



COMPLETE EDITION

Groundzero Runtime Environments

Developer documentation for the Spark, DuckDB, Standard and Lakehouse runtimes.

Developer documentation

Edition 2026-10-02

Runtime environments: Spark, DuckDB, Standard, Lakehouse

Contents

Part 1 - Choosing a Runtime and Shared Concepts

1.1	Choosing a runtime	9
1.1.1	At a glance	10
1.1.2	Decision guide	10
1.1.3	Where the choice is made	11
1.2	How a run starts	12
1.2.1	From Run to results	12
1.2.2	Datasets, jobs and notebooks compared	12
1.2.3	Sessions and warm starts	12
1.2.4	Timeouts and retries	13
1.2.5	Compute sizes, workers and GPU size	13
1.3	What your code can use during a run	15
1.3.1	Run details	15
1.3.2	Job placeholders	15
1.3.3	Imported models	15
1.4	Connections and secrets	16
1.5	The lakehouse (Iceberg)	17
1.6	Custom runtimes	18
1.7	ML variants	20
1.8	Jobs: lifecycle, status and logs	21
1.8.1	Statuses	21
1.8.2	Logs	21
1.8.3	Scheduling	21
1.9	Glossary	22

Part 2 - Spark Runtime

2.1	Overview	23
2.1.1	What the Spark runtime gives you	24
2.1.2	Choose the Spark runtime	24
2.1.3	How a run works	25
2.1.4	Versions	26
2.1.5	What changed recently	26
2.1.6	Machine-learning variants	27
2.2	Quickstart	28
2.2.1	Before you begin	28
2.2.2	Step 1: Create the job	28
2.2.3	Step 2: Run and read the result	29
2.2.4	Step 3: Try it in a notebook	30
2.2.5	Step 4: Schedule it	30
2.3	How-to guides	34

2.3.1	Read from a connection	34
2.3.2	Query and write Lakehouse (Iceberg) tables	35
2.3.3	Work with outputs and previews	37
2.3.4	Use parameters and placeholders	38
2.3.5	Chain jobs	39
2.3.6	Schedule through ETL jobs	39
2.3.7	Install and use extra packages	39
2.3.8	Tune performance	40
2.3.9	Log from your job	41
2.3.10	Check data quality	42
2.3.11	Parse PDFs and run Agents over documents	42
2.4	API reference	44
2.4.1	Conventions	44
2.4.2	Session	44
2.4.3	Lakehouse (Iceberg) methods	45
2.4.4	Object storage and files	49
2.4.5	Relational databases and warehouses	50
2.4.6	LLM methods	51
2.4.7	Data quality	53
2.4.8	PDF extraction	55
2.4.9	Agents	56
2.4.10	Jobs, mail and clients	58
2.4.11	Placeholders and outputs	60
2.4.12	Connector method index	61
2.5	Installed packages	68
2.5.1	Python	68
2.5.2	Data, SQL and cloud libraries	68
2.5.3	LLM and AI libraries	68
2.5.4	Document extraction	69
2.5.5	Notebook and plotting libraries	69
2.5.6	JVM libraries	69
2.5.7	System packages	69
2.6	Limits and troubleshooting	70
2.6.1	Limits and quotas	70
2.6.2	Restrictions in job scripts	70
2.6.3	Things to know	71
2.6.4	Troubleshooting	71
2.6.5	FAQ	73
2.7	Examples gallery	74
2.7.1	Daily revenue rollup in the lakehouse	74
2.7.2	Incremental load from PostgreSQL to Parquet	74
2.7.3	Deduplicate and enrich with window functions	75
2.7.4	Stream Kafka events into an Iceberg table	76
2.7.5	Classify support tickets with an LLM	77
2.7.6	Train and apply a scikit-learn model	78
2.7.7	Load a warehouse query into the lakehouse in bulk	79

2.7.8	Extract structured data from a folder of PDFs with an Agent	80
2.7.9	Quality gate with notification	80
2.7.10	Orchestrate sub-jobs	81

Part 3 - DuckDB Runtime

3.1	Overview	82
3.1.1	What the DuckDB runtime gives you	83
3.1.2	Choose the DuckDB runtime	83
3.1.3	How a run works	84
3.1.4	Versions	84
3.1.5	What changed recently	85
3.1.6	Machine-learning variants	85
3.2	Quickstart	86
3.2.1	Your first job	86
3.2.2	Your first dataset	88
3.2.3	Your first notebook	88
3.3	How-to guides	91
3.3.1	Create and configure a session	91
3.3.2	Read from connections	91
3.3.3	Query across sources	93
3.3.4	Query the Groundzero Lakehouse	94
3.3.5	Write results	96
3.3.6	Return output and previews	97
3.3.7	Use parameters and placeholders	98
3.3.8	Run jobs and schedule them	99
3.3.9	Use extra packages	100
3.3.10	Work with files	100
3.3.11	Tune performance	101
3.3.12	Log and debug	102
3.3.13	Handle secrets	102
3.3.14	Extract text from PDFs	103
3.3.15	Run Groundzero Agents over documents	103
3.3.16	Check data quality	104
3.4	API reference	105
3.4.1	Conventions	105
3.4.2	Session	105
3.4.3	Groundzero Lakehouse	108
3.4.4	Object storage	110
3.4.5	SQL databases	111
3.4.6	Cloud warehouses	112
3.4.7	Outputs and run context	113
3.4.8	Clients and connection details	114
3.4.9	Connectors	115
3.4.10	LLM providers	121
3.4.11	MCP servers	123
3.4.12	PDF extraction	124

3.4.13	Groundzero Agents	126
3.4.14	Data quality and record linkage	127
3.4.15	Talend migration helpers	127
3.5	Installed packages	129
3.5.1	Python and system	129
3.5.2	Connector and data libraries	129
3.5.3	LLM and AI libraries	131
3.5.4	Document extraction	131
3.5.5	DuckDB extensions	132
3.5.6	Packages added by the ML variants	132
3.6	Limits and troubleshooting	133
3.6.1	Limits and quotas	133
3.6.2	Things to know	133
3.6.3	Troubleshooting	134
3.6.4	FAQ	136
3.7	Examples gallery	137
3.7.1	Example 1: Partitioned daily export	137
3.7.2	Example 2: Rolling averages with window functions	137
3.7.3	Example 3: Flatten JSON events	138
3.7.4	Example 4: Churn scoring with scikit-learn	139
3.7.5	Example 5: Deduplicate and validate before loading	139
3.7.6	Example 6: Customer 360 across lakehouse, PostgreSQL and S3	140
3.7.7	Example 7: Incremental load from Snowflake	141
3.7.8	Example 8: Classify support tickets with an LLM	142
3.7.9	Example 9: Ingest a REST API	143
3.7.10	Example 10: Geospatial distance query	144
3.7.11	Example 11: Extract policy fields from PDFs with an Agent	145
3.7.12	Example 12: A two-step pipeline with executeJob	145

Part 4 - Standard Runtime

4.1	Overview	146
4.1.1	What Standard is	147
4.1.2	How it runs	147
4.1.3	Versions	147
4.1.4	Choose the Standard runtime	148
4.1.5	The Python (pandas) job runtime	148
4.1.6	The base runtime and custom runtimes	148
4.2	Quickstart	149
4.2.1	1. Pick a connection	149
4.2.2	2. Write the source query	149
4.2.3	3. Add Prep code	149
4.2.4	4. Run and save	150
4.2.5	5. Read it from code	150
4.3	How-to guides	153
4.3.1	Read from a connection	153

4.3.2	Query lakehouse tables	153
4.3.3	Work with datasets, outputs and previews	154
4.3.4	Use parameters and placeholders	154
4.3.5	Filter without changing the query text	155
4.3.6	Reshape results with Prep code	155
4.3.7	Schedule with ETL jobs	156
4.3.8	Install extra packages (custom runtime)	156
4.3.9	Tune performance	156
4.3.10	Logging and errors	156
4.4	API reference	157
4.4.1	Prep code contract	157
4.4.2	Data request	157
4.4.3	Data response	158
4.4.4	Structured filters	158
4.4.5	Error messages	159
4.5	Installed packages	160
4.5.1	Python (pandas) job runtime and base runtime	160
4.5.2	Standard query engine	160
4.6	Limits and troubleshooting	161
4.6.1	Limits and quotas	161
4.6.2	Known issues	161
4.6.3	Troubleshooting	161
4.6.4	FAQ	162
4.7	Examples gallery	163
4.7.1	1. Shipped orders in the last quarter	163
4.7.2	2. Revenue by region with a Prep rollout	163
4.7.3	3. Monthly trend	163
4.7.4	4. Largest order per region	164
4.7.5	5. Pivot status counts by region	164
4.7.6	6. Orders owned by the signed-in user	164
4.7.7	7. A single row of headline numbers	164
4.7.8	8. Customers with their segment (join in SQL)	165
4.7.9	9. Defensive Prep for empty results	165
4.7.10	10. Test Prep locally before you save it	165

Part 5 - Lakehouse Runtime

5.1	Overview	166
5.1.1	What the Lakehouse is	167
5.1.2	Warehouses, databases and tables	167
5.1.3	Choose the Lakehouse	167
5.1.4	How a query runs	168
5.1.5	Versions	169
5.2	Quickstart	170
5.2.1	1. Open the Lakehouse and pick a warehouse	170
5.2.2	2. Choose a compute and start a session	170

5.2.3	3. Create a database and a table	173
5.2.4	4. Insert rows	174
5.2.5	5. Query the table	174
5.2.6	6. Save and export	175
5.2.7	7. Stop the session	175
5.3	How-to guides	176
5.3.1	Browse databases, tables and columns	176
5.3.2	Write queries	177
5.3.3	Create and alter Iceberg tables	180
5.3.4	Insert, update, delete and merge	182
5.3.5	Time travel and snapshots	183
5.3.6	Metadata tables	184
5.3.7	Table properties	185
5.3.8	Maintenance	186
5.3.9	Long-running queries and stopping a query	188
5.3.10	Results, paging and exports	189
5.3.11	Saved queries	192
5.3.12	Governed tables, data policies and table access	193
5.3.13	Read lakehouse tables from jobs and datasets	196
5.3.14	Backups, schedules and restore	196
5.3.15	Audit logs	198
5.4	SQL reference	199
5.4.1	Statements and the permission each needs	199
5.4.2	Queries	200
5.4.3	Time travel	200
5.4.4	Creating and changing tables	201
5.4.5	Changing data	201
5.4.6	Databases and information	201
5.4.7	Table procedures	202
5.4.8	Blocked or restricted statements	202
5.4.9	Behaviour of SQL in the Lakehouse	203
5.5	Limits and behaviour	204
5.5.1	Sessions	204
5.5.2	Queries and results	204
5.5.3	Exports	204
5.5.4	Other values	205
5.6	Troubleshooting	206
5.7	Examples gallery	209
5.7.1	Example 1: Revenue by channel and month	209
5.7.2	Example 2: Top three products in each category	209
5.7.3	Example 3: Running total and seven-day average	210
5.7.4	Example 4: Customers who have never ordered	210
5.7.5	Example 5: Low-stock list	210
5.7.6	Example 6: Upsert customer changes with MERGE	211
5.7.7	Example 7: Correct statuses after returns	211

5.7.8	Example 8: Build a partitioned daily rollup	211
5.7.9	Example 9: Compare today with yesterday using time travel	212
5.7.10	Example 10: Inspect a table's snapshots and files	212
5.7.11	Example 11: Pivot channel revenue into columns	212
5.7.12	Example 12: Evolve a table without rewriting it	212
5.7.13	Example 13: Tags stored as an array	213
5.7.14	Example 14: Safe division and clean text	213
5.7.15	Example 15: Clean up a busy table	213
5.7.16	Example 16: Find slow statements (administrators)	213

PART 1

Choosing a Runtime and Shared Concepts

Compare the runtimes and learn the concepts they share.

- 1.1 Choosing a runtime
- 1.2 How a run starts
- 1.3 What your code can use during a run
- 1.4 Connections and secrets
- 1.5 The lakehouse (Iceberg)
- 1.6 Custom runtimes
- 1.7 ML variants
- 1.8 Jobs: lifecycle, status and logs
- 1.9 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.

1.1 Choosing a runtime

1.1.1 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

1.1.2 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.

1.1.3 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.

1.2 How a run starts

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

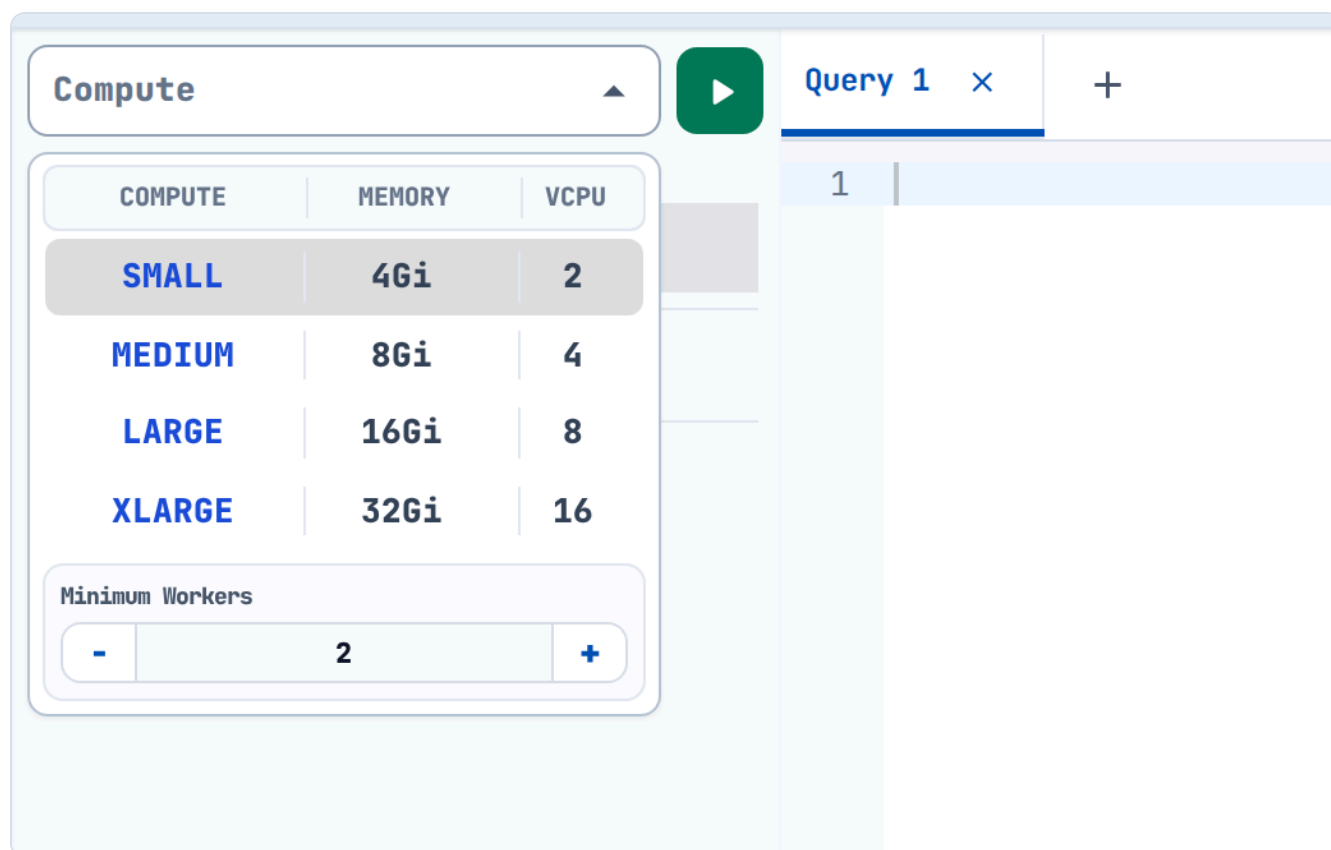
For Spark:

