed data

### 7.4.9 Next Steps

After successfully configuring catalog integration:

1. Establish token refresh procedures for production environments
2. Configure monitoring for catalog connectivity

3. Implement data lifecycle policies for automated offloading
4. Optimize query patterns for mixed PostgreSQL-Iceberg workloads

## 7.5 How-To Lakehouse Read With/Without A Catalog

### 7.5.1 Overview

This guide demonstrates how to configure PGAA Lakehouse clusters for accessing data lake formats, both with and without catalog integration. PGAA supports direct access to Delta Lake and Apache Iceberg tables, enabling analytical workloads on object storage without data movement.

### 7.5.2 Prerequisites

#### System Requirements

- PGAA Lakehouse cluster provisioned and accessible
- PostgreSQL File System (PGFS) version 2.0.1 or later recommended
- Administrative privileges for catalog and storage configuration

#### Access Requirements

- For catalog integration: Valid credentials for target catalog service
- For direct access: Read permissions on target S3 buckets or storage locations
- Network connectivity to storage endpoints

### 7.5.3 Configuration Approaches

PGAA supports three primary approaches for lakehouse data access:

1. **Catalog-managed access:** Full integration with external catalog services
2. **Direct Delta Lake access:** Query Delta tables without catalog dependency
3. **Direct Iceberg access:** Query Iceberg tables using metadata files

### 7.5.4 AWS S3 Tables Integration

#### Python Client Configuration

AWS S3 Tables requires REST API compatibility for Python clients. Configure PyIceberg as follows:

```
from pyiceberg.catalog import load_catalog

Define S3 Tables parameters

REGION = "eu-north-1" # Your S3 Tables bucket region
ARN = "arn:aws:s3tables:eu-north-1:0123456789:bucket/your-bucket"

s3tables_catalog = load_catalog(
 "s3tables_catalog",
 **{
 "type": "rest",
 "warehouse": ARN,
```

(continues on next page)

(continued from previous page)

```

 "uri": f"https://s3tables.{REGION}.amazonaws.com/iceberg",
 "rest.sigv4-enabled": "true",
 "rest.signing-name": "s3tables",
 "rest.signing-region": REGION,
 }
)

```

**Note:** PyIceberg versions  $\leq 0.9.1$  perform credential resolution at each API call and cannot pass S3 options directly during catalog definition.

## PGAA Native Configuration

PGAA implements a native Rust S3 Tables client, eliminating REST proxy overhead:

```

SELECT bdr.replicate_ddl_command($$
SELECT pgaa.delete_catalog(s3tables_catalog);
SELECT pgaa.add_catalog(
 s3tables_catalog,
 iceberg-s3tables,
 {"arn": "arn:aws:s3tables:eu-north-1:0123456789:bucket/your-bucket",
 "region": "eu-north-1"}
);
SELECT pgaa.import_catalog(s3tables_catalog);
$$);

```

### Key Differences:

- Catalog type: iceberg-s3tables (not iceberg-rest)
- Required parameters: ARN and region only
- Authentication: AWS credential chain (profiles, environment variables, IMDS)
- Explicit credentials not supported; relies on SDK credential resolution

## 7.5.5 Direct Delta Lake Access

### Storage Location Setup

Configure storage locations for Delta Lake tables:

```

-- PGFS 2.0.1 and later
SELECT pgfs.create_storage_location(
 biganimal-sample-data,
 s3://beacon-analytics-demo-data-us-east-1-prod,
 {"aws_skip_signature": "true"}
);

-- Legacy syntax (before PGFS 2.0.1)
SELECT pgfs.create_storage_location(
 biganimal-sample-data,
 s3://beacon-analytics-demo-data-eu-west-1-prod,
 NULL, -- managed storage location ID (unused)
 {"aws_skip_signature": "true"}, -- options

```

(continues on next page)

(continued from previous page)

```
{ } -- credentials
);
```

## Creating Delta Tables

Map Delta Lake tables using PGAA table access method:

```
- - Create customer table
CREATE TABLE public.customer () USING PGAA
WITH (
 pgaa.storage_location = biganimal-sample-data,
 pgaa.path = tpch_sf_1/customer
);

- - Verify table access
SELECT count(*) FROM public.customer;

- - Create additional TPC-H tables
CREATE TABLE public.lineitem () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/lineitem);

CREATE TABLE public.orders () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/orders);

CREATE TABLE public.nation () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/nation);
```

## Working with Large Datasets

Configure separate schema for scale testing:

```
CREATE SCHEMA tpch;

- - Create 1TB scale factor tables
CREATE TABLE tpch.lineitem () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1000/
↳lineitem);

CREATE TABLE tpch.customer () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1000/
↳customer);

- - Additional tables follow same pattern...
```

## 7.5.6 Direct Iceberg Access

### Configuring Iceberg Storage

```
- - Create storage location for Iceberg data
SELECT pgfs.create_storage_location(
 biganimal-sample-data-dev,
```

(continues on next page)

(continued from previous page)

```

s3://beacon-analytics-demo-data-us-east-1-dev,
{"aws_skip_signature": "true"}
);

-- Create Iceberg table reference
CREATE TABLE iceberg_table () USING PGAA
WITH (
 pgaa.storage_location = biganimal-sample-data-dev,
 pgaa.path = iceberg-example/default.db/iceberg_table,
 pgaa.format = iceberg -- Explicitly specify format
);

-- Query Iceberg data
SELECT * FROM iceberg_table ORDER BY key ASC;

```

## 7.5.7 Catalog Attachment

### Connecting to Existing Catalogs

Attach and query data from configured catalogs:

```

-- Add REST catalog
SELECT pgaa.add_catalog(
 lakekeeper-test,
 iceberg-rest,
 {
 "url": "https://catalog-endpoint.example.com",
 "token": "your-api-token",
 "warehouse": "warehouse-id",
 "danger_accept_invalid_certs": "true"
 }
);

-- Attach catalog to current session
SELECT pgaa.attach_catalog(lakekeeper-test);

-- Query catalog tables
SELECT COUNT(*) FROM tpch_sf_1.lineitem;

