



DEVELOPER DOCUMENTATION

Groundzero Lakehouse Runtime

SQL over Apache Iceberg tables: warehouses, sessions, queries and governance.

Developer documentation

Edition 2026-10-02

Runtime environments: Spark, DuckDB, Standard, Lakehouse

Contents

Guide - Lakehouse Runtime

1	Overview	5
1.1	What the Lakehouse is	6
1.2	Warehouses, databases and tables	6
1.3	Choose the Lakehouse	6
1.4	How a query runs	7
1.5	Versions	8
2	Quickstart	9
2.1	1. Open the Lakehouse and pick a warehouse	9
2.2	2. Choose a compute and start a session	9
2.3	3. Create a database and a table	12
2.4	4. Insert rows	13
2.5	5. Query the table	13
2.6	6. Save and export	14
2.7	7. Stop the session	14
3	How-to guides	15
3.1	Browse databases, tables and columns	15
3.2	Write queries	16
3.3	Create and alter Iceberg tables	19
3.4	Insert, update, delete and merge	21
3.5	Time travel and snapshots	22
3.6	Metadata tables	23
3.7	Table properties	24
3.8	Maintenance	25
3.9	Long-running queries and stopping a query	27
3.10	Results, paging and exports	28
3.11	Saved queries	31
3.12	Governed tables, data policies and table access	32
3.13	Read lakehouse tables from jobs and datasets	35
3.14	Backups, schedules and restore	35
3.15	Audit logs	37
4	SQL reference	38
4.1	Statements and the permission each needs	38
4.2	Queries	39
4.3	Time travel	39
4.4	Creating and changing tables	40
4.5	Changing data	40
4.6	Databases and information	40
4.7	Table procedures	41
4.8	Blocked or restricted statements	41
4.9	Behaviour of SQL in the Lakehouse	42
5	Limits and behaviour	43

5.1	Sessions	43
5.2	Queries and results	43
5.3	Exports	43
5.4	Other values	44
6	Troubleshooting	45
7	Examples gallery	48
7.1	Example 1: Revenue by channel and month	48
7.2	Example 2: Top three products in each category	48
7.3	Example 3: Running total and seven-day average	49
7.4	Example 4: Customers who have never ordered	49
7.5	Example 5: Low-stock list	49
7.6	Example 6: Upsert customer changes with MERGE	50
7.7	Example 7: Correct statuses after returns	50
7.8	Example 8: Build a partitioned daily rollout	50
7.9	Example 9: Compare today with yesterday using time travel	51
7.10	Example 10: Inspect a table's snapshots and files	51
7.11	Example 11: Pivot channel revenue into columns	51
7.12	Example 12: Evolve a table without rewriting it	51
7.13	Example 13: Tags stored as an array	52
7.14	Example 14: Safe division and clean text	52
7.15	Example 15: Clean up a busy table	52
7.16	Example 16: Find slow statements (administrators)	52

Appendix A - Choosing a Runtime and Shared Concepts

A.1	Choosing a runtime	54
A.1.1	At a glance	55
A.1.2	Decision guide	55
A.1.3	Where the choice is made	56
A.2	How a run starts	57
A.2.1	From Run to results	57
A.2.2	Datasets, jobs and notebooks compared	57
A.2.3	Sessions and warm starts	57
A.2.4	Timeouts and retries	58
A.2.5	Compute sizes, workers and GPU size	58
A.3	What your code can use during a run	60
A.3.1	Run details	60
A.3.2	Job placeholders	60
A.3.3	Imported models	60
A.4	Connections and secrets	61
A.5	The lakehouse (Iceberg)	62
A.6	Custom runtimes	63
A.7	ML variants	65
A.8	Jobs: lifecycle, status and logs	66
A.8.1	Statuses	66

A.8.2	Logs	66
A.8.3	Scheduling	66
A.9	Glossary	67

GUIDE

Lakehouse Runtime

SQL over Apache Iceberg tables: warehouses, sessions, queries and governance.

-
- 1 Overview
 - 2 Quickstart
 - 3 How-to guides
 - 4 SQL reference
 - 5 Limits and behaviour
 - 6 Troubleshooting
 - 7 Examples gallery

The Groundzero Lakehouse is a SQL engine over Apache Iceberg tables. You open a warehouse, start a SQL session on the compute size you choose, and write Spark SQL in the Lakehouse editor: create tables, load and change data, query it, travel back in time, protect it with policies, and export the results. It is the runtime to reach for when your work is SQL and your data lives in Iceberg tables, with no Python or job code to write.

This guide documents the Lakehouse end to end: what it is, how sessions and queries behave, how to do each common task, the SQL you can rely on, the limits that apply, how to recover from the messages you may see, and a gallery of worked examples.

WHO THIS GUIDE IS FOR

Analysts, data engineers and developers who write SQL in Groundzero. The examples use fictional data from a retail company, Acme Retail, in a warehouse named `acme-retail-lake`. Replace the warehouse, database and table names with your own.

1 Overview

1.1 What the Lakehouse is

The Lakehouse combines the low cost and openness of files in object storage with the tables, transactions and SQL of a database.

- **Tables are Apache Iceberg tables.** Data is stored as Parquet files in an S3 bucket you own; Iceberg keeps the table's schema, partitioning and a history of snapshots, so every change is an atomic commit you can look back at.
- **SQL is Spark SQL.** Queries run on Apache Spark 4.0 with Iceberg 1.10, so you get joins, window functions, common table expressions, `MERGE`, and the Iceberg-specific statements for partitioning, schema changes and time travel.
- **Compute is yours while you need it.** A session starts a dedicated engine sized by the compute you pick and the minimum number of workers; you stop it when you are done.
- **Access is governed.** Permissions on warehouses, databases and tables, row filters, column masks and an audit log of every statement apply to everything you run.

1.2 Warehouses, databases and tables

Term	Meaning
Warehouse	An S3 bucket registered with Groundzero. A warehouse is one Iceberg catalog, and its name is the catalog name.
Database	A grouping of tables inside a warehouse, also called a schema. Write it before the table name.
Table	An Iceberg table. Always refer to it as <code>database.table</code> .
Session	A running SQL engine for one warehouse, with the compute size and workers you chose. SQL does not run until a session is started.
Compute	A size (memory and vCPU) chosen from the list your site offers.
Minimum workers	How many workers the session starts with. It scales up from there as the load grows.

The system database `audit` records every statement that runs in the warehouse. Administrators can query it (see Audit logs in the How-to guides).

1.3 Choose the Lakehouse

Groundzero offers four ways to run your data work. Pick the one that matches the shape of the task.

	Lakehouse	Spark	DuckDB	Standard
What you write	SQL in the Lakehouse editor	PySpark and SQL in jobs and notebooks	Python and SQL in jobs and notebooks	One SQL query on a dataset

	Lakehouse	Spark	DuckDB	Standard
Engine	Spark SQL 4.0 with Iceberg 1.10	Apache Spark cluster	In-process DuckDB	Managed query engine
Where it runs	Your session, driver plus autoscaled workers	Cluster per run	One machine	Shared service
Start	Start a session, then run many queries	Cluster start per run	Quick start, then kept warm	No session
Tables	Create, alter and drop Iceberg tables	Read and write through connections	Read only	Read only
Schedules and pipelines	Backups only; queries are interactive	Jobs and schedules	Jobs and schedules	Datasets and dashboards
Governance	Policies, masks and table access built in	Applied as the caller	Applied as the caller	Applied as the caller
Best for	Exploring, modelling and maintaining Iceberg tables with SQL	Large or write-heavy ETL	Fast single-node analytics	Charts, dashboards and quick lookups

Choose the Lakehouse when:

- you want to create, change and look after Iceberg tables with SQL;
- you explore data interactively and want results, paging and exports in a SQL editor;
- you need `MERGE`, partition and schema changes, time travel or table maintenance;
- you administer access: data policies, column masks and table access rules are edited on the same page.

Choose another runtime when:

- the work has to run on a schedule or inside a pipeline: use a Spark or DuckDB job, which can read and write lakehouse tables;
- you need Python libraries, machine learning or files that are not tables: use Spark or DuckDB;
- a dashboard or API only needs rows from one query: a Standard dataset on a Groundzero lakehouse connection does that without a session.

1.4 How a query runs

1. You press Run. The whole editor tab is sent to your session.
2. Groundzero checks your permission for every table and database the statement touches, and applies the data policies that protect those tables.
3. The engine runs the statement. If it finishes within about 55 seconds, the first page of rows comes back immediately.
4. Otherwise the query keeps running in your session and the Messages tab reads "Still running. Results load here when the query finishes." The page shows the result as soon as it is ready.
5. The first 100,000 rows of a result are kept in the session so you can page through them quickly. Larger results are prepared as files in your warehouse bucket when you page past them or export.
6. The statement is written to the audit log with its status and duration.

1.5 Versions

Component	Version
Apache Spark (SQL engine)	4.0.0
Apache Iceberg	1.10.0
Java	17
Table data files	Parquet
Spark SQL mode	ANSI SQL mode on, as in Spark 4.0

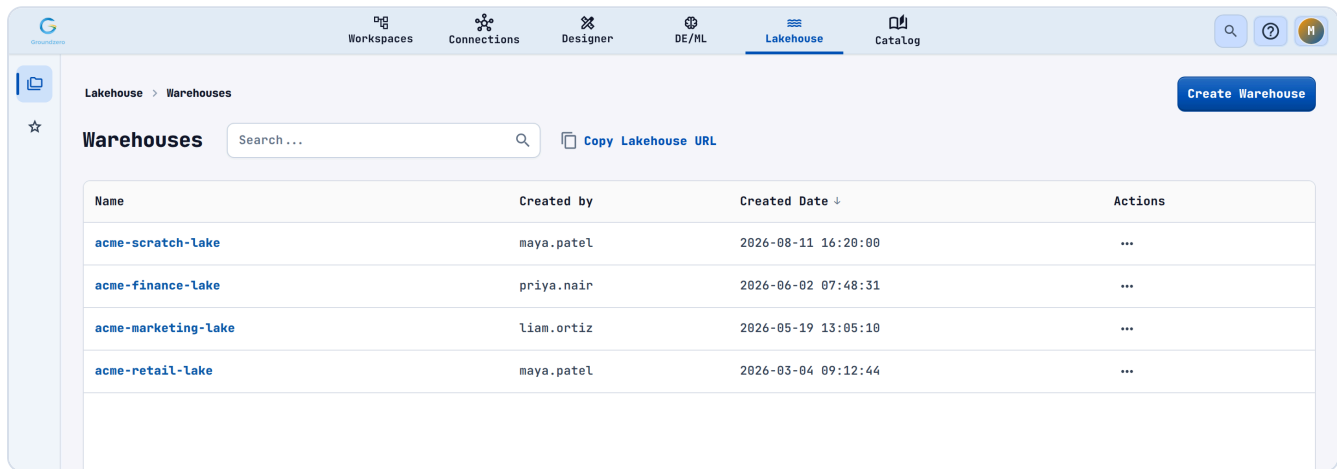
ANSI mode matters for everyday SQL: an invalid cast or a division by zero raises an error instead of returning `NULL`. The functions `try_cast`, `try_divide` and `nullif` are the safe alternatives (see the SQL reference).

2 Quickstart

This section takes you from an empty warehouse to a query result you export. You need access to a warehouse (the list in **Lakehouse** shows the warehouses you can see) and permission to create databases and tables in it. Your administrator sets those permissions.

2.1 1. Open the Lakehouse and pick a warehouse

Choose **Lakehouse** in the top navigation bar. The **Warehouses** page lists the warehouses you can see, with who created them and when. Type in the **Search...** box to filter by name, and click a name to open its workspace.



The screenshot shows the 'Warehouses' page in the Groundzero interface. The top navigation bar includes 'Workspaces', 'Connections', 'Designer', 'DE/HL', 'Lakehouse' (selected), and 'Catalog'. The 'Lakehouse' section has a 'Create Warehouse' button. The 'Warehouses' table lists four warehouses: 'acme-scratch-lake', 'acme-finance-lake', 'acme-marketing-lake', and 'acme-retail-lake'. Each row shows the creator's name and the creation date. A search bar and a 'Copy Lakehouse URL' link are also visible.

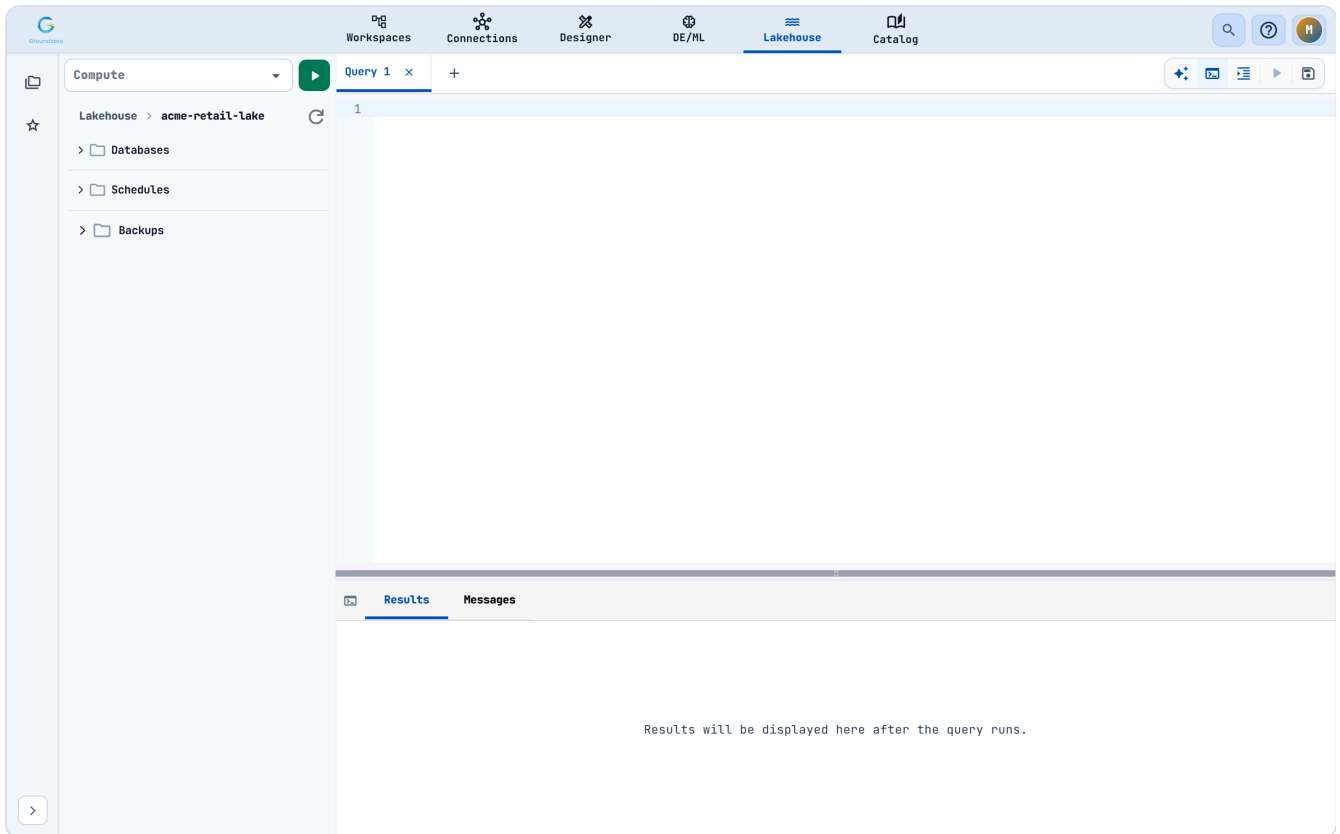
Name	Created by	Created Date ↓	Actions
acme-scratch-lake	maya.patel	2026-08-11 16:28:00	...
acme-finance-lake	priya.nair	2026-06-02 07:48:31	...
acme-marketing-lake	liam.ortiz	2026-05-19 13:05:10	...
acme-retail-lake	maya.patel	2026-03-04 09:12:44	...

The Warehouses page lists each warehouse with its creator and creation date.

To register a new bucket as a warehouse, use **Create Warehouse** on this page; the Lakehouse chapter of the Data features guide walks through it.

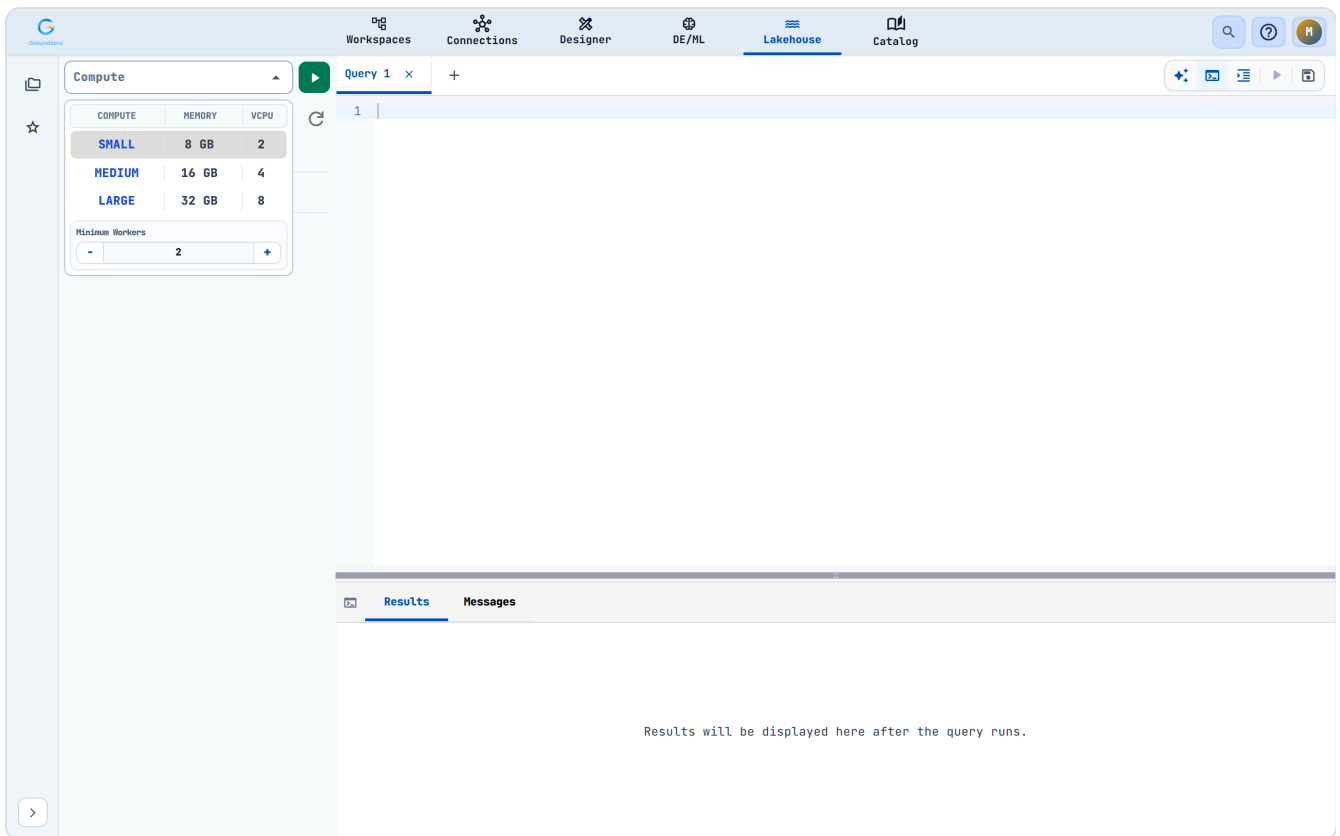
2.2 2. Choose a compute and start a session

The workspace opens with the compute selector at the top left, the databases tree below it and an empty SQL editor. SQL cannot run yet.