- 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

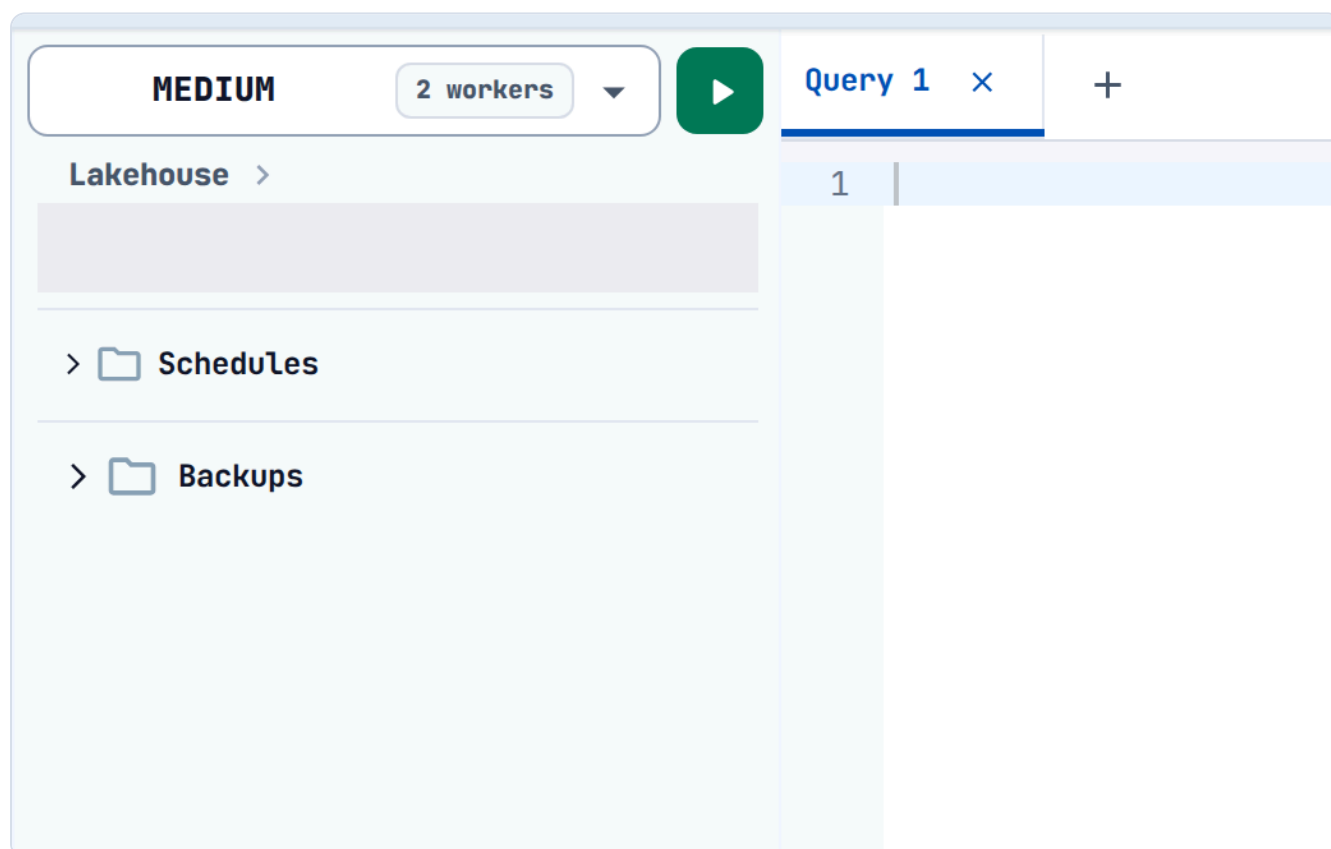
Minimum Workers

- 2 +

Query 1 × +

1

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



MEDIUM 2 workers

Lakehouse >

> Schedules

> Backups

Query 1 × +

1

A chosen compute size with its worker count.

1.3 What your code can use during a run

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.4 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.

1.5 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)

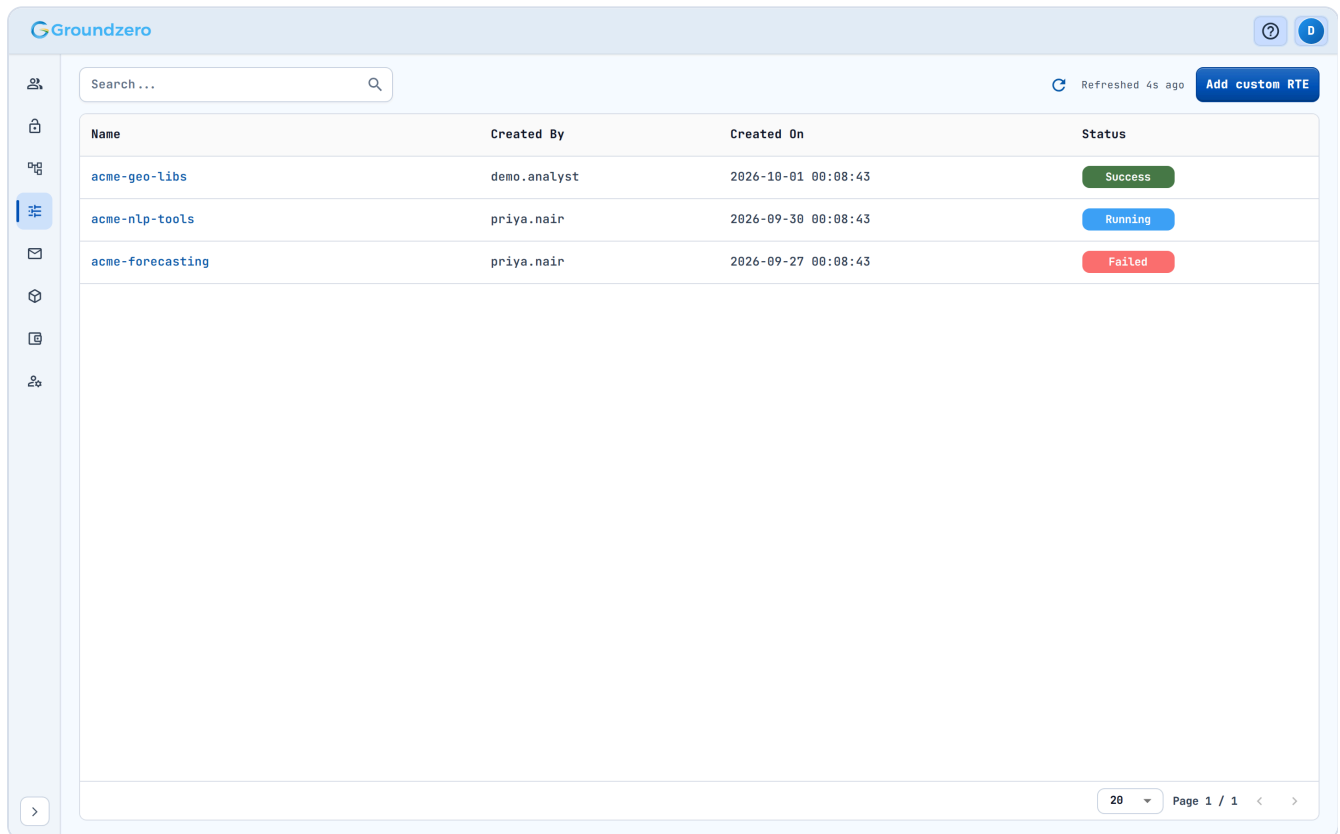
1.6 Custom runtimes

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:

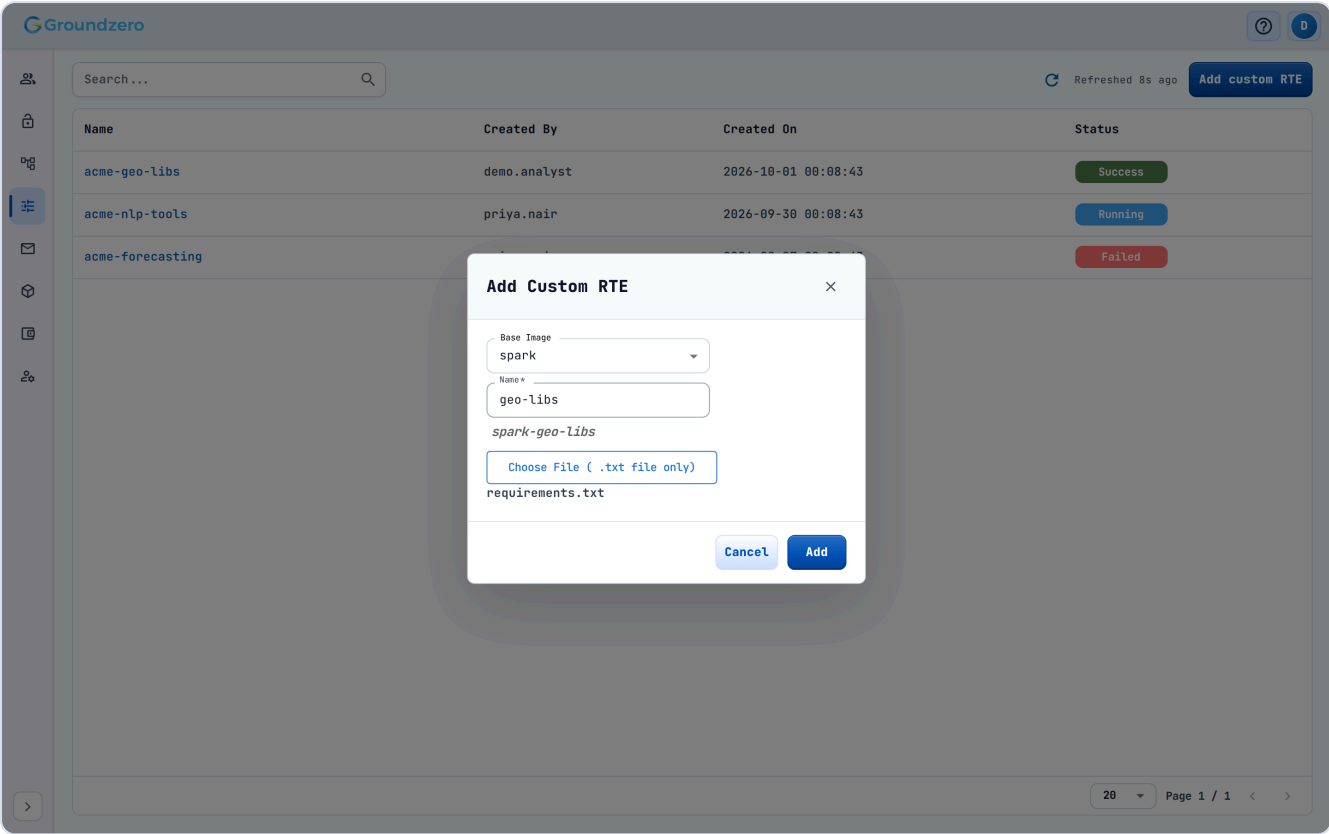
- Extend the runtime you actually run on: a Spark base for Spark jobs, a DuckDB base for DuckDB jobs.
- Pin versions. Do not change packages the base already includes (NumPy, pandas, scikit-learn) unless you need a different version.
- Pure-Python packages install most reliably. A package that needs 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 shows the Groundzero web interface. A table lists runtime environments with columns: Name, Created By, Created On, and Status. The table contains three rows: 'acme-geo-libs' (Success), 'acme-nlp-tools' (Running), and 'acme-forecasting' (Failed). An 'Add custom RTE' button is in the top right. A modal dialog titled 'Add Custom RTE' is open in the center. It has a 'Base Image' dropdown set to 'spark', a 'Name' input field with 'geo-libs', and a preview of 'spark-geo-libs'. Below is a 'Choose File (.txt file only)' button and a file named 'requirements.txt'. 'Cancel' and 'Add' buttons are at the bottom right of the dialog.

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.

1.7 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.

1.8 Jobs: lifecycle, status and logs

1.8.1 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.

1.8.2 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.

1.8.3 Scheduling

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

1.9 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

PART 2

Spark Runtime

Distributed processing with Apache Spark and Iceberg on Groundzero.

- [2.1 Overview](#)
- [2.2 Quickstart](#)
- [2.3 How-to guides](#)
- [2.4 API reference](#)
- [2.5 Installed packages](#)
- [2.6 Limits and troubleshooting](#)
- [2.7 Examples gallery](#)

The Groundzero Spark runtime runs Python jobs, dataset queries and notebooks on Apache Spark. A job is a Python script. It gets one ready-made `sparkSession`, a set of governed connectors, and a place to leave its result (`outputDF`). Each run starts its own Spark cluster with a driver and workers. You choose the size in Compute, so the same script runs on a small sample or on a large table.

This guide shows how to run your first job, read and write data, use the Groundzero helper functions, and handle the limits and errors you may meet. It ends with a gallery of working examples.

2.1 Overview

2.1.1 What the Spark runtime gives you

- Apache Spark with Python 3.12 (PySpark, pandas, NumPy, scikit-learn, DuckDB and more).
- The Groundzero helper library (`gZ`). It creates your Spark session and adds every connector as a method on that session.
- More than 50 connector families: databases, warehouses, object stores, streams, SaaS APIs, vector stores and LLM providers.
- Apache Iceberg support for Groundzero Lakehouse tables, with per-table permissions, audit logging and lineage.
- A paged preview of your result in the UI, and logs streamed while the job runs.
- JupyterLab notebooks on the same runtime.

You do not install or deploy anything. Pick the Spark runtime and a compute size when you create a job, dataset or notebook.

2.1.2 Choose the Spark runtime

Groundzero offers three runtime families. Use the table to decide which one fits your workload.

	Spark	DuckDB	Standard
Engine	Distributed Apache Spark	Single-node DuckDB	Managed query engine
Language	Python (PySpark)	Python (DuckDB)	SQL plus pandas Prep
Scales by	Adding workers	Larger compute size	Managed by Groundzero
Best for	Large joins, wide ETL, streaming, ML	Fast single-node analysis	Simple governed datasets
Chosen for	Jobs, notebooks, datasets	Jobs, notebooks, datasets	Datasets only
Compute picker	Yes, with workers	Yes	None

Choose Spark when:

- Your data does not fit comfortably in the memory of one machine.
- You need Structured Streaming (Kafka, Kinesis, Pulsar, Iceberg streaming).
- You need the broadest connector coverage, including bulk loads from Snowflake and Redshift straight into the lakehouse.
- You train or score models over a DataFrame, or call an LLM for every row.

Choose another runtime when the data is small (a few gigabytes) and start-up time matters more than scale. Spark starts a driver and at least one worker before your code runs, so a single-node runtime starts faster.

WHERE THE OTHER RUNTIMES ARE DOCUMENTED

The DuckDB and Standard runtimes have their own guides. This guide only covers Spark.