```

### Verifying Catalog Access

```

-- Check available schemas
SELECT * FROM information_schema.schemata
WHERE schema_name LIKE tpch%;

-- Verify offloaded data views
SELECT * FROM partitioned_table_offloaded LIMIT 10;

```

## 7.5.8 Query Validation

### Sample Analytical Query

Validate configuration with complex joins:

```
SELECT
 c_custkey,
 c_name,
 sum(l_extendedprice * (1 - l_discount)) AS revenue,
 c_acctbal,
 n_name,
 c_address,
 c_phone,
 c_comment
FROM
 customer,
 orders,
 lineitem,
 nation
WHERE
 c_custkey = o_custkey
 AND l_orderkey = o_orderkey
 AND o_orderdate >= CAST(1993-10-01 AS date)
 AND o_orderdate < CAST(1994-01-01 AS date)
 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
LIMIT 20;
```

#### Expected Results:

- Customer#000057040: revenue 734235.2455
- Customer#000143347: revenue 721002.6948
- Customer#000060838: revenue 679127.3077

## 7.5.9 Best Practices

### Storage Configuration

1. Use PGFS 2.0.1+ simplified syntax when available
2. Configure region-appropriate endpoints for optimal performance
3. Implement proper credential management through AWS credential chain

### Catalog Management

1. Regularly refresh OAuth tokens for REST catalogs
2. Use native clients when available (S3 Tables)
3. Import catalog metadata after configuration changes

## **Performance Optimization**

1. Partition large datasets appropriately
2. Create materialized views for frequently accessed data
3. Monitor query execution plans for optimization opportunities

### **7.5.10 Troubleshooting**

#### **Common Issues**

##### **Storage Access Errors**

- Verify S3 bucket permissions and network connectivity
- Check credential chain configuration for AWS authentication

##### **Catalog Connection Failures**

- Validate catalog endpoint URLs and credentials
- Ensure proper SSL certificate validation settings

##### **Query Performance Degradation**

- Review table statistics and partition pruning
- Consider data locality and caching strategies

### **7.5.11 Next Steps**

1. Configure monitoring for lakehouse queries
2. Establish data refresh schedules for materialized views
3. Implement cost optimization through intelligent caching
4. Develop governance policies for catalog access control

## **7.6 Configure Analytics Storage and Data Tiering with PGAA and PGD**

### **7.6.1 Table of Contents**

#### **Part 1: Foundation Setup**

- *Step 1: Initialize PGAA Extension and Performance Settings*
- *Step 2: Configure Storage Locations*
- *Step 3: Configure Analytics Write Location*
- *Step 4: Verify Seafoam Query Engine Activation*

#### **Part 2: Read-Only Analytics: Access Existing Data**

- *Step 5: Delta Lake Table Access*
- *Step 6: Execute Analytics Queries Across Delta Lake Data*
- *Step 7: Iceberg Table Access*
- *Step 8: Analyze Storage Characteristics*

### Part 3: Transactional Table Analytics Offload

- *Step 9: Create and Populate Transactional Table*
- *Step 10: Configure Analytics Offload*
- *Step 11: Create Analytics Reader Interface*
- *Step 12: Convert Transactional Table to Analytics-Only ]*

### Part 4: Advanced Table Management Operations

- *Step 13: Restore Table from Analytics to Transactional Storage*
- *Step 14: Handle Data Updates with Analytics Synchronization*
- *Step 15: Implement Partial Table Offload*

### Part 5: AutoPartition with Analytics Offload

- *Step 16: Configure Automated Partitioning*
- *Step 17: Load Test Data and Analyze Baseline Storage*
- *Step 18: Baseline Storage Analysis*
- *Step 19: Configure Automated Analytics Offload*
- *Step 20: Monitor Automated Offload Operations*
- *Step 21: Validate Storage Tier Distribution*
- *Step 22: Post-Migration Storage Analysis and ROI Calculation*

### Part 6: Catalog-Based Operations

- *Step 23: Environment Cleanup and Catalog Setup*
- *Step 24: External Data Loading and Catalog Configuration*
- Distributed Catalog Access
- *Step 26: Dynamic Catalog Discovery and Management*
- Catalog-Managed Analytics Operations
- Cross-Node Data Access Validation

## 7.6.2 Part 7: External Tool Integration

- *PyIceberg Catalog Access*
  - Unified Cold Storage View
- 

### 7.6.3 Steps Overview

This guide implements a complete analytics storage architecture through six progressive phases:

**Foundation (Steps 1-4):** Configure PGAA infrastructure, establish storage locations, and activate query engines for lakehouse operations.

**Read-Only Analytics (Steps 5-8):** Access existing Delta Lake and Iceberg datasets without data movement, demonstrating immediate analytics capabilities.

**Basic Offload (Steps 9-12):** Implement transactional table offload to analytics storage with data integrity validation and format conversion.

**Advanced Management (Steps 13-15):** Master bi-directional data movement, update synchronization, and selective archival strategies for complex scenarios.

**Automated Tiering (Steps 16-22):** Deploy AutoPartition with automated lifecycle management, capturing before/after storage optimization metrics.

**Catalog Integration (Steps 23-28):** Integrate external catalogs for enterprise-scale data sharing, multi-node access, and third-party tool compatibility.

---

### 7.6.4 Prerequisites

#### Technical Requirements

PostgreSQL with PGD High Availability must be operational with administrative privileges for catalog management, partition operations, and storage configuration. PGAA extension requires installation with network connectivity to external storage systems including S3-compatible endpoints. Sufficient bandwidth supports data transfer operations between storage tiers during offload processes.

#### Knowledge Requirements

Understanding of PostgreSQL partitioning concepts, system catalog structures, and distributed database management enables effective implementation. Familiarity with object storage systems, lakehouse architectures, and analytics workload patterns helps optimize configuration decisions. Basic knowledge of SQL performance analysis supports troubleshooting and optimization activities.

