



EDB

Postgres® for the AI Generation

Lakehouse analytics

Release January 28, 2025

EnterpriseDB Corporation.

Jan 29, 2025

Contents

1	Introduction	1
1.1	Rapid analytics for Postgres	1
1.2	Postgres native	1
1.3	Vectorized execution	2
1.4	Columnar storage	2
1.5	Lakehouse Sync	2
1.6	Fully managed service	2
1.7	Try it today	2
2	Concepts - EDB Postgres Lakehouse	3
2.1	Major concepts	3
2.1.1	Lakehouse node	4
2.1.2	Lakehouse tables	4
2.1.3	Managed storage location	4
2.1.4	Lakehouse Sync	4
2.2	Terminology	4
2.2.1	Lakehouse	4
2.2.2	Vectorized query engine	5
2.2.3	Delta tables	5
2.3	How it works	5
2.3.1	Level 100	5
2.3.2	Level 200	6
2.3.3	Level 300	6
3	Quick Start - EDB Postgres Lakehouse	9
3.1	Introduction	9
3.1.1	What to know	9
3.1.2	Basic architecture	10
3.2	Getting started	10
3.2.1	Create a Lakehouse node	10
3.3	Operating a Lakehouse node	11
3.3.1	Connect to the node	11
3.3.2	Understand the constraints	12
3.4	Inspect the benchmark datasets	12
3.5	Query the benchmark datasets	12

4	Querying Delta Lake Tables in S3-compatible object storage	15
4.1	Overview	15
4.2	Prerequisites	15
4.3	Creating an External Table	16
4.4	Querying an External Table	16
5	Lakehouse Sync	17
5.1	Overview	17
5.2	Performing a Lakehouse Sync	17
5.2.1	Prerequisites	17
5.2.2	Navigate to Lakehouse Sync	17
5.2.3	Define Lakehouse Sync	18
5.2.4	Selecting Tables	19
5.2.5	Start Lakehouse Sync	19
6	Reference - EDB Postgres® AI Lakehouse	21
6.1	Supported cloud providers and regions	21
6.2	Supported AWS instances	22
6.3	Benchmarking datasets	22
6.4	Queries	23
6.5	Delta Lake Table tools	23
6.5.1	Creating Delta Lake Tables	23
6.6	PGAA functions reference	24
6.6.1	pgaa.create_storage_location	24
6.7	DirectScan, fallback modes, and EXPLAIN	24
6.8	User management	25
6.9	Loading data (sync or bring your own)	25
6.9.1	Loading data with Lakehouse sync	25
6.9.2	Bringing your own data	26

CHAPTER 1

Introduction

EDB Postgres Lakehouse extends the power of Postgres to analytical workloads, by adding a vectorized query engine and separating storage from compute. Building a data Lakehouse has never been easier: just use Postgres.

1.1 Rapid analytics for Postgres

Postgres Lakehouse is a core offering of the EDB Postgres[®] AI platform, extending Postgres to support analytical queries over columnar data in object storage, while keeping the simplicity and ease of use that Postgres users love.

With Postgres Lakehouse, you can query your Postgres data with a Lakehouse node, an ephemeral, scale-to-zero compute resource powered by Postgres that's optimized for vectorized query execution over columnar data.

1.2 Postgres native

Never leave the Postgres ecosystem.

Postgres Lakehouse nodes run either EDB Advanced Server (EPAS) or EDB Postgres Extended (PGE) as the Postgres engine, with data for analytics stored as columnar tables in object storage using the open source Delta Lake protocol.

EDB Postgres Lakehouse is “just Postgres” – you can query it with any Postgres client, and it fully supports all Postgres queries, functions and statements, so there's no need to change existing queries or reconfigure business intelligence software.

1.3 Vectorized execution

Postgres Lakehouse uses Apache DataFusion's vectorized SQL query engine to execute analytical queries 5-100x faster (30x on average) compared to native Postgres, while still falling back to native execution when necessary.

1.4 Columnar storage

Postgres Lakehouse is optimized to query "Lakehouse Tables" in object storage, extending the power of open source database to open table formats. Currently, it supports querying "Delta Tables" stored according to the Delta Lake protocol.