2.1.3 How a run works

Compute size and workers

You choose a named compute size in the runtime picker. Each size sets the CPU and memory of the driver and of every worker.

Setting	Value	Notes
Minimum workers	Default 1	Set in the runtime picker
Maximum workers	10	Fixed per session
Scale-up trigger	70 percent	Average CPU or memory use across workers
Executor memory	65 percent of worker memory	Never more than worker memory minus 1 GiB

Workers are added automatically when CPU or memory use passes 70 percent, up to a maximum of 10 workers. Workers added during a heavy workload stay until the session ends. Spark releases idle executors after 120 seconds. To get more memory per executor, choose a larger compute size.

What happens when a job runs

1. Groundzero loads your script.
2. It replaces placeholders such as `{STARTTIME}` and `{SITEID}` in the code (see Placeholders in the API reference).
3. It runs the script. Standard output and standard error stream to the job log.
4. If the script defined a variable named `outputDF`, Groundzero reads up to 100,000 rows of it for the preview and shows the first page. Later pages are served without recomputing the result.
5. Groundzero records the final status and lineage, and sends the completion notification if one is configured.

Notebooks skip these steps. They start JupyterLab with a kernel that has already imported `sparkSession`, `pandas`, `numpy`, `pyspark` and `executeJob`.

Sessions and datasets

A dataset that uses the Spark runtime runs its query in a Spark session that is shared between users of the same dataset. One session handles one run at a time. When every session for the dataset is busy, Groundzero starts another instead of queueing your run. A session stays available for 280 seconds after its last run and shuts down after 300 seconds of inactivity.

Lakehouse permissions are checked as the user who starts each run, and the session clears its views when the user changes. One user never sees another user's governed rows.

KEEP A NOTEBOOK OR DATASET WARM

If your next run starts within a few minutes, it reuses the running session and skips the Spark start-up. After the idle window, the next run starts a new session.

2.1.4 Versions

Component	Version
Python	3.12
Apache Spark (and PySpark)	3.5.6
Java	OpenJDK 17
Apache Iceberg	1.11.0
Delta Lake	3.3.2
Apache Hudi	1.1.1
NumPy	2.3.5
pandas	2.2 or later, below 3
DuckDB (Python)	1.5.5
scikit-learn	1.5.2
PyTorch (CPU)	2.14.0
MLflow	2.20.4
Qualipilot (data quality)	3.5.0
Docling (PDF extraction)	2.116.0

Connector driver versions you may need when you read vendor documentation:

Connector	Version
PostgreSQL JDBC	42.7.11
MySQL	8.4.0
SQL Server JDBC	13.4.0
Oracle JDBC	23.6.0.24.10
Snowflake Spark / JDBC	3.1.7 / 3.28.0
MongoDB Spark	10.5.0
Kafka clients	3.4.1
Elasticsearch Spark	9.4.1
BigQuery Spark	0.44.0

2.1.5 What changed recently

The Spark runtime is updated continuously and has no numbered releases. Recent changes that affect your code, newest first:

- Bulk loads from Snowflake and Redshift into lakehouse tables (`bulkLoadGzLakehouseFromSnowflake` , `bulkLoadGzLakehouseFromRedshift`).
- Lineage is recorded for every connector read and write.
- Dataset previews of wide nested columns no longer run out of memory.
- Faster start-up for Spark sessions.
- The `*Native` REST methods are now named `*Arrow` .
- Qualipilot 3.5.0 data quality, PDF extraction with OCR, Agent calls over documents, an xAI connector and LLM completion across providers.

2.1.6 Machine-learning variants

Three machine-learning variants add libraries for model work. They keep every Spark and Groundzero feature. Choose a variant in the runtime picker when you need it.

Variant	Adds
scikit-learn	<code>opencv-python-headless</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code>
PyTorch	PyTorch 2.5.1 and torchvision 0.20.1 (CUDA 12.1 build), <code>shap</code> , <code>lime</code> , <code>albumentations</code> , <code>transformers</code> , <code>huggingface_hub</code> , <code>fastai</code> , <code>simsimd</code> , plus the scikit-learn set
TensorFlow	<code>tensorflow</code> , <code>nlTK</code> , <code>spacy</code> , <code>h5py</code> , <code>shap</code> , <code>lime</code> , <code>scipy</code> , <code>opencv</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code>

All variants include scikit-learn 1.5.2, NumPy and pandas. The PyTorch variant uses the CUDA 12.1 build of PyTorch, so it can use an NVIDIA GPU when you set a GPU size.

CHECK THE EXACT VERSIONS

The TensorFlow variant installs its packages without version pins. To see a version, run `import tensorflow as tf; print(tf.__version__)` (or the equivalent for any package) in a notebook on the variant.

For LLM fine-tuning libraries (`unsloth` , `trl` , `peft` , `bitsandbytes`), use a GPU variant and install them with `%pip` in a notebook (see Install and use extra packages).

2.2 Quickstart

This quickstart creates a Spark job that builds a small DataFrame, aggregates it, writes the result to object storage and exposes it as the job output. It uses fictional data so you can run it as soon as you have an S3 connection.

2.2.1 Before you begin

You need:

- A project in which you can create ETL jobs.
- Permission to use a compute size (the runtime picker lists the sizes you may use).
- An S3 connection named `acme-s3` (any object-store connection works with the matching `read / write` method; this example uses S3).

2.2.2 Step 1: Create the job

1. Open the project and create a new ETL job.
2. In the editor toolbar, open the runtime picker, choose a Spark runtime and a compute size.
3. Paste the code below.

PYTHON

```

from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("QuickstartOrders", "WARN")

orders = spark.createDataFrame(
    [
        (1001, "emea", "widget", 3, 19.99),
        (1002, "emea", "gadget", 1, 49.50),
        (1003, "apac", "widget", 10, 19.99),
        (1004, "amer", "gizmo", 2, 120.00),
        (1005, "apac", "gadget", 4, 49.50),
    ],
    ["order_id", "region", "product", "quantity", "unit_price"],
)

revenue = (
    orders.withColumn("revenue", F.col("quantity") * F.col("unit_price"))
    .groupBy("region")
    .agg(
        F.count("*").alias("orders"),
        F.round(F.sum("revenue"), 2).alias("revenue"),
    )
    .orderBy(F.desc("revenue"))
)

spark.writeS3(
    df=revenue,
    connName="acme-s3",
    filePath="quickstart/revenue_by_region",
    format="parquet",
    mode="overwrite",
)

outputDF = revenue

```

2.2.3 Step 2: Run and read the result

Select **Run**. The log panel streams the job log: Spark start-up messages first, then anything your script prints. When the job completes, the preview shows the rows of `outputDF` :

TEXT

region	orders	revenue
apac	2	397.9
amer	1	240.0
emea	2	109.47

The same `revenue_by_region` data now exists as Parquet files under `quickstart/revenue_by_region` in the bucket of the `acme-s3` connection.

WHY OUTPUTDF ?

The runtime looks for a top-level variable named `outputDF` after your script finishes. A Spark DataFrame, a pandas DataFrame or a list of dictionaries can all be used. If the variable is missing, the job still succeeds, but there is no preview.

2.2.4 Step 3: Try it in a notebook

Create a notebook, choose the same runtime, and run the first cell. The kernel has already imported `sparkSession`, `pandas as pd`, `numpy as np`, `pyspark` and `executeJob`:

PYTHON

```
spark = sparkSession("NotebookScratch", "WARN")
df = spark.createDataFrame([(1, "a"), (2, "b")], ["id", "label"])
df.show()
```

2.2.5 Step 4: Schedule it

From the job page, add a schedule. The scheduler dialog has a single **RTE** field that opens the same runtime and compute picker. Pick a runtime, a compute size, and save. Scheduled runs use the runtime and compute stored with the schedule.

In the UI

The runtime is chosen in the **runtime picker**, a compact control that shows the selected runtime name and, next to it, a small badge with the compute name. Selecting it opens a panel with these fields:

Field	Meaning
Runtime	The runtime to use: <code>Standard</code> (datasets only) or one of the registered runtimes
Compute	A named size; each entry lists its name, memory and CPU
Minimum Workers	Shown for Spark runtimes; the starting worker count, never below 1
GPU Size	Shown for every runtime except <code>Standard</code> ; defaults to 0

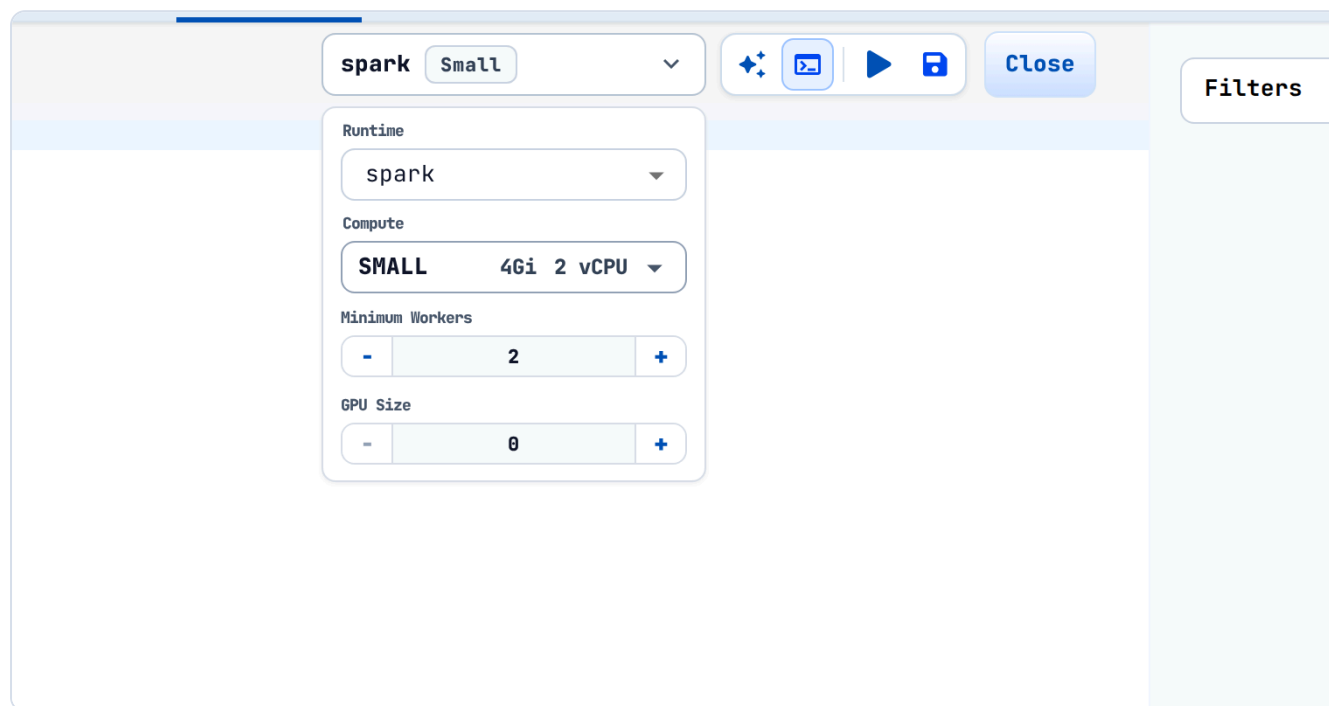
Where you find it:

- **ETL job editor.** The picker sits in the editor toolbar. `Standard` is not offered for jobs, so you choose a registered runtime directly.
- **Notebooks.** The picker is in the notebook header and also omits `Standard`. When the notebook's session is disconnected, the control turns red and shows "Session disconnected".
- **Datasets.** The dataset designer lists `Standard` first, followed by the registered runtimes. Choosing `Standard` hides the Compute field; any other runtime requires a compute size before **Save** is enabled.
- **Flow builder.** The picker sits in the top bar of the flow builder and, like the job editor, lists the registered runtimes.
- **Scheduler.** The scheduler dialog shows one **RTE** text field that displays the chosen runtime and compute together. Selecting it opens a menu with Runtime and Compute drop-downs and a **Save** button.

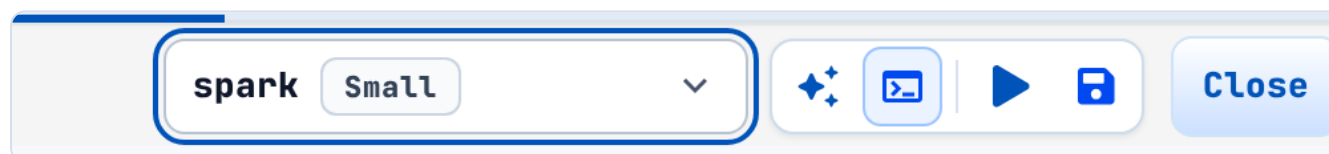
The Runtime list is loaded from your site, so it shows only the runtimes your administrator has registered. If the list is empty, the picker shows "No RTEs Available".

COMPUTE SIZES YOU CANNOT SEE

If the Compute field reads "Insufficient Permission", your user has no access to any compute size. Ask a site administrator to grant you access.

The image shows a web interface for selecting a runtime environment. At the top, there's a header bar with 'spark' and a 'Small' badge, followed by icons for a list, a play button, and a save icon, and a 'Close' button. Below this, a 'Runtime' dropdown menu is open, showing 'spark' selected. Underneath, the 'Compute' section displays 'SMALL' and '4Gi 2 vCPU'. The 'Minimum Workers' section has a value of '2' with minus and plus buttons. The 'GPU Size' section has a value of '0' with minus and plus buttons. On the right side of the interface, there's a 'Filters' button.

Runtime picker with Spark selected: Runtime, Compute, Minimum Workers and GPU Size.

This image shows the same interface as the previous one, but the 'Runtime' dropdown menu is now closed. The header bar now only shows 'spark' and the 'Small' badge, along with the same set of icons and the 'Close' button. The 'Filters' button remains on the right.

The closed picker shows the runtime name and a compute badge.

The screenshot shows the Groundzero job editor for a pipeline named 'inventory_archive'. The 'Code' tab is selected, displaying a Spark script that reads data from S3 and writes it to GCS. The 'Messages' tab is currently empty, showing a placeholder message: 'Messages will be displayed here.'

```

1 from GZ.SparkUtils import sparkSession
2
3 # Configuration
4 S3_CONNECTION_NAME = "AcmeWarehouseFiles"
5 S3_FILE_PATH = "inventory/2026-09-30.csv"
6 FILE_FORMAT = "csv"
7 GCS_CONNECTION_NAME = "AcmeArchiveGCS"
8 GCS_TARGET_PATH = "archive/inventory/2026-09-30"
9 WRITE_MODE = "overwrite"
10
11 # Initialize Spark session
12 spark = sparkSession("InventoryArchive", "WARN")
13
14 # Read from S3
15 df = spark.readS3(
16     connName=S3_CONNECTION_NAME,
17     fileName=S3_FILE_PATH,
18     format=FILE_FORMAT,
19     options={"header": "true"},
20 )
21
22 print(f"Read {df.count()} rows from {S3_FILE_PATH}")
23
24 # Write to GCS
25 spark.writeGcs(
26     df=df,
27     connName=GCS_CONNECTION_NAME,
28     fileName=GCS_TARGET_PATH,
29     format=FILE_FORMAT,
30 )
  
```

The job editor with the Spark runtime and Small compute selected.