#### Environment Setup

Active PGD cluster with multiple nodes provides high availability foundation while PGAA extension enables analytics storage integration. Network security configurations must permit access to external storage endpoints with appropriate firewall and authentication settings. Monitoring infrastructure should track background task processing and storage utilization patterns.

### 7.6.5 Setup: Configure Analytics Infrastructure

#### Step 1: Initialize PGAA Extension and Performance Settings

Configure the analytics infrastructure with optimized settings for testing and production deployment scenarios.

```
-- Setup PGD configuration for efficient testing cycles
SET bdr.taskmgr_nap_time = 10000; -- AutoPartition worker runs every 10s vs default ↪ 290s
SET pgaa.max_replication_lag_s = 5; -- Analytics replication lag threshold
SET pgaa.flush_task_interval_s = 5; -- Frequent storage flush for testing

-- Initialize PGAA extension
CREATE EXTENSION IF NOT EXISTS pgaa CASCADE;
```

The performance settings accelerate background task processing during development and testing phases. Production environments should evaluate these settings against system resources and operational requirements. The PGAA extension provides the foundation for analytics storage integration and lakehouse table management.

## Step 2: Configure Storage Locations

Establish storage locations for both read-only analytics access and write operations with appropriate security configurations.

```
-- Create read-only storage location for existing Delta Lake data
SELECT bdr.replicate_ddl_command($$
 SELECT pgfs.create_storage_location(
 biganimal-sample-data,
 s3://beacon-analytics-demo-data-us-east-1-prod,
 {"aws_skip_signature": "true"}
);
$$);

-- Create development storage location for Iceberg tables
SELECT bdr.replicate_ddl_command($$
 SELECT pgfs.create_storage_location(
 biganimal-sample-data-dev,
 s3://beacon-analytics-demo-data-us-east-1-dev,
 {"aws_skip_signature": "true"}
);
$$);
```

Storage location configuration establishes the foundation for both read-only analytics access and write operations. The `aws_skip_signature` setting applies to public buckets but production environments require proper AWS credential configuration through IAM roles or credential files.

## Step 3: Configure Analytics Write Location

Set up the primary analytics storage location for automated data offload operations.

```
-- Create appliance storage location with dynamic path construction
SELECT bdr.replicate_ddl_command($$
 SELECT pgfs.create_storage_location(
 appliance-bucket,
 ${BUCKET_URL} || / ||
 (SELECT node_group_name FROM bdr.node_group WHERE node_group_name != world LIMIT
 ↪1) ||
 /analytics-offload
)
 $$);

-- Validate storage location accessibility
CREATE TABLE sentinel_sl_read_test () USING PGAA
WITH (pgaa.format = iceberg, pgaa.storage_location = appliance-bucket, pgaa.path =
 ↪nonexistent);

-- Should return ERROR: No valid Iceberg metadata file found (confirms bucket access)

-- Set as primary analytics write location
SELECT bdr.alter_node_group_option(
 (SELECT node_group_name FROM bdr.node_group WHERE node_group_name != world LIMIT 1),
 analytics_storage_location,
 appliance-bucket
);
```

The dynamic path construction includes the PGD node group name to prevent conflicts in multi-tenant environments. Storage location validation through deliberate error testing confirms proper authentication and network connectivity. Setting the analytics storage location activates the Seafowl query engine for lakehouse operations.

#### Step 4: Verify Seafowl Query Engine Activation

Confirm that the analytics query engine has started and is ready for lakehouse operations.

```
- - Verify Seafowl query engine is running
SELECT application_name FROM pg_stat_activity WHERE application_name LIKE %seafowl%;
- - Must show pgd seafowl_manager for proper operation
```

The Seafowl query engine provides optimized execution for analytics workloads across columnar storage formats. Proper activation is essential for performance optimization in lakehouse architectures.

### 7.6.6 Read-Only Analytics: Access Existing Data

#### Step 5: Delta Lake Table Access

Create analytics tables that provide direct access to existing Delta Lake datasets without data movement or ETL processes.

```
- - Create Delta Lake analytics tables pointing to existing data
CREATE TABLE public.customer () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/customer);

CREATE TABLE public.lineitem () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/lineitem);

CREATE TABLE public.nation () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/nation);

CREATE TABLE public.orders () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/orders);

CREATE TABLE public.part () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/part);

CREATE TABLE public.partsupp () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/partsupp);

CREATE TABLE public.region () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/region);

CREATE TABLE public.supplier () USING PGAA
WITH (pgaa.storage_location = biganimal-sample-data, pgaa.path = tpch_sf_1/supplier);
```

These table definitions create PostgreSQL interfaces to existing Delta Lake data without requiring data migration or format conversion. The PGAA access method automatically handles Delta Lake format parsing and query optimization.

## Step 6: Execute Analytics Queries Across Delta Lake Data

Validate analytics functionality through complex multi-table queries that demonstrate lakehouse performance characteristics.

```
-- Complex analytics query spanning multiple Delta Lake tables
SELECT
 c_custkey, c_name,
 sum(l_extendedprice * (1 - l_discount)) AS revenue,
 c_acctbal, n_name, c_address, c_phone, c_comment
FROM customer, orders, lineitem, nation
WHERE c_custkey = o_custkey
 AND l_orderkey = o_orderkey
 AND o_orderdate >= CAST(1993-10-01 AS date)
 AND o_orderdate < CAST(1994-01-01 AS date)
 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
LIMIT 20;
```

The query demonstrates complex join operations across multiple Delta Lake tables while maintaining familiar SQL syntax. Query execution leverages the Seafoal analytics engine for optimized performance across columnar data formats.