1.5 Lakehouse Sync

You can sync your own data from tables in transactional sources (initially, EDB Postgres® AI Cloud Service databases) into Lakehouse Tables in Storage Locations (initially, managed locations in S3 object storage).

1.6 Fully managed service

You can launch Postgres Lakehouse nodes using the EDB Postgres AI Cloud Service (formerly EDB BigAnimal). Point a Lakehouse node at a storage bucket with some Delta Tables in it, and get results of analytical (OLAP) queries in less time than if you queried the same data in a transactional Postgres database.

Postgres Lakehouse nodes are available now for customers using EDB Postgres AI - Hosted environments on AWS, and will be rolling out to additional cloud environments soon.

1.7 Try it today

It's easy to start using Postgres Lakehouse. Provision a Lakehouse node in five minutes, and start querying pre-loaded benchmark data like TPC-H, TPC-DS, Clickbench, and the 1 Billion Row challenge.

Concepts - EDB Postgres Lakehouse

EDB Postgres Lakehouse is the solution for running Rapid Analytics against operational data on the EDB Postgres[®] AI platform.

2.1 Major concepts

- **Lakehouse nodes** query **Lakehouse tables** in **managed storage locations**.
- **Lakehouse Sync** can create **Lakehouse tables** from **Transactional tables** in a source database.

Here's how it fits together:

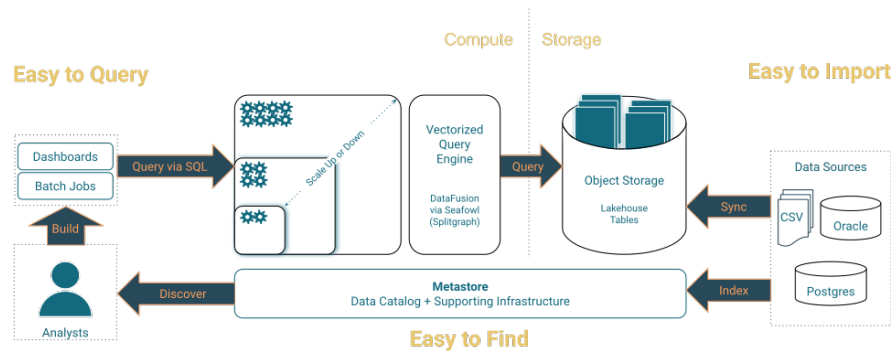


Fig. 1: Level 50 basic architecture

2.1.1 Lakehouse node

A Postgres Lakehouse node is Postgres, with a Vectorized Query Engine that's optimized to query Lakehouse tables, but still fall back to Postgres for full compatibility.

Lakehouse nodes are stateless and ephemeral. Scale them up or down based on workload requirements.

2.1.2 Lakehouse tables

Lakehouse Tables are stored using highly compressible, columnar storage formats optimized for analytics and interoperable with the rest of the Analytics ecosystem. Currently, Postgres Lakehouse nodes can read tables stored using the Delta Protocol ("delta tables"), and Lakehouse Sync can write them.

2.1.3 Managed storage location

A *managed storage location* is where you can organize Lakehouse tables in object storage, so that Postgres Lakehouse can query them.

A managed storage location is a location in object storage where we control the file layout and write Lakehouse Tables on your behalf. Technically, it's an implementation detail that we store these in buckets. This is really a subset of an upcoming "Storage Location" feature that will also support "External Storage Locations," where you bring your own bucket.

2.1.4 Lakehouse Sync

Lakehouse Sync is a data migration service offered as part of the EDB Postgres AI platform. It can "sync" tables from a transactional database, to Lakehouse Tables in a destination Storage Location. Currently, it supports source databases hosted in the EDB Postgres AI Cloud Service (formerly known as BigAnimal). In the future, it will support more source databases (such as Oracle) and deployment locations (on-premises databases). It's built using [Debezium](#).

2.2 Terminology

2.2.1 Lakehouse

The " [Lakehouse architecture](#) " is a data engineering practice, which is a portmanteau of "Data *Lake*" and "Data Ware_house_," offering the best of both. The central tenet of the architecture is that data is stored in Object Storage, generally in columnar formats like Parquet, where different query engines can process it for their own specialized purposes, using the optimal compute resources for a given query.