The screenshot shows the same job editor after a successful run. The 'Messages' tab now displays a 'Job completed successfully' message and a table of timestamped logs. Two success notifications are visible in the bottom right corner: 'Job saved successfully' and 'ETL job submitted successfully'.

```

1 from GZ.SparkUtils import sparkSession
2
3 # Configuration
4 S3_CONNECTION_NAME = "AcmeWarehouseFiles"
5 S3_FILE_PATH = "inventory/2026-09-30.csv"
6 FILE_FORMAT = "csv"
7 GCS_CONNECTION_NAME = "AcmeArchiveGCS"
8 GCS_TARGET_PATH = "archive/inventory/2026-09-30"
9 WRITE_MODE = "overwrite"
10
11 # Initialize Spark session
12 spark = sparkSession("InventoryArchive", "WARN")
13
14 # Read from S3
15 df = spark.readS3(
16     connName=S3_CONNECTION_NAME,
17     fileName=S3_FILE_PATH,
18     format=FILE_FORMAT,
19     options={"header": "true"},
20 )
21
22 print(f"Read {df.count()} rows from {S3_FILE_PATH}")
23
24 # Write to GCS
25 spark.writeGcs(
26     df=df,
27     connName=GCS_CONNECTION_NAME,
28     fileName=GCS_TARGET_PATH,
29     format=FILE_FORMAT,
30 )
  
```

Timestamp	Logs
2026-10-01 14:30:01	Read 18402 rows from inventory/2026-09-30.csv
2026-10-01 14:30:04	Wrote archive to archive/inventory/2026-09-30

After a run, the Messages tab shows the job status and the timestamped logs.

New Task ×

Select Project*
Retail Pipelines ▼

Select Job*
inventory_archive ▼

RTE*
spark | Medium | 8Gi | 4

☒ Streaming ☐ Batch

Name*

Description

Start date
10/02/2026 

Start time
12:06 AM 

Cancel

Save

Scheduling a job: the RTE field records the runtime and compute as one value.

2.3 How-to guides

2.3.1 Read from a connection

Connections are created once in Groundzero (credentials stay in the connection) and referenced by name from code. Every connector adds `read*`, `execute*` and `write*` methods to the Spark session; the exact set depends on the connector (see the connector table in the API reference).

Pattern: the first argument to every read or execute method is `connName`, the name of the connection. For write methods the DataFrame comes first, followed by `connName`.

Object storage

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("ReadSales")

raw = spark.readS3(
    connName="acme-s3",
    fileName="landing/sales/2026/",
    format="csv",
    options={"header": "true", "inferSchema": "true"},
)
raw.printSchema()
```

`readS3` builds the path `s3a://<bucket>/<fileName>` from the connection's bucket. Pass a schema as `schema=` to skip inference on large inputs.

Relational databases

All the JDBC-based connectors (PostgreSQL, MySQL, SQL Server, Oracle, Redshift, Teradata, ClickHouse, Databricks, Trino and others) share one reader:

PYTHON

```
orders = spark.readJDBC(
    connName="acme-postgres",
    query="SELECT order_id, region, amount FROM sales.orders "
    "WHERE order_date ≥ DATE '2026-01-01'",
)
```

Run DDL or DML (create, insert, update, delete) with `executeJDBC`:

PYTHON

```
spark.executeJDBC(
    connName="acme-postgres",
    query="DELETE FROM sales.staging_orders WHERE loaded = true",
)
```

PUSH WORK TO THE DATABASE

`readJDBC` sends your `query` to the database as a subquery, so filters and aggregations in the SQL run there. Select only the columns and rows you need before Spark sees them.

Snowflake, BigQuery and other warehouses

PYTHON

```
df = spark.executeSnowflake(
    connName="acme-snowflake",
    query="SELECT customer_id, SUM(total) AS spend FROM orders GROUP BY 1",
)
```

Most warehouse connectors follow the same `executeX(connName, query)` shape and return a `DataFrame`, plus a `writeX(df, connName, tableName, ...)` writer. See the connector table for the exact method names.

Streams

PYTHON

```
events = spark.readKafka(
    connName="acme-kafka",
    topic="orders.created",
    startingOffsets="earliest",
)
```

`readKafka` returns a streaming `DataFrame`. Streaming writers such as `writeKafka`, `writeStreamToS3` and `writeGzLakehouse(..., streaming=True)` require a `checkpointLocation`.

Other helpers for credentials

If you need the native client library rather than a Spark `DataFrame` (for example `boto3`, `pymongo` or a SQLAlchemy engine), use the client factory:

PYTHON

```
from GZ.Client import getClient

client = getClient()
s3 = client.getS3Client("acme-s3")
buckets = s3.list_objects_v2(Bucket="acme-data", MaxKeys=5)
```

Credentials are never visible to your script. The factory returns an object that is already authenticated.

2.3.2 Query and write Lakehouse (Iceberg) tables

Groundzero Lakehouse tables are Apache Iceberg tables. A **GZLAKEHOUSE** Groundzero Lakehouse tables are **Apache Iceberg tables**. A `GZLAKEHOUSE connection**` names the warehouse and the default database. Two methods cover most work: `executeQuery` to read with SQL, and `writeGzLakehouse` to append or overwrite. To check it worked, call `orders.show()` or read the table back with `executeQuery`.

Query with SQL

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("LakehouseRead", siteName="acme")

orders = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="""
        SELECT region, SUM(amount) AS total
        FROM sales.orders
        GROUP BY region
    """,
)
orders.show()
```

Always qualify tables as `database.table`. The database name must be one you can see in the Lakehouse pages.

NAME THE SITE FOR LAKEHOUSE CONNECTIONS

A lakehouse connection routes every call to a site. Groundzero normally finds your site by itself. If it cannot, pass `siteName="your-site"` to `sparkSession` (applies to all such connections), pass `siteNames={"connection-name": "site"}` for one site per connection, or pass `siteName=` to the individual call. Otherwise the call raises `ValueError: siteName is required for GZLAKEHOUSE connection`.

Statements that only read data (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`) need no extra check. Statements that change data are checked against your permission on each table first.

Write a table

PYTHON

```
daily = orders.withColumnRenamed("total", "daily_total")

spark.writeGzLakehouse(
    df=daily,
    connName="acme-lakehouse",
    tableName="orders_daily",
    database="sales",
    mode="append",
)
```

For connections with a lakehouse endpoint, `mode` must be `append` or `overwrite`. `overwrite` replaces the snapshot of an existing table. External (read-only) lakehouse connections reject all writes.

Stream into a table

PYTHON

```
query = spark.writeGzLakehouse(
    df=events_df,
    connName="acme-lakehouse",
    tableName="orders_stream",
    database="sales",
    streaming=True,
    checkpointLocation="s3://acme-data/checkpoints/orders_stream",
    outputMode="append",
    trigger={"availableNow": True},
)
```

Create the target table, including its partition specification, before streaming into it. `partitionBy` is rejected for streaming writes.

Check your connection first

`verifyLakehouseConnection` checks that the connection reaches the expected warehouse. It turns a routing problem into an immediate, readable error:

PYTHON

```
info = spark.verifyLakehouseConnection("acme-lakehouse", siteName="acme")
print(info["warehouse"], info["site_used"])
```

Bulk-load from Snowflake or Redshift

For large one-time loads, let the source warehouse unload Parquet files directly into the lakehouse table's storage and register them with one Iceberg commit. No row passes through Spark:

PYTHON

```
summary = spark.bulkLoadGzLakehouseFromSnowflake(
    connName="acme-snowflake",
    query="SELECT * FROM analytics.orders WHERE order_date ≥ '2026-01-01'",
    lakehouseConn="acme-lakehouse",
    tableName="orders_history",
    database="sales",
    mode="overwrite",
    siteName="acme",
)
print(summary)
```

2.3.3 Work with outputs and previews

The runtime turns your result into a preview that the UI pages through.

- Assign the DataFrame you want to show to the top-level variable `outputDF`.
- For a Spark DataFrame, the runtime reads up to 100,000 rows. If your result has more rows, the preview is the first 100,000.
- A pandas DataFrame is converted to a list of records. Any list of dictionaries also works.
- The schema shown in the preview is taken from `outputDF`.

- When the same code runs again with the same inputs within a short period, the runtime may serve the preview from its cache. Results from governed lakehouse tables are cached per caller policy, so users share cached rows only when their policies allow it.

PYTHON

```
import pandas as pd

summary = pd.DataFrame(
    {"metric": ["rows", "regions"], "value": [1_250_000, 4]}
)
outputDF = summary
```

KEEP PREVIEWS FAST

Order and limit the result before you assign it to `outputDF`. A deterministic plan without shuffles lets the runtime return the first page without reading the whole preview. Joins, aggregates and sorts need the whole result.

Dataset runs page the preview with a page size that defaults to 1,000 rows and is capped at 10,000.

2.3.4 Use parameters and placeholders

Groundzero replaces a fixed set of placeholders in your code with run-time values before the script executes. They are replaced as text, so write them where a Python string expression is valid.

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("Parameters")

run_started = {STRTTM}
site = {SITENM}
print("Started:", run_started, "for site", site)
```

The line `run_started = {STRTTM}` becomes `run_started = "'2026-10-01 09:30:00'"`. The substituted value is already quoted for SQL, so it can be concatenated into a statement:

PYTHON

```
sql = "SELECT * FROM sales.orders WHERE created_at ≥ " + {STRTTM}
df = spark.executeQuery("acme-lakehouse", sql)
```

The placeholders are listed in the API reference. The `#{NAME}` spelling is also accepted.

PLACEHOLDERS ARE NOT PYTHON F-STRING FIELDS

Write `"... ≥ " + {STRTTM}`, not `f"... ≥ {STRTTM}"`. Inside a string literal the replacement text would break the quoting.

2.3.5 Chain jobs

A job that was submitted with sub-jobs can run them in sequence from code using `executeJob` :

PYTHON

```
from GZ.JobUtils import executeJob

results = executeJob(["load_orders", "build_marts"])
print(list(results))
```

Each sub-job runs with the same restrictions as a normal job. If one fails, `executeJob` raises and the remaining jobs do not run.

2.3.6 Schedule through ETL jobs

1. Save the job with its runtime and compute.
2. Open the scheduler for the job and set the frequency, start and stop times.
3. Choose the RTE with the runtime picker.

If you set a stop time, the session ends at that time and aborts any job still running. Use a stop time as a safety cap for long-running streaming jobs.

When a job finishes, Groundzero sends the notification configured for the job's status (success or failure), if there is one. Its subject and body can use the same placeholders. From code, you can also send your own mail (see `sendEmail`).

2.3.7 Install and use extra packages

Use this when your code needs a library that the runtime does not include. Check Installed packages first, because most common libraries are already there.

Need	Route
Try a package in a notebook	<code>%pip install name</code> in a notebook cell
Machine-learning libraries	Pick the scikit-learn, PyTorch or TensorFlow variant
Packages for every job	A custom runtime that extends a Spark runtime

Notebook installs

In a notebook you can install a package for the life of the session:

TEXT

```
%pip install rapidfuzz
```

To check it worked, `import rapidfuzz` in the next cell. The package disappears when the session ends. Job scripts run with restrictions, so do not install packages from inside a job script.

Custom runtimes

A custom runtime is a Spark runtime with extra Python packages added. Administrators create it on the **Custom RTE** page:

1. Prepare a `requirements.txt` with one pip requirement per line.
2. Select **Add custom RTE**, choose the **Base Image** to extend (for example a Spark variant), give the runtime a **Name**, and upload the file.
3. Wait for the build. The table on the page shows each runtime's name, creator, creation date and build **Status**.
4. When the status shows success, choose the new runtime in the runtime picker.

An example `requirements.txt`:

TEXT

```
splink
sqlparse
s3fs
```

PICK THE CLOSEST BASE

Extend the Spark variant that already has most of what you need (for example the scikit-learn variant for model work) so you only add the difference.

If the build fails, the **Status** column shows the failure. Check that each package name and version in `requirements.txt` exists and supports Python 3.12.

2.3.8 Tune performance

Partitions and shuffles

Spark runs one task per partition. Match the partition count to the cores you have: too few leaves workers idle, too many wastes time on scheduling.

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("Tuned", **{"spark.sql.shuffle.partitions": "64"})

df = spark.executeQuery("acme-lakehouse", "SELECT * FROM sales.orders")
df = df.repartition(64, "region")
```

Any keyword argument you pass to `sparkSession` is applied as a Spark setting. Setting names contain dots, so pass them with `**{ ... }` as shown. Settings apply only when the session is first created. A reused session ignores them.

Defaults the runtime already sets:

Setting	Value
<code>spark.serializer</code>	Kryo, buffer up to 1 GB

Setting	Value
<code>spark.driver.maxResultSize</code>	0 (unlimited)
<code>spark.network.timeout</code>	600 seconds
<code>spark.sql.broadcastTimeout</code>	600 seconds
<code>spark.sql.adaptive.enabled</code>	true when a lakehouse connection exists
<code>spark.dynamicAllocation.enabled</code>	true, executors idle out after 120 seconds

Memory

- Executor memory is set from the compute size (65 percent of worker memory, at most worker memory minus 1 GiB). To get more, pick a larger compute size.
- Avoid `collect()` and `toPandas()` on large DataFrames: the data lands in the driver's memory. Aggregate or limit first.
- Cache a DataFrame (`df.cache()`) only when several actions reuse it, and release it with `df.unpersist()` .

THE DRIVER HAS NO RESULT LIMIT

`spark.driver.maxResultSize` is 0, so an oversized `collect()` fails with an out-of-memory error instead of a friendly limit message.

Arrow

Arrow speeds up transfers between Spark and Python.

- Use the `*Arrow` connector methods (for example `readRESTArrow` , `readSalesforceArrow` , `readSFTPArrow`) for Python-side API connectors. They fetch and write through Arrow tables, and many accept a `columns` list to prune data.
- Enable Arrow for `toPandas()` and pandas UDFs with `spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")` .
- Dataset previews of nested columns are read through Arrow for you, so you need no setting for that.

Read less

- Select columns and filter early; Iceberg and Parquet push both down.
- Prefer Parquet or Iceberg over CSV for intermediate data.
- Pass an explicit `schema=` to `readS3` to skip schema inference.

2.3.9 Log from your job

Everything your script writes to standard output or standard error is captured and shown in the job log, tagged with the job's execution identifier. Use `print` or the standard `logging` module.

PYTHON

```
import logging

log = logging.getLogger("orders-job")
log.warning("Loaded %d rows", orders.count())
print("done")
```

`sparkSession(appName, logLevel)` sets the Spark log level (`ALL` , `DEBUG` , `INFO` , `WARN` , `ERROR` , `FATAL` , `OFF` , `TRACE`). The default is `WARN` . Errors raised by your script are reported with the exception type, message and the line number in your code, for example `ValueError: bad region at line 14` .

The runtime also collects Spark metrics for each run (tasks, stages, shuffle and spill, GC time, input and output records) and attaches them to the execution so you can inspect them with the run details. No code is needed for that.

2.3.10 Check data quality

Qualipilot runs deterministic checks inside Spark and, optionally, sends a compact aggregate to an LLM connection for a narrative summary. Row samples and top values are never sent.