## Step 7: Iceberg Table Access

Configure access to Apache Iceberg tables through direct storage location mapping.

```
-- Create Iceberg analytics table
CREATE TABLE iceberg_table () USING PGAA
WITH (
 pgaa.storage_location = biganimal-sample-data-dev,
 pgaa.path = iceberg-example/default.db/iceberg_table,
 pgaa.format = iceberg
);

-- Query Iceberg data
SELECT * FROM iceberg_table ORDER BY key ASC;
```

Iceberg format specification enables access to Apache Iceberg tables with full metadata support including schema evolution, partition management, and time travel capabilities. The explicit format declaration ensures proper parsing of Iceberg metadata structures.

## Step 8: Analyze Storage Characteristics

Use analytics functions to understand storage utilization and performance characteristics of lakehouse tables.

```
-- Analyze storage statistics for Delta Lake tables
SELECT * FROM pgaa.lakehouse_table_stats(lineitem::regclass);

-- Analyze storage statistics for Iceberg tables
SELECT * FROM pgaa.lakehouse_table_stats(iceberg_table::regclass);
```

Storage statistics provide insights into compression ratios, file organization, and query performance characteristics

that inform optimization decisions. These metrics support capacity planning and performance tuning for analytics workloads.

## 7.6.7 Transactional Table Analytics Offload

### Step 9: Create and Populate Transactional Table

Establish a transactional table with representative data for offload testing and validation.

```
-- Create transactional table for offload testing
CREATE TABLE transactional_table (
 key BIGINT,
 inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 revenue DOUBLE PRECISION,
 comment TEXT,
 PRIMARY KEY (key, inserted_at)
);

-- Insert test data with temporal distribution
INSERT INTO transactional_table (
 SELECT
 g,
 2010-01-01 00:00:00::timestamp + INTERVAL 1 second *
 (EXTRACT (EPOCH FROM (CURRENT_TIMESTAMP - 2010-01-01 00:00:00::timestamp)) /
↪1000000 * g),
 g * 0.001,
 encode(sha256(g::text::bytea), hex)
 FROM generate_series(1, 1000000) g
);
```

The data generation creates realistic temporal distribution patterns while providing sufficient volume for performance testing. SHA256 hash generation simulates variable-length text content found in production systems.

### Step 10: Configure Analytics Offload

Enable automatic replication of transactional data to analytics storage for cost optimization and performance scaling.

```
-- Enable analytics replication for transactional table
ALTER TABLE transactional_table SET (pgd.replicate_to_analytics = TRUE);

-- Monitor replication progress
SELECT * FROM bdr.analytics_table;

-- Wait until table appears in results (~120s for initial sync)
```

Analytics replication creates an Iceberg representation of transactional data in the configured storage location. The initial synchronization process handles format conversion and establishes ongoing replication streams for data consistency.

### Step 11: Create Analytics Reader Interface

Establish direct access to the analytics-tier data through dedicated reader tables.

```
-- Create reader interface for analytics data
CREATE TABLE transactional_table_analytics_reader ()
USING PGAA
```

(continues on next page)

(continued from previous page)

```

WITH (
 pgaa.format = iceberg,
 pgaa.storage_location = appliance-bucket,
 pgaa.path = public.transactional_table
);

-- Verify data integrity across storage tiers
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue
FROM transactional_table_analytics_reader;
-- Should return: count=1000000, total_revenue=500000500

```

The analytics reader provides direct access to cost-effective storage while maintaining query transparency. Data integrity validation confirms successful format conversion and replication processes.

## Step 12: Convert Transactional Table to Analytics-Only

Transform the transactional table to use analytics storage directly while discarding local data.

```

-- Convert to analytics-only access
ALTER TABLE transactional_table
SET ACCESS METHOD PGAA,
SET (
 pgaa.auto_truncate = true,
 pgaa.format = iceberg,
 pgaa.storage_location = appliance-bucket,
 pgaa.path = public.transactional_table
);

-- Verify conversion maintained data access
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM transactional_
↪table;

```

The conversion process eliminates local storage overhead while maintaining transparent SQL access through the analytics tier. Auto-truncate removes local data after confirming analytics tier availability.

## 7.6.8 Advanced Table Management Operations

### Step 13: Restore Table from Analytics to Transactional Storage

Demonstrate the reversibility of analytics offload for scenarios requiring local high-performance access.

```

-- Restore table to transactional storage
ALTER TABLE transactional_table SET ACCESS METHOD heap, SET (pgd.replicate_to_
↪analytics = FALSE);

-- Verify restoration with query plan analysis
EXPLAIN SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM_
↪transactional_table;

-- Should show Parallel Seq Scan indicating local heap access

```

Restoration capabilities provide flexibility for changing access patterns or performance requirements. Query plan analysis confirms the transition back to local PostgreSQL heap storage with parallel processing capabilities.

## Step 14: Handle Data Updates with Analytics Synchronization

Demonstrate data modification workflows that maintain consistency across storage tiers.

```
- - Update data in transactional table
UPDATE transactional_table SET revenue = revenue * 2;

- - Re-enable analytics replication to propagate changes
ALTER TABLE transactional_table SET (pgd.replicate_to_analytics = TRUE);

- - Wait for synchronization (~120s) then verify changes propagated
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue
FROM transactional_table_analytics_reader;
- - Should show doubled revenue: total_revenue=1000001000
```

Update propagation maintains consistency between transactional and analytics tiers while handling data modifications efficiently. The synchronization process recreates the analytics representation with current data.