2.2.2 Vectorized query engine

A vectorized query engine is a query engine that's optimized for running queries on columnar data. Most analytics engines use vectorized query execution. Postgres Lakehouse uses [Apache DataFusion](#).

2.2.3 Delta tables

We use the term “Lakehouse tables” to avoid overcommitting to a particular format (since we might eventually support Iceberg or Hudi, for example). But technically, we're using [Delta Tables](#). A Delta Table is a well-defined container of Parquet files and JSON metadata, according to the “Delta Lake” spec and open protocol. Delta Lake is a Linux Foundation project.

2.3 How it works

Postgres Lakehouse is built using a number of technologies:

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

2.3.1 Level 100

The most important thing to understand about Postgres Lakehouse, is that it separates storage from compute. This allows you to scale them independently, which is ideal for analytical workloads where queries can be unpredictable and spiky. You would not 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.

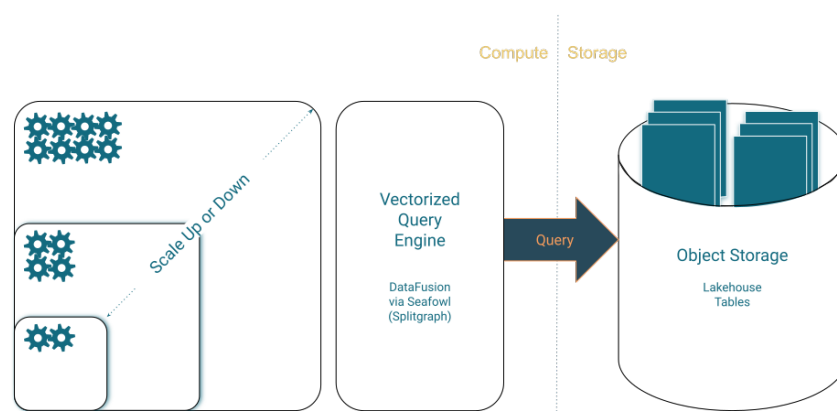


Fig. 2: Level 100 Architecture

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.

2.3.2 Level 200

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

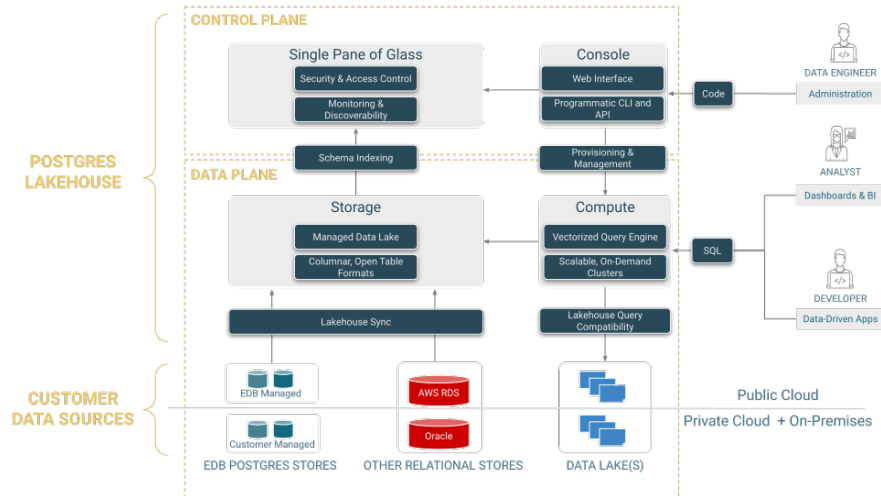


Fig. 3: Level 200 Architecture

2.3.3 Level 300

Here' s the more detailed, zoomed-in view of “what’ s in the box” :