PYTHON

```
from GZ.SparkUtils import sparkSession
from qualipilot import CheckConfig, QualipilotConfig
from qualipilot.models.config import ColumnRange

spark = sparkSession("QualityCheck")
orders = spark.createDataFrame(
    [(1001, "ana@example.com", 125.0), (1002, None, -25.0)],
    ["order_id", "customer_email", "amount"],
)
config = QualipilotConfig(
    checks=CheckConfig(
        required_columns=["order_id", "customer_email", "amount"],
        column_ranges={"amount": ColumnRange(min=0, max=10_000)},
    )
)
report = spark.runDataQualityChecks(
    df=orders, connectionName="acme-openai", config=config
)
print(report.to_json())
spark.saveDataQualityReport(report, "reports/orders.html", "html")
```

2.3.11 Parse PDFs and run Agents over documents

`parsePDF` extracts one document from S3, `parsePDFCorpus` extracts a folder restartably, and `invokeAgentPDFCorpus` runs a Groundzero Agent over every document in a prefix. Corpus runs are content-addressed: re-running the same source and output prefix skips page ranges that are already committed.

PYTHON

```
result = spark.parsePDFCorpus(  
    s3ConnName="acme-s3",  
    sourcePrefix="incoming/policies/",  
    outputS3ConnName="acme-s3",  
    outputPrefix="pdf-results/policies",  
    pageRangeSize=64,  
)  
print(result["status"], result["documentCount"], result["manifestUri"])
```

See the [API reference](#) for the full parameter list and the [examples gallery](#) for an end-to-end Agent run.

2.4 API reference

This section documents the Groundzero helper API that is available in the Spark runtime. Standard PySpark (DataFrame, functions, types, MLlib) is documented by Apache Spark and is not repeated here.

2.4.1 Conventions

- Import the session factory once: `from GZ.SparkUtils import sparkSession`.
- The object it returns (called `spark` in this guide) is the **Groundzero session**. Every connector method is a method on it.
- Connections are always referenced by `connName`, the connection name in Groundzero. Credentials never appear in code.
- Read and execute methods return a Spark DataFrame. Write methods take the DataFrame as the first argument (`df`) and generally return `None`.
- Method and parameter names use camelCase and are case sensitive.
- Argument names in the signatures below are the real parameter names; pass them by keyword for readability.

PARAMETERS NOT DESCRIBED HERE

Parameters that a method accepts but that this guide does not describe are passed straight through to the underlying connector.

2.4.2 Session

sparkSession

PYTHON

```
sparkSession(appName, logLevel="WARN", siteName=None, siteNames=None, **kwargs)
```

Creates (or returns) the Groundzero session for the current run.

Parameter	Type	Default	Description
<code>appName</code>	<code>str</code>	required	Application name shown in Spark and the logs
<code>logLevel</code>	<code>str</code>	<code>"WARN"</code>	Spark log level, such as <code>INFO</code> or <code>ERROR</code>
<code>siteName</code>	<code>str</code>	<code>None</code>	Site used for every REST-backed lakehouse connection
<code>siteNames</code>	<code>dict</code>	<code>None</code>	Mapping of connection name to site
<code>**kwargs</code>	<code>any</code>	<code>none</code>	Extra Spark settings, for example <code>**{"spark.x": "y"}</code>

`siteNames` also accepts a list of site names (a single entry applies to every REST-backed connection; a list of the same length maps by connection order) or a list of dictionaries with `connName` (or `connectionName`, `connection`, `name`) and `siteName` (or `site`).

Returns. The Groundzero session object.

Raises. `ValueError` if you try to move a lakehouse connection to a different site in the same session:

`GZLAKEHOUSE connection 'x' is already configured for site 'a'; cannot reconfigure it to 'b' in the same Spark session.`

Behavior:

- Calling `sparkSession` again in the same notebook or reused session returns the existing session. Extra Spark settings in `**kwargs` take effect only on the first call.
- Your settings are applied on top of the Groundzero defaults (see Tune performance).
- Lakehouse catalogs are registered for every GZLAKEHOUSE connection in the run.

PYTHON

```
spark = sparkSession(
    "DailyOrders",
    "INFO",
    siteNames={"acme-lakehouse": "acme", "partner-lakehouse": "partner"},
)
```

Session object

The session is a restricted wrapper around a Spark session.

Member	Behavior
<code>spark.sql(sqlQuery, *args, **kwargs)</code>	Runs Spark SQL; catalog and policy checks apply
<code>spark.createDataFrame(...)</code>	Passed through to Spark
<code>spark.table(...)</code> , <code>spark.range(...)</code>	Passed through to Spark
<code>spark.conf</code> , <code>spark.catalog</code> , <code>spark.udf</code>	Passed through to Spark
<code>spark.read</code> , <code>spark.write</code>	Blocked; use the connector methods
<code>spark.sparkContext</code>	Blocked

Accessing a blocked member raises `AttributeError: 'sparkSession' object has no attribute 'read'`. Use the connector's `read*` method instead. On a `DataFrame`, `df.write` works as in Spark, but connectors (which attach credentials, authorization, audit and lineage) are the supported route.

2.4.3 Lakehouse (Iceberg) methods

executeQuery

PYTHON

```
spark.executeQuery(connName, sqlQuery, siteName=None)
```

Runs SQL against a GZLAKEHOUSE connection and returns the result.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	GZLAKEHOUSE connection name
<code>sqlQuery</code>	<code>str</code>	required	SQL; reference tables as <code>database.table</code>
<code>siteName</code>	<code>str</code>	None	Site to route to; falls back to the session site

Returns. A Spark DataFrame.

Raises.

- `PermissionError`: authentication failed, the user is not authorized for a referenced table, or the connection is external (read-only) and the statement is not `SELECT`, `SHOW`, `DESCRIBE` or `EXPLAIN`.
- `ConnectionError`: the lakehouse endpoint could not be reached.
- `RuntimeError`: the endpoint returned a non-success status, or routed to a warehouse that does not match the connection.
- `ValueError`: the site is unknown or the connection has no warehouse name.

Every call is recorded in the lakehouse audit log (success or failure, with the duration), and lineage is reported for the tables it touches.

PYTHON

```
df = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT id, status FROM sales.orders WHERE status = 'open'",
    siteName="acme",
)
```

writeGzLakehouse

PYTHON

```
spark.writeGzLakehouse(
    df, connName, tableName, database=None, mode="overwrite",
    partitionBy=None, options=None, siteName=None, streaming=False,
    checkpointLocation=None, outputMode="append", triggerTime=None,
    trigger=None,
)
```

Writes a DataFrame to a lakehouse table.

Parameter	Type	Default	Description
<code>df</code>	DataFrame	required	Data to write; streaming frames start a stream
<code>connName</code>	<code>str</code>	required	GZLAKEHOUSE connection name
<code>tableName</code>	<code>str</code>	required	Target table
<code>database</code>	<code>str</code>	None	Target database; defaults to the connection's
<code>mode</code>	<code>str</code>	"overwrite"	<code>append</code> or <code>overwrite</code> (endpoint connections)
<code>partitionBy</code>	<code>list</code>	None	Partition columns; batch writes on legacy connections only

Parameter	Type	Default	Description
<code>options</code>	<code>dict</code>	<code>None</code>	Extra writer options, such as <code>fanout-enabled</code> for streams
<code>siteName</code>	<code>str</code>	<code>None</code>	Site to route to
<code>streaming</code>	<code>bool</code>	<code>False</code>	Start a Structured Streaming write
<code>checkpointLocation</code>	<code>str</code>	<code>None</code>	Required for streaming writes
<code>outputMode</code>	<code>str</code>	<code>"append"</code>	<code>append</code> or <code>complete</code> for streams
<code>triggerTime</code>	<code>str</code>	<code>None</code>	Processing-time trigger such as <code>"1 minute"</code>
<code>trigger</code>	<code>dict</code>	<code>None</code>	Trigger arguments such as <code>{"availableNow": True}</code>

Returns. For a batch write to a lakehouse endpoint, a `dict` with `status`, `table`, `warehouse`, `catalog`, `site`, `mode` and `via`. For a stream, the running streaming query.

Raises.

- `ValueError`: `mode` is not `append` or `overwrite`; `database` is missing; the site is missing; `partitionBy` is used with a stream.
- `PermissionError`: the user may not write the table, or the connection is external.
- `ConnectionError`, `RuntimeError`: the lakehouse endpoint is unreachable or rejected the commit.

A `DataFrame` whose `isStreaming` flag is true is written as a stream even when `streaming` is not set. The write is authorized per table, audited, and its schema and lineage are captured.

getLakehouseCatalog

PYTHON

```
spark.getLakehouseCatalog(connName, siteName)
```

Returns the Spark catalog alias for a REST-backed connection and site, so you can write fully qualified SQL for `spark.sql`.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	GZLAKEHOUSE connection name
<code>siteName</code>	<code>str</code>	required	Site that fixes the catalog

Returns. `str`, the catalog alias. **Raises.** `ValueError` if the connection has no lakehouse endpoint.

PYTHON

```
catalog = spark.getLakehouseCatalog("acme-lakehouse", "acme")
spark.sql(f"SELECT COUNT(*) FROM {catalog}.sales.orders").show()
```

setLakehouseSite

PYTHON

```
spark.setLakehouseSite(connName, siteName)
```

Registers the catalog for a site ahead of use. It does not change the active catalog. **Returns.** `None`. **Raises.** `ValueError` if the connection has no warehouse name or no lakehouse endpoint.

verifyLakehouseConnection

PYTHON

```
spark.verifyLakehouseConnection(connName, siteName=None)
```

Probes the lakehouse endpoint and checks that it routes to the connection's warehouse. The probe is anonymous: it checks reachability and routing, not authorization.

Returns. `dict` with the keys `provider_url`, `warehouse`, `site_used` and `prefix`. **Raises.** `ConnectionError`, `RuntimeError` or `ValueError` with a descriptive message.

Bulk loads into the lakehouse

PYTHON

```
spark.bulkLoadGzLakehouseFromSnowflake(
    connName, query, lakehouseConn, tableName, database=None,
    mode="overwrite", warehouseSize=None, storageIntegration=None,
    stagingLocation=None, maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES,
    siteName=None,
)
spark.bulkLoadGzLakehouseFromRedshift(
    connName, query, lakehouseConn, tableName, database=None,
    mode="overwrite", iamRole=None, stagingLocation=None,
    maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES, siteName=None,
)
```

The warehouse unloads the query result as Parquet under the target table's location, and the files are registered with one Iceberg commit. No row passes through Spark. The default for `maxFileSize` is 256 MiB.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	Source Snowflake or Redshift connection
<code>query</code>	<code>str</code>	required	Query whose rows are loaded
<code>lakehouseConn</code>	<code>str</code>	required	GZLAKEHOUSE connection of the target
<code>tableName</code>	<code>str</code>	required	Target table; created from the file schema if missing
<code>database</code>	<code>str</code>	<code>None</code>	Target database; defaults to the connection's
<code>mode</code>	<code>str</code>	<code>"overwrite"</code>	<code>overwrite</code> replaces the rows, <code>append</code> adds
<code>warehouseSize</code>	<code>str</code>	<code>None</code>	Snowflake only; warehouse size during the unload
<code>storageIntegration</code>	<code>str</code>	<code>None</code>	Snowflake only; storage integration to write with
<code>iamRole</code>	<code>str</code>	<code>None</code>	Redshift only; role to unload with
<code>stagingLocation</code>	<code>str</code>	<code>None</code>	Prefix inside the table location for the files
<code>maxFileSize</code>	<code>int</code>	256 MiB	Upper bound in bytes per unloaded file

Parameter	Type	Default	Description
siteName	str	None	Site to route to

Returns. A summary of the rows and files registered, or a summary with `via: "spark_write"` when the files had to be written with Spark. **Raises.** `ValueError` if the Redshift method is called with a connection that is not a Redshift connection.

2.4.4 Object storage and files

readS3

PYTHON

```
spark.readS3(connName, fileName, format="parquet", schema=None, options=None)
```

Parameter	Type	Default	Description
connName	str	required	S3 connection name
fileName	str	required	Key or prefix inside the connection's bucket
format	str	"parquet"	Any Spark format: <code>csv</code> , <code>json</code> , <code>parquet</code> , <code>orc</code>
schema	schema	None	Spark schema; skips inference
options	dict	None	Reader options, for example <code>{"header": "true"}</code>

Returns. A `DataFrame`. **Raises.** `Exception("Error reading file from S3: ...")` wrapping the underlying cause.

writeS3

PYTHON

```
spark.writeS3(df, connName, filePath, format, mode, options=None, coalesce=False)
```

Parameter	Type	Default	Description
df	<code>DataFrame</code>	required	Data to write
connName	str	required	S3 connection name
filePath	str	required	Target key or prefix
format	str	required	Output format
mode	str	required	<code>overwrite</code> , <code>append</code> , <code>ignore</code> or <code>errorifexists</code>
options	dict	None	Writer options
coalesce	bool	False	Write a single file (<code>df.coalesce(1)</code>)

Returns. `None`.

COALESCE FUNNELS ALL DATA THROUGH ONE TASK

`coalesce=True` makes one executor write the entire result. Use it only for small outputs.

writeStreamToS3

PYTHON

```
spark.writeStreamToS3(df, connName, filePath, format, outputMode,
                      checkpointLocation, coalesce=False, options=None)
```

Writes each micro-batch of a streaming DataFrame to S3 in overwrite mode. **Returns.** The running streaming query.

2.4.5 Relational databases and warehouses

readJDBC

PYTHON

```
spark.readJDBC(connName, query, options=None)
```

Reads the result of `query` from any JDBC connection (PostgreSQL, MySQL, SkySQL, Oracle, SQL Server, Redshift, Teradata, IBM Db2, SAP and SAP HANA, Synapse, CockroachDB, TimescaleDB, ClickHouse, Databricks, Trino, Presto, Vertica, Dremio).

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	JDBC connection name
<code>query</code>	<code>str</code>	required	SQL sent to the database
<code>options</code>	<code>dict</code>	<code>None</code>	Extra Spark JDBC options, such as <code>fetchsize</code>

Returns. A DataFrame. **Raises.** `ValueError("Connection x not found or is not a JDBC type.")`. TLS options for Redshift are enforced by the runtime.

writeJDBC

PYTHON

```
spark.writeJDBC(df, connName, tableName, mode, options=None)
```

Parameter	Type	Default	Description
<code>df</code>	DataFrame	required	Data to write
<code>connName</code>	<code>str</code>	required	JDBC connection name
<code>tableName</code>	<code>str</code>	required	Target table
<code>mode</code>	<code>str</code>	required	<code>append</code> , <code>overwrite</code> , <code>ignore</code> or <code>errorifexists</code>
<code>options</code>	<code>dict</code>	<code>None</code>	Extra Spark JDBC options

Raises. `ValueError` for an unknown connection, and `TypeError("Use writeSnowflake() for connection x.")` for Snowflake connections.

executeJDBC

PYTHON

```
spark.executeJDBC(connName, query)
```

Runs a DDL or DML statement through the JDBC driver and commits it. Prints `DDL statement executed successfully!` and returns `None`. Errors are printed and re-raised.

Snowflake

PYTHON

```
spark.executeSnowflake(connName, query, options=None)
spark.writeSnowflake(df, connName, tableName, mode="append", extraOptions=None)
spark.executeSnowflakeStatement(connName, statement)
```

Method	Returns
<code>executeSnowflake</code>	A <code>DataFrame</code> with the query result
<code>writeSnowflake</code>	<code>None</code> ; repartitions to at least 200 partitions first
<code>executeSnowflakeStatement</code>	The string <code>Snowflake</code> statement executed successfully.

`options` and `extraOptions` are merged into the connector options. Use `executeSnowflakeStatement` for DDL and session statements.