## Step 15: Implement Partial Table Offload

Execute selective data archival where only historical portions of tables move to analytics storage.

```
- - Perform partial offload with consistent transaction boundary
BEGIN;
CREATE TABLE transactional_table_before_2020 AS (
 SELECT * FROM transactional_table
 WHERE DATE_PART(year, inserted_at) < 2020
);
DELETE FROM transactional_table WHERE DATE_PART(year, inserted_at) < 2020;
COMMIT;

- - Configure partial table for analytics offload
ALTER TABLE transactional_table_before_2020 ADD PRIMARY KEY (key, inserted_at);
ALTER TABLE transactional_table_before_2020 SET (pgd.replicate_to_analytics = TRUE);

- - Convert to analytics-only access
ALTER TABLE transactional_table_before_2020
SET ACCESS METHOD PGAA,
SET (
 pgaa.auto_truncate = true,
 pgaa.format = iceberg,
 pgaa.storage_location = appliance-bucket,
 pgaa.path = public.transactional_table_before_2020
);

- - Verify temporal data distribution
SELECT
 COUNT(*) FILTER (WHERE inserted_at < 2020-01-01) AS before_2020,
 COUNT(*) FILTER (WHERE inserted_at >= 2020-01-01) AS after_2020
FROM transactional_table_before_2020;
```

Partial offload enables fine-grained data lifecycle management where different time periods have different storage tier requirements. Transaction boundaries ensure data consistency during the archival process.

## 7.6.9 AutoPartition with Analytics Offload

### Step 16: Configure Automated Partitioning

Establish time-based partitioning with automated lifecycle management for scalable data organization.

```
-- Create partitioned table structure
CREATE TABLE partitioned_table (
 key BIGINT,
 inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 revenue DOUBLE PRECISION,
 comment TEXT,
 PRIMARY KEY (key, inserted_at)
) PARTITION BY RANGE(inserted_at);

-- Configure AutoPartition without immediate offload
SELECT bdr.autopartition (
 relation := partitioned_table,
 partition_increment := 1 year,
 partition_initial_lowerbound := 2010-01-01 00:00:00,
 managed_locally := TRUE
);

-- Wait for partition creation across cluster
SELECT bdr.autopartition_wait_for_partitions_on_all_nodes(partitioned_table::regclass,
 → CURRENT_TIMESTAMP::text);
```

AutoPartition configuration creates annual partitions with boundaries that align with typical data access patterns. The initial configuration excludes offload settings to prevent premature data migration during setup and data loading phases.

### Step 17: Load Test Data and Analyze Baseline Storage

Populate partitions with representative data and establish baseline storage metrics for optimization analysis.

```
-- Load test data across temporal partitions
INSERT INTO partitioned_table (
 SELECT
 g,
 2010-01-01 00:00:00::timestamp + INTERVAL 1 second *
 (EXTRACT (EPOCH FROM (CURRENT_TIMESTAMP - 2010-01-01 00:00:00::timestamp)) /
 → 1000000 * g),
 g * 0.001,
 encode(sha256(g::text::bytea), hex)
 FROM generate_series(1, 1000000) g
);
```

Data loading creates approximately 65,000 records per annual partition with consistent storage characteristics that enable predictable cost modeling and performance analysis.

### Step 18: Baseline Storage Analysis

Create comprehensive storage analysis capabilities and capture pre-offload metrics for comparison purposes.

```
-- Create optimized row counting function
CREATE OR REPLACE FUNCTION temp_get_table_rowcount(schema_name text, table_name text)
```

(continues on next page)

(continued from previous page)

```

RETURNS bigint AS $$
DECLARE count bigint;
BEGIN
 -- Uses DirectScan for PGAA tables (1.8s vs 20s without optimization)
 EXECUTE FORMAT(SELECT COUNT(*) FROM %I.%I, schema_name, table_name) INTO count;
 RETURN count;
END $$ LANGUAGE plpgsql;

-- Comprehensive partition storage analysis
WITH partitioned_table_info AS (
 SELECT
 nspc.nspname AS schema_name,
 child.relname AS table_name,
 pg_catalog.pg_get_expr(child.relpartbound, child.oid) AS partition_constraint,
 pg_get_partition_constraintdef(child.oid) AS partition_constraint_full
 FROM pg_inherits inh
 JOIN pg_class parent ON inh.inhparent = parent.oid
 JOIN pg_class child ON inh.inhrelid = child.oid
 JOIN pg_namespace nspp ON parent.relnamespace = nspp.oid
 JOIN pg_namespace nspc ON child.relnamespace = nspc.oid
 WHERE parent.relname = partitioned_table AND nspp.nspname = public
),

pgaa_tables AS (
 SELECT
 m.schema_name, m.table_name, m.format,
 CASE
 WHEN m.storage_location_name IS NOT NULL THEN FORMAT(pgfs://%s/%s, m.storage_
↪location_name, m.path)
 WHEN (m.managed_by IS NOT NULL AND m.catalog_namespace IS NOT NULL AND m.
↪catalog_table IS NOT NULL)
 THEN FORMAT(%s.%s.%s, m.managed_by, m.catalog_namespace, m.catalog_table)
 END AS path,
 temp_get_table_rowcount(m.schema_name, m.table_name) AS num_rows,
 s.total_size AS total_size,
 pti.partition_constraint
 FROM pgaa.get_all_analytics_table_settings() m
 LEFT JOIN LATERAL pgaa.lakehouse_table_stats(FORMAT(%I.%I, m.schema_name, m.table_
↪name)::regclass) s ON TRUE
 JOIN partitioned_table_info pti ON m.schema_name = pti.schema_name AND m.table_name_
↪= pti.table_name
 WHERE m.storage_location_name IS NOT NULL OR (m.managed_by IS NOT NULL AND m.
↪catalog_namespace IS NOT NULL AND m.catalog_table IS NOT NULL)
),

local_tables AS (
 SELECT
 pn.nspname AS schema_name, pc.relname AS table_name,
 temp_get_table_rowcount(pn.nspname, pc.relname) AS num_rows,
 pg_total_relation_size(FORMAT(%I.%I, pn.nspname, pc.relname)::regclass) AS total_
↪size,
 pti.partition_constraint

```

(continues on next page)