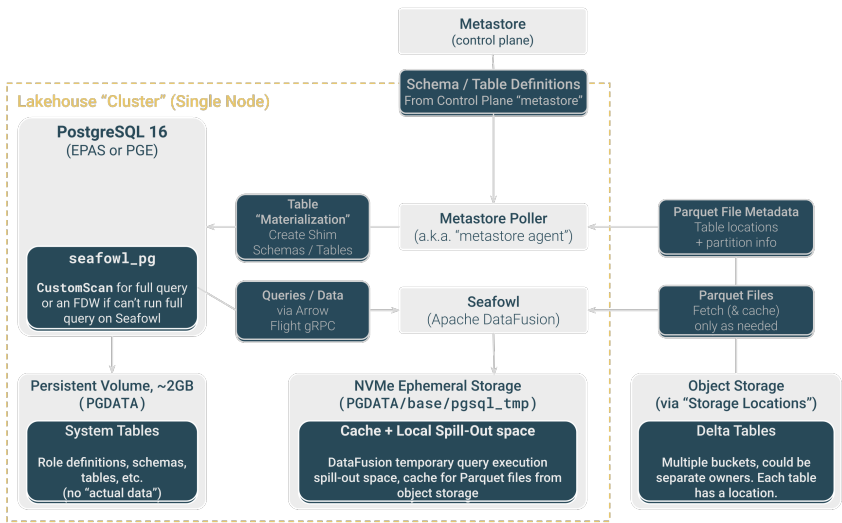


Fig. 4: Level 300 Architecture

Quick Start - EDB Postgres Lakehouse

In this guide, you will:

1. Create a Lakehouse node
2. Connect to the node with your preferred Postgres client
3. Query sample data (TPC-H, TPC-DS, Clickbench, or 1BRC) in object storage

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

3.1 Introduction

Postgres Lakehouse is a new type of Postgres “cluster” (it’s really just one node) that you can provision in EDB Postgres® AI Cloud Services (formerly known as “BigAnimal”). It includes a vectorized query engine (based on Apache [DataFusion](#)) for fast queries over columnar “Lakehouse Tables” in object storage (using [Delta Lake](#) protocol).

3.1.1 What to know

For more details, see [Concepts - EDB Postgres Lakehouse](#) and [PGAA functions reference](#), but this should cover the gist of it:

- Postgres Lakehouse nodes are ephemeral, but they have NVMe hard drives for local cache and spill out space. Currently we support deploying them to EDB-hosted regions in AWS.
- Postgres Lakehouse nodes are pre-configured 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 will persist between instance restarts and will be saved as part of backup/restore operations. Otherwise, Lakehouse tables will not be part of backups, since they are ultimately stored in object storage.

3.1.2 Basic architecture

Here's what's in the box of a Lakehouse node:

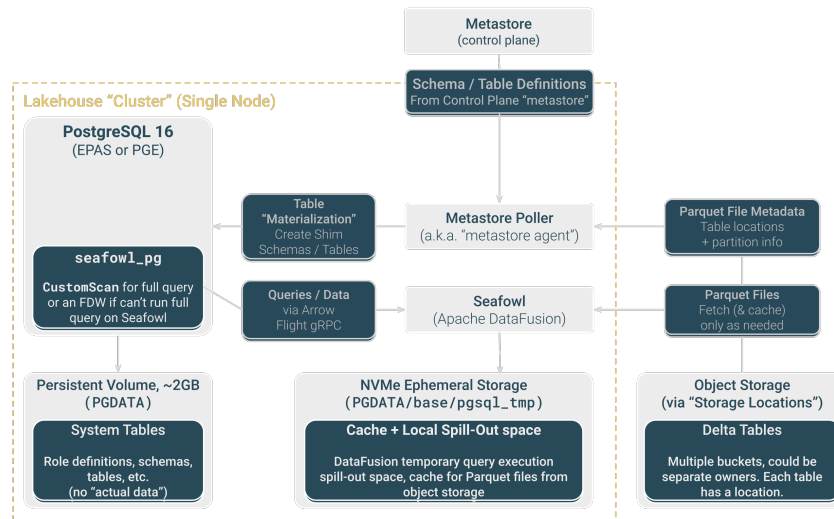


Fig. 1: Level 300 Architecture

3.2 Getting started

You will need an EDB Postgres AI account. Once you've logged in and created a project, you can create a cluster.