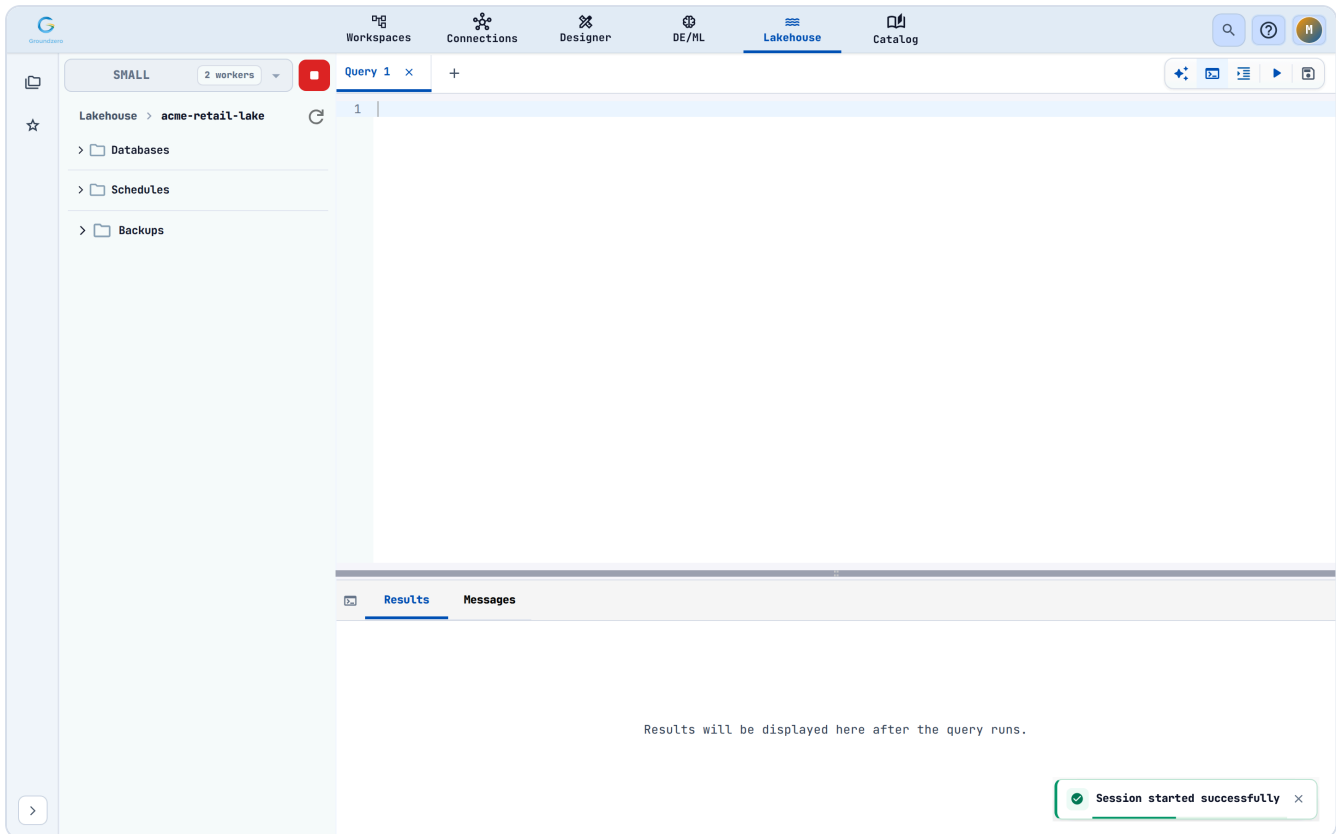
A new workspace before a session starts: compute selector, tree and editor.

1. Open the **Compute** list. Each option shows its name, memory and vCPU.
2. Set **Minimum Workers** under the options (the default is 2, the lowest value is 1). Use more workers for scans over very large tables.
3. Click the green play button (**Start session**).



The compute list with the Minimum Workers stepper.

The button shows a spinner while the engine starts. When the notification "Session started successfully" appears, the button turns into a red stop button and **Run query** is enabled. The compute is locked until you stop the session.



A running session: the compute is locked and the stop button is red.

2.3 3. Create a database and a table

Type the statements below in the editor. Run them one at a time: each press of **Run query** sends the whole tab.

SQL

```
CREATE DATABASE IF NOT EXISTS sales
```

SQL

```
CREATE TABLE sales.orders (
  order_id    BIGINT,
  customer_id BIGINT,
  store_id    INT,
  order_ts    TIMESTAMP,
  order_total DECIMAL(12, 2),
  channel     STRING,
  status      STRING
) USING iceberg
PARTITIONED BY (days(order_ts))
```

After each statement the **Messages** tab reads "Query executed successfully." Statements that create or change things return no rows, so there is no results grid. Click the refresh icon next to the breadcrumb to reload the tree and see the new database and table.

2.4 4. Insert rows

SQL

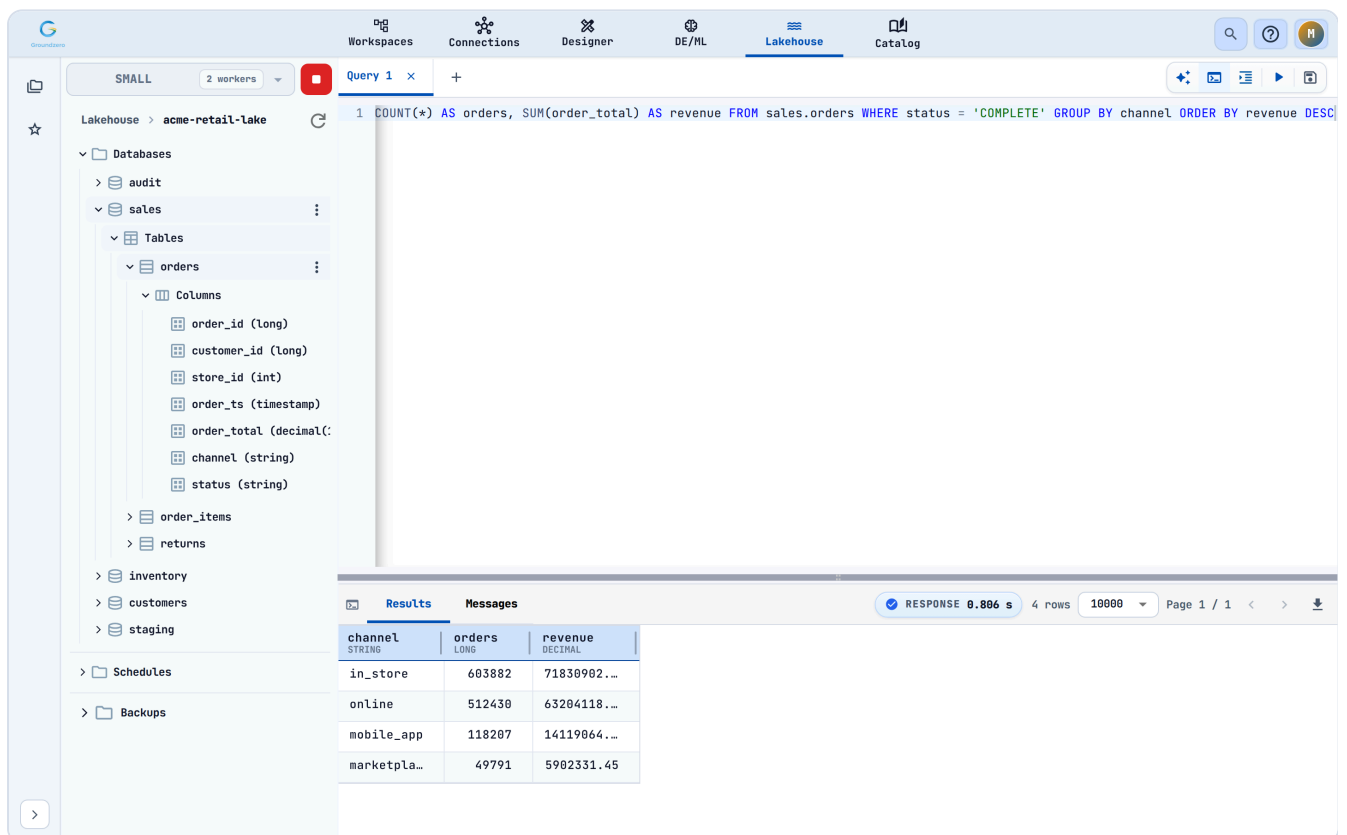
```
INSERT INTO sales.orders VALUES
(1001, 501, 12, TIMESTAMP '2026-09-29 09:14:00', 129.90, 'online', 'COMPLETE'),
(1002, 502, 12, TIMESTAMP '2026-09-29 11:02:00', 45.00, 'in_store', 'COMPLETE'),
(1003, 503, 7, TIMESTAMP '2026-09-30 16:45:00', 310.25, 'mobile_app', 'COMPLETE'),
(1004, 501, 12, TIMESTAMP '2026-09-30 18:20:00', 18.75, 'online', 'RETURNED'),
(1005, 504, 7, TIMESTAMP '2026-10-01 08:05:00', 99.00, 'marketplace', 'COMPLETE')
```

2.5 5. Query the table

SQL

```
SELECT channel,
       COUNT(*) AS orders,
       SUM(order_total) AS revenue
FROM sales.orders
WHERE status = 'COMPLETE'
GROUP BY channel
ORDER BY revenue DESC
```

While the query runs, the Messages tab shows "Executing Query..." and a pill counts the seconds. When it finishes, the **Results** grid opens. Each column header shows its name and its type, and the toolbar shows the row count, rows per page, the page number and the response time.



The screenshot shows the Groundzero Lakehouse interface. On the left, a sidebar displays the database structure for 'acme-retail-lake', including tables like 'orders' and 'order_items'. The main area shows a SQL query: `COUNT(*) AS orders, SUM(order_total) AS revenue FROM sales.orders WHERE status = 'COMPLETE' GROUP BY channel ORDER BY revenue DESC`. Below the query, the 'Results' tab is active, displaying a table with 4 rows and 3 columns: 'channel' (STRING), 'orders' (LONG), and 'revenue' (DECIMAL). The toolbar indicates a response time of 0.806 s, 4 rows, and 10000 rows per page.

channel STRING	orders LONG	revenue DECIMAL
in_store	603882	71830902...
online	512430	63204118...
mobile_app	118207	14119064...
marketpla...	49791	5902331.45

A result: typed columns, a row count, rows per page and the response time.

2.6 6. Save and export

Click **Save query**, enter a name and press **Save** to keep the SQL for later (see Saved queries). To keep the rows, use the download icon in the results toolbar: **Download current page as CSV** saves the rows on screen. Results larger than the 100,000 rows kept in the session also offer **Export**, which writes every row to one CSV or Parquet file (see Exports).

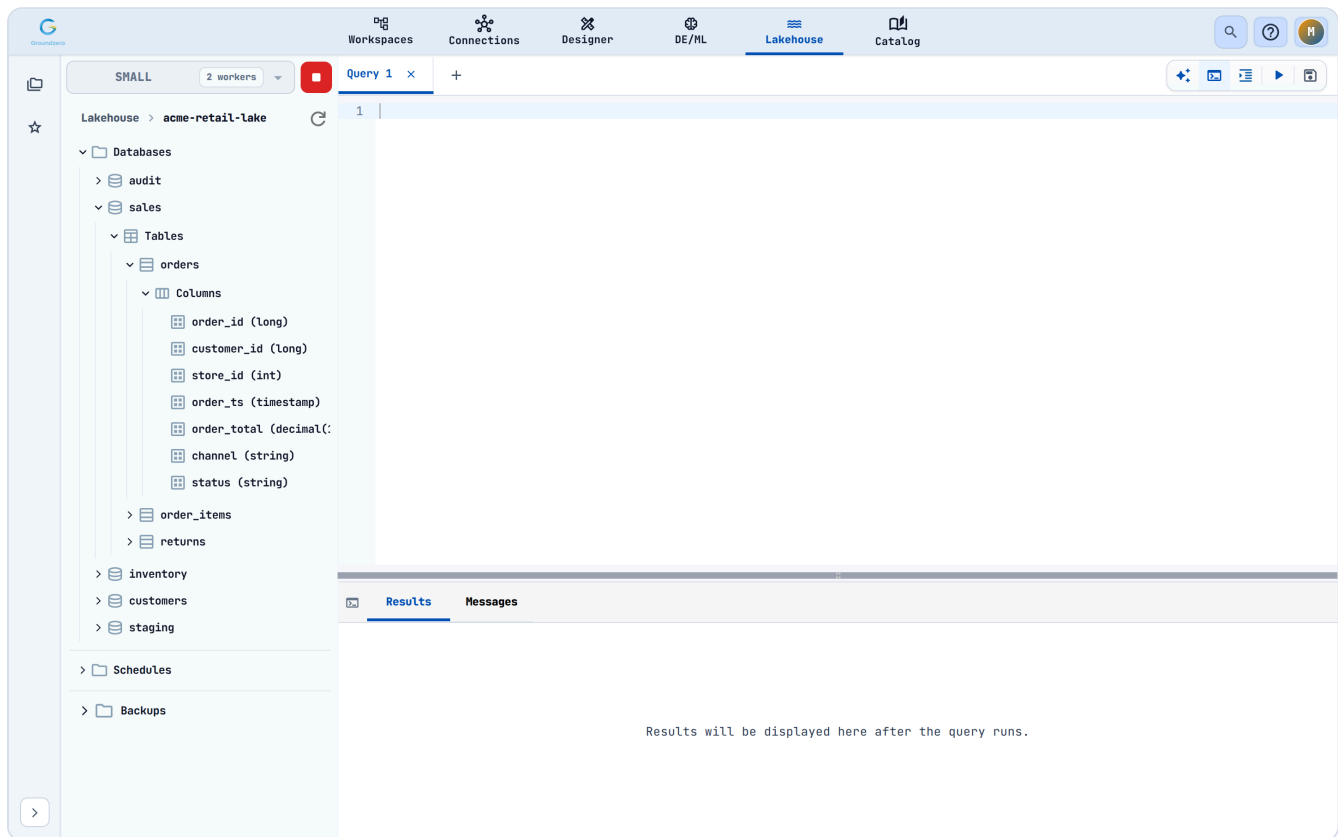
2.7 7. Stop the session

Click the red stop button when you finish. The notification "Session stopped successfully" confirms that the compute is released. A session you forget is stopped for you after 30 minutes without activity.

3 How-to guides

3.1 Browse databases, tables and columns

Expand **Databases** in the tree to see each database (the system database `audit` is listed first), then **Tables**, a table and its **Columns**. A column is shown as `name (type)`. Hover over a database or table and click **Open row actions** for its menu.



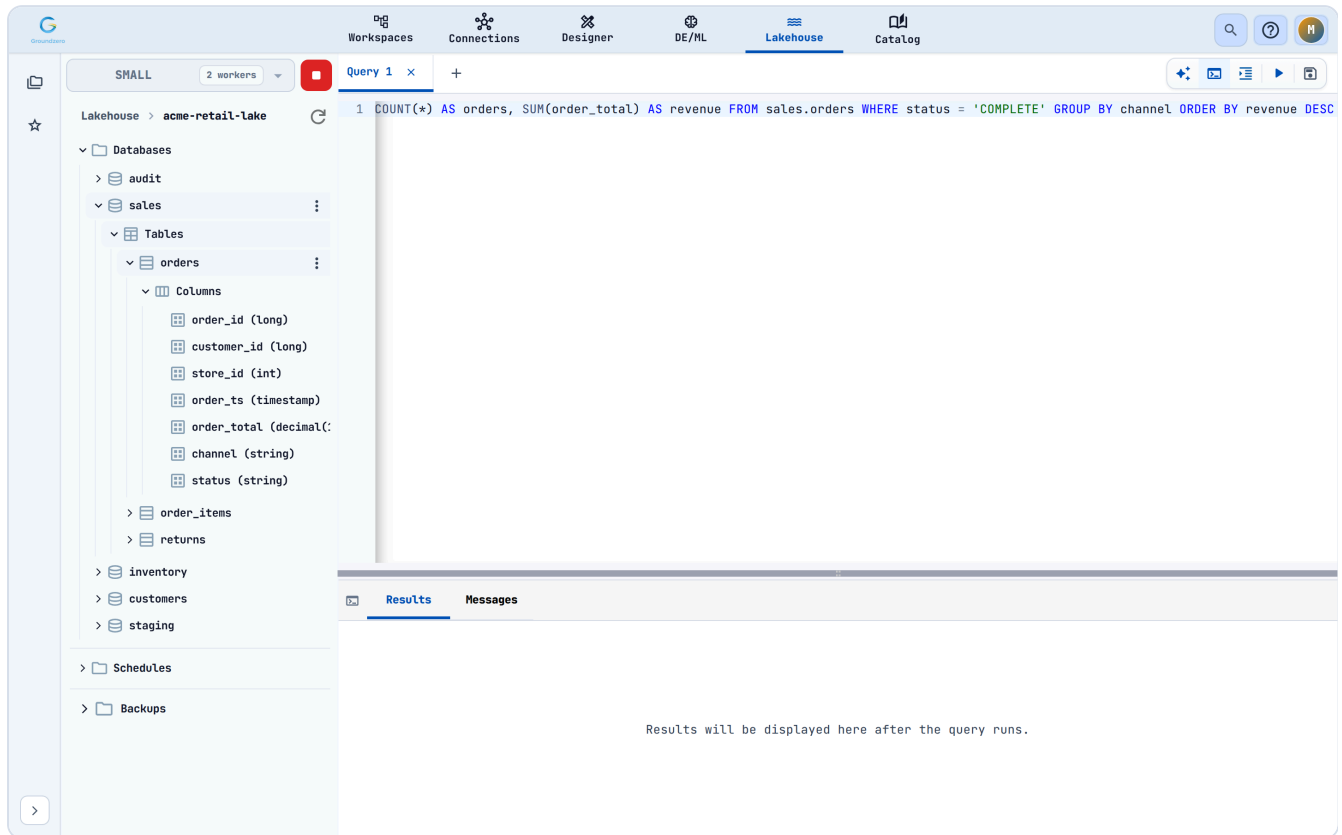
Tables expand to their columns with their types.

Node	Menu items
Databases	Import Schema, Restore Database, Mark stale backups failed
A database	Properties, Export schema, Backup Database, Restore Database
A table	Add table on cursor, Properties, Data policy, Table Access Manager

- **Add table on cursor** types `database.table` at the cursor of the active query tab.
- **Export schema** downloads a text file with the column definitions of every table in a database, without data. **Import Schema** takes such a file and a database name and creates the database with one empty table per entry; tables that already exist are skipped.
- The editor's autocomplete suggests the warehouse, databases, tables (as `table` and `database.table`), columns and connection names from the tree, plus common SQL keywords and functions.