(continued from previous page)

```

FROM pg_class pc
 JOIN pg_namespace pn ON pc.relnamespace = pn.oid
 JOIN pg_am am ON am.oid = pc.relam
 JOIN partitioned_table_info pti ON pn.nspname = pti.schema_name AND pc.relname =
↳pti.table_name
WHERE pn.nspname NOT IN (pg_catalog, information_schema, pg_toast, pg_internal,
↳pgaa, pgfs, bdr)
 AND pc.relkind = r AND am.amname != pgaa
),

all_leaf_tables AS (
 SELECT schema_name, table_name, format AS pgaa_format, path AS pgaa_path, num_rows,
↳total_size, partition_constraint FROM pgaa_tables
 UNION
 SELECT schema_name, table_name, NULL AS pgaa_format, NULL AS pgaa_path, num_rows,
↳total_size, partition_constraint FROM local_tables
),

final_table_info AS (
 SELECT
 schema_name, table_name,
 CASE
 WHEN pgaa_format = iceberg THEN iceberg
 WHEN pgaa_format = delta THEN delta
 ELSE heap
 END AS type,
 COALESCE(pgaa_path, local) AS location,
 num_rows, total_size AS total_size_bytes,
 SUBSTRING(partition_constraint, FROM []((^[^]+)) AS start_bound,
 SUBSTRING(partition_constraint, TO []((^[^]+)) AS end_bound
 FROM all_leaf_tables
 ORDER BY partition_constraint ASC
)

SELECT * FROM final_table_info;

```

**Baseline Results:** All partitions show type = 'heap' and location = 'local' with approximately 10MB per historical partition, totaling ~154MB for complete dataset in expensive transactional storage.

### Step 19: Configure Automated Analytics Offload

Enable automated partition tiering based on data age thresholds while maintaining query transparency.

```

-- Configure analytics offload for partitions older than one year
SELECT bdr.autopartition (
 relation := partitioned_table,
 partition_increment := 1 year,
 partition_initial_lowerbound := 2010-01-01 00:00:00,
 managed_locally := TRUE,
 analytics_offload_period := 1 year
);

```

The one-year threshold balances operational access requirements with storage cost optimization by keeping recent data

in high-performance storage while migrating historical partitions to cost-effective analytics tiers.

Step 20: Monitor Automated Offload Operations

Track background migration progress and identify completion status for storage tier transitions.

```
- - Monitor offload work queue until completion
SELECT
 wqs.workid,
 wis.ap_wi_started_at,
 wis.ap_wi_finished_at,
 wis.ap_wi_status,
 "partition",
 sql
FROM bdr.taskmgr_local_workitem_status wis
 RIGHT JOIN bdr.taskmgr_local_work_queue_status wqs ON wis.ap_wi_workid = wqs.workid
WHERE relname = public.partitioned_table
AND sql NOT LIKE %bdr%.autopartition_create_partition%
ORDER BY ap_wi_finished_at DESC;
```

Monitor until all ap\_wi\_finished\_at values are NOT NULL, indicating successful completion. Processing typically requires 10+ minutes for 14 historical partitions (2010-2023 when running in 2025).

Step 21: Validate Storage Tier Distribution

Confirm successful migration through storage tier assignment analysis and local storage reduction verification.

```
- - Examine storage tier assignments
dt+
- - Historical partitions should show "pgaa" with "0 bytes"
- - Recent partitions should show "heap" with actual storage allocation
```

Expected results show historical partitions with “pgaa” access method and zero local storage, confirming successful migration to analytics tier. Recent partitions maintain “heap” access with local storage for optimal operational performance.

Step 22: Post-Migration Storage Analysis and ROI Calculation

Execute comprehensive storage analysis to quantify optimization results and calculate return on investment.

```
- - Run same storage analysis query as Step 18 to capture post-migration metrics
- - Verify data integrity across storage tiers
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM partitioned_table;
- - Should maintain: count=1000000, total_revenue=500000500
```

Storage Optimization Results:

Metric	Before Migration	After Migration	Savings
Local Storage (Historical)	144MB	16MB	128MB (89% reduction)
Storage Cost Profile	100% premium storage	11% premium, 89% standard	60-80% cost reduction

continues on next page

Table 2 – continued from previous page

Metric		Before Migration	After Migration	Savings
<b>Query Performance</b>		Uniform fast access	Recent fast, historical acceptable	<b>Optimized by access pattern</b>
<b>Administrative head</b>	<b>Over-</b>	Manual management	Automated lifecycle	<b>Eliminated manual intervention</b>

The automated tiering delivers substantial infrastructure cost reductions while maintaining transparent query access across storage tiers. Recent data remains in high-performance storage for operational requirements while historical data optimizes for cost-effective analytics storage.

## 7.6.10 Catalog-Based Operations

### Step 23: Environment Cleanup and Catalog Setup

Prepare the environment for catalog-based operations by removing previous test configurations and establishing external catalog integration.

```
-- Clean up previous test configurations
DROP TABLE IF EXISTS partitioned_table CASCADE;
DROP TABLE IF EXISTS transactional_table CASCADE;
DROP TABLE IF EXISTS transactional_table_before_2020 CASCADE;
DROP TABLE IF EXISTS transactional_table_analytics_reader CASCADE;

-- Reset analytics storage location
SELECT bdr.alter_node_group_option(
 (SELECT node_group_name FROM bdr.node_group WHERE node_group_name != world LIMIT 1),
 analytics_storage_location,
 NULL
);
```

Environment cleanup ensures clean state for catalog integration testing while removing potential conflicts from previous configurations.