3.2.1 Create a Lakehouse node

You will see a "Lakehouse Analytics" option under the "Create New" dropdown on your project page:

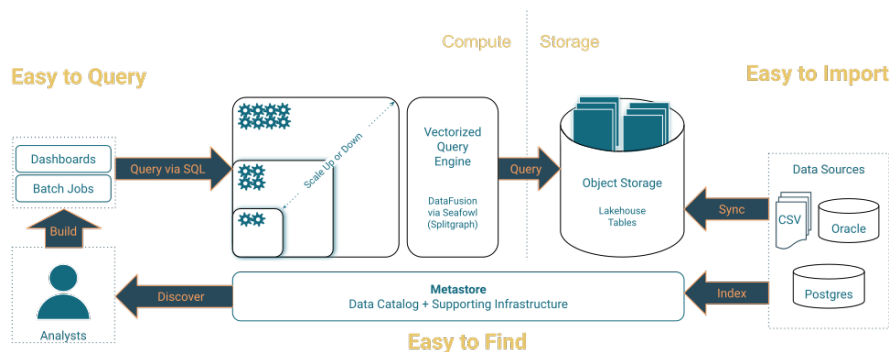


Fig. 2: Level 50 basic architecture

Selecting the "Lakehouse Analytics" button starts a configuration wizard that looks like this:

Provisioning the cluster may take 10-15 minutes.

Fig. 3: Create Lakehouse Node Wizard Step 1

A few notes:

- You will need to provide your own password. Create a secure password and save it to your password manager. We will not save it on our side.
- Only AWS is supported, in BigAnimal Hosted regions.
- You can choose EDB Postgres Advanced Server (EPAS) or EDB Postgres Extended (PGE), and the pricing calculator uses the same formula with license costs as it would for a normal EPAS or PGE single-node cluster.
- Only instances with NVMe drives are supported. This is the M6ID family. The NVMe drives are used for spill-out space while processing queries. All data on the NVMe drives will be lost on cluster shutdown. Persistent data in system tables (users, roles, etc) is stored in an attached block storage device and will survive a restart or backup/restore cycle.
- Only Postgres 16 is supported.

For more notes about supported instance sizes, see [Reference - Supported AWS instances](#).

3.3 Operating a Lakehouse node

3.3.1 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 Cloud Service (formerly known as BigAnimal): 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 should be able to 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.

3.3.2 Understand the constraints

- Every cluster uses EPAS or PGE. 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 pre-loaded 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 supported at the moment. Bring Your Own Account (BYOA) is not supported.
- You can deploy a cluster in any region that is activated in your EDB Postgres AI Account. Each region has a bucket with a copy of the benchmarking data, and so when you launch a cluster, it will use 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, e.g. roles and users and grants. If you restart the cluster, or backup the cluster and then restore it, it will restore these system tables. But the data in object storage will remain untouched.
- The cluster supports READ ONLY queries of the data in object storage (but it supports write queries to system tables for creating users, etc.). You cannot write directly to object storage. You cannot create new tables.
- If you want to load your own data into object storage, see [Reference - Bring your own data](#) .

3.4 Inspect the benchmark datasets

Inspect the Benchmark Datasets. Every cluster has some benchmarking data available out of the box. If you are 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](#)](./reference/datasets).

3.5 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;
```

(continues on next page)

(continued from previous page)

```
+-----+
| count |
+-----+
| 1000000000 |
+-----+
SELECT 1
Time: 0.651s
```

Note: Do not use `search_path` ! Read the [PGAA functions reference](#) page for more gotchas and information about syntax/query compatibility.

Querying Delta Lake Tables in S3-compatible object storage

4.1 Overview

External tables allow you to access and query data stored in S3-compatible object storage using SQL. You can create an external table that references data in S3-compatible object storage and query the data using standard SQL commands.

4.2 Prerequisites

- An EDB Postgres AI account and a Lakehouse node.
- An S3-compatible object storage location with data stored as Delta Lake Tables. * See *Bringing your own data* for more information on how to prepare your data.
- Credentials to access the S3-compatible object storage location, unless it is a public bucket. * These credentials will be stored within the database. We recommend creating a separate user with limited permissions for this purpose.
- be slow because of cross-region latencies
- will incur AWS costs (between \$0.01 and \$0.02 / GB) for data transfer. Currently these egress costs are not passed through to you but we do track them and reserve the right to terminate an instance.

The first step is to create an external storage location which references S3-compatible object storage where your data resides. A storage location is an object within the database which you refer to to access the data; each storage location has a name for this purpose.

Creating a named storage location is performed with SQL by executing the `pgaa.create_storage_location` function. `pgaa` is the name of the extension and namespace that provides the functionality to query external storage locations. The `create_storage_location` function takes a name for the new storage location, and the URI of the S3-compatible object storage location as parameters. The function optionally can take a third parameter, `options`, which is a JSON object for specifying optional settings, detailed in the *PGAA functions reference*. For example, in the options, you can specify the access key ID and secret access key for the storage location to enable access to a private bucket.

The following example creates an external table that references a public S3-compatible object storage location:

```
SELECT pgaa.create_storage_location(  
  name => sample-data,  
  url => s3://pgaa-sample-data-eu-west-1,  
  options => {},  
  msl_id => NULL  
);
```

The next example creates an external storage location that references a private S3-compatible object storage location:

```
SELECT pgaa.create_storage_location(  
  name => private-data,  
  url => s3://my-private-bucket,  
  options => {"access_key_id": "my-access-key-id", "secret_access_key": "my-secret-access-  
key"},  
  msl_id => NULL  
);
```

4.3 Creating an External Table

After creating the external storage location, you can create an external table that references the data in the storage location. The following example creates an external table that references a Delta Lake Table in the S3-compatible object storage location:

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

Note that the schema is not defined in the CREATE TABLE statement. The pgaa extension expects the schema to be defined in the storage location, and the schema itself is derived from the schema stored at the path specified in the pgaa.path option. The pgaa extension will infer the best Postgres-equivalent data types for the columns in the Delta Table.

4.4 Querying an External Table

After creating the external table, you can query the data in the external table using standard SQL commands. The following example queries the external table created in the previous step:

```
SELECT COUNT(*) FROM public.customer;
```

5.1 Overview

Performing a Lakehouse Sync is a way to capture information from a transactional database at a point in time and sync that information to a Managed Store Location (MSL).

The Lakehouse sync process organizes the transactional database data into Lakehouse tables stored in the MSL. This process allows the data to be queried by a Lakehouse node, which is optimized for higher-performance queries using a vectorized query engine designed for Lakehouse tables.

5.2 Performing a Lakehouse Sync

5.2.1 Prerequisites

- a Postgres cluster hosted and managed by EDB Postgres AI[®] Cloud Service

5.2.2 Navigate to Lakehouse Sync

1. Go to the [EDB Postgres AI Console](#) .
2. From the landing page, select the project with the database instance you want to sync. If it's not shown on the landing page, select the **View Projects** link in the **Projects** section and select your project from there.
3. Select the **Migrate** dropdown in the left navigation bar and then select **Migrations**.
4. Select the **Create New Migration** button.

5.2.3 Define Lakehouse Sync

5. Give the sync a **Name**, then select a **Source Cluster** and the **Database** you want to sync.
6. If you have already created an MSL you want to use, select that MSL from the list of available MSLs and move on to *Selecting Tables* below. If not, select the **Create New Managed Storage Location** button to open the **Add Managed Storage Location** dialog.
7. Select the AWS region for the new MSL.
8. Set a location prefix in the form near the bottom of the **Add Managed Storage Location** dialog to complete the definition of the MSL. A location prefix is a unique name used to identify any resources and assets associated with the MSL.

Provider	Region	Deployment	Storage Locations
☐ aws AWS	AP South 1	EDB Postgres® AI Cloud Service	0
☐ aws AWS	AP SouthEast 2	EDB Postgres® AI Cloud Service	0
☐ aws AWS	EU Central 1	EDB Postgres® AI Cloud Service	0
☐ aws AWS	EU West 1	EDB Postgres® AI Cloud Service	1
☐ aws AWS	EU West 2	EDB Postgres® AI Cloud Service	0

Region: No region selected

Prefix:

Cancel Create Managed Storage Location

Fig. 1: List of MSLs

9. Select the **Create Managed Storage Location** button.

5.2.4 Selecting Tables

10. Select the **Tables** tab next to the **Get Started** tab near the top of the page and select which tables and columns you wish to be included in the migration.

5.2.5 Start Lakehouse Sync

11. Select the **Start Lakehouse Sync** button.
12. If successful, you'll see your Lakehouse sync with the 'Creating' status under 'MOST RECENT' migrations on the Migrations page. The time taken to perform a sync can depend upon how much data is being synchronized and may take several hours.

Note: The first sync in a project will take a couple of hours due to the provisioning of the required infrastructure.

Reference - EDB Postgres® AI Lakehouse

EDB Postgres® AI Lakehouse is an early product. Eventually, it will support deployment modes across multiple clouds and on-premises. However, currently it's fairly limited in terms of where you can deploy it and what data you can query with it.

To get the best experience with Lakehouse, you should follow the [Quick Start - EDB Postgres Lakehouse](#) to query benchmarking data. Then you can try loading your own data with Lakehouse Sync or you can bring your own data and use [external tables](#). If you're intrigued, reach out to us and we can talk more about your use case and potential opportunities.

This section details some of the important things you should know.

6.1 Supported cloud providers and regions

AWS only: Currently, support for all Lakehouse features (Lakehouse nodes, Managed Storage Locations, and Lakehouse Sync) is limited to AWS.

EDB-hosted only: “Bring Your Own Account” (BYOA) regions are NOT currently supported for Lakehouse resources. Support is limited to ONLY **EDB Postgres® AI Hosted** environments on AWS (a.k.a. “EDB-Hosted AWS regions”).

This means you can select from one of the following regions:

- North America * US East 1 * US East 2 * US West 2
- Europe * EU Central 1 * EU West 1 * EU West 2
- Asia * AP South 1
- Australia * AP SouthEast 2

To be precise:

- Lakehouse nodes can only be provisioned in EDB-hosted AWS regions
- Managed Storage Locations can only be created in EDB-hosted AWS regions
- Lakehouse Sync can only sync from source databases in EDB-hosted AWS regions

These limitations will be removed as we continue to improve the product. Eventually, we will support BYOA, as well as Azure and GCP, for all Lakehouse use cases.

6.2 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)

6.3 Benchmarking datasets

When you provision a Lakehouse node, it comes pre-configured to point to a public S3 bucket in its same region, containing sample benchmarking datasets.

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; !!!
```