3.2 Write queries

Every query names its tables as `database.table`. Run a statement with the **Run query** button; there is no keyboard shortcut. The whole tab is sent, not only the selected text, so keep one statement per tab and open a new tab with **+** for the next one.



Typing a query in a tab. Use **+** to open more tabs.

USE BLOCK COMMENTS

Before a query is sent, every run of spaces and line breaks is collapsed to a single space. A `--` comment would therefore comment out the rest of the statement. Write comments as `/*` like this `*/`, which survives.

PAGE AND EXPORT ANY RESULT

Wrap a `UNION` or a `VALUES` list in a select, for example `SELECT * FROM (SELECT ... UNION ALL SELECT ...)`. Results of a plain `SELECT` (including `WITH ... SELECT`) can be paged beyond 100,000 rows and exported; other row-returning forms show at most the first 100,000 rows.

SELECT, filters and ordering

SQL

```
SELECT order_id, customer_id, order_total
FROM sales.orders
WHERE status = 'COMPLETE'
      AND order_ts ≥ TIMESTAMP '2026-09-01 00:00:00'
      AND channel IN ('online', 'mobile_app')
ORDER BY order_total DESC
LIMIT 20
```

`GROUP BY ALL` groups by every non-aggregate column in the select list, and `SELECT DISTINCT`, `HAVING`, `LIMIT` and `OFFSET` work as in standard SQL.

Joins

SQL

```
SELECT o.order_id, c.name, c.segment, o.order_total
FROM sales.orders AS o
JOIN customers.profiles AS c
      ON c.customer_id = o.customer_id
LEFT JOIN sales.returns AS r
      ON r.order_id = o.order_id
WHERE r.order_id IS NULL
```

`INNER`, `LEFT`, `RIGHT`, `FULL`, `CROSS`, `LEFT SEMI` and `LEFT ANTI` joins are supported. A semi or anti join is a clear way to say "customers who have (or have not) ordered":

SQL

```
SELECT c.customer_id, c.name
FROM customers.profiles AS c
LEFT ANTI JOIN sales.orders AS o
      ON o.customer_id = c.customer_id
```

Common table expressions and subqueries

SQL

```
WITH monthly AS (
  SELECT date_trunc('month', order_ts) AS month,
         SUM(order_total)              AS revenue
  FROM sales.orders
  WHERE status = 'COMPLETE'
  GROUP BY 1
)
SELECT month,
       revenue,
       revenue - LAG(revenue) OVER (ORDER BY month) AS change
FROM monthly
ORDER BY month
```

Subqueries work in `FROM`, `WHERE` (`IN`, `EXISTS`, scalar) and the select list.

Window functions

SQL

```
SELECT customer_id,
       order_id,
       order_total,
       ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_ts) AS nth_order,
       SUM(order_total) OVER (
         PARTITION BY customer_id ORDER BY order_ts
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS lifetime_value
FROM sales.orders
```

The ranking functions (`ROW_NUMBER` , `RANK` , `DENSE_RANK` , `NTILE`), the value functions (`LAG` , `LEAD` , `FIRST_VALUE` , `LAST_VALUE`) and every aggregate can be used with `OVER` .

Dates, strings and other everyday functions

Task	Functions
Current date and time	<code>current_date()</code> , <code>current_timestamp()</code>
Truncate and extract	<code>date_trunc('month', ts)</code> , <code>trunc(date, 'MM')</code> , <code>year(d)</code> , <code>month(d)</code> , <code>dayofweek(d)</code> , <code>extract(HOUR FROM ts)</code>
Date arithmetic	<code>date_add(d, 7)</code> , <code>date_sub(d, 7)</code> , <code>add_months(d, 1)</code> , <code>datediff(end, start)</code> , <code>months_between(a, b)</code> , <code>last_day(d)</code>
Convert	<code>to_date(s, 'yyyy-MM-dd')</code> , <code>to_timestamp(s)</code> , <code>date_format(ts, 'yyyy-MM')</code> , <code>cast(x AS type)</code> , <code>try_cast(x AS type)</code>
Text	<code>concat</code> , <code>concat_ws</code> , <code>substring</code> , <code>lower</code> , <code>upper</code> , <code>trim</code> , <code>replace</code> , <code>regexp_replace</code> , <code>regexp_extract</code> , <code>split</code> , <code>length</code> , <code>lpad</code>
Numbers	<code>round</code> , <code>floor</code> , <code>ceil</code> , <code>abs</code> , <code>greatest</code> , <code>least</code> , <code>try_divide(a, b)</code>
Nulls and conditions	<code>coalesce</code> , <code>nullif</code> , <code>ifnull</code> , <code>CASE WHEN ... END</code> , <code>if(cond, a, b)</code>
Aggregates	<code>count</code> , <code>sum</code> , <code>avg</code> , <code>min</code> , <code>max</code> , <code>count(DISTINCT x)</code> , <code>approx_count_distinct</code> , <code>percentile_approx</code> , <code>collect_list</code> , <code>collect_set</code> , <code>max_by</code> , <code>min_by</code>
Arrays and maps	<code>array</code> , <code>size</code> , <code>explode</code> , <code>array_contains</code> , <code>transform</code> , <code>map_keys</code> , <code>map_values</code> , <code>element_at</code>
JSON	<code>from_json</code> , <code>to_json</code> , <code>get_json_object</code>

Data types

Type	Notes	Shown in results as
<code>TINYINT</code> , <code>SMALLINT</code> , <code>INT</code> , <code>BIGINT</code>	Whole numbers of 1, 2, 4 and 8 bytes	byte, short, integer, long
<code>FLOAT</code> , <code>DOUBLE</code>	Binary floating point	float, double
<code>DECIMAL(p, s)</code>	Exact numbers; use for money	decimal
<code>STRING</code>	Text of any length	string

Type	Notes	Shown in results as
BOOLEAN	true or false	boolean
DATE	Calendar date	date
TIMESTAMP	Instant, with time zone	timestamp
TIMESTAMP_NTZ	Date and time without time zone	timestamp_ntz
BINARY	Bytes	binary
ARRAY<T> , MAP<K,V> , STRUCT<a: T, ...>	Nested values; shown as text in the grid	array, map, struct

Format SQL

Format SQL (the indent icon in the editor toolbar) lays the active tab out over several lines with upper-case keywords, using the Spark dialect. It shows "No SQL query to format" for an empty tab and "Unable to format this SQL query" if the text cannot be formatted.

3.3 Create and alter Iceberg tables

Create a table

SQL

```
CREATE TABLE sales.order_items (
  order_id  BIGINT,
  line_no   INT,
  sku       STRING,
  quantity  INT,
  unit_price DECIMAL(10, 2)
) USING iceberg
COMMENT 'One row per order line'
```

`CREATE TABLE IF NOT EXISTS` skips the statement when the table exists. To create a table from a query, use `CREATE TABLE ... AS SELECT`; `REPLACE TABLE ... AS SELECT` swaps the contents and keeps the table's history.

SQL

```
CREATE TABLE sales.daily_revenue USING iceberg
PARTITIONED BY (day)
AS SELECT CAST(order_ts AS DATE) AS day,
         channel,
         SUM(order_total)      AS revenue
FROM sales.orders
GROUP BY 1, 2
```

Partitioning

Partitioning groups rows into files so queries that filter on the partition column read less data. Iceberg partitions by a *transform* of a column, so you filter on the column itself and never on a derived partition column:

Transform	Use for
<code>years(ts)</code> , <code>months(ts)</code> , <code>days(ts)</code> , <code>hours(ts)</code>	Timestamp and date columns
<code>bucket(16, id)</code>	Spreading a high-cardinality key over a fixed number of buckets
<code>truncate(4, text)</code>	Grouping by the first characters or a numeric range
plain column	Low-cardinality columns such as <code>region</code>

SQL

```
CREATE TABLE inventory.stock_moves (
  sku      STRING,
  store_id INT,
  moved_at TIMESTAMP,
  delta    INT
) USING iceberg
PARTITIONED BY (days(moved_at), bucket(8, store_id))
```

A query such as `WHERE moved_at ≥ TIMESTAMP '2026-09-30 00:00:00'` prunes partitions automatically. Choose coarse partitions: daily or monthly for most tables, and avoid partitioning on a column with thousands of distinct values.

You can change how a table is partitioned later. New data is written with the new layout and old data keeps the old one; both stay queryable.

SQL

```
ALTER TABLE sales.orders ADD PARTITION FIELD bucket(4, store_id)
ALTER TABLE sales.orders DROP PARTITION FIELD bucket(4, store_id)
ALTER TABLE sales.orders REPLACE PARTITION FIELD days(order_ts) WITH months(order_ts)
```

The table's **Properties** tab (see Table properties) shows the current **Partitioned By** value.

Schema evolution

Iceberg tracks columns by an internal id, so these changes are metadata-only and safe: they never rewrite data and never change the meaning of existing columns.

SQL

```
ALTER TABLE sales.orders ADD COLUMN coupon_code STRING
ALTER TABLE sales.orders ADD COLUMN discount DECIMAL(10, 2) AFTER order_total
ALTER TABLE sales.orders RENAME COLUMN channel TO sales_channel
ALTER TABLE sales.orders ALTER COLUMN order_id TYPE BIGINT
ALTER TABLE sales.orders ALTER COLUMN status COMMENT 'COMPLETE, RETURNED or OPEN'
ALTER TABLE sales.orders DROP COLUMN coupon_code
```

A type change must widen the type (`INT` to `BIGINT`, `FLOAT` to `DOUBLE`, or a larger `DECIMAL` precision). Other table changes:

SQL

```
ALTER TABLE sales.orders RENAME TO sales.orders_archive
ALTER TABLE sales.orders SET TBLPROPERTIES ('comment' = 'All customer orders')
COMMENT ON TABLE sales.orders IS 'All customer orders'
ALTER TABLE sales.orders WRITE ORDERED BY order_ts
```

Dropping a table removes it from the catalog:

SQL

```
DROP TABLE IF EXISTS sales.orders_archive
```

3.4 Insert, update, delete and merge

All of these are transactional: a statement either commits as one new snapshot or leaves the table unchanged.

INSERT

SQL

```
INSERT INTO sales.order_items VALUES (1001, 1, 'SKU-100', 2, 49.95)
INSERT INTO sales.order_items SELECT order_id, 1, 'SKU-100', 1, 19.00 FROM sales.orders
INSERT OVERWRITE sales.daily_revenue SELECT day, channel, revenue FROM sales.stage
```

`INSERT OVERWRITE` replaces only the partitions the new rows touch, not the whole table. To replace everything, use `REPLACE TABLE ... AS SELECT` or delete first.

UPDATE and DELETE

SQL

```
UPDATE sales.orders SET status = 'CANCELLED' WHERE order_id = 1004
DELETE FROM sales.orders WHERE status = 'CANCELLED' AND order_ts < TIMESTAMP '2025-10-01 00:00:00'
TRUNCATE TABLE sales.staging_orders
```

MERGE: upsert from a staging table

`MERGE INTO` updates rows that match, inserts rows that do not, and can delete. It is the way to load changes into a table without duplicating rows.

SQL

```
MERGE INTO customers.profiles AS t
USING customers.profile_updates AS s
  ON t.customer_id = s.customer_id
 WHEN MATCHED AND s.deleted THEN DELETE
 WHEN MATCHED THEN UPDATE SET
   t.name = s.name, t.email = s.email, t.segment = s.segment
 WHEN NOT MATCHED THEN INSERT *
```

MAKE THE JOIN KEY UNIQUE

A source row set that matches the same target row more than once makes the `MERGE` fail. De-duplicate the staging data first, for example with `ROW_NUMBER()` in a subquery.

Updates, deletes and merges need the **Update** permission on the table; inserts need **Insert**. See the permission table in the SQL reference.

3.5 Time travel and snapshots

Every commit creates a *snapshot*. You can query a table as it was at an earlier snapshot or time.

SQL

```
SELECT COUNT(*) FROM sales.orders TIMESTAMP AS OF '2026-09-30 12:00:00'  
SELECT COUNT(*) FROM sales.orders VERSION AS OF 4821763650912831907
```

To find a snapshot id, open the table's **Properties** tab and click the **Snapshot History** value, which lists the ten most recent snapshots with their id, parent, time, operation and counts of added and deleted records, or query the snapshots metadata table (next section).

Snapshot History

```

[
  {
    "snapshot_id": 4821907344556677,
    "parent_snapshot_id": 4821907343456674,
    "timestamp": "2026-09-30 22:15:00",
    "operation": "append",
    "added_records": 8800,
    "deleted_records": 0
  },
  {
    "snapshot_id": 4821907343456674,
    "parent_snapshot_id": 4821907342356671,
    "timestamp": "2026-09-29 22:15:00",
    "operation": "append",
    "added_records": 8930,
    "deleted_records": 0
  },
  {
    "snapshot_id": 4821907342356671,
    "parent_snapshot_id": 4821907341256668,
    "timestamp": "2026-09-28 22:15:00",
    "operation": "overwrite",
    "added_records": 9060,
    "deleted_records": 412
  },
  {
    "snapshot_id": 4821907341256668,
    "parent_snapshot_id": 4821907340156665,
    "timestamp": "2026-09-27 22:15:00",
    "operation": "append"
  }
]

```

Compression Codec

zstd

Snapshot History lists the most recent snapshots as JSON.

To undo a mistake, roll the table back to an earlier snapshot. This needs the **AlterTable** permission:

SQL

```
CALL system.rollback_to_snapshot('sales.orders', 4821763650912831907)
```

SNAPSHOTS ARE NOT BACKUPS

Maintenance removes old snapshots (see [Expire old snapshots](#)), after which you can no longer travel back to them. Use database backups for long-term recovery.

3.6 Metadata tables

Every Iceberg table has read-only metadata tables that describe its snapshots, files and partitions. Query them by adding a suffix to the table name; you need **Select** on the table.

Metadata table	Shows
<code>sales.orders.snapshots</code>	Every snapshot: id, commit time, operation, summary
<code>sales.orders.history</code>	The table's history of current snapshots
<code>sales.orders.files</code>	Data files with row counts, sizes and column statistics
<code>sales.orders.partitions</code>	Row and file counts per partition
<code>sales.orders.manifests</code>	The manifest files that track the data files
<code>sales.orders.refs</code>	Branches and tags
<code>sales.orders.metadata_log_entries</code>	Earlier versions of the table metadata
<code>sales.orders.entries</code> , <code>all_entries</code> , <code>all_files</code> , <code>all_data_files</code> , <code>all_delete_files</code> , <code>all_manifests</code> , <code>data_files</code> , <code>delete_files</code> , <code>position_deletes</code>	Lower-level file and manifest detail

SQL

```
SELECT snapshot_id, committed_at, operation
FROM sales.orders.snapshots
ORDER BY committed_at DESC
LIMIT 10
```

SQL

```
SELECT partition, record_count, file_count
FROM sales.orders.partitions
ORDER BY record_count DESC
```

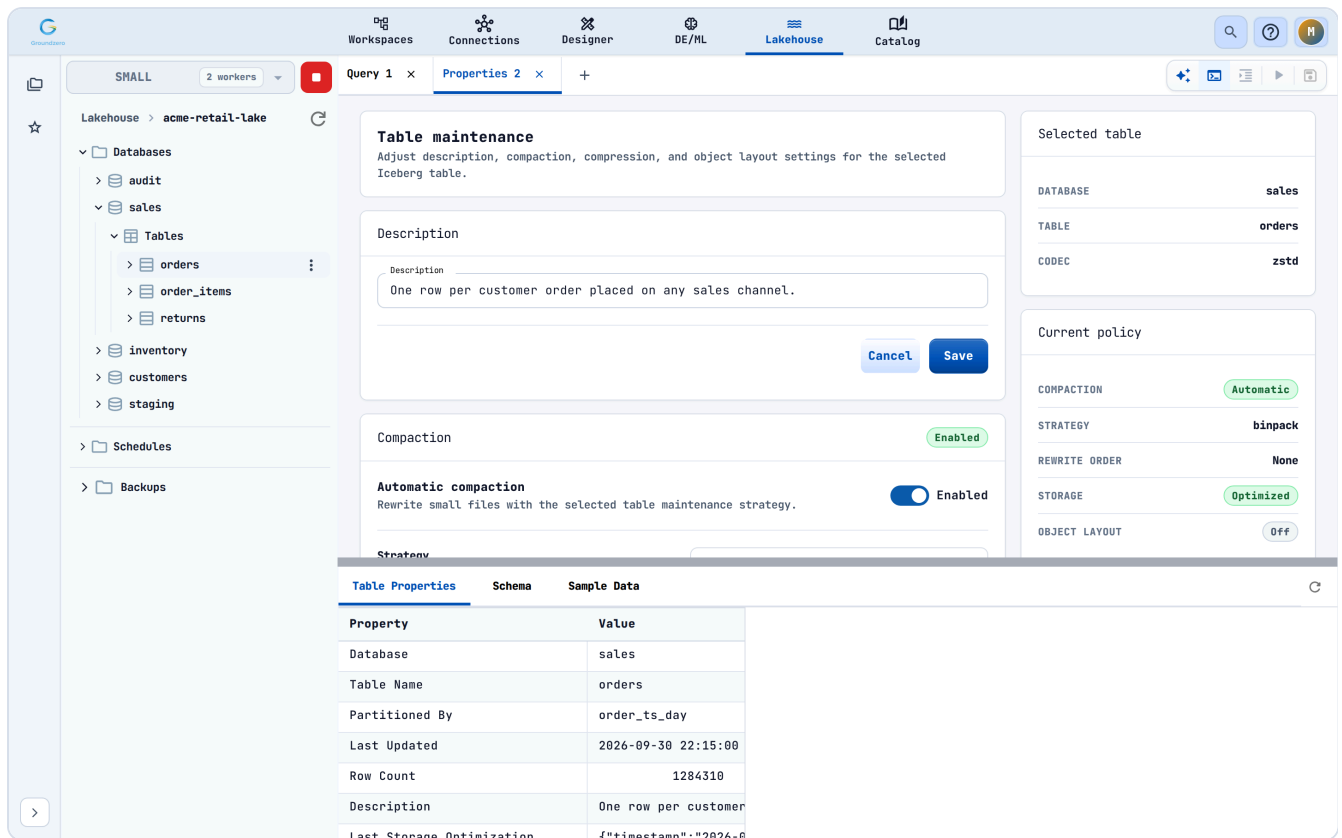
SQL

```
SELECT file_path, record_count, file_size_in_bytes
FROM sales.orders.files
ORDER BY file_size_in_bytes
LIMIT 20
```

If the table has a data policy (a row filter, a column mask or protected access), its metadata tables refuse to run with "Metadata tables of `<table>` are ...: the table has data policies, and its metadata tables expose column statistics that would reveal masked or filtered values. Query the table itself instead." Query the table instead.

3.7 Table properties

Choose **Properties** in a table's menu to open a **Properties** tab. The top of the tab holds the maintenance panels; the bottom pane has three tabs.



The Properties tab: maintenance panels on top; properties, schema and sample data below.

Property	Value
Database	sales
Table Name	orders
Partitioned By	order_ts_day
Last Updated	2026-09-30 22:15:00
Row Count	1284310
Description	One row per customer
Last Storage Optimization	{ "timestamp": "2026-09-30 22:15:00" }

The Properties tab: maintenance panels on top; properties, schema and sample data below.