2.4.6 LLM methods

complete

PYTHON

```
spark.complete(connName, df, model=None, promptColumn="prompt",
               promptTemplate=None, systemPrompt=None, systemPromptColumn=None,
               outputColumn="llm_response", temperature=0.0, maxTokens=None,
               responseFormat=None, continueOnError=False, timeout=120,
               **providerOptions)
```

Adds a completion to every row of a `DataFrame` using the provider of the connection (OpenAI, Azure OpenAI, Anthropic, AWS Bedrock, Cohere, Gemini, Fireworks AI, Together AI, Hugging Face and xAI). The calls run inside the executors, so the work is distributed and no rows are collected to the driver.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	LLM connection name
<code>df</code>	<code>DataFrame</code>	required	Input rows
<code>model</code>	<code>str</code>	<code>None</code>	Model; for Azure the deployment name; for Bedrock the model ID

Parameter	Type	Default	Description
<code>promptColumn</code>	<code>str</code>	<code>"prompt"</code>	Column holding the prompt text
<code>promptTemplate</code>	<code>str</code>	<code>None</code>	Template with <code>{{column}}</code> or <code>\${column}</code> fields
<code>systemPrompt</code>	<code>str</code>	<code>None</code>	System prompt applied to every row
<code>systemPromptColumn</code>	<code>str</code>	<code>None</code>	Column with a per-row system prompt
<code>outputColumn</code>	<code>str</code>	<code>"llm_response"</code>	Name of the response column
<code>temperature</code>	<code>float</code>	<code>0.0</code>	Sampling temperature
<code>maxTokens</code>	<code>int</code>	<code>None</code>	Response limit
<code>responseFormat</code>	<code>str</code>	<code>None</code>	Provider response format, for example <code>json</code>
<code>continueOnError</code>	<code>bool</code>	<code>False</code>	Record per-row errors instead of failing
<code>timeout</code>	<code>int</code>	<code>120</code>	Request timeout in seconds (not used by Bedrock)

`providerOptions` accepts provider-only settings, such as `useConverse` for Bedrock or `returnFullText` for Hugging Face.

Returns. The input DataFrame plus these columns, where `<out>` is `outputColumn : <out>`, `<out>_status` (`success` or `error`), `<out>_error`, `<out>_provider`, `<out>_model`, `<out>_prompt_tokens`, `<out>_completion_tokens`, `<out>_total_tokens` and `<out>_latency_ms`.

Raises. `ValueError` if the connection is unknown (`Connection 'x' was not found.`), has an unsupported type (`Connection 'x' has unsupported LLM type 'y'. Supported types: ...`), a referenced column is missing (`Missing columns: [...]. Available: [...]`), or a Bedrock call has no model (`AWS Bedrock model must be set in the connection or call.`). With `continueOnError=False` the first failing row fails the job.

Requests are retried with exponential backoff on HTTP 429 and 5xx responses.

PYTHON

```

tickets = spark.createDataFrame(
    [(1, "Printer jams on page two"), (2, "Cannot reset my password")],
    ["id", "text"],
)
labeled = spark.complete(
    connName="acme-openai",
    df=tickets,
    promptTemplate="Label the ticket as hardware or account: {{text}}",
    outputColumn="label",
    maxTokens=8,
    continueOnError=True,
)

```

Fine-tuning and batch operations

Provider-neutral entry points route by connection type:

PYTHON

```
spark.fineTune(connName, df, **kwargs)
spark.distill(connName, df, **kwargs)
spark.listFineTuneJobs(connName, **kwargs)
spark.getFineTuneJob(connName, **kwargs)
spark.cancelFineTuneJob(connName, **kwargs)
spark.listModels(connName, **kwargs)
```

Each forwards its arguments to the provider method named by the operation plus the provider suffix, for example `fineTuneOpenAI`, `fineTuneAwsBedrock` or `listModelsTogetherAI`. For AWS Bedrock, cancel maps to `stopFineTuneJobAwsBedrock`. If the provider does not implement the operation, a `NotImplementedError` is raised: `fineTune is not available for connection type 'x'`.

Provider suffix	Typical methods
OpenAI, AzureOpenAI	<code>fineTune</code> , <code>distill</code> , jobs, batch jobs, deployments
AwsBedrock	<code>fineTune</code> , <code>distill</code> , jobs (<code>stop</code>), batch jobs, deployments
TogetherAI, FireworksAI	<code>fineTune</code> , jobs, batch jobs, deployments
Cohere, Gemini, HuggingFace, Anthropic	<code>complete</code> , models, batch jobs, deployments
Unsloth	<code>fineTuneUnsloth</code> , <code>listModelsUnsloth</code> (GPU runtime)
XAI	<code>completeXAI</code> , <code>listModelsXAI</code>

Direct fine-tuning is unavailable for Anthropic, Cohere, the Gemini API and the generic Hugging Face connector. Use AWS Bedrock for eligible Claude models, Vertex AI for Gemini tuning, and Unsloth or AutoTrain for Hugging Face models. OpenAI and Azure OpenAI require at least 10 training examples.

The batch method families follow the same naming: `createBatchJob<Provider>`, `listBatchJobs<Provider>`, `getBatchJob<Provider>`, `cancelBatchJob<Provider>` and `getBatchResults<Provider>`. Deployment methods follow `createDeployment<Provider>`, `listDeployments<Provider>`, `getDeployment<Provider>` and `deleteDeployment<Provider>`. Which ones exist for a provider is listed in the table above; the method listing for each connector is in the connector reference below.

2.4.7 Data quality

PYTHON

```
spark.createDataQualityChecker(df, connectionName=None, config=None,
                               references=None)
spark.runDataQualityChecks(df, connectionName=None, config=None,
                           includeLlm=True, references=None)
spark.runTableDataQualityChecks(tableName, connectionName=None, config=None,
                                includeLlm=True, references=None)
spark.runQueryDataQualityChecks(query, connectionName=None, config=None,
                                includeLlm=True, references=None)
spark.saveDataQualityReport(report, path, reportFormat="json")
```

Parameter	Type	Default	Description
<code>df</code>	<code>DataFrame</code>	<code>required</code>	Data to check (streaming frames are not allowed for local work)
<code>tableName</code>	<code>str</code>	<code>required</code>	Catalog table to load and check
<code>query</code>	<code>str</code>	<code>required</code>	Spark SQL whose result is checked
<code>connectionName</code>	<code>str</code>	<code>None</code>	LLM connection used for the narrative
<code>config</code>	<code>QualipilotConfig</code>	<code>None</code>	Checks to run
<code>includeLlm</code>	<code>bool</code>	<code>True</code>	Ask the LLM to narrate the results
<code>references</code>	<code>dict</code>	<code>None</code>	Reference data forwarded to Qualipilot
<code>report</code>	<code>QualityReport</code>	<code>required</code>	Report returned by a run method
<code>path</code>	<code>str</code>	<code>required</code>	Output path
<code>reportFormat</code>	<code>str</code>	<code>"json"</code>	Output format, for example <code>html</code>

Returns. `runDataQualityChecks`, `runTableDataQualityChecks` and `runQueryDataQualityChecks` return a `QualityReport` with the check results and `llm_report`. `createDataQualityChecker` returns a checker you can use as a context manager and call `run(include_llm=...)` on.

Only a compact aggregate of the deterministic results goes to the LLM. Row samples and top values are excluded.

Local record linkage

TEXT

```
spark.runLocalRecordLinkage(data, config, rightData=None, *, maxRows)
spark.runLocalRecordDeduplication(data, config, consolidation, *, maxRows)
spark consolidateLocalRecords(data, idColumn, clusters, config, *, maxRows)
```

Run Qualipilot's native linkage on small datasets, materialized directly as driver-local Polars DataFrames. `maxRows` is mandatory and must be positive; the runtime applies `limit(maxRows + 1)` and fails if the data exceeds it.

Raises. `ValueError` with messages such as `maxRows must be a positive integer`, `data exceeds maxRows=N`, `rightData is required in link mode`, `rightData is only valid in link mode`, and `driver-local Qualipilot operations reject streaming data`. Nested, variant, duplicate or case-conflicting columns fail before collection. `maxRows` limits rows, not width or candidate pairs, so select only the columns you need.

2.4.8 PDF extraction

parsePDF

PYTHON

```
spark.parsePDF(s3ConnName=None, inputUri=None, outputPrefix=None,
               outputFormat="csv", outputFormats=None, inputFile=None,
               connName=None, hybrid=False, ocrMode="auto",
               ocrStack="ppocrv6", strict=False, password=None,
               language="eng", maxPages=None, maxFileBytes=None,
               llmProvider=None, llmConnName=None, llmModel=None,
               llmBaseUrl=None, llmApiKey=None, outputS3ConnName=None,
               returnAgentPages=False, contentAddressed=False,
               inlineAgentPages=True, pageRange=None)
```

Parses one PDF stored in S3 and writes the result back to S3. The `llm*` parameters configure the LLM-assisted hybrid mode and are used together with `hybrid=True`.

Parameter	Type	Default	Description
<code>s3ConnName</code>	<code>str</code>	<code>None</code>	S3 connection of the source (or use <code>connName</code>)
<code>inputUri</code>	<code>str</code>	<code>None</code>	PDF location; <code>inputFile</code> is an alias
<code>outputPrefix</code>	<code>str</code>	<code>generated</code>	Output prefix; defaults to <code>pdf-extractor/<uuid></code>
<code>outputFormat</code>	<code>str</code>	<code>"csv"</code>	<code>json</code> , <code>md</code> , <code>txt</code> , <code>csv</code> or <code>xlsx</code>
<code>outputFormats</code>	<code>list</code>	<code>None</code>	Several formats at once
<code>ocrMode</code>	<code>str</code>	<code>"auto"</code>	<code>off</code> , <code>auto</code> or <code>all</code>
<code>ocrStack</code>	<code>str</code>	<code>"ppocrv6"</code>	<code>classic</code> or <code>ppocrv6</code>
<code>language</code>	<code>str</code>	<code>"eng"</code>	OCR language
<code>strict</code>	<code>bool</code>	<code>False</code>	Fail instead of returning partial results
<code>password</code>	<code>str</code>	<code>None</code>	Password for encrypted PDFs
<code>maxPages</code>	<code>int</code>	<code>None</code>	Page limit
<code>maxFileBytes</code>	<code>int</code>	<code>None</code>	File size limit
<code>outputS3ConnName</code>	<code>str</code>	<code>None</code>	S3 connection for output; defaults to the source

Returns. A `dict` with `status`, `input_uri`, `output_prefix`, `output_format`, `result_uri`, `artifacts` (including a `manifest`), `page_count` and `sourceSha256`.

Raises. `ValueError("s3ConnName is required")` and `ValueError("inputUri or inputFile is required")`; `ValueError("pageRange requires returnAgentPages=True")`.

readPDF and readPDFResult

PYTHON

```
spark.readPDF(s3ConnName, inputUri=None, outputPrefix=None, outputFormat="csv",
              outputFormats=None, readOptions=None, outputS3ConnName=None,
              **options)
spark.readPDFResult(resultUri, outputFormat="csv", s3ConnName=None,
                    readOptions=None)
```

`readPDF` parses a PDF and returns the result as a DataFrame. `readPDFResult` reads an existing result. Spark reads `csv` (without a header by default), `json` and the text formats `txt`, `text`, `md`, `markdown`. Other formats raise `ValueError("Spark cannot read 'x' directly")`.

parsePDFCorpus

PYTHON

```
spark.parsePDFCorpus(s3ConnName, sourcePrefix=None, inputUri=None,
                     outputPrefix=None, outputS3ConnName=None,
                     pageRangeSize=64, partitions=None, ocrMode="auto",
                     language="eng", strict=False, password=None,
                     maxFileBytes=None, maxDocuments=10_000)
```

Parses an S3 prefix, or an explicit list, as restartable distributed work. Discovery is recursive and finds `.pdf` files regardless of case. Results are content-addressed: re-running with the same source and output prefix skips page ranges that were already committed.

Parameter	Type	Default	Description
<code>s3ConnName</code>	<code>str</code>	required	S3 connection of the source
<code>sourcePrefix</code>	<code>str</code>	<code>None</code>	Folder to scan (give this or <code>inputUri</code>)
<code>inputUri</code>	<code>list</code>	<code>None</code>	Explicit PDF list
<code>outputPrefix</code>	<code>str</code>	<code>None</code>	Where results go
<code>pageRangeSize</code>	<code>int</code>	<code>64</code>	Pages per unit of work
<code>partitions</code>	<code>int</code>	<code>None</code>	Parallel parsers, capped (default cap 2)
<code>maxDocuments</code>	<code>int</code>	<code>10_000</code>	Document limit

Returns. A `dict` including `status`, `documentCount` and `manifestUri`.

The cap on `partitions` exists because every parser worker loads a complete document-model set; size executor memory before raising it.

2.4.9 Agents

listAgents

PYTHON

```
spark.listAgents(connName)
```

Returns a `list[dict]` of the Agents visible to the connection's service account.

invokeAgent

TEXT

```
spark.invokeAgent(connName, agentId, message, sessionId=None, *,
                  requestId=None, expectedAgentVersion=None,
                  disableTools=False, pdfSourceConnName=None,
                  pdfSourceObject=None, pdfSinkConnName=None,
                  pdfSinkPrefix=None)
```

Submits the message in the background and waits for the final result, polling for up to 15 minutes so no HTTP request stays open for the whole run.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	Agent connection
<code>agentId</code>	<code>str</code>	required	Agent identifier (UUID)
<code>message</code>	<code>str</code>	required	Non-empty message
<code>sessionId</code>	<code>str</code>	None	UUID of an existing conversation
<code>requestId</code>	<code>str</code>	None	Idempotency identifier
<code>expectedAgentVersion</code>	<code>int</code>	None	Fail if the live version differs; must be ≥ 1
<code>disableTools</code>	<code>bool</code>	False	Run without tools; not allowed with PDFs
<code>pdfSourceConnName</code>	<code>str</code>	None	S3 connection of a PDF to bind
<code>pdfSourceObject</code>	<code>str</code>	None	PDF location
<code>pdfSinkConnName</code>	<code>str</code>	None	S3 connection for extracted pages
<code>pdfSinkPrefix</code>	<code>str</code>	None	Prefix for extracted pages

Returns. The final invocation result as a `dict`, including `pdfArtifacts` when a PDF was bound.

Raises. `ValueError` (message must be a non-empty string, `expectedAgentVersion` must be a positive integer, `disableTools` must be a boolean, `disableTools` cannot be used with PDF tool bindings), and `TimeoutError` with the run ID if polling times out. A timeout does not cancel or resubmit the run; inspect the existing run before starting another.

invokeAgentPDFCorpus

TEXT

```
spark.invokeAgentPDFCorpus(connName, agentId, message, pdfSourceConnName,
                           pdfSinkPrefix, *, pdfSourcePrefix=None,
                           pdfSourceObjects=None, pdfSinkConnName=None,
                           pageRangeSize=64, partitions=None,
                           summaryWorkers=4, summaryInputChars=100_000,
                           maxDocuments=10_000, maxAgentCalls=50_000,
                           ocrMode="auto", language="eng", strict=False,
                           password=None, maxFileBytes=None)
```

Runs an approved Agent over every complete PDF under a prefix. Pass exactly one of `pdfSourcePrefix` or `pdfSourceObjects`.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	Agent connection
<code>agentId</code>	<code>str</code>	required	Agent identifier
<code>message</code>	<code>str</code>	required	Instruction applied to every document
<code>pdfSourceConnName</code>	<code>str</code>	required	S3 connection of the documents
<code>pdfSinkPrefix</code>	<code>str</code>	required	Prefix for results
<code>summaryWorkers</code>	<code>int</code>	4	Concurrent document workflows, 1 to 32
<code>maxDocuments</code>	<code>int</code>	10_000	Document limit
<code>maxAgentCalls</code>	<code>int</code>	50_000	Shared Agent-call budget

Returns. A `dict` with `status`, `outputMode`, `resultsUri`, `manifestUri`, `processedDocumentCount` and `skippedDocumentCount`. Each row at `resultsUri` links to a JSON object with the validated `data`. The older names `summaryResultsUri`, `summaryManifestUri`, `summaryUri` and `summarizedDocumentCount` remain aliases.