6.4 Queries

In general, **READ ONLY** queries are supported. You cannot write directly to object storage. This includes all Postgres built-in functions, statements and types. It also includes any of those provided by EPAS or PGE, depending on which distribution you choose to deploy.

In general, you cannot insert, update, delete or otherwise modify data. You can use `CREATE TABLE` but only to create a normal Postgres table on the node, bearing in mind that the node is ephemeral and will be destroyed when you terminate it. You can also `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 with Lakehouse Sync. See [Loading data \(sync or bring your own\)](#) 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 `CREATE USER` or `CREATE ROLE` or `GRANT USAGE`, and these users/roles/grants will survive restarts and restores.

6.5 Delta Lake Table tools

6.5.1 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 has 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 will export 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 now 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 [External Tables](#) for the details on how to do that.

6.6 PGAA functions reference

6.6.1 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

6.7 DirectScan, fallback modes, and EXPLAIN

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, this will be more fine-grained 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 will fall 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.

You can check which mode is being used by running an 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_linenumber, 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 only involved 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, we 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.

6.8 User management

When you provision a Lakehouse node, you must provide a password. We do not save this password. You will need it to login as the `edb_admin` user. This is not a superuser account, but it does have the ability to create users and roles and grants. Thus, you can either share the credentials for `edb_admin` itself, or you can create other users and distribute those.

6.9 Loading data (sync or bring your own)

6.9.1 Loading data with Lakehouse sync

If you have a transactional database running in EDB Postgres AI Cloud Service, then you can sync tables from this database into a Managed Storage Location. See [How to lakehouse sync](#) for further details.

6.9.2 Bringing your own data

It's possible to point your Lakehouse node at an arbitrary S3 bucket with Delta Tables inside of it. However, this comes with some major caveats (which will eventually be resolved):

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 obtain and use that utility.

For further details, see the [External Tables](#) documentation.