### Step 24: External Data Loading and Catalog Configuration

Load sample data into external catalog using PyIceberg data loader and configure PGD cluster for catalog access.

```
Load sample data into HCP Lakekeeper catalog

python loader.py \
 --ca-cert ca.crt \
 --bucket beacon-analytics-demo-data-eu-west-1-prod \
 --prefix tpch_sf_1/lineitem \
 --endpoint $PARAMS_URI \
 --token $API_WRITE \
 --warehouse $PARAMS_WAREHOUSE \
 --namespace tpch_sf_1 \
 --table-name lineitem
```

External data loading creates Iceberg tables in the catalog that will be accessible through PostgreSQL interfaces, demonstrating hybrid catalog capabilities.

```

-- Configure catalog connection with authentication
SELECT bdr.replicate_ddl_command($$
 SELECT pgaa.add_catalog(
 lakekeeper-test,
 iceberg-rest,
 {
 "url": "$PARAMS_URI",
 "token": "$API_WRITE",
 "warehouse": "$PARAMS_WAREHOUSE_ID",
 "danger_accept_invalid_certs": "true"
 }
)
$$);

-- Set catalog as analytics write destination
SELECT bdr.alter_node_group_option(
 (SELECT node_group_name FROM bdr.node_group WHERE node_group_name != world LIMIT 1),
 analytics_write_catalog,
 lakekeeper-test
);

-- Verify Seafoal activation for catalog operations
SELECT application_name FROM pg_stat_activity WHERE application_name LIKE %seafowl%;
-- Should show pgd seafowl_manager

```

Catalog configuration establishes authenticated connection to external Iceberg catalog while activating query engine capabilities for distributed analytics operations.

## Step 25: Distributed Catalog Operations

Attach external catalog and validate data accessibility across distributed PGD cluster nodes.

```

-- Attach catalog for distributed access
SELECT bdr.replicate_ddl_command($$
 SELECT pgaa.attach_catalog(
 lakekeeper-test
)
$$);

-- Validate external data access
SELECT COUNT(*) FROM tpch_sf_1.lineitem;
-- Expected: 6001215 rows

-- Examine sample data structure
SELECT * FROM tpch_sf_1.lineitem LIMIT 1;

-- Test namespace isolation with local transactional table
CREATE TABLE tpch_sf_1.sentinel_tx(value TEXT);
INSERT INTO tpch_sf_1.sentinel_tx VALUES (still alive);

```

Distributed catalog attachment provides uniform access to external data across all cluster nodes while maintaining namespace isolation between local and external tables. The sentinel table validates that catalog operations don't interfere with local transactional data.

## Step 26: Dynamic Catalog Discovery and Management

Demonstrate dynamic catalog content updates and automatic discovery of new tables.

```
Add additional table to external catalog

python loader.py \
 --ca-cert ca.crt \
 --bucket beacon-analytics-demo-data-eu-west-1-prod \
 --prefix tpch_sf_1/customer \
 --endpoint $PARAMS_URI \
 --token $API_WRITE \
 --warehouse $PARAMS_WAREHOUSE \
 --namespace tpch_sf_1 \
 --table-name customer
```

edb\_notranlate\_27 sql – Verify new table accessibility `SELECT * FROM tpch_sf_1.customer LIMIT 1;`

- Confirm sentinel table preservation `SELECT * FROM tpch_sf_1.sentinel_tx;` – Should return ‘still alive’

Dynamic catalog discovery enables real-time integration of new datasets while preserving local table isolation. The catalog refresh mechanism supports evolving data architectures without requiring manual configuration updates.

```
``` sql -- Disable automatic catalog refresh to prevent test interference SELECT bdr.
  replicate_ddl_command($$ SELECT pgaa.detach_catalog('lakekeeper-test'); $$);
```

Catalog detachment prevents automatic refresh operations from interfering with subsequent testing while maintaining established table mappings for continued access.

Step 27: Catalog-Managed Analytics Operations

Execute the complete analytics workflow using catalog-managed storage instead of direct storage locations.

```
``` sql – Recreate transactional table for catalog-based offload CREATE TABLE transactional_table ( key BIGINT,
 inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, revenue DOUBLE PRECISION, comment TEXT,
 PRIMARY KEY (key, inserted_at));
```

- Load test data `INSERT INTO transactional_table ( SELECT g, '2010-01-01 00:00:00'::timestamp + INTERVAL '1 second' * (EXTRACT (EPOCH FROM (CURRENT_TIMESTAMP - '2010-01-01 00:00:00'::timestamp)) / 1000000 * g), g * 0.001, encode(sha256(g::text::bytea), 'hex') FROM generate_series(1, 1000000) g);`
- Enable analytics replication to catalog `ALTER TABLE transactional_table SET (pgd.replicate_to_analytics = TRUE);`
- Wait for initial sync completion (~120s) `SELECT * FROM bdr.analytics_table;`

Catalog-managed operations provide centralized metadata management while maintaining the same operational patterns as direct storage approaches. The external catalog handles schema registration and metadata coordination across different analytics tools.

```
``` sql -- Create catalog-managed analytics reader CREATE TABLE transactional_table_
  analytics_reader () USING PGAA WITH ( pgaa.format = 'iceberg', pgaa.managed_by =
  'lakekeeper-test', pgaa.catalog_namespace = 'public', pgaa.catalog_table =
  'transactional_table' );
```

(continues on next page)

(continued from previous page)

```

- - Verify data integrity SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_
↪revenue FROM transactional_table_analytics_reader; -- Should return: count=1000000,↪
↪total_revenue=500000500

- - Convert original table to catalog-managed access ALTER TABLE transactional_table↪
↪SET ACCESS METHOD PGAA, SET ( pgaa.format = 'iceberg', pgaa.managed_by =
↪'lakekeeper-test', pgaa.catalog_namespace = 'public', pgaa.catalog_table =
↪'transactional_table', pgaa.auto_truncate = 'true' );

```

Catalog-managed table access provides the same functionality as direct storage while enabling integration with external analytics tools and providing centralized metadata management across different processing engines.

Step 28: Cross-Node Data Access Validation

Validate that catalog-managed data remains accessible across all PGD cluster nodes for high availability scenarios.