- **Table Properties** lists Database, Table Name, Partitioned By, Last Updated, Row Count, Description, Last Storage Optimization, Last Compaction, Storage Optimization Enabled, Format Version, Snapshot Id, File Count, Total Size Bytes, Created At, Created By, Compression Codec, Object Storage Layout, Snapshot History and Sort Order.
- **Schema** lists each column with its type and governance tags.
- **Sample Data** shows the first rows of the table for administrators.

To read the table's Iceberg properties, run `SHOW TBLPROPERTIES sales.orders`. For a database, choose **Properties** in its menu to see the total tables, total size and a per-table size and file count.

3.8 Maintenance

Tables that receive many small writes accumulate many small files and old snapshots. Maintenance keeps them fast and small. It needs the `AlterTable` permission through SQL, and an administrator account in the Properties tab.

Compact small files

In the Properties tab, the **Compaction** panel has these controls:

Control	Meaning
Automatic compaction	Switch on to compact the table automatically
Strategy	Binpack merges small files; Sort also orders rows by a sort order
Rewrite job order	None (default), bytes-asc, bytes-desc, files-asc or files-desc

Control	Meaning
Commit group	Largest amount of data rewritten in one group (default 100 GB)
Partial progress	Maximum number of commits (default 10)
Concurrent groups	File groups rewritten in parallel (default 1)

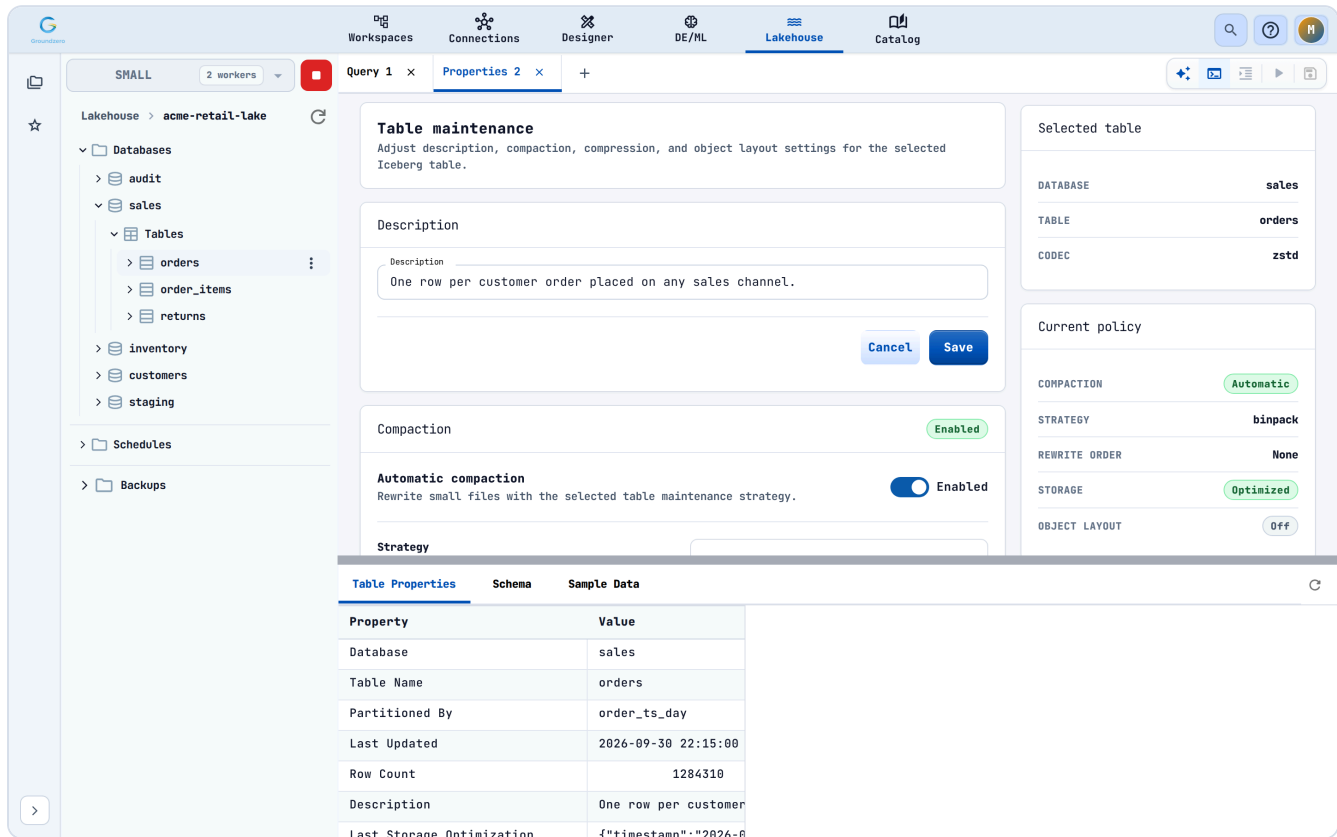


Table maintenance
Adjust description, compaction, compression, and object layout settings for the selected Iceberg table.

Description
Description
One row per customer order placed on any sales channel.

Compaction Enabled

Automatic compaction
Rewrite small files with the selected table maintenance strategy. Enabled

Strategy

Selected table

DATABASE	sales
TABLE	orders
CODEC	zstd

Current policy

COMPACTION	Automatic
STRATEGY	binpack
REWRITE ORDER	None
STORAGE	Optimized
OBJECT LAYOUT	Off

Table Properties

Property	Value
Database	sales
Table Name	orders
Partitioned By	order_ts_day
Last Updated	2026-09-30 22:15:00
Row Count	1284310
Description	One row per customer
Last Storage Optimization	{ "timestamp": "2026-09-30 22:15:00" }

The Compaction panel.

Saving shows "Compaction process initiated successfully!" and the work runs in the background. The same operation is available in SQL:

SQL

```
CALL system.rewrite_data_files(table => 'sales.orders')
CALL system.rewrite_data_files(table => 'sales.orders', strategy => 'sort',
                               sort_order => 'order_ts ASC NULLS LAST')
```

Storage optimization

The **Storage** panel's **Storage optimization** switch manages the table's file layout and cleanup in the background. When it is off, you choose the **Compression codec** (zstd, snappy or gzip) and the **Object storage layout** yourself. Saving shows a confirmation, and **Last Storage Optimization** in the properties records the time.

Expire old snapshots

Expiring snapshots deletes snapshot history and the files only those snapshots used. Time travel to an expired snapshot stops working.

SQL

```
CALL system.expire_snapshots(table => 'sales.orders',
                             older_than => TIMESTAMP '2026-09-01 00:00:00',
                             retain_last => 5)
```

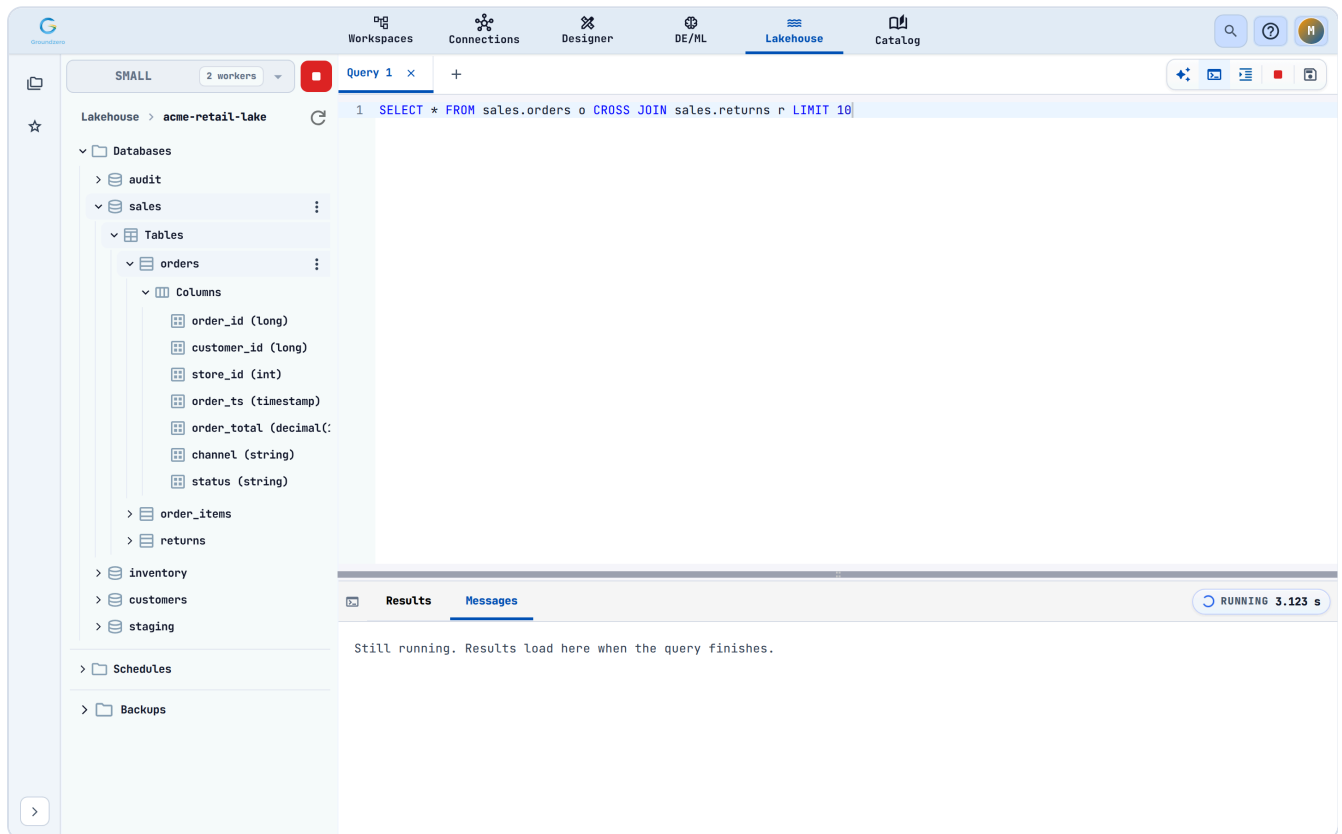
Other table procedures you can call with the AlterTable permission are `rewrite_manifests` (merge manifest files) and `remove_orphan_files` (delete files no snapshot references):

SQL

```
CALL system.rewrite_manifests('sales.orders')
```

3.9 Long-running queries and stopping a query

A query that is still running after about 55 seconds keeps going in your session. The Messages tab reads "Still running. Results load here when the query finishes." and the page checks for you; when the query completes, the results open. You can leave the page open and keep working in another tab.



A long query keeps running and shows its elapsed time.

- **Stop query** (the red square in place of Run) cancels it: the Messages tab reads "Query stopped."
- A query that runs longer than 30 minutes is stopped by the session with "Query ran longer than 30 minutes and was stopped."
- A session runs up to four queries at a time. A fifth is refused with "This session is already running 4 queries. Wait for one to finish or stop it, then run this query again." Preparing a full result and exporting also count.

- If the page loses contact with the session while a query runs, it shows "Lost contact with the session. Retrying..." and picks the result up again when it can. A result kept for a detached query is held for 15 minutes; after that the Messages tab reads "The session no longer has this query. Run it again."

Re-running the same query within five minutes may return the earlier result from a cache. Any statement that changes data, such as an `INSERT`, clears that cache, so results after a change are always fresh.

3.10 Results, paging and exports

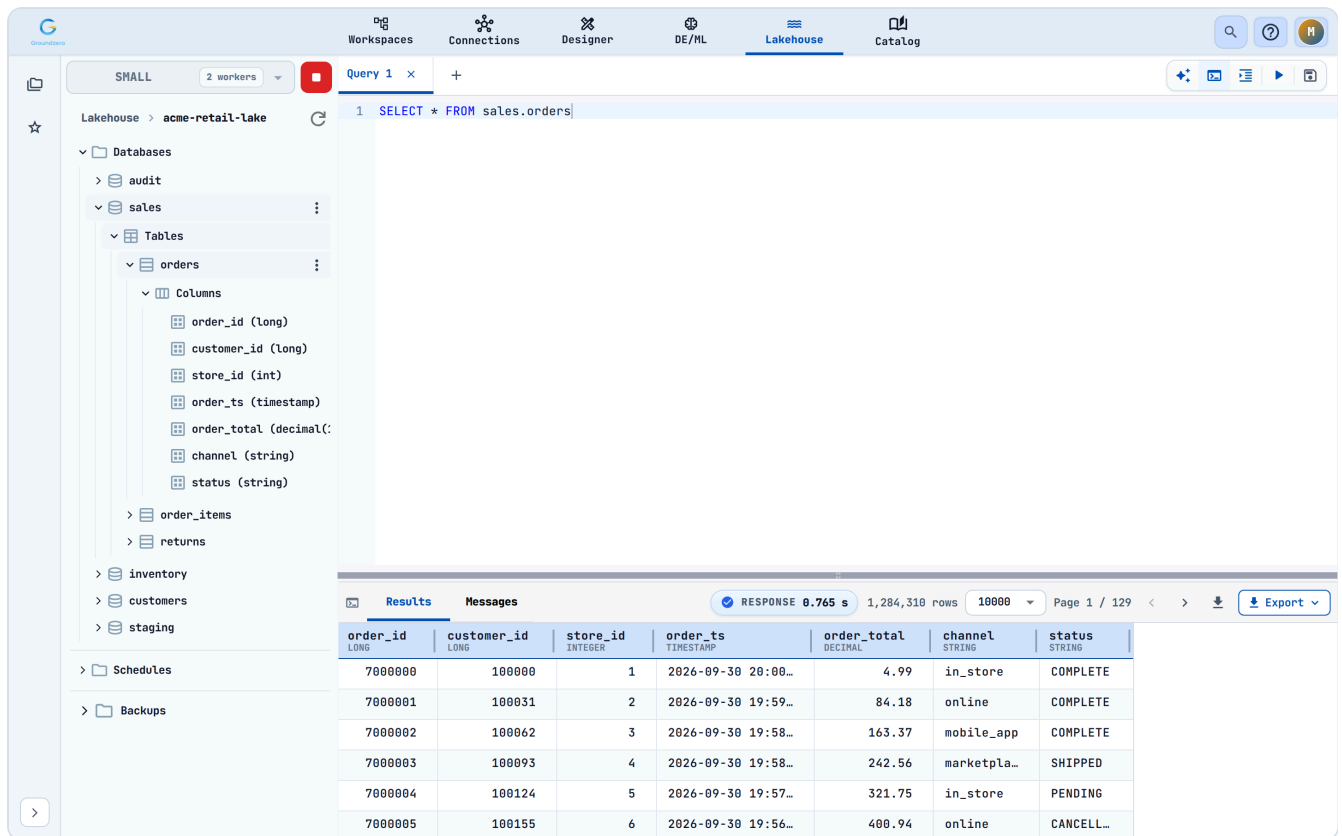
The results grid

The grid shows each column with its type. Drag a header to reorder columns, or right-click it for **Autosize this column**, **Autosize all columns**, **Auto-align columns** and **Reset column order**. There is no sorting or filtering in the grid: use `ORDER BY` and `WHERE`. Click a cell that holds nested data or long text to read the full value and copy it. Results with very many columns load the first 300 columns, then further groups of 200 as you scroll right.

The toolbar above the grid shows the response time, the row count and:

Control	Behaviour
Rows per page	50, 100, 500, 1000 or 10000 (default 10000). Changing it goes back to page 1.
Page n / total , previous and next	Move through the result
Download current page as CSV	A CSV of the rows on screen, named <code>query_<date>.csv</code> , with every value quoted
Export	Appears when the result has more than 100,000 rows. Writes every row to one file.

For results of 100,000 rows or more the toolbar shows the exact total, for example "1,284,310 rows". If the exact total cannot be counted, the label reads "Showing top 100K rows".



The screenshot shows the Groundzero Lakehouse interface. On the left, a sidebar displays the database structure for 'acme-retail-lake', including databases (audit, sales), tables (orders, order_items, returns), and other entities (inventory, customers, staging, Schedules, Backups). The 'orders' table is selected, showing its columns: order_id (long), customer_id (long), store_id (int), order_ts (timestamp), order_total (decimal), channel (string), and status (string). The main area shows a query editor with the query: `1 SELECT * FROM sales.orders`. Below the query editor, the 'Results' tab is active, displaying a table with 7 columns: order_id, customer_id, store_id, order_ts, order_total, channel, and status. The table contains 6 rows of data. Above the table, the 'Messages' tab shows a response of 0.765 s and 1,284,310 rows. The 'Export' button is visible in the top right corner of the results area.

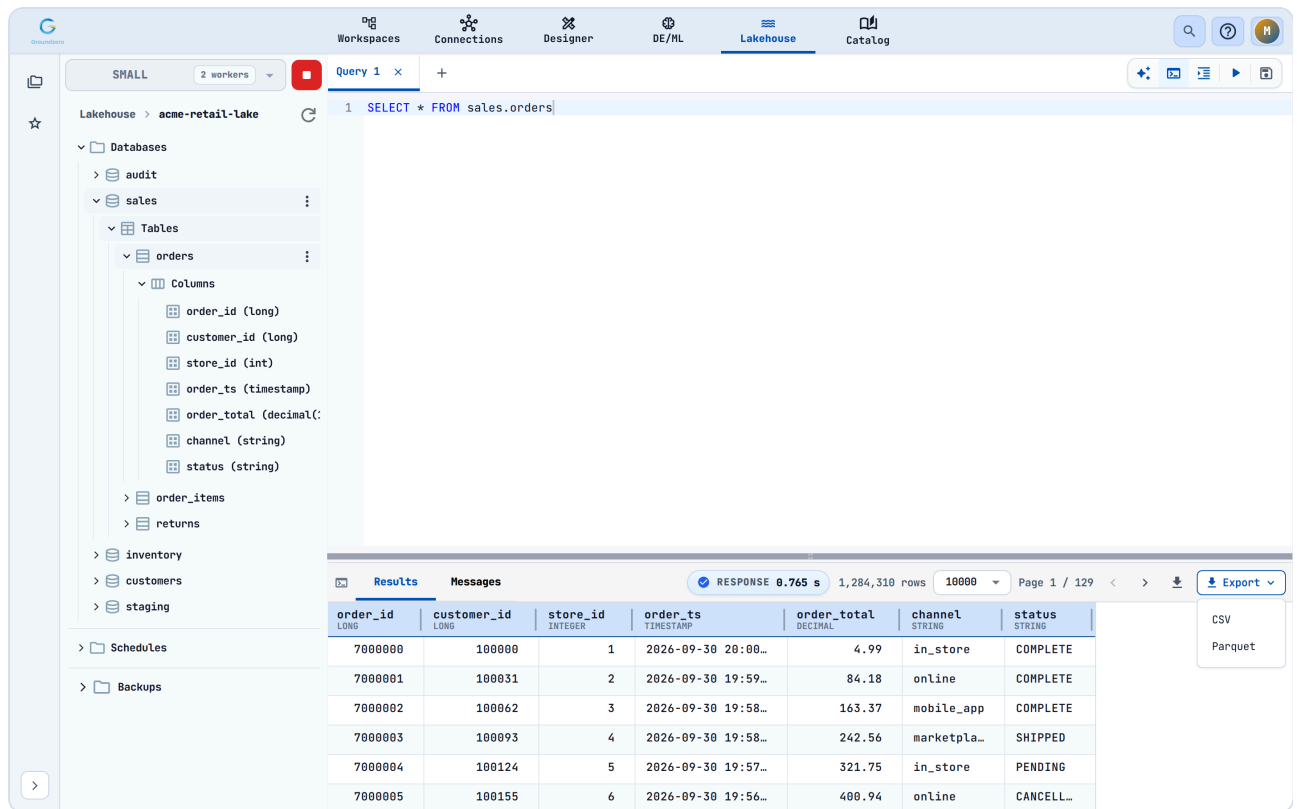
order_id	customer_id	store_id	order_ts	order_total	channel	status
7000000	100000	1	2026-09-30 20:00...	4.99	in_store	COMPLETE
7000001	100031	2	2026-09-30 19:59...	84.18	online	COMPLETE
7000002	100062	3	2026-09-30 19:58...	163.37	mobile_app	COMPLETE
7000003	100093	4	2026-09-30 19:58...	242.56	marketpla...	SHIPPED
7000004	100124	5	2026-09-30 19:57...	321.75	in_store	PENDING
7000005	100155	6	2026-09-30 19:56...	400.94	online	CANCELL...