The corpus resolves the approved Agent version and response schema before any PDF work. With a JSON response schema, the output mode is `structured`; without one, the Agent returns cited facts. Unchanged results are reused on later runs. Documents that fail are recorded in the run index and do not stop the others.

2.4.10 Jobs, mail and clients

executeJob

PYTHON

```
from GZ.JobUtils import executeJob

executeJob(jobNames)
```

Runs the named sub-jobs of the current job in sequence, each under the same restrictions as a normal job.

Parameter	Type	Default	Description
<code>jobNames</code>	<code>list[str]</code>	required	Names of sub-jobs to run, in order

Returns. `dict` mapping each job name to its output. **Raises.** `ValueError` when none of the names match a sub-job, and any exception from a sub-job (the remaining jobs do not run).

sendEmail

PYTHON

```
from GZ.SendEmail import sendEmail

sendEmail(subject, body, to_email, from_email, **kwargs)
```

Sends an email through the Groundzero notification service. Extra keyword arguments are added to the message.

Parameter	Type	Default	Description
<code>subject</code>	<code>str</code>	required	Subject line
<code>body</code>	<code>str</code>	required	Message body
<code>to_email</code>	<code>str</code>	required	Recipient address
<code>from_email</code>	<code>str</code>	required	Sender address

Returns. `None` . **Raises.** Any error from the notification service.

getClient

PYTHON

```
from GZ.Client import getClient

client = getClient()
```

Returns a client factory. Its methods build authenticated native clients from a connection name; each takes `connName` (the SQL engine method takes `conn_name`).

Method	Returns
<code>getSQLEngine(conn_name)</code>	SQLAlchemy engine (PostgreSQL, MySQL, Oracle, Db2, SQL Server, SAP, Redshift, Exasol, CockroachDB)
<code>getS3Client(connName)</code>	boto3 S3 client
<code>getS3FSCClient(connName)</code>	s3fs file system
<code>getSnowflakeClient(connName)</code>	Snowflake connection
<code>getMongoClient(connName)</code>	MongoDB client
<code>getAzureBlobClient(connName)</code>	Azure Blob client
<code>getKinesisClient</code> , <code>getDynamoDBClient</code> , <code>getAthenaClient</code>	AWS clients
<code>getBigQueryClient</code> , <code>getGCSCClient</code> , <code>getGoogleDriveClient</code>	Google clients
<code>getCassandraSession</code> , <code>getRedshiftClient</code> , <code>getElasticClient</code>	Database clients

Method	Returns
<code>getOpenAIClient</code> , <code>getAzureOpenAIClient</code> , <code>getAwsBedrockClient</code>	LLM clients
<code>getCohereClient</code> , <code>getTogetherAIClient</code> , <code>getHuggingFaceClient</code>	LLM clients
<code>getFireworksAIClient</code> , <code>getXAIClient</code> , <code>getMcpClient</code>	LLM and MCP clients

Further methods exist for Salesforce, SFTP, CosmosDB, Exasol, Timescale, Pinecone, Firebolt, Teradata, Neo4j, InfluxDB, DuckDB, Business Central, Shopify, Marketo, Couchbase, ClickHouse, OneDrive, Splunk, Databricks SQL, Synapse, Event Hubs, SharePoint, ScyllaDB and Weaviate, plus `read_rest` and `write_rest` helpers for REST connections.

2.4.11 Placeholders and outputs

Placeholders

Placeholder	Replaced with
<code>{STRTTM}</code>	Run start time, <code>YYYY-MM-DD HH:MM:SS</code>
<code>{ENDTM}</code>	Timestamp taken just before your script starts; in notifications, the end time
<code>{CLNTNM}</code>	Client name
<code>{SITENM}</code>	Site name
<code>{PRJNM}</code>	Project name
<code>{JOBNM}</code>	Job name
<code>{STS}</code>	Status
<code>{ERR}</code>	Error message
<code>{RTE}</code>	Runtime name
<code>{RGN}</code>	Region
<code>{USR}</code>	User name of the caller
<code>{USRGRPS}</code>	Groups of the caller

`{STS}` and `{ERR}` are meant for notification messages. In job code they are empty. In code, each placeholder becomes a Python string literal that already includes single quotes (for example `"'2026-10-01 09:30:00'"`). `{USRGRPS}` expands to a comma-separated list of such literals. A name that is not in the table is left unchanged, so ordinary braces in strings and dictionaries are safe unless they match a name exactly. The `${NAME}` spelling is equivalent.

outputDF

After your script ends, the runtime reads the global `outputDF`:

Value	Handling
Spark DataFrame	Up to 100,000 rows are collected; schema from the DataFrame
pandas DataFrame	Converted with <code>to_dict(orient="records")</code>
Any other iterable	Converted with <code>list(...)</code>
Not defined	No preview; the job still succeeds

2.4.12 Connector method index

Every connector below is available as methods on the session. `df` is a Spark DataFrame; `connName` is the Groundzero connection name; `mode` is a Spark save mode unless noted. Methods ending in `Arrow` exchange data through Apache Arrow tables.

LOOK UP SYNTAX

The Groundzero assistant can look up the Spark syntax of any connector method, so you do not need to memorize this list.

Object storage and files

TEXT

```
# S3
spark.readS3(connName, fileName, format='parquet', schema=None, options=None)
spark.writeS3(df, connName, filePath, format, mode, options=None, coalesce=False)
spark.writeStreamToS3(df, connName, filePath, format, outputMode, checkpointLocation,
    coalesce=False, options=None)
# GCS
spark.getCredentialsFileGCS(credentialsFile)
spark.readGcs(connName, fileName, format, options)
spark.writeGcs(df, connName, fileName, format, mode, options=None, coalesce=False)
# AZUREBLOB
spark.readAzureBlob(connName, filePath, options=None)
spark.writeAzureBlob(df, connName, filePath, format, mode, options=None)
# SFTP
spark.readSFTP(connName, fileType, filePath)
spark.writeSFTP(df, connName, fileType, filePath, mode='overwrite')
spark.listSFTP(connName, remotePath='.')
spark.deleteSFTP(connName, remotePath)
spark.readSFTPArrow(connName, fileType, filePath, **options)
spark.writeSFTPArrow(df, connName, fileType, filePath, mode='overwrite', **options)
spark.listSFTPArrow(connName, remotePath='.', recursive=False)
spark.deleteSFTPArrow(connName, remotePath, recursive=False)
spark.mkdirSFTPArrow(connName, remotePath)
spark.renameSFTPArrow(connName, srcPath, dstPath)
spark.existsSFTPArrow(connName, remotePath)
# GOOGLEDRIVE
spark.readGoogleDriveFiles(connName, folderId=None, query=None, limit=None)
spark.readGoogleDriveFileContent(connName, fileId)
spark.writeGoogleDriveFiles(df, connName, parentFolderId=None, batchSize=10,
    maxPartitions=None)
spark.deleteGoogleDriveFile(connName, fileId)
spark.createGoogleDriveFolder(connName, folderName, parentFolderId=None)
spark.readGoogleDriveFilesArrow(connName, folderId=None, query=None, limit=None,
    columns=None, options=None)
```

```
# ONEDRIVE
spark.readOneDriveFiles(connName, folderPath=None, fileExtensions=None, limit=None,
    batchSize=1000, userPrincipalName=None, driveId=None)
spark.writeOneDriveFiles(df, connName, folderPath=None, batchSize=10,
    maxPartitions=None, userPrincipalName=None, driveId=None)
spark.deleteOneDriveFile(connName, fileId=None, filePath=None, userPrincipalName=None,
    driveId=None)
spark.readOneDriveFilesArrow(connName, folderPath=None, limit=None, batchSize=100,
    userPrincipalName=None, driveId=None, columns=None, options=None)
# SHAREPOINT
spark.readSharePoint(connName, listName, select=None, filter=None, orderBy=None,
    expand=None, top=None, maxResults=5000)
spark.writeSharePoint(df, connName, listName, operation='create', batchSize=100)
spark.readSharePointArrow(connName, listName, select=None, filter=None, orderBy=None,
    expand=None, top=None, maxResults=None, columns=None, options=None)
spark.writeSharePointArrow(df, connName, listName, operation='create', batchSize=100,
    options=None)
```

Table formats

TEXT

```
# DELTA
spark.readDelta(connName, path=None, tableName=None, options=None, **cloudOptions)
spark.writeDelta(df, connName, path=None, tableName=None, mode='append', options=None,
    **cloudOptions)
spark.executeDelta(query)
# HUDI
spark.readHudi(connName, tableName, options=None)
spark.writeHudi(df, connName, tableName, writeOperation='upsert', options=None)
```

Relational databases and warehouses

TEXT

```
# JDBC
spark.readJDBC(connName, query, options=None)
spark.writeJDBC(df, connName, tableName, mode, options=None)
spark.executeJDBC(connName, query)
spark.bulkLoadGzLakehouseFromRedshift(connName, query, lakehouseConn, tableName,
    database=None, mode='overwrite', iamRole=None, stagingLocation=None,
    maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES, siteName=None)
# SNOWFLAKE
spark.executeSnowflakeStatement(connName, statement)
spark.executeSnowflake(connName, query, options=None)
spark.writeSnowflake(df, connName, tableName, mode='append', extraOptions=None)
spark.bulkLoadGzLakehouseFromSnowflake(connName, query, lakehouseConn, tableName,
    database=None, mode='overwrite', warehouseSize=None, storageIntegration=None,
    stagingLocation=None, maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES, siteName=None)
# GOOGLEBIGQUERY
spark.getCredentialsFileBQ(credentialsFile)
spark.executeBigQuery(connName, tableName, query=None)
spark.writeBigQuery(df, connName, tableName, writeMethod, mode)
# ATHENA
spark.readAthena(connName, region=None, query=None, options=None, workGroup=None,
    catalog=None, sessionToken=None, accessKey=None, secretAccessKey=None)
spark.executeAthena(connName, region=None, query=None, options=None, workGroup=None,
    catalog=None, sessionToken=None, accessKey=None, secretAccessKey=None)
spark.writeAthena(df, connName, tableName, mode, options=None, region=None,
    workGroup=None, catalog=None, sessionToken=None, accessKey=None,
    secretAccessKey=None, path=None, createTable=None, replaceTable=None,
```

```

        repairTable=None)
# DATABRICKS
spark.executeDatabricks(connName, query, options=None)
spark.writeDatabricks(df, connName, tableName, mode='append', options=None)
spark.executeDatabricksStatement(connName, query)
# TRINO
spark.executeTrino(connName, query, options=None)
spark.writeTrino(df, connName, tableName, mode='append', options=None)
spark.executeTrinoStatement(connName, query)
# PRESTO
spark.executePresto(connName, query, options=None)
spark.writePresto(df, connName, tableName, mode='append', options=None)
spark.executePrestoStatement(connName, query)
# VERTICA
spark.executeVertica(connName, query, options=None)
spark.writeVertica(df, connName, tableName, mode='append', options=None)
spark.executeVerticaStatement(connName, query)
# EXASOL
spark.executeExasol(connName, query, fingerprint=None)
spark.writeExasol(df, connName, tableName, fingerprint=None, mode='append',
    options=None)
# DORIS
spark.readDoris(connName, tableIdentifier=None, options=None)
spark.executeDoris(connName, tableIdentifier=None, filterQuery=None, options=None)
spark.writeDoris(df, connName, tableIdentifier=None, mode='append', options=None)
# DREMIO
spark.executeDremio(connName, query, options=None)
spark.writeDremio(df, connName, tableName, mode='append', options=None)
spark.executeDremioStatement(connName, query)
# DREMIOFLIGHT
spark.executeDremioFlight(connName, query, options=None)
spark.writeDremioFlight(df, connName, tableName, mode='append', options=None)
spark.executeDremioFlightStatement(connName, query)
# FIREBOLT
spark.executeFirebolt(connName, sqlQuery)
spark.readFirebolt(connName, sqlQuery)
spark.writeFirebolt(df, connName, tableName, mode='append', options=None)
spark.executeFireboltSQL(connName, sqlStatement)
spark.listFireboltTables(connName)
# SINGLESTORE
spark.readSingleStore(connName, query, options=None)
spark.readSingleStoreTable(connName, tableName, options=None)
spark.writeSingleStore(df, connName, tableName, mode, options=None)
spark.executeSingleStore(connName, query, options=None)
# MOTHERDUCK
spark.executeMotherDuck(connName, query)
spark.readMotherDuck(connName, query)
spark.executeMotherDuckSQL(connName, statement)
spark.writeMotherDuck(df, connName, tableName, mode='append')

```

NoSQL, search and caches

TEXT

```

# MONGODB
spark.readMongodb(connName, databaseName, collectionName, options=None)
spark.writeMongodb(df, connName, databaseName, collectionName, mode)
# DYNAMODB
spark.readDynamoDB(connName, tableName, options=None, schema=None)
spark.readDynamoDBIndex(connName, tableName, indexName, options=None, schema=None)
spark.queryDynamoDB(connName, tableName, hashKeyName, hashKeyValue, rangeKeyName=None,

```

```

    rangeKeyValue=None, rangeKeyOperator='=', options=None, schema=None)
spark.writeDynamoDB(df, connName, tableName, mode='append', options=None)
spark.updateDynamoDB(df, connName, tableName, mode='append', options=None)
spark.deleteDynamoDB(df, connName, tableName, mode='append', options=None)
# COSMOSDB
spark.readCosmosDb(connName)
spark.writeCosmosDb(df, connName, mode)
# COUCHBASE
spark.readCouchbase(connName, scope=None, collection=None, service='query', **kwargs)
spark.writeCouchbase(df, connName, scope=None, collection=None, service='query',
    mode='overwrite', **kwargs)
# CASSANDRA
spark.executeCassandra(connName, tableName, query, isQuery=False)
spark.writeCassandra(df, connName, table, mode)
spark.executeCassandraStatement(connName, statement)
# SCYLLADB CLOUD
spark.readScyllaDbCloud(connName, tableName, keyspace=None, options=None)
spark.executeScyllaDbCloud(connName, query, keyspace=None)
spark.writeScyllaDbCloud(df, connName, tableName, keyspace=None, mode='append',
    options=None)
spark.executeScyllaDbCloudCQL(connName, cqlStatement, keyspace=None)
spark.listScyllaDbCloudTables(connName, keyspace=None)
spark.createScyllaDbCloudTable(connName, tableName, columns, keyspace=None)
# ELASTICSEARCH
spark.readElasticsearch(connName, indexName)
spark.writeElasticsearch(df, connName, indexName, mode)
# REDIS
spark.readRedis(connName, table=None, options=None)
spark.writeRedis(df, connName, table=None, mode='append', options=None)
# NEO4J
spark.executeNeo4j(connName, query, options=None)
spark.writeNeo4j(df, connName, nodeLabel=None, relationship=None, mode='overwrite',
    options=None)
# INFLUXDB
spark.executeInfluxDB(connName, query, database=None)
spark.writeInfluxDB(df, connName, measurement, timeColumn='time', tagColumns=None,
    fieldColumns=None, batchSize=5000, timeUnit='ns', useForeachPartition=True,
    maxPartitions=None)
spark.writeInfluxDBLineProtocol(df, connName, batchSize=5000, maxPartitions=None)