```
`` sql - Identify non-leader node for testing SELECT * FROM bdr.node_group_routing_summary;
```

- - Connect to non-leader node and validate data access – (Use kubectl or direct connection to non-leader node)
 - kubectl -n exec -c postgres -it -- psql bdrdb

```
SELECT COUNT(*) FROM tpch_sf_1.lineitem; -- Should return: 6001215
```

```
SELECT * FROM tpch_sf_1.lineitem LIMIT 1; -- Should return sample data
```

```
SELECT COUNT(*) AS count, ROUND(SUM(revenue)) AS total_revenue FROM transactional_table; -- Should re-
turn: count=1000000, total_revenue=500000500
```

```

Cross-node validation confirms that catalog-managed data access functions correctly↪
↪across all cluster members, providing high availability for analytics workloads.↪
↪Distributed access patterns ensure consistent query results regardless of↪
↪connection endpoint.

```

```
## External Tool Integration
```

```
### Step 29: PyIceberg Catalog Access
```

```

Validate that external tools can access catalog-managed data created through PGD↪
↪analytics operations.

```

```

`` python from pyiceberg.catalog import load_catalog from pyiceberg.catalog.rest↪
↪import RestCatalog import os

```

```

URI = os.getenv("PARAMS_URI") TOKEN = os.getenv("API_WRITE") WAREHOUSE = os.getenv(
↪"PARAMS_WAREHOUSE")

```

```

catalog = RestCatalog( name="PGAA Test", warehouse=WAREHOUSE, uri=URI, token=TOKEN,↪
↪ssl={"cabundle": "ca.crt"} )

```

```
# List all available tables across namespaces
```

```

print("Available tables:") for namespace in catalog.list_namespaces(): print(f"
↪{namespace[0]}: ") for _, table in catalog.list_tables(namespace): print(f"
↪{table}")

```

(continues on next page)

(continued from previous page)

```
# Verify PGD-created data accessibility

table = catalog.load_table(("public", "transactional_table")) print(f"Transactional_
↪table row count: {table.scan().count()}") print("Sample record:", table.
↪scan(limit=1).to_arrow())

# Validate partition data from AutoPartition operations

partition_tables = [t for _, t in catalog.list_tables("public") if t.startswith(
↪"partitioned_table")] if partition_tables: sample_partition = partition_tables[0]
↪table = catalog.load_table(("public", sample_partition)) print(f"Partition {sample_
↪partition} row count: {table.scan().count()}")
```

External tool validation confirms that data written by PGD analytics operations maintains full compatibility with standard Iceberg tooling, enabling integration with diverse analytics ecosystems.

Step 30: Unified Cold Storage View

Create and validate unified views of partitioned data that span both local and analytics storage tiers.

```
```sql – Verify unified partition view exists SELECT COUNT(*) FROM public.partitioned_table_offloaded;
```

- – If view doesn't exist, reimport catalog SELECT bdr.replicate\_ddl\_command(

*SELECT pgaa.import\_catalog('lakekeeper – test')*

);
- – Analyze hot vs cold data distribution DO \$\$ DECLARE total\_count INTEGER DEFAULT 0; table\_name TEXT; row\_count INTEGER; cold\_row\_count INTEGER DEFAULT 0; BEGIN
  - \* Count rows in local “hot” partitions FOR table\_name IN ( SELECT relname FROM pg\_class WHERE relname LIKE 'partitioned\_table%' AND relam = 2 – heap access method ) LOOP EXECUTE FORMAT('SELECT COUNT(\*) FROM public.%I', table\_name) INTO row\_count; total\_count := total\_count + row\_count; END LOOP;
  - \* Count rows in analytics “cold” view EXECUTE 'SELECT COUNT(\*) FROM partitioned\_table\_offloaded' INTO cold\_row\_count;

```
RAISE NOTICE 'Hot storage rows: %', total_count; RAISE NOTICE 'Cold storage rows: %', cold_row_count;
RAISE NOTICE 'Total rows: %', total_count + cold_row_count; END $$; ```
```

The unified view provides seamless access to complete datasets spanning multiple storage tiers while maintaining performance optimization through intelligent data placement. This architecture enables comprehensive analytics across both operational and historical data without complex data federation requirements.

## 7.6.11 Troubleshooting and Validation

### Performance Monitoring and Optimization

Query performance characteristics vary between storage tiers, with local PostgreSQL storage optimized for low-latency transactional operations while analytics storage optimizes for high-throughput analytical workloads. Monitor query execution plans to understand access patterns and optimize accordingly.

Storage utilization analysis through `pgaa.lakehouse_table_stats` provides insights into compression ratios, file organization, and access patterns that inform optimization decisions. Regular analysis helps identify opportunities for further cost reduction or performance improvements.

### Common Issues and Resolution

**Catalog Connectivity:** Verify network access to external catalog endpoints and validate authentication credentials. Use deliberate error testing to confirm proper connectivity and permissions.

**Background Task Processing:** Monitor work queue status through `bdr.taskmgr_local_workitem_status` and adjust processing intervals for development environments while maintaining production defaults for resource optimization.

**Storage Tier Synchronization:** Ensure consistent storage tier assignments across distributed cluster nodes and verify data integrity after migration operations through row counts and aggregate calculations.

**External Tool Compatibility:** Validate that catalog-managed tables maintain full compatibility with external analytics tools through direct testing with PyIceberg, PySpark, and other ecosystem components.

### Security and Operational Considerations

Production deployments require proper SSL certificate validation, secure credential management, and network access controls that protect authentication flows and data transfers. Regular security audits should validate access permissions and credential rotation procedures.

Backup and recovery strategies must account for data distributed across multiple storage systems with different backup mechanisms and recovery characteristics. Develop procedures that ensure consistent recovery points across storage tiers.

Capacity planning should consider both immediate storage reduction benefits and long-term growth patterns as new data arrives and ages through different storage tiers. Monitor cost optimization effectiveness and adjust policies based on actual access patterns.

The comprehensive analytics storage configuration provides automated data tiering capabilities that balance operational performance requirements with storage cost optimization while maintaining transparent SQL access across hybrid storage architectures. The integration with external catalogs enables enterprise-scale data sharing and analytics tool compatibility while preserving PostgreSQL as the primary interface for both transactional and analytical workloads.