A large result shows the exact row count, page controls and the Export button.

Paging past the first 100,000 rows needs the full result prepared as files in your warehouse bucket. The page shows "Preparing the full result for paging. Pages load here when it is ready." and pages become readable as they are written. Without an `ORDER BY`, row order can differ from the first run, so always sort a result you page through.

Exports

1. Click **Export** and choose **CSV** or **Parquet** in the **Export format** menu.

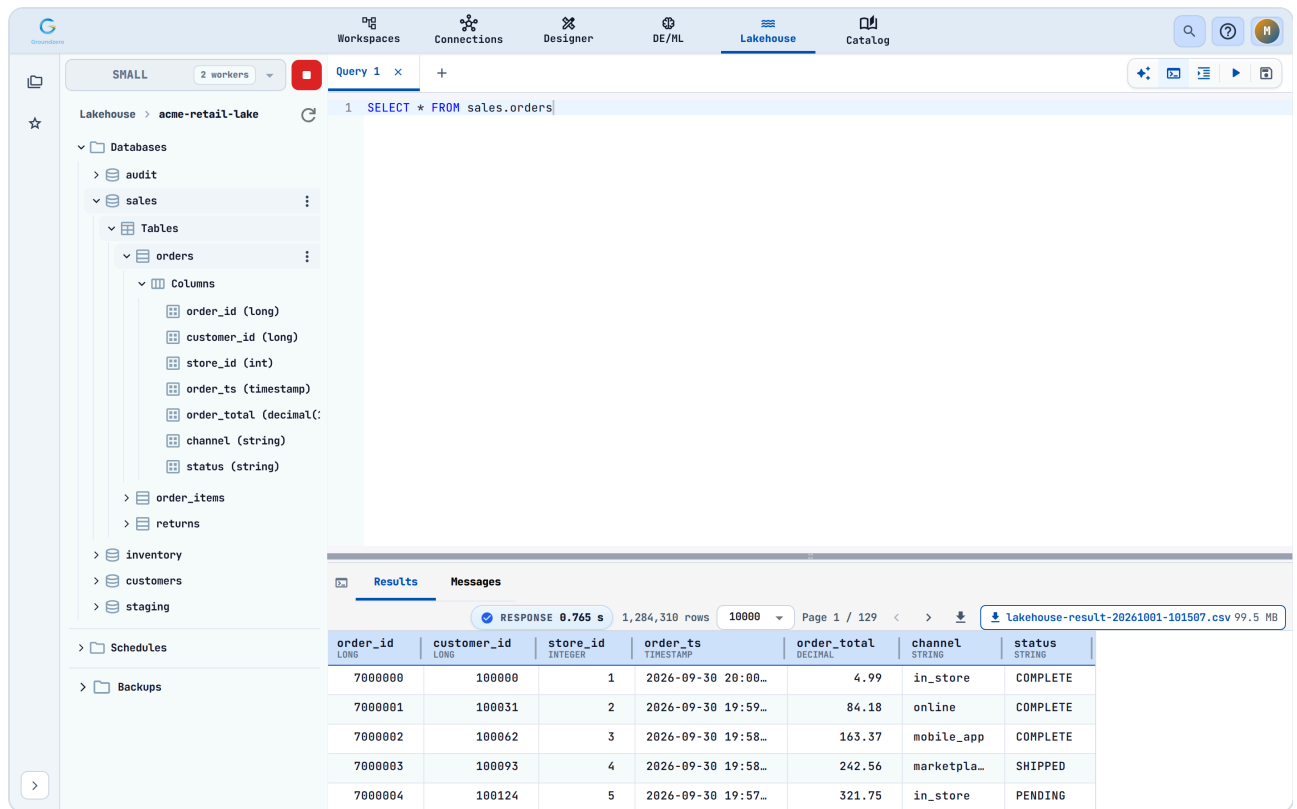


The screenshot shows the Groundzero Lakehouse interface. On the left, a sidebar displays the database structure for 'acme-retail-lake', including databases (audit, sales), tables (orders, order_items, returns), and other entities (inventory, customers, staging, Schedules, Backups). The 'orders' table is selected, showing its columns: order_id (Long), customer_id (Long), store_id (Int), order_ts (Timestamp), order_total (Decimal), channel (String), and status (String). The main panel shows a query editor with the text '1 SELECT * FROM sales.orders'. Below the editor, the 'Results' tab is active, displaying a table with 7 columns and 6 rows of data. The table headers are: order_id (LONG), customer_id (LONG), store_id (INTEGER), order_ts (TIMESTAMP), order_total (DECIMAL), channel (STRING), and status (STRING). The data rows show various order details, including order IDs, customer IDs, store IDs, timestamps, totals, channels, and statuses. To the right of the table, an 'Export' dropdown menu is open, showing options for 'CSV' and 'Parquet'. Above the table, a status bar indicates 'RESPONSE 0.765 s', '1,284,310 rows', and '10000' rows per page. The page number is 'Page 1 / 129'.

order_id LONG	customer_id LONG	store_id INTEGER	order_ts TIMESTAMP	order_total DECIMAL	channel STRING	status STRING
7000000	100000	1	2026-09-30 20:00...	4.99	in_store	COMPLETE
7000001	100031	2	2026-09-30 19:59...	84.18	online	COMPLETE
7000002	100062	3	2026-09-30 19:58...	163.37	mobile_app	COMPLETE
7000003	100093	4	2026-09-30 19:58...	242.56	marketpla...	SHIPPED
7000004	100124	5	2026-09-30 19:57...	321.75	in_store	PENDING
7000005	100155	6	2026-09-30 19:56...	400.94	online	CANCELL...

The Export menu offers CSV and Parquet.

- The toolbar shows "Preparing CSV file" with a timer and a **Stop export** button. If a status check fails it shows "Export: retrying...".
- When the file is ready, the toolbar shows a download link with the file name and size, for example `lakehouse-result-20261001-101507.csv`. The link works for about one hour; hover over it to see when it ends.



The screenshot shows the Groundzero Lakehouse interface. On the left, a sidebar displays the database structure for 'acme-retail-lake', including tables like 'orders' and 'order_items'. The main area shows a query editor with the text 'SELECT * FROM sales.orders'. Below the editor, the 'Results' tab is active, displaying a table with 8 columns: 'order_id', 'customer_id', 'store_id', 'order_ts', 'order_total', 'channel', and 'status'. The table contains 5 rows of data. Above the table, a status bar indicates 'RESPONSE 0.765 s', '1,284,310 rows', and a download link for 'Lakehouse-result-20261001-101507.csv 99.5 MB'.

order_id	customer_id	store_id	order_ts	order_total	channel	status
7000000	100000	1	2026-09-30 20:00...	4.99	in_store	COMPLETE
7000001	100031	2	2026-09-30 19:59...	84.18	online	COMPLETE
7000002	100062	3	2026-09-30 19:58...	163.37	mobile_app	COMPLETE
7000003	100093	4	2026-09-30 19:58...	242.56	marketpla...	SHIPPED
7000004	100124	5	2026-09-30 19:57...	321.75	in_store	PENDING

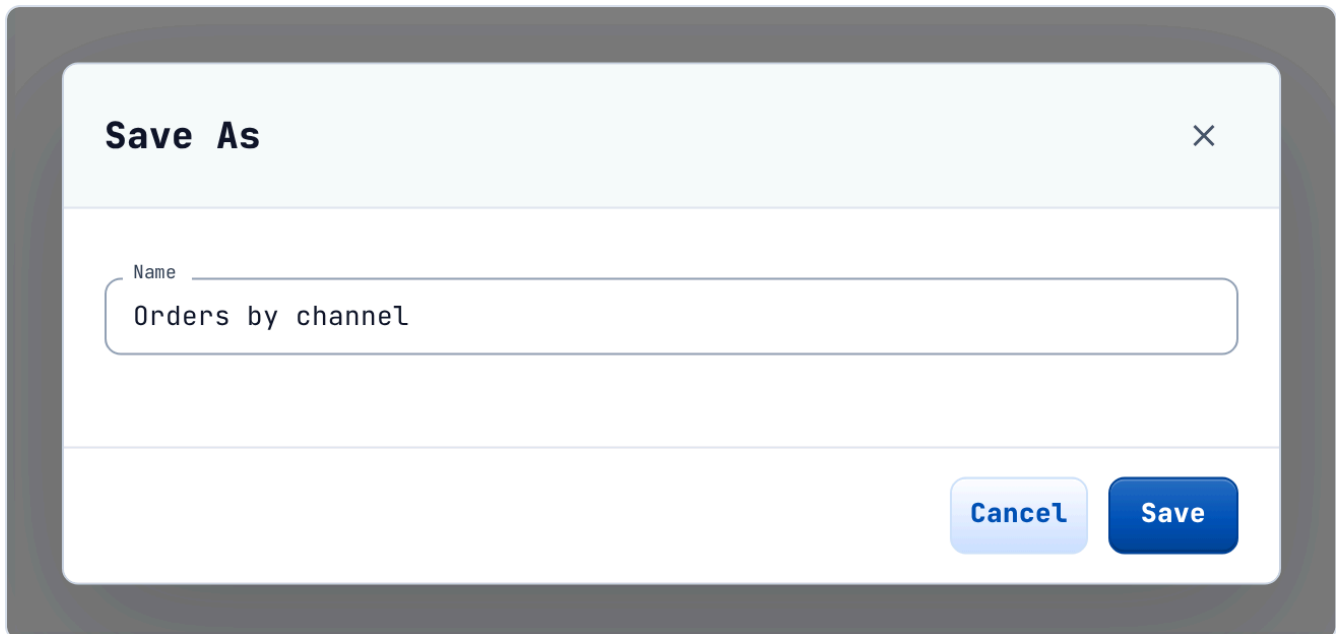
A finished export is a download link.

Exports re-run your query, so row filters and column masks apply exactly as in the grid. CSV exports have one header row, write nested values as JSON text and write time-zone timestamps in UTC. Each export is recorded in the audit log with the operation `EXPORT`. If an export fails, the red message beside **Retry export** says why (see Troubleshooting).

If you only need the result inside the Lakehouse, keep it there with `CREATE TABLE ... AS SELECT` instead of exporting it.

3.11 Saved queries

Save query (the disk icon) asks for a name in the **Save As** dialog and stores the SQL for you with the warehouse. The tab takes the saved name, and the notification reads "Query saved successfully". Saving a name that already exists is refused with "Key already exists."



The Save As dialog names the query.

Open **Saved Queries** in the left rail to list your queries with their warehouse, who changed them and when. Click a name to open the warehouse with the SQL in one tab (start a session to run it). **Rename** and **Delete** are in each row's **Actions** menu.

Name	Warehouse	Modified By	Modified On	Actions
Daily revenue by channel	acme-retail-lake	maya.patel	2026-09-29 10:12:00	...
Low stock by store	acme-retail-lake	maya.patel	2026-09-22 15:40:00	...
Supplier lead times	acme-retail-lake	maya.patel	2026-09-15 08:05:00	...
Email sends last week	acme-marketing-lake	maya.patel	2026-09-27 12:30:00	...

Saved Queries lists your queries and their warehouses.

Unsaved text lives only in the page. Reloading the browser loses open tabs, so save a query you want to keep.

3.12 Governed tables, data policies and table access

Two features decide what each person sees. Both apply to the editor, exports, saved queries and any statement that reads the table, including `CREATE TABLE ... AS SELECT`, which stores the masked or filtered values it was given.

Permissions you may be missing

Groundzero checks every statement at three levels. A refusal reads "You do not have All or `<Action>` permission", for example "You do not have All or Select permission".

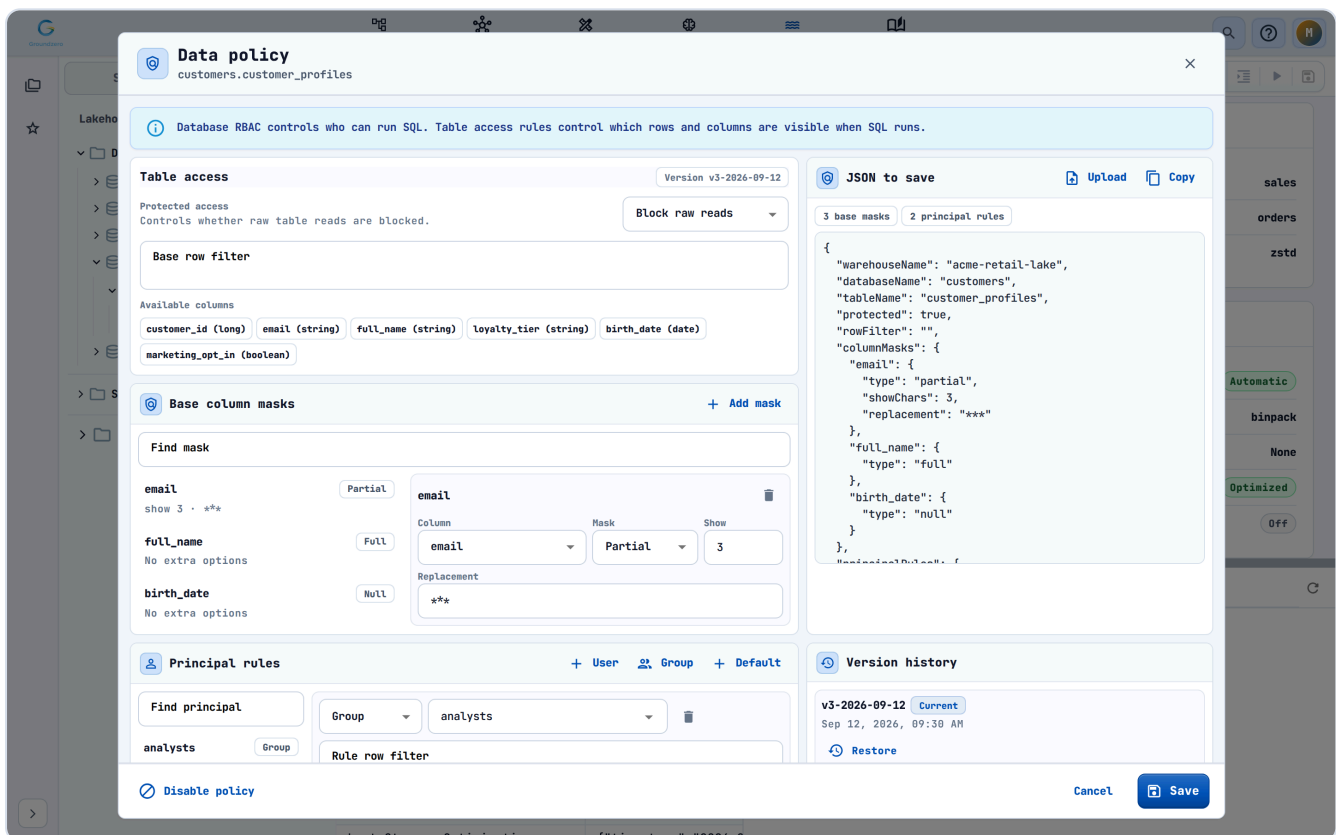
1. **Warehouse:** create, delete and list warehouses, create and drop databases.
2. **Database:** select, insert, update, create table, alter table and drop table on the tables of a database.
3. **Table:** allow or deny rules for single tables, set in the Table Access Manager.

A table rule that denies an action always wins. Ask your administrator for the grant named in the message.

Data policies

Choose **Data policy** in a table's menu to see or edit the policy for `database.table`. You need Select on the database to open it and All to change it. A policy has:

- **Protected access: Policy only** applies the policy to every read; **Block raw reads** stops all reads of the table with "Table access blocked by protected policy: `<database>.<table>`".
- **Base row filter:** a SQL condition added as a `WHERE` to every read for everybody. Click a column chip under **Available columns** to insert a column name. The templates `${user}`, `${site}`, `${groups}` and `${attr.<name>}` expand to the signed-in user, site, groups and user attributes.
- **Base column masks:** a mask per column.
- **Principal rules:** extra filters and masks for one **User**, one **Group** or the **Default**. A rule's filter is added to the base filter.



The Data policy dialog: protected access, base row filter, masks and principal rules.

Mask	What a reader sees
full	**** (or your replacement) for text; NULL for other types
partial	The first characters (3 by default) followed by *** (or your replacement)
hash	The SHA-256 hash of the value

Mask	What a reader sees
null	NULL
deny	The query fails with "Column access denied by masking policy"
unmask	Removes a base mask for the user, group or default of a principal rule

When several masks apply to a column, the strictest wins: deny, then full, null, hash and partial. `hash` and `partial` apply only when the column is selected directly; an expression such as `upper(email)` gets the full mask.

Masked columns cannot be used in `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `JOIN ... ON` or window clauses: the query fails with "Masked columns cannot be used in `<clause>` clauses". Select the masked column and filter on another one. `SELECT *` is expanded with the masks applied, but a `*` cannot be masked when the query has common table expressions, subqueries, joins or several tables; list the columns instead. A masked column that exists in several tables must be qualified ("Ambiguous unqualified masked column 'email'; qualify it").

JSON to save shows the whole policy as you edit it; **Upload** loads a policy and **Copy** copies it. **Version history** lists every saved version, newest first, with **Restore**. **Save**, **Disable policy** and **Restore** each create a new version, and a change applies from the next query.

Table Access Manager

Choose **Table Access Manager** in a table's menu to grant or deny actions on one table to users or groups. You need All on the database. Pick an **Action** (Select, Insert, Update, Alter, Drop or All), **Allow** or **Deny**, and move users or groups from **Available** to **Selected**. Rules inherited from the database are listed, and a deny always wins. Every add or remove is saved immediately.

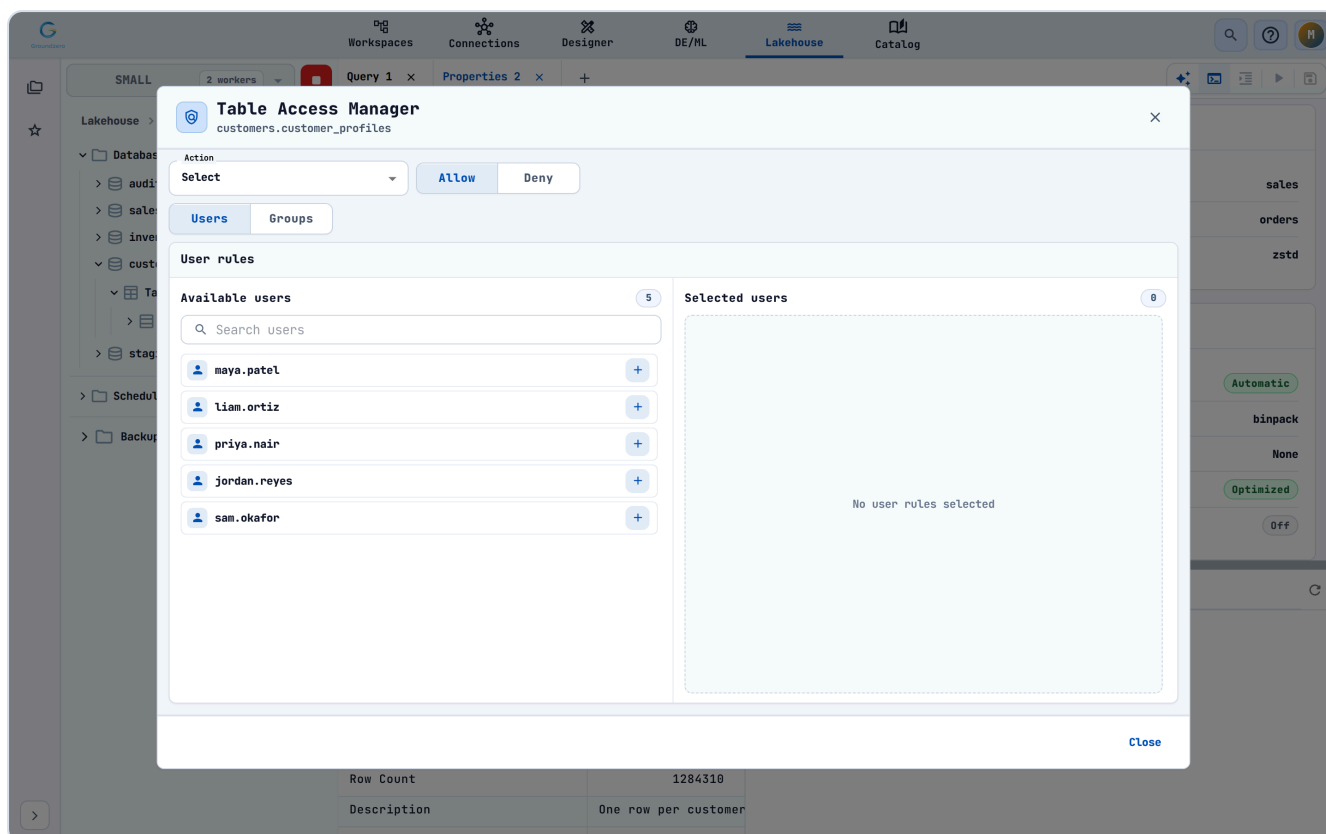


Table Access Manager: action, effect and the users or groups it applies to.

3.13 Read lakehouse tables from jobs and datasets

The tables you create here are ordinary Iceberg tables, so every other runtime reads them through a **Groundzero lakehouse connection** (Connections, then the Groundzero Lakehouse type). The connection's **Lakehouse URL** is the value copied by **Copy Lakehouse URL** on the Warehouses page. The same permissions, masks and row filters apply to the caller of every run.

From	How	See
Spark job or notebook	<code>spark.executeQuery(connName, sql)</code> reads with SQL; <code>spark.writeGzLakehouse(...)</code> appends or overwrites	Spark guide, Query and write Lakehouse (Iceberg) tables
DuckDB job or notebook	<code>session.readGzLakehouse(connName=..., tableName="sales.orders")</code> ; read only	DuckDB guide, Query the Groundzero Lakehouse
Standard dataset	A SQL query on the connection, for example <code>SELECT ... FROM sales.orders</code>	Standard guide, Query lakehouse tables

In every case, write tables as `database.table` . To write lakehouse tables from code, use the Spark runtime; DuckDB and Standard read only.

3.14 Backups, schedules and restore

A backup copies a database's data files and table definitions inside the warehouse bucket. It is a safety net for mistakes that snapshots cannot undo, such as dropping a table.

Time	User	Operation	Status	Database	Table	Duration
10/1/2026, 12:00:00...	maya.patel	Select	SUCCESS	sales	orders	1,840 ms
10/1/2026, 9:36:00 ...	liam.ortiz	Select	SUCCESS	inventory	stock_levels	625 ms
10/1/2026, 7:12:00 ...	priya.nair	Select	FAILED	sales	ordrs	26 ms
10/1/2026, 4:48:00 ...	maya.patel	Insert	SUCCESS	sales	returns	2,249 ms
10/1/2026, 2:24:00 ...	maya.patel	EXPORT	SUCCESS	sales	orders	41,262 ms
10/1/2026, 12:00:00...	liam.ortiz	Select	FAILED	customers	customer_profiles	65 ms
9/30/2026, 9:36:00 ...	priya.nair	Create	SUCCESS	staging	daily_rollup	9,398 ms
9/30/2026, 7:12:00 ...	maya.patel	Select	SUCCESS	sales	orders	1,840 ms
9/30/2026, 4:48:00 ...	liam.ortiz	Select	SUCCESS	inventory	stock_levels	625 ms
9/30/2026, 2:24:00 ...	priya.nair	Select	FAILED	sales	ordrs	26 ms
9/30/2026, 12:00:00...	maya.patel	Insert	SUCCESS	sales	returns	2,249 ms
9/30/2026, 9:36:00 ...	maya.patel	EXPORT	SUCCESS	sales	orders	41,262 ms
9/30/2026, 7:12:00 ...	liam.ortiz	Select	FAILED	customers	customer_profiles	65 ms
9/30/2026, 4:48:00 ...	priya.nair	Create	SUCCESS	staging	daily_rollup	9,398 ms
9/30/2026, 2:24:00 ...	maya.patel	Select	SUCCESS	sales	orders	1,840 ms
9/30/2026, 12:00:00...	liam.ortiz	Select	SUCCESS	inventory	stock_levels	625 ms
9/29/2026, 9:36:00 ...	priya.nair	Select	FAILED	sales	ordrs	26 ms
9/29/2026, 7:12:00 ...	maya.patel	Insert	SUCCESS	sales	returns	2,249 ms
9/29/2026, 4:48:00 ...	maya.patel	EXPORT	SUCCESS	sales	orders	41,262 ms

The Backups section lists each database's backups with their status.

Back up now. Choose **Backup Database** in a database's menu and enter a **Backup Name**: letters, digits and underscores, 3 to 50 characters, starting with a letter and not beginning or ending with `_` or `-`. The message "Database backup created successfully!" means the backup *started*; it runs in the background and shows **Running** until it finishes with **Completed** or **Failed**. A name that exists is refused ("Backup name `<name>` already exists").

Schedule backups. Expand **Schedules**, open a database's menu and choose **Schedule Backup**. Set the **Backup Name**, **Retention (days)** from 1 to 30, the start and **Repeat** (Hourly, Daily, Weekly, Monthly or Yearly). Each run appends a timestamp to the name. A database has one schedule; **Pause Schedule**, **Resume Schedule** and **Delete Schedule** are in the same menu, and deleting a schedule keeps the backups it already made.

Restore. Choose **Restore Database**, then the backup and a **Target Database Name** (letters, digits and underscores). "Database restoration started!" confirms the start; refresh the tree when it finishes.

RESTORING INTO THE ORIGINAL DATABASE

Restoring a backup into the database it came from replaces that database's tables with the backed-up version. Changes made after the backup are lost.

A backup whose retention has passed is removed, and restoring it fails with "Backup `<name>` has expired in S3 and was removed from the catalog. Take a fresh backup before restoring." **Mark stale backups failed** (in the **Databases** menu) marks backups that have been running for more than 180 minutes as failed.

3.15 Audit logs

Every statement that runs in a warehouse is written to the `audit.logs` table, with the time, user, operation, status, database, table, duration and the SQL. Only administrators can read it: expand `audit`, then **Tables**, `logs` and choose **Open audit logs**, or query it.

Time	User	Operation	Status	Database	Table	Duration
10/1/2026, 12:00:00...	maya.patel	Select	SUCCESS	sales	orders	1,840 ms
10/1/2026, 9:36:00 ...	liam.ortiz	Select	SUCCESS	inventory	stock_levels	625 ms
10/1/2026, 7:12:00 ...	priya.nair	Select	FAILED	sales	ordrs	26 ms
10/1/2026, 4:48:00 ...	maya.patel	Insert	SUCCESS	sales	returns	2,249 ms
10/1/2026, 2:24:00 ...	maya.patel	EXPORT	SUCCESS	sales	orders	41,262 ms
10/1/2026, 12:00:00...	liam.ortiz	Select	FAILED	customers	customer_profiles	65 ms
9/30/2026, 9:36:00 ...	priya.nair	Create	SUCCESS	staging	daily_rollup	9,398 ms
9/30/2026, 7:12:00 ...	maya.patel	Select	SUCCESS	sales	orders	1,840 ms
9/30/2026, 4:48:00 ...	liam.ortiz	Select	SUCCESS	inventory	stock_levels	625 ms
9/30/2026, 2:24:00 ...	priya.nair	Select	FAILED	sales	ordrs	26 ms
9/30/2026, 12:00:00...	maya.patel	Insert	SUCCESS	sales	returns	2,249 ms
9/30/2026, 9:36:00 ...	maya.patel	EXPORT	SUCCESS	sales	orders	41,262 ms
9/30/2026, 7:12:00 ...	liam.ortiz	Select	FAILED	customers	customer_profiles	65 ms
9/30/2026, 4:48:00 ...	priya.nair	Create	SUCCESS	staging	daily_rollup	9,398 ms
9/30/2026, 2:24:00 ...	maya.patel	Select	SUCCESS	sales	orders	1,840 ms
9/30/2026, 12:00:00...	liam.ortiz	Select	SUCCESS	inventory	stock_levels	625 ms
9/29/2026, 9:36:00 ...	priya.nair	Select	FAILED	sales	ordrs	26 ms
9/29/2026, 7:12:00 ...	maya.patel	Insert	SUCCESS	sales	returns	2,249 ms
9/29/2026, 4:48:00 ...	maya.patel	EXPORT	SUCCESS	sales	orders	41,262 ms

The audit log lists every statement with its user, status and duration.

SQL

```
SELECT timestamp, userId, operation, status, execution_time_ms, query
FROM audit.logs
WHERE status = 'FAILED'
ORDER BY timestamp DESC
LIMIT 100
```

The audit viewer has a range (7 days by default, 30, 90 or all), a page size (20, 50 or 100), refresh, **Download current audit logs page as CSV**, **Compact audit logs** (expires snapshots older than 30 days, keeping the last 20) and **Rewrite audit files**. Click a cell to open the details drawer. New rows can appear up to about 30 seconds after a statement finishes, and `audit.logs` can be read but never changed: "Audit logs can only be queried. Modifications are not permitted."

4 SQL reference

4.1 Statements and the permission each needs

Statement	Permission
<code>SELECT</code> , <code>WITH ... SELECT</code> , time travel (<code>VERSION AS OF</code> , <code>TIMESTAMP AS OF</code>)	Select on each table
<code>SELECT</code> from a metadata table (<code>db.t.snapshots</code>)	Select on the base table
<code>SHOW TABLES IN db</code> , <code>SHOW VIEWS IN db</code>	Select on the database
<code>SHOW DATABASES</code> , <code>DESCRIBE DATABASE</code>	List on the warehouse
<code>DESCRIBE [EXTENDED \ FORMATTED] db.t</code> , <code>SHOW TBLPROPERTIES</code> , <code>SHOW PARTITIONS</code> , <code>SHOW COLUMNS</code> , <code>SHOW CREATE TABLE</code> , <code>ANALYZE TABLE</code> , <code>REFRESH TABLE</code>	Select on the table
<code>INSERT INTO</code>	Insert on the target, Select on the sources
<code>INSERT OVERWRITE</code> , <code>UPDATE</code> , <code>DELETE</code> , <code>MERGE INTO</code> , <code>TRUNCATE TABLE</code>	Update on the target, Select on the sources
<code>CREATE TABLE</code> , <code>CREATE TABLE ... AS SELECT</code> , <code>CREATE VIEW</code>	CreateTable on the database, Select on the sources
<code>CREATE TEMPORARY VIEW</code>	Select on the sources
<code>DROP TABLE</code> , <code>DROP VIEW</code>	DropTable
<code>ALTER TABLE</code> , <code>COMMENT ON</code> , <code>REPLACE TABLE ... AS SELECT</code>	AlterTable, Select on the sources
<code>CREATE DATABASE</code>	CreateDatabase on the warehouse
<code>DROP DATABASE</code>	DropDatabase on the warehouse
<code>ALTER DATABASE</code> , <code>SET key = value</code>	All on the warehouse
<code>CALL system.<procedure>(...)</code>	AlterTable on the table (see below)

A permission named **All** at a level covers every action at that level. An explicit table-level deny always wins.

4.2 Queries

SQL

```
SELECT [DISTINCT] expr [AS alias], ...
FROM database.table [AS alias]
  [[INNER | LEFT | RIGHT | FULL | CROSS] JOIN database.table ON condition]
  [LEFT SEMI JOIN ... | LEFT ANTI JOIN ...]
[WHERE condition]
[GROUP BY expr, ... | GROUP BY ALL]
[HAVING condition]
[ORDER BY expr [ASC | DESC] [NULLS FIRST | NULLS LAST], ...]
[LIMIT n]
```

Prefix with `WITH name AS (SELECT ...), ...` for common table expressions. Set operators `UNION [ALL]`, `INTERSECT` and `EXCEPT` combine queries. `PIVOT`, `LATERAL VIEW explode(...)`, `TABLESAMPLE` and `GROUPING SETS`, `ROLLUP` and `CUBE` are supported:

SQL

```
SELECT * FROM (
  SELECT channel, status, order_total FROM sales.orders
) PIVOT (SUM(order_total) FOR status IN ('COMPLETE', 'RETURNED'))
```

4.3 Time travel

SQL

```
SELECT ... FROM db.t TIMESTAMP AS OF '2026-09-30 12:00:00'
SELECT ... FROM db.t VERSION AS OF 4821763650912831907
SELECT ... FROM db.t VERSION AS OF 'tag_name'
```

`VERSION AS OF` takes a snapshot id or the name of a branch or tag. `FOR SYSTEM_TIME AS OF` and `FOR SYSTEM_VERSION AS OF` are not accepted by the Lakehouse; use the forms above.

4.4 Creating and changing tables

SQL

```
CREATE [OR REPLACE] TABLE [IF NOT EXISTS] db.t (col type [COMMENT 'text'], ...)
  USING iceberg
  [PARTITIONED BY (col | years(col) | months(col) | days(col) | hours(col)
    | bucket(n, col) | truncate(n, col), ...)]
  [COMMENT 'text']
  [TBLPROPERTIES ('key' = 'value', ...)]
```

```
CREATE TABLE db.t USING iceberg [PARTITIONED BY (...)] AS SELECT ...
REPLACE TABLE db.t USING iceberg AS SELECT ...
```

```
ALTER TABLE db.t ADD COLUMN col type [COMMENT 'text'] [FIRST | AFTER other]
ALTER TABLE db.t DROP COLUMN col
ALTER TABLE db.t RENAME COLUMN col TO new_name
ALTER TABLE db.t ALTER COLUMN col TYPE wider_type
ALTER TABLE db.t ALTER COLUMN col {SET | DROP} NOT NULL
ALTER TABLE db.t ADD | DROP PARTITION FIELD transform
ALTER TABLE db.t REPLACE PARTITION FIELD old_transform WITH new_transform
ALTER TABLE db.t WRITE ORDERED BY col [ASC | DESC], ...
ALTER TABLE db.t SET TBLPROPERTIES ('key' = 'value')
ALTER TABLE db.t UNSET TBLPROPERTIES ('key')
ALTER TABLE db.t RENAME TO db.new_name
COMMENT ON TABLE db.t IS 'text'
DROP TABLE [IF EXISTS] db.t
```

4.5 Changing data

SQL

```
INSERT INTO db.t [(col, ...)] VALUES (...), (...)
```

```
INSERT INTO db.t SELECT ...
```

```
INSERT OVERWRITE db.t SELECT ...
```

```
UPDATE db.t SET col = expr [, ...] [WHERE condition]
```

```
DELETE FROM db.t [WHERE condition]
```

```
TRUNCATE TABLE db.t
```

```
MERGE INTO db.target AS t USING (db.source | (SELECT ...)) AS s ON condition
  WHEN MATCHED [AND condition] THEN UPDATE SET col = expr [, ...] | UPDATE SET * | DELETE
  WHEN NOT MATCHED [AND condition] THEN INSERT (cols) VALUES (...) | INSERT *
```

4.6 Databases and information

SQL

```
CREATE DATABASE [IF NOT EXISTS] db
```

```
DROP DATABASE [IF EXISTS] db
```

```
SHOW DATABASES
```

```
SHOW TABLES IN db
```

```
DESCRIBE [EXTENDED | FORMATTED] db.t
```

```
SHOW CREATE TABLE db.t
```

```
SHOW TBLPROPERTIES db.t
```

```
SHOW PARTITIONS db.t
```

```
SHOW COLUMNS IN db.t
```

A database must be empty before you can drop it.

4.7 Table procedures

Iceberg procedures run with `CALL`. Name arguments with `=>`.

Procedure	Purpose
<code>rewrite_data_files(table => 't')</code>	Compact small files; add <code>strategy</code> , <code>sort_order</code> or <code>where</code>
<code>rewrite_manifests('db.t')</code>	Merge manifest files
<code>expire_snapshots(table => 't', older_than => ts, retain_last => n)</code>	Delete old snapshots
<code>remove_orphan_files(table => 't')</code>	Delete files that no snapshot references
<code>rollback_to_snapshot('db.t', snapshot_id)</code>	Move the table back to a snapshot
<code>rollback_to_timestamp('db.t', ts)</code>	Move the table back to a time
<code>set_current_snapshot('db.t', snapshot_id)</code>	Make a snapshot current
<code>cherrypick_snapshot('db.t', snapshot_id)</code>	Apply one snapshot's changes
<code>ancestors_of('db.t')</code>	List a snapshot's ancestors (needs only Select)

4.8 Blocked or restricted statements

What you ran	What you see
<code>USE database</code>	'Use' keyword not allowed. Use fully qualified table names.
A table without its database, for example <code>FROM orders</code>	Use fully qualified table names.
A statement that cannot be parsed for authorization	Access denied: Unable to parse query for authorization.
A statement that names no table, such as <code>SELECT 1</code> or <code>EXPLAIN ...</code>	Access denied: Unable to determine operations for authorization. Query a table instead.
A database that does not exist	Following namespaces specified in the query does not exist: followed by the names
<code>DROP DATABASE audit</code>	The system audit database cannot be dropped.
Changing <code>audit.logs</code>	Audit logs can only be queried. Modifications are not permitted.
Reading <code>audit.logs</code> without the administrator role	Access to audit.logs table is restricted to Admin group members.
The word <code>namespace</code> as a separate word anywhere in the text	No resource named 'namespace'.
Metadata tables of a table with data policies	Metadata tables of <code><table></code> are ...: the table has data policies ... Query the table itself instead.
A table whose policy is Block raw reads	Table access blocked by protected policy: <code><database>.<table></code>

What you ran	What you see
A masked column in a filter, grouping, ordering or join	Masked columns cannot be used in <clause> clauses

ONE STATEMENT PER RUN

Send one statement at a time. When a text contains a `SELECT`, only the first statement is run and the rest is ignored, without an error.

4.9 Behaviour of SQL in the Lakehouse

- **ANSI mode.** An invalid cast, an arithmetic overflow or a division by zero fails the query. Use `try_cast(x AS INT)`, `try_divide(a, b)` or `a / nullif(b, 0)` to get `NULL` instead.
- **Names are case-insensitive.** `Sales.Orders` and `sales.orders` are the same table; Iceberg stores the names as written.
- **Overwrites are dynamic.** `INSERT OVERWRITE` replaces only the partitions present in the new rows.
- **SET lasts for the session.** A value set with `SET key = value` stays until the session stops.
- **Temporary views.** `CREATE TEMPORARY VIEW name AS SELECT ...` creates a view that exists only until the session ends and can be queried without a database.
- **No keyboard shortcut to run.** Use the **Run query** button.

5 Limits and behaviour

5.1 Sessions

Item	Value
Time Groundzero waits for a session to become ready	180 seconds
Session idle shutdown	30 minutes without requests (a running query counts as activity)
Browser remembers your running session	30 minutes
Minimum workers	At least 1, default 2

When a session stops, whatever lived only in memory is gone: temporary views, `SET` values and unread results. Tables and data are never affected.

5.2 Queries and results

Item	Value
Time a query is answered directly	About 55 seconds, then it continues as a long-running query
Longest run of one query	30 minutes
Queries running at once in a session	4 (result preparation and exports count)
Rows kept in the session for fast paging	100,000 per query
Rows per page	50, 100, 500, 1000, 10000 (default 10000)
Columns loaded at first, then per group	300, then 200
Largest result that can be paged or exported	50,000,000 rows or 20 GiB
Prepared result pages remain readable	About 15 minutes after you last read them
Preparation is stopped if nobody is waiting	After 3 minutes
Same query served from cache	Up to 5 minutes, cleared by any data change
Finished detached query kept	15 minutes

5.3 Exports

Item	Value
Formats	CSV, Parquet (one file each)
Largest single export file	5 GB
Download link	About 1 hour

Item	Value
File name	lakehouse-result- <code><date>-<time></code> .csv or .parquet
Nested columns in CSV	Written as JSON text
Row filters and masks	Applied
Audit	One <code>EXPORT</code> entry

Temporary result files are written under your warehouse bucket and expire after one day.

5.4 Other values

Item	Value
Snapshots listed in Snapshot History	10 most recent
Sample Data rows	10 (administrators)
Compaction defaults	Commit group 100 GB, partial progress 10 commits, 1 concurrent group
Backup retention in a schedule	1 to 30 days
Backup name	3 to 50 characters
Backups marked failed as stale	Running for more than 180 minutes
Audit rows appear	Within about 30 seconds
Audit viewer pages cached	60 seconds
Audit compaction	Expires snapshots older than 30 days, keeps the last 20

6 Troubleshooting

Symptom	Cause	Fix
Run query is greyed out	No session is running, or no query tab is active	Pick a compute, start a session and select a query tab
"please select Compute"	You pressed start without choosing a compute	Open the Compute list and pick a size
"Session start failed"	The session was not ready in 180 seconds	Try again in a moment
"The lakehouse session is not reachable. Start the session and try again."	The session stopped (30 minutes idle, or you stopped it)	Start a new session; tables are unchanged
"The lakehouse session did not answer in time. Try again in a moment."	The session was busy or restarting	Wait a moment and run again
Use fully qualified table names.	A table was written without its database, or you used <code>USE</code>	Write <code>database.table</code> everywhere
'Use' keyword not allowed.	You ran <code>USE db</code>	Remove it and qualify names
Access denied: Unable to determine operations for authorization.	The statement names no table (<code>SELECT 1 , EXPLAIN</code>)	Query a table, or run a different statement
Access denied: Unable to parse query for authorization.	The SQL has a syntax error	Fix the syntax; Format SQL can help spot it
Following namespaces specified in the query does not exist:	A database name is misspelled or missing	Check the tree, or <code>CREATE DATABASE</code>
You do not have All or Select permission	You lack that action on the table or database	Ask an administrator for the grant, or check Table Access Manager for a deny
Table access blocked by protected policy	The table's policy uses Block raw reads	Ask the table's owner for a different access route
Masked columns cannot be used in WHERE clauses	The query filters, groups, sorts or joins on a masked column	Select the column and use a different column to filter
Metadata tables of ... have data policies	A policy protects the table	Query the table itself
Text after <code>--</code> is ignored	Line breaks are removed before sending	Use <code>/* ... */</code> comments
Only the first statement ran	Several statements in one tab	Run one statement per tab
A division or cast raises an error	ANSI mode	Use <code>try_divide</code> , <code>nullif</code> or <code>try_cast</code>
"Still running. Results load here when the query finishes."	The query takes more than about 55 seconds	Wait, or press Stop query

Symptom	Cause	Fix
"Query ran longer than 30 minutes and was stopped."	The limit was reached	Filter earlier, use more workers, or write the result with <code>CREATE TABLE ... AS SELECT</code>
"This session is already running 4 queries. ..."	Four queries are running	Stop one, or wait
"The session no longer has this query. Run it again."	The 15-minute hold on the result ended	Run the query again
Export button is missing	The result fits in the 100,000 rows kept	Use Download current page as CSV
"This export is larger than 5 GB, the most one download can hold. ..."	The single-file limit	Add a filter, select fewer columns, or <code>CREATE TABLE ... AS SELECT</code>
"This result is too large to page through or export here (limit: 50,000,000 rows or 20 GiB). ..."	The result is larger than the maximum	Filter, or save it as a table
"Preparing the full result took too long and was stopped. ..."	Preparation exceeded its time	Narrow the query, then try again
"Stopped because nobody was waiting for it any more."	The page was closed for 3 minutes while preparing	Run and export again
"Deep paging and export aren't available for this warehouse: its bucket encryption doesn't allow Groundzero to write temporary results."	The bucket's encryption settings block temporary files	Ask the bucket owner to allow Groundzero to write encrypted results
"Could not get access to the warehouse bucket for this result. Try again in a moment."	Temporary access could not be obtained	Retry
"Could not read the prepared result. Run the query again."	The prepared result files could not be read	Run the query again
"Only a finished SELECT result can be paged or exported."	The result came from a <code>UNION</code> , <code>VALUES</code> or other form, or is not finished	Wrap in <code>SELECT * FROM (...)</code>
"Page <code>n</code> is out of range. Total pages: <code>t</code> for page size <code>s</code> ."	The page does not exist for that page size	Go to page 1
"This page is past the rows kept in memory. Prepare the full result to page further."	You paged beyond the first 100,000 rows	Use the prepare option in the grid
"Key already exists."	A saved query has that name	Choose another name
Warehouse cannot be deleted while it contains resources. ...	It still holds databases, tables or views	Drop them first
"Delete its tables and views first."	A database to be deleted is not empty	Drop its tables and views
"Backup name ... already exists"	The backup name is used	Choose a new name
"Backup ... has expired in S3 and was removed from the catalog. ..."	Its retention ended	Take a fresh backup

Symptom	Cause	Fix
Audit logs: "Unauthorized access"	Only administrators can read them	Ask an administrator
Recent audit rows are missing	Rows are written in batches	Refresh after about 30 seconds
A <code>MERGE</code> fails because several source rows match one target row	The staging data has duplicate keys	De-duplicate the source

7 Examples gallery

The examples use fictional data for Acme Retail, held in these tables of the `acme-retail-lake` warehouse. Run each statement on its own.

Table	Columns
<code>sales.orders</code>	<code>order_id</code> , <code>customer_id</code> , <code>store_id</code> , <code>order_ts</code> , <code>order_total</code> , <code>channel</code> , <code>status</code>
<code>sales.order_items</code>	<code>order_id</code> , <code>line_no</code> , <code>sku</code> , <code>quantity</code> , <code>unit_price</code>
<code>sales.returns</code>	<code>order_id</code> , <code>returned_at</code> , <code>reason</code>
<code>customers.profiles</code>	<code>customer_id</code> , <code>name</code> , <code>email</code> , <code>segment</code> , <code>signup_date</code> , <code>country</code> , <code>tags</code>
<code>customers.profile_updates</code>	<code>customer_id</code> , <code>name</code> , <code>email</code> , <code>segment</code> , <code>deleted</code>
<code>inventory.products</code>	<code>sku</code> , <code>name</code> , <code>category</code> , <code>unit_cost</code>
<code>inventory.stock</code>	<code>sku</code> , <code>store_id</code> , <code>on_hand</code> , <code>reorder_point</code> , <code>updated_at</code>

7.1 Example 1: Revenue by channel and month

Group completed orders by month and sales channel; `date_trunc` turns each timestamp into the first day of its month.

SQL

```
SELECT date_trunc('month', order_ts) AS month,
       channel,
       COUNT(*)                      AS orders,
       ROUND(SUM(order_total), 2)    AS revenue
FROM sales.orders
WHERE status = 'COMPLETE'
GROUP BY 1, 2
ORDER BY month, revenue DESC
```

7.2 Example 2: Top three products in each category

A window function ranks products inside each category, and an outer query keeps the first three.

SQL

```
WITH product_sales AS (
  SELECT p.category, p.name,
         SUM(i.quantity * i.unit_price) AS revenue
  FROM sales.order_items AS i
  JOIN inventory.products AS p ON p.sku = i.sku
  GROUP BY p.category, p.name
),
ranked AS (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY category ORDER BY revenue DESC) AS rnk
  FROM product_sales
)
SELECT category, name, revenue
FROM ranked
WHERE rnk ≤ 3
ORDER BY category, rnk
```

7.3 Example 3: Running total and seven-day average

Daily revenue with a cumulative total and a moving average over the current and six previous days.

SQL

```
WITH daily AS (
  SELECT CAST(order_ts AS DATE) AS day, SUM(order_total) AS revenue
  FROM sales.orders
  WHERE status = 'COMPLETE'
  GROUP BY 1
)
SELECT day,
       revenue,
       SUM(revenue) OVER (ORDER BY day) AS cumulative,
       AVG(revenue) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
       AS avg_7_day
FROM daily
ORDER BY day
```

7.4 Example 4: Customers who have never ordered

An anti join returns the customers with no matching order, a quick list for a win-back campaign.

SQL

```
SELECT c.customer_id, c.name, c.email
FROM customers.profiles AS c
LEFT ANTI JOIN sales.orders AS o
  ON o.customer_id = c.customer_id
WHERE c.signup_date < current_date() - INTERVAL 30 DAYS
```

7.5 Example 5: Low-stock list

Join stock levels to products and list items at or under their reorder point, with the shortfall.

SQL

```
SELECT p.category, p.name, s.store_id, s.on_hand, s.reorder_point,
       s.reorder_point - s.on_hand AS shortfall
FROM inventory.stock AS s
JOIN inventory.products AS p ON p.sku = s.sku
WHERE s.on_hand ≤ s.reorder_point
ORDER BY shortfall DESC, p.category
```

7.6 Example 6: Upsert customer changes with MERGE

Apply a batch of changes: delete flagged customers, update the rest that match and insert the new ones.

`INSERT *` fills every column by name.

SQL

```
MERGE INTO customers.profiles AS t
USING customers.profile_updates AS s
  ON t.customer_id = s.customer_id
 WHEN MATCHED AND s.deleted THEN DELETE
 WHEN MATCHED THEN UPDATE SET
   t.name = s.name, t.email = s.email, t.segment = s.segment
 WHEN NOT MATCHED AND NOT s.deleted THEN INSERT
   (customer_id, name, email, segment)
 VALUES (s.customer_id, s.name, s.email, s.segment)
```

7.7 Example 7: Correct statuses after returns

Mark orders that appear in the returns table as `RETURNED` with an `UPDATE` that uses a subquery.

SQL

```
UPDATE sales.orders
SET status = 'RETURNED'
WHERE status = 'COMPLETE'
      AND order_id IN (SELECT order_id FROM sales.returns)
```

7.8 Example 8: Build a partitioned daily rollup

Materialise a summary table, partitioned by day, from the orders table. Later queries read this small table instead of scanning orders.

SQL

```
CREATE TABLE sales.daily_channel_revenue USING iceberg
PARTITIONED BY (day)
AS SELECT CAST(order_ts AS DATE) AS day,
       channel,
       COUNT(*) AS orders,
       SUM(order_total) AS revenue
FROM sales.orders
WHERE status = 'COMPLETE'
GROUP BY 1, 2
```

7.9 Example 9: Compare today with yesterday using time travel

Count the rows now and as they were at the start of the day to see what a load added.

SQL

```
SELECT 'now' AS version, COUNT(*) AS orders FROM sales.orders
UNION ALL
SELECT 'start of day', COUNT(*)
FROM sales.orders TIMESTAMP AS OF '2026-10-01 00:00:00'
```

7.10 Example 10: Inspect a table's snapshots and files

Use the metadata tables to see recent commits and whether the table has many small files.

SQL

```
SELECT snapshot_id, committed_at, operation,
       summary['added-records'] AS added_records
FROM sales.orders.snapshots
ORDER BY committed_at DESC
LIMIT 5
```

SQL

```
SELECT COUNT(*) AS data_files,
       ROUND(AVG(file_size_in_bytes) / 1048576, 1) AS avg_mb
FROM sales.orders.files
```

7.11 Example 11: Pivot channel revenue into columns

Turn one row per channel into one column per channel for a dashboard.

SQL

```
SELECT * FROM (
  SELECT CAST(order_ts AS DATE) AS day, channel, order_total
  FROM sales.orders
  WHERE status = 'COMPLETE'
)
PIVOT (
  SUM(order_total)
  FOR channel IN ('online', 'in_store', 'mobile_app', 'marketplace')
)
ORDER BY day
```

7.12 Example 12: Evolve a table without rewriting it

Add a column and change the partitioning. Existing data is untouched, and new rows use the new layout.

SQL

```
ALTER TABLE sales.orders ADD COLUMN coupon_code STRING AFTER status
```

SQL

```
ALTER TABLE sales.orders REPLACE PARTITION FIELD days(order_ts) WITH months(order_ts)
```

7.13 Example 13: Tags stored as an array

Explode a customer's tag array to count customers per tag; `LATERAL VIEW` turns each array element into its own row.

SQL

```
SELECT tag, COUNT(*) AS customers
FROM customers.profiles
LATERAL VIEW explode(tags) t AS tag
GROUP BY tag
ORDER BY customers DESC
```

7.14 Example 14: Safe division and clean text

ANSI mode raises an error when you divide by zero. `try_divide` returns `NULL` instead, and `try_cast` does the same for a value that is not a number.

SQL

```
SELECT p.sku,
       TRIM(UPPER(p.name)) AS name,
       try_divide(SUM(i.unit_price * i.quantity), SUM(i.quantity)) AS avg_price,
       try_cast(p.category AS INT) AS category_code
FROM inventory.products AS p
LEFT JOIN sales.order_items AS i ON i.sku = p.sku
GROUP BY p.sku, p.name, p.category
```

7.15 Example 15: Clean up a busy table

Compact small files, then expire snapshots older than a month while keeping the last five.

SQL

```
CALL system.rewrite_data_files(table => 'sales.orders')
```

SQL

```
CALL system.expire_snapshots(table => 'sales.orders',
                             older_than => TIMESTAMP '2026-09-01 00:00:00',
                             retain_last => 5)
```

7.16 Example 16: Find slow statements (administrators)

The audit log shows which statements took longest in the last day.

SQL

```
SELECT timestamp, userId, operation, execution_time_ms / 1000 AS seconds, query
FROM audit.logs
WHERE timestamp ≥ current_timestamp() - INTERVAL 1 DAY
      AND status = 'SUCCESS'
ORDER BY execution_time_ms DESC
LIMIT 20
```

APPENDIX A

Choosing a Runtime and Shared Concepts

Compare the runtimes and learn the concepts they share.

- A.1 Choosing a runtime
- A.2 How a run starts
- A.3 What your code can use during a run
- A.4 Connections and secrets
- A.5 The lakehouse (Iceberg)
- A.6 Custom runtimes
- A.7 ML variants
- A.8 Jobs: lifecycle, status and logs
- A.9 Glossary

Groundzero offers four ways to run your data work: the **Standard** runtime, the **DuckDB** runtime, the **Spark** runtime and the **Lakehouse** SQL engine. This chapter covers what they have in common, so the per-runtime guides can stay focused. It explains how to choose a runtime, how a run starts, how to size compute, what your code can use during a run, how connections and the lakehouse work, how custom and machine-learning runtimes are added, and how to follow a run to the end.

A.1 Choosing a runtime

A.1.1 At a glance

	Standard	DuckDB	Spark	Lakehouse
Engine	Managed query engine, SQL plus pandas Prep	DuckDB 1.5.5 on one machine	Apache Spark 3.5.6 cluster	Spark SQL 4.0 with Iceberg 1.10
Compute size	None	You choose	You choose, plus minimum workers	You choose, plus minimum workers
Start	Nothing to start	Quick start, then kept warm for datasets	Cluster start, then kept warm for datasets	Start a session, then run many queries
Connections	One SQL source or the lakehouse	Many, in one query	Many, in one job	The warehouse you open
Writes	No (read queries)	Yes, with the helper library	Yes, with the helper library	Yes, with SQL on Iceberg tables
Scale	Pages of rows	Fits in one machine's memory, or streams from storage	Distributed across workers	Distributed across workers, results paged
Languages	SQL and Python (pandas)	Python and SQL	Python (PySpark) and SQL	SQL (Spark dialect)
Helper library	None	<code>GZ.DuckDBUtils</code>	<code>GZ.SparkUtils</code>	None
ML variants	None	scikit-learn, PyTorch, TensorFlow	scikit-learn, PyTorch, TensorFlow	None
Best for	Charts, dashboards, quick lookups	Fast analytics, ETL, joins across sources	Large data, wide shuffles, distributed ETL	Exploring and maintaining Iceberg tables with SQL

A.1.2 Decision guide

Work down this list and stop at the first match.

1. Is it one query on one SQL source or the lakehouse, feeding a chart, dashboard or API read? Use **Standard**.
2. Is the work interactive SQL on Iceberg tables: exploring, creating or altering tables, `MERGE`, time travel, table maintenance, policies? Use the **Lakehouse**: open a warehouse, start a session and write SQL.
3. Does it join several connections, write results, run on a schedule, or run Python beyond a few lines of pandas? Continue.
4. Does the data fit comfortably in the memory of one machine, or can DuckDB stream it from object storage (Parquet, Iceberg)? Use **DuckDB**. It starts quickly and needs no cluster.

5. **Is the data too large for one machine, or does the job need distributed shuffles, joins of very large tables or Spark-native libraries?** Use **Spark**.
6. **Do you need scikit-learn, PyTorch or TensorFlow?** Pick the matching ML variant of DuckDB or Spark.
7. **Do you need packages that are not installed?** Ask an administrator for a custom runtime built on the runtime you picked.

START SMALL

Prototype on Standard or DuckDB. Move to Spark when a job outgrows one machine, not before. The Groundzero helper libraries of DuckDB and Spark follow the same shape, which keeps a port short.

A.1.3 Where the choice is made

You choose a runtime in three places: when you create a dataset, when you create or schedule an ETL job, and when you start a notebook. The runtime control lists the runtimes available to you and, except for Standard, a compute size. Standard appears only in the dataset designer. You choose the Lakehouse by opening a warehouse in **Lakehouse**, where you pick a compute size and start a session.

A.2 How a run starts

A.2.1 From Run to results

The steps are the same everywhere you can start a run.

1. You press Run, a schedule fires, or an API client submits a job.
2. Groundzero checks that you may use the chosen compute size and the dataset or job.
3. It prepares the connections you selected for the run.
4. It records the run with a unique identifier and starts lineage tracking.
5. It starts your runtime with the compute size you chose, or reuses a warm session (see Sessions and warm starts).
6. Your code runs, and its status, output and logs appear as it goes.
7. For a dataset run, the first page of rows is returned to the caller. For a job, the run page follows status and logs until the run ends.

A.2.2 Datasets, jobs and notebooks compared

	Dataset run	ETL job run	Notebook
Starts from	Dataset designer, chart, dashboard	Job page, schedule, API	Notebook page
Output	A page of rows	Whatever the code writes, plus logs	Interactive
Session	Kept warm and shared (DuckDB, Spark)	Ends when the run ends	Stays up until stopped
Default page size	1000 rows	not applicable	not applicable

In a Spark or DuckDB notebook the Groundzero helper library is already imported (`sparkSession` and `executeJob` for Spark; `duckSession` for DuckDB), together with NumPy, pandas and boto3. Notebooks are saved to your notebook storage each time you save.

A.2.3 Sessions and warm starts

Starting a runtime takes a moment, so Groundzero keeps a **warm session** after a dataset run on DuckDB or Spark. Your next run starts faster if it arrives within a few minutes.

- A session handles one run at a time. A new run takes an idle session that matches its dataset, compute size and connections, and prefers the one you used last.
- If every matching session is busy, Groundzero starts another one. Runs never wait in a queue behind each other.
- A warm session stays available for about 280 seconds after a run and ends after about 300 seconds without a run. After that, the next run starts a new session.
- Lakehouse permissions follow the **user who runs**, not the session, so sharing a session never widens anyone's access.

- Do not keep state in memory between runs. Any run can start in a new session.

Jobs do not share sessions. Each job run gets its own session and ends it when the job finishes. A schedule can also set a stop time, at which the run ends.

A.2.4 Timeouts and retries

Setting	Value
Wait for a dataset run's output	300 seconds by default
Wait for a new session to become ready	60 seconds
Spark workers	1 minimum by default, 10 maximum

If a run request fails before it reaches the runtime, Groundzero retries it automatically. If it may already have reached the runtime, Groundzero does not send it again, so a run never executes twice by accident.

A.2.5 Compute sizes, workers and GPU size

A **compute size** is a named CPU and memory pair that your administrators define. The runtime control lists each as `Name | memory | cpu`. The size you choose is the CPU and memory your code gets. You need Describe permission on a compute size to use it.

For Spark:

- The compute size applies to the driver and to each worker.
- **Minimum workers** sets how many workers start. If you leave it empty, one starts. Workers are added, up to a maximum of 10, when average CPU or memory use passes 70 percent.
- Each executor gets about 65 percent of its worker's memory, and never more than the worker's memory minus 1 GiB.
- Spark starts with one executor and releases idle executors after 120 seconds.

The runtime control also shows a **GPU Size** field for every runtime except Standard.

CHOOSING MEMORY

DuckDB and the Standard runtime hold working data in the memory of one machine. Spark spreads it over workers and spills to disk. When a run stops because it ran out of memory, pick a larger compute size first, then reduce the data you read.

Compute

COMPUTE	MEMORY	VCPU
SMALL	4Gi	2
MEDIUM	8Gi	4
LARGE	16Gi	8
XLARGE	32Gi	16

Minimum Workers

-2+

▶

Query 1

×

+

1

Compute sizes are chosen from a table of name, memory and vCPU.

MEDIUM

2 workers

▼

▶

Query 1

×

+

Lakehouse

>

> Schedules

> Backups

1

A chosen compute size with its worker count.

A.3 What your code can use during a run

A.3.1 Run details

Every run gives your code a fixed set of values. You can read them from `os.environ` in notebooks and, where listed, in job code.

Name	Meaning
<code>USERNAME</code>	User who started the run
<code>USERGROUPS</code>	The user's groups, comma-separated
<code>SITE_NAME</code>	Site (tenant) name
<code>REGION</code>	Cloud region
<code>JOB_IDENTIFIER</code>	Id of this run
<code>SESSION_ID</code>	Id of the session
<code>WORKLOAD</code>	<code>ETL_JOB</code> or <code>NOTEBOOK</code>
<code>PAGE_NUMBER</code> , <code>PAGE_SIZE</code>	Paging of a dataset run

Do not rely on any other values or try to override these.

A.3.2 Job placeholders

Text in a job can contain `${NAME}` placeholders that the runtime fills in when the job runs.

Placeholder	Meaning
<code>\${STRTTM}</code>	Run start time
<code>\${ENDTM}</code>	Run end time
<code>\${CLNTNM}</code> , <code>\${SITENM}</code>	Client and site name
<code>\${PRJNM}</code> , <code>\${JOBNM}</code>	Project and job name
<code>\${STS}</code> , <code>\${ERR}</code>	Run status and error text
<code>\${RTE}</code> , <code>\${RGN}</code>	Runtime name and region
<code>\${USR}</code> , <code>\${USRGRPS}</code>	User and groups

The Standard runtime uses a smaller set; see the Standard chapter.

A.3.3 Imported models

A run can be given registered ML models. Groundzero finds each model's latest stored version and makes it available to your code.

A.4 Connections and secrets

A **connection** stores how to reach a source (host, database, credentials or role) under a name. Your code refers to the connection by name and never contains credentials.

- Groundzero prepares your connections when a run starts.
- Only the connections you selected for the job or dataset are available to its code.
- Lakehouse connections use short-lived access that Groundzero refreshes for you, so long-running sessions keep working.
- Do not print connection details or credentials to run logs. Logs are stored and shown on the run page.

NEVER HARD-CODE SECRETS

Put credentials in a connection. Do not paste keys or passwords into job code, parameters or notebooks.

A.5 The Lakehouse (Iceberg)

The Groundzero lakehouse stores tables in Apache Iceberg format on object storage, with one warehouse per site. The lakehouse console and its SQL editor run lakehouse SQL for you. Runtimes read and write the table data directly, with temporary access that applies your Groundzero roles and policies.

Runtime	Engine versions	How it reaches the lakehouse
Standard	DuckDB 1.5.5	Reads Iceberg tables (read-only)
DuckDB	DuckDB 1.5.5, iceberg extension	Reads through the helper library (read-only)
Spark	Spark 3.5.6, Iceberg 1.11.0	Reads and writes through the helper library
Lakehouse	Spark SQL 4.0.0, Iceberg 1.10.0	Your SQL session in the Lakehouse pages (see the Lakehouse guide)

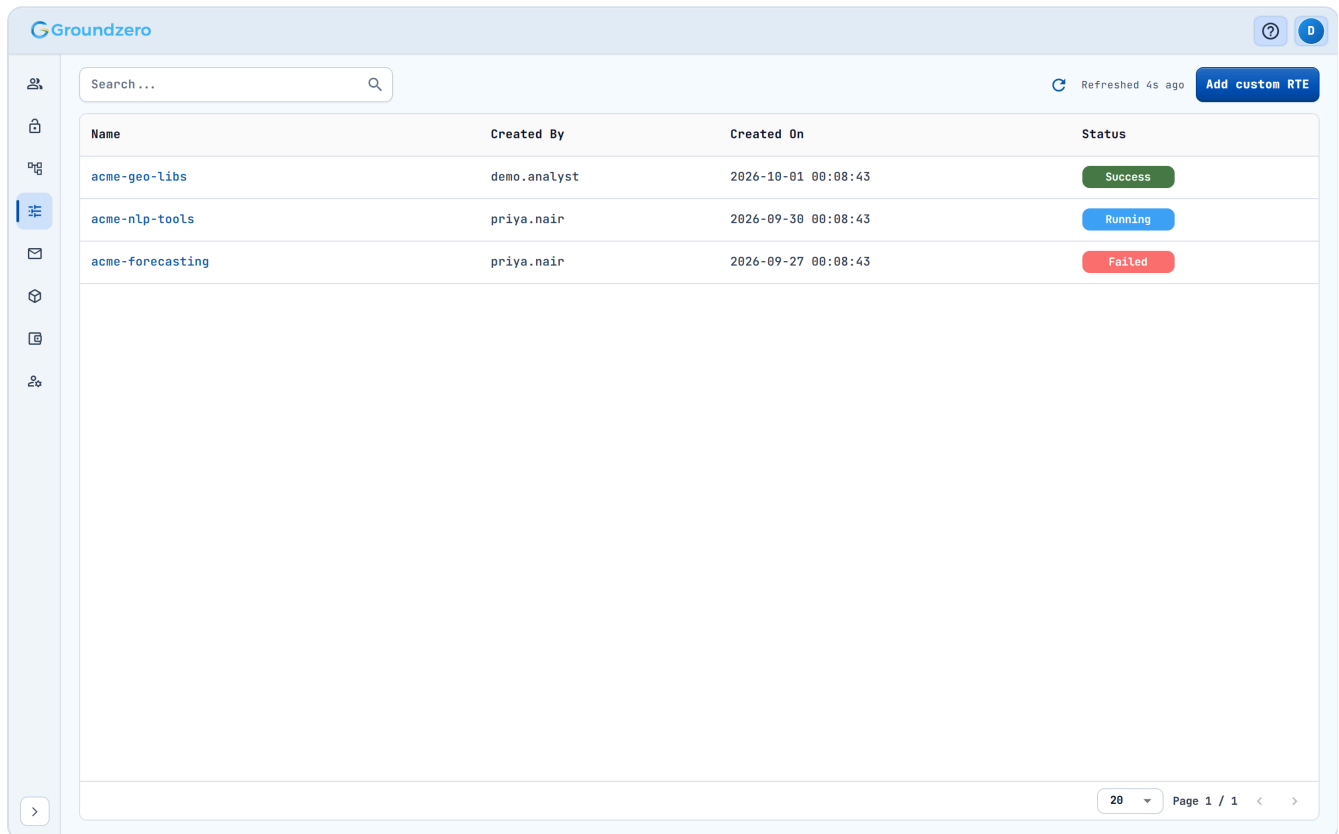
A.6 Custom runtimes

A custom runtime is an existing runtime plus your own pip packages. Administrators create it on the **Custom RTE** page.

1. Upload a `requirements.txt` and choose the **Base Image** (the runtime to extend) and a name.
2. Groundzero builds the runtime with the packages from `requirements.txt`.
3. The build status shows as QUEUED, IN PROGRESS, then SUCCESS or FAILURE.
4. When the build succeeds, the runtime appears in the runtime list.

Guidelines:

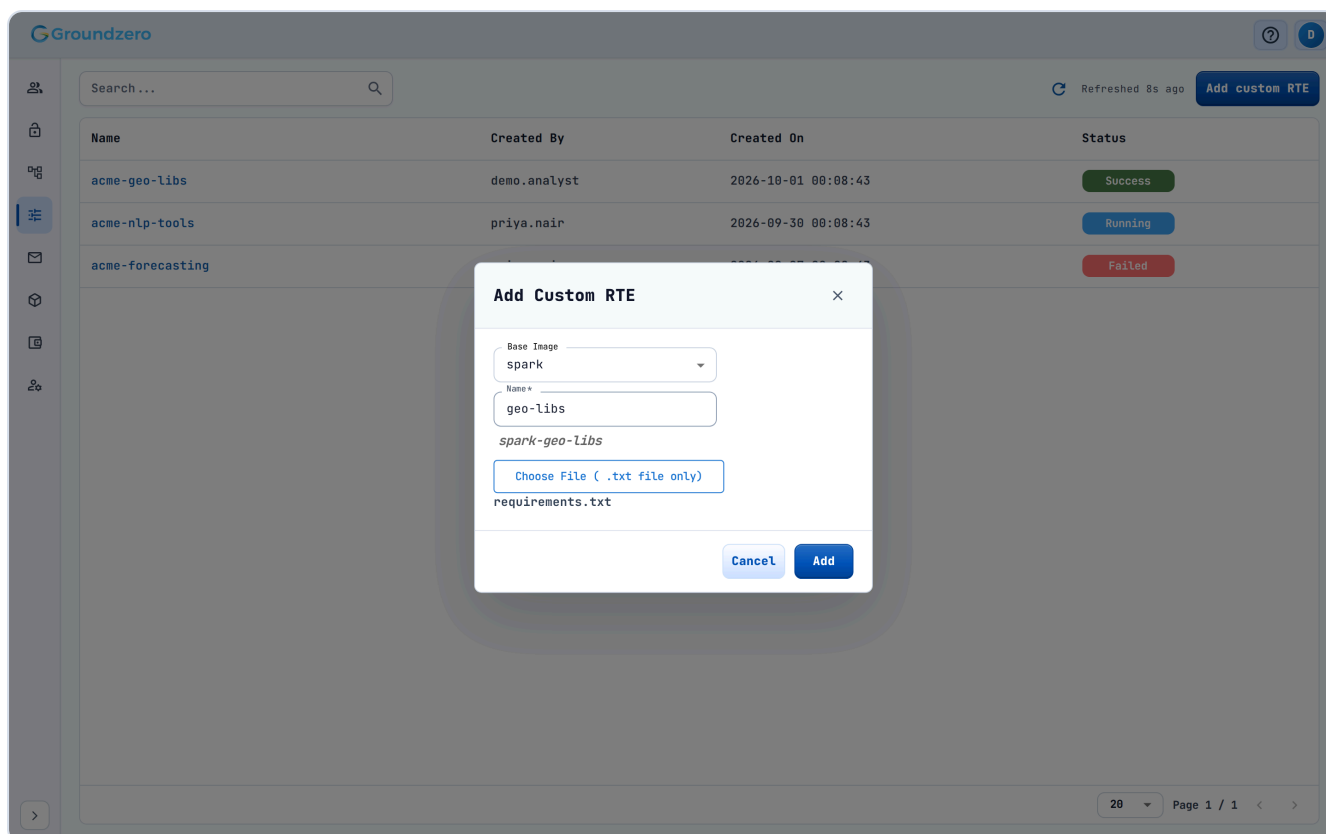
- Extend the runtime you actually run on: a Spark base for Spark jobs, a DuckDB base for DuckDB jobs.
- Pin versions. Do not change packages the base already includes (NumPy, pandas, scikit-learn) unless you need a different version.
- Pure-Python packages install most reliably. A package that needs extra system libraries may fail to build. Check the FAILURE status and the package's requirements.



The screenshot shows the Groundzero admin console interface. At the top, there's a search bar and a refresh button. Below is a table with columns: Name, Created By, Created On, and Status. The table lists three custom runtimes: 'acme-geo-lib' (Success), 'acme-nlp-tools' (Running), and 'acme-forecasting' (Failed). The status is indicated by colored buttons: green for Success, blue for Running, and red for Failed. At the bottom right, there's a pagination control showing 'Page 1 / 1'.

Name	Created By	Created On	Status
acme-geo-lib	demo.analyst	2026-10-01 00:08:43	Success
acme-nlp-tools	priya.nair	2026-09-30 00:08:43	Running
acme-forecasting	priya.nair	2026-09-27 00:08:43	Failed

The Custom RTE page in the admin console lists each custom runtime and its build status.



The screenshot displays the Groundzero Lakehouse Runtime interface. At the top, there is a search bar and a refresh button labeled 'Refreshed 8s ago'. A table lists custom RTEs with columns for Name, Created By, Created On, and Status. A modal window titled 'Add Custom RTE' is open, showing a form to add a new RTE. The form includes a 'Base Image' dropdown set to 'spark', a 'Name' input field containing 'geo-libs', and a 'Choose File (.txt file only)' button. Below the button, the filename 'requirements.txt' is displayed. The modal also has 'Cancel' and 'Add' buttons.

Name	Created By	Created On	Status
acme-geo-libs	demo.analyst	2026-10-01 00:08:43	Success
acme-nlp-tools	priya.nair	2026-09-30 00:08:43	Running
acme-forecasting			Failed

Add Custom RTE

Base Image: spark

Name: geo-libs

spark-geo-libs

Choose File (.txt file only)

requirements.txt

Cancel Add

Add custom RTE: choose a base image, name the runtime and upload a requirements file.

A.7 ML variants

Spark and DuckDB each come as a base runtime and three ML variants. A variant is the base plus extra libraries.

Variant	Adds
Base	Spark or DuckDB, Groundzero helper libraries, scikit-learn 1.5.2, CPU PyTorch 2.14.0 and torchvision 0.29.0, document parsing
sklearn	OpenCV (headless), XGBoost, LightGBM, CatBoost
pytorch	sklearn extras, PyTorch 2.5.1 and torchvision 0.20.1 (GPU build), SHAP, LIME, Albumentations, Hugging Face Hub and Transformers, simsimd, fastai
tensorflow	TensorFlow, OpenCV, XGBoost, LightGBM, CatBoost, SHAP, LIME, NLTK, spaCy, h5py

Notes:

- The pytorch variant uses the GPU build of PyTorch, so GPU-sized runs can train on GPUs.
- Variants appear in the runtime list with names such as `gz-rte-duckdb-sklearn` and a version tag such as `1.0.0`.
- The Standard runtime has no ML variants.

A.8 Jobs: lifecycle, status and logs

A.8.1 Statuses

Status	Meaning
Queued	Accepted, waiting to start
Running	Your code is executing
Success	Finished without error
Failed	The code or Groundzero reported an error
Aborted	Stopped before it could finish

Queued and Running are active. Success, Failed and Aborted are final.

A.8.2 Logs

Run logs stream to the run page while the run works.

- Lines never repeat when the page reloads the stream.
- If a run stops without reporting a final state, Groundzero marks it Failed rather than leaving it Running.
- If the page is asked for logs too often, it retries after a short delay.

Print and logging output from your code appears in the run log as the run works.

A.8.3 Scheduling

A schedule names the job, its runtime and its compute size. It may also set a start and stop time.

A.9 Glossary

Term	Meaning
Runtime (RTE)	The environment and engine your code runs in
Compute size	A named CPU and memory pair for a runtime
Session	A running runtime that can accept runs
Warm session	A dataset session kept ready for a few minutes and shared between runs
Connection	A named, stored way of reaching a source
Dataset	A saved query (and code) that returns rows
Prep	The pandas code that reshapes a Standard dataset's rows
Lakehouse	Groundzero's Iceberg tables on object storage
Warehouse	A site's lakehouse storage location
Custom runtime	A runtime extended with your pip packages
ML variant	A runtime with machine-learning libraries added
Lineage	The record of which assets a run read and wrote