```

Streams

TEXT

```
# KAFKA
spark.readKafka(connName, topic, startingOffsets=None, options=None)
spark.writeKafka(df, connName, topic, checkpointLocation, outputMode, triggerTime,
    options=None, s3AccessKey=None, s3SecretKey=None, s3SessionToken=None,
    s3Region=None, s3Endpoint=None)

# KINESIS
spark.readKinesisStream(connName, startingPosition, consumerType, options=None)
spark.writeKinesisStream(connName, df, checkpointLocation, outputMode, triggerTime,
    outputType, options=None, forEachBatch=None)

# EVENTHUBS
spark.readAzureEventHubs(connName, consumerGroup, **kwargs)
spark.writeAzureEventHubs(df, connName, **kwargs)

# PULSAR
spark.readPulsar(connName, topic=None, options=None)
spark.readPulsarStream(connName, topic=None, options=None)
spark.writePulsar(df, connName, topic=None, mode='append', options=None)
spark.writePulsarStream(df, connName, checkpointLocation, topic=None,
    outputMode='append', triggerTime='1 seconds', options=None)
```

SaaS and APIs

TEXT

```
# REST
spark.readREST(connName, endpoint=None, method=None, params=None, data=None,
    options=None)
spark.writeREST(df, connName, endpoint=None, method='POST', chunkSize=1000,
    options=None)
spark.readRESTArrow(connName, endpoint=None, method=None, params=None, data=None,
    options=None, columns=None)
spark.writeRESTArrow(df, connName, endpoint=None, method='POST', chunkSize=1000,
    options=None)

# SALESFORCE
spark.executeSalesforce(connName, sqlQuery, includeDeleted=False)
spark.readSalesforce(connName, sqlQuery, includeDeleted=False)
spark.writeSalesforce(df, connName, objectName, operation='insert', batchSize=2000,
    useBulk=False, externalIdField='Id', dropNulls=False, useSerial=False,
    raiseOnError=True)
spark.executeSalesforceArrow(connName, sqlQuery, includeDeleted=False, columns=None,
    options=None)
spark.readSalesforceArrow(connName, sqlQuery, includeDeleted=False, columns=None,
    options=None)
spark.writeSalesforceArrow(df, connName, objectName, operation='insert', batchSize=200,
    externalIdField='Id', options=None, **legacyOpts)

# HUBSPOT
spark.readHubSpot(connName, objectType, properties=None, limit=None, offset=None,
    filters=None, searchTerm=None, rateLimit=10)
spark.writeHubSpot(df, connName, objectType, operation='create', batchSize=100,
    rateLimit=10)
spark.readHubSpotArrow(connName, objectType, properties=None, limit=None, filters=None,
    searchTerm=None, rateLimit=None, columns=None, options=None)
spark.writeHubSpotArrow(df, connName, objectType, operation='create', batchSize=100,
    rateLimit=None, idProperty=None, options=None)

# SERVICENOW
spark.readServiceNow(connName, table, query=None, limit=1000, displayValue=True,
    options=None)
spark.executeServiceNow(connName, sourceQuery, limit=1000, options=None)
spark.writeServiceNow(df, connName, table, method='POST', batchSize=100)
```

```

spark.readServiceNowArrow(connName, table, query=None, limit=None, displayValue=None,
    columns=None, options=None)
spark.executeServiceNowArrow(connName, sourceQuery, columns=None, options=None)
spark.writeServiceNowArrow(df, connName, table, operation='create', idColumn='sys_id',
    options=None)
# SHOPIFY
spark.readShopify(connName, resource, params=None, maxResults=None,
    shopifyTier='enterprise', rateLimit=None, batchSize=250)
spark.writeShopify(df, connName, resource, operation='create', batchSize=50,
    shopifyTier='enterprise', rateLimit=None)
spark.readShopifyArrow(connName, resource, params=None, maxResults=None, rateLimit=None,
    batchSize=250, recordsPath=None, columns=None, options=None)
spark.writeShopifyArrow(df, connName, resource, operation='create', batchSize=50,
    wrapperKey=None, options=None)
# ZOHOCRm
spark.readZohoCRM(connName, module, perPage=200, maxPages=None)
spark.executeZohoCRM(connName, module)
spark.readZohoCRMArrow(connName, module, perPage=None, maxPages=None, columns=None,
    options=None)
spark.executeZohoCRMArrow(connName, module, columns=None, options=None)
# M365BC
spark.readM365BC(connName, entity, operation='get_all', params=None)
spark.writeM365BC(df, connName, entity, operation='create', batchSize=100)
spark.readM365BCArrow(connName, entity, params=None, columns=None, options=None)
spark.writeM365BCArrow(df, connName, entity, operation='create', batchSize=100,
    options=None)
# SPLUNK
spark.executeSplunk(connName, searchQuery, operation='get_all', params=None,
    requestTimeout=30)
spark.writeSplunk(df, connName, batchSize=100, index=None, source=None, sourcetype=None,
    hecUrl=None, hecHost=None, hecPort=None, hecPath='/services/collector/event',
    requestTimeout=30, requestChannel=None)
spark.executeSplunkArrow(connName, searchQuery, earliestTime=None, latestTime=None,
    requestTimeout=None, columns=None, options=None)
spark.writeSplunkArrow(df, connName, index=None, source=None, sourcetype=None,
    hecUrl=None, hecHost=None, hecPort=8088, hecPath='/services/collector/event',
    requestChannel=None, options=None)

```

Vector stores

TEXT

```

# PINECONE
spark.readPinecone(connName, ids_to_fetch=None, schema=None, options=None)
spark.writePinecone(df, connName=None, options=None, mode='upsert')
# QDRANT
spark.readQdrant(connName, collectionName=None, schema=None, options=None)
spark.writeQdrant(df, connName, collectionName=None, options=None)
# WEAVIATECLOUD
spark.createWeaviateCloudClass(connName, className, properties, vectorizer='none',
    moduleConfig=None)
spark.readWeaviateCloud(connName, className, properties=None, where_filter=None,
    limit=None, batchSize=10000)
spark.writeWeaviateCloud(df, connName, className, batchSize=100, maxPartitions=None,
    idColumn=None, vectorColumn=None, mode='append', options=None, writer='spark')
spark.listWeaviateCloudClasses(connName)
spark.deleteWeaviateCloudObject(connName, className, objectId)

```

Model Context Protocol

TEXT

```
# MCP
spark.listMcpTools(connName)
spark.callMcpTool(connName, toolName, arguments=None)
spark.listMcpResources(connName)
spark.readMcpResource(connName, uri)
spark.listMcpPrompts(connName)
spark.getMcpPrompt(connName, promptName, arguments=None)
spark.completeWithMcpTool(llmProvider, llmConnName, mcpConnName, df, toolName,
    arguments=None, argumentsColumn=None, promptColumn='prompt', promptTemplate=None,
    mcpPromptTemplate=None, systemPrompt=None, systemPromptColumn=None,
    outputColumn='llm_response', mcpResultColumn='mcp_tool_result',
    mcpErrorColumn='mcp_tool_error', model=None, temperature=0.0, maxTokens=None,
    responseFormat=None, continueOnError=False, timeout=120, llmOptions=None)
spark.getMcpLlmConfig(connName, provider='openai', serverName=None,
    requireApproval='always', includeSecrets=False)
```

2.5 Installed packages

Packages marked "pinned" have a fixed version. The others install at their latest version when the runtime is built. To read the exact version in your session, run `pip show <name>` in a notebook.

2.5.1 Python

Package	Version
Python	3.12
numpy	2.3.5 (pinned)
pandas	2.2 or later, below 3
pyspark	3.5.6 (pinned)
scikit-learn	1.5.2 (pinned)
duckdb	1.5.5 (pinned)
torch, torchvision (CPU)	2.14.0, 0.29.0 (pinned)
mlflow	2.20.4 (pinned)

2.5.2 Data, SQL and cloud libraries

Package	Purpose
<code>SQLAlchemy</code> , <code>psycopg2-binary</code> , <code>pymysql</code> , <code>pyodbc</code> , <code>cx-Oracle</code>	Database access
<code>jaydebeapi</code>	JDBC from Python
<code>snowflake-connector-python</code> , <code>simple-salesforce</code> , <code>hubspot-api-client</code>	Warehouse and SaaS clients
<code>cassandra-driver</code> , <code>redis</code> , <code>azure-eventhub</code> , <code>influxdb3-python</code>	Store and stream clients
<code>pinecone</code> , <code>msgpack</code> , <code>paramiko</code>	Vector store, serialization, SFTP
<code>boto3</code> , <code>urllib3</code> , <code>requests</code> , <code>httpx</code> , <code>httpcore</code>	AWS and HTTP
<code>pyiceberg[s3fs,sql-postgres,sql-sqlite,pyarrow]</code>	Iceberg from Python
<code>sparkmeasure</code>	Spark metrics

2.5.3 LLM and AI libraries

Package	Purpose
<code>openai</code> , <code>anthropic</code> , <code>cohere</code> , <code>google-generativeai</code>	Provider SDKs
<code>huggingface-hub</code>	Model hub client
<code>mcp</code>	Model Context Protocol, 1.29 up to 2.0

Package	Purpose
<code>qualipilot[gz,linking]</code>	Data quality and record linkage, 3.5.0 (pinned)

2.5.4 Document extraction

Package	Version
<code>docling[rapidocr]</code>	2.116.0 (pinned)
<code>rapidocr</code>	3.9.2 (pinned)
<code>onnxruntime</code>	1.28.0 (pinned)
<code>pdfplumber</code> , <code>pypdf</code> , <code>pdf2image</code> , <code>pytesseract</code> , <code>pillow</code> , <code>openpyxl</code>	0.11, 4.2, 1.17, 0.3, 10.0, 3.1 or later

The layout, table-structure and OCR models come with the runtime, so PDF extraction works without internet access.

2.5.5 Notebook and plotting libraries

`matplotlib`, `jupyterlab`, `ipython`, `ipykernel`. The Groundzero helper packages (`GZ` and others) are already importable.

2.5.6 JVM libraries

Library	Version
Spark	3.5.6
Iceberg Spark runtime and AWS bundle	1.11.0
Delta Lake	3.3.2
Hudi Spark bundle	1.1.1
AWS SDK v2 bundle	2.25.70
AWS SDK v1 bundle	1.12.781

Connector drivers (Snowflake, MongoDB, Kafka, Cassandra, Elasticsearch, BigQuery, JDBC drivers and others) are preinstalled and loaded for the connection types your run uses.

2.5.7 System packages

OpenJDK 17, the AWS CLI, ODBC (`unixodbc`) and time-zone data, so timestamps with a time zone work with Arrow.

2.6 Limits and troubleshooting

2.6.1 Limits and quotas

Limit	Value
Workers per session	1 minimum (default), 10 maximum
Worker scale-up threshold	70 percent CPU or memory
Executor memory	65 percent of worker memory, at most worker memory minus 1 GiB
Idle executor release	120 seconds
Rows collected for a preview	100,000
Dataset page size	Default 1,000, maximum 10,000
Dataset run wait	300 seconds
Session kept after its last run	280 seconds
Session inactivity shutdown	300 seconds
Network and broadcast timeout	600 seconds
Agent invocation wait	15 minutes
Agent corpus documents, calls	10,000, 50,000 by default
Agent corpus workers	1 to 32, default 4
LLM request timeout	120 seconds by default
LLM retries	3, for HTTP 429 and 5xx

What the limits mean in practice:

- **Dataset runs are bounded.** A dataset run must answer within 300 seconds. Move long transformations into a scheduled job and let the dataset read the result.
- **Sessions end.** A session is released after about 5 minutes of inactivity. Never keep state in the driver's memory that you cannot rebuild.
- **Previews are samples.** The preview holds at most 100,000 rows. Write the complete result to a table or file when you need all rows.
- **Stop times end the session.** When a schedule's stop time is reached, the session ends and aborts running work.

2.6.2 Restrictions in job scripts

Job scripts run with restrictions. Notebook kernels do not apply these rules.

Area	Rule
Imports	<code>socket</code> , <code>urllib</code> , <code>http</code> , <code>ssl</code> , <code>threading</code> , <code>multiprocessing</code> , <code>pickle</code> , <code>shutil</code> , <code>pathlib</code> , <code>glob</code> , <code>gzip</code> , <code>zipfile</code> , <code>sqlite3</code> , <code>signal</code> , <code>asyncio</code> and similar modules cannot be imported
Processes	<code>subprocess</code> , <code>os.system</code> , <code>os.popen</code> , <code>os.exec*</code> , <code>os.spawn*</code> and <code>os.fork</code> are blocked
Files	<code>open</code> works under <code>/tmp</code> , <code>/home/workspace</code> and <code>/home</code> ; other system paths are refused
Credentials	Secret values in <code>os.environ</code> show as <code>****</code> ; changes stay private to your script
Dynamic code	<code>eval</code> , <code>exec</code> , <code>compile</code> , <code>input</code> and <code>breakpoint</code> are blocked
AWS	<code>boto3</code> clients cannot be created; use the connectors or <code>getClient()</code>
Exit	<code>sys.exit()</code> is blocked

Use connectors for network and storage access, and finish a script by assigning `outputDF` rather than calling `exit`.

2.6.3 Things to know

- **Spark settings apply once.** In a reused session (notebook, shared dataset session) `sparkSession(..., **settings)` returns the existing session and ignores new settings. Restart the session to change them.
- **`collect()` has no result limit.** A very large `collect()` exhausts the driver's memory.
- **`coalesce=True` writes through a single task.** Use it for small outputs only.
- **A lakehouse catalog belongs to one run.** Do not keep a catalog name from an earlier run (see `belongs to another execution` below).
- **Unpinned packages move.** The TensorFlow variant and any package without a pinned version can change when the runtime is rebuilt.
- **Streaming into a lakehouse table needs its partitioning first.** Create the table with its partition spec, then stream into it.

2.6.4 Troubleshooting

Each entry gives the message exactly as you see it, then the cause and the fix.

Symptom	Cause	Fix
<code>AttributeError: 'sparkSession' object has no attribute 'read'</code>	You used <code>spark.read</code> , <code>spark.write</code> or <code>spark.sparkContext</code> .	Use the connector methods, for example <code>spark.readS3(...)</code> , or <code>DataFrame</code> operations.
<code>ImportError: Import of 'x' is not allowed for security reasons</code>	The module is blocked in job scripts.	Use a connector for the same task, or run the code in a notebook.
<code>PermissionError: os.system() is not allowed (and similar for subprocess)</code>	Job code cannot start processes.	Use library calls instead.

Symptom	Cause	Fix
<code>PermissionError: Access to '/data/x' outside sandbox-safe directories is not allowed</code>	The script wrote outside the allowed folders.	Write temporary files under <code>/tmp</code> or <code>/home/workspace</code> . Use S3 or a lakehouse table for anything that must outlive the run.
<code>ImportError: Direct access to 'boto3' is restricted</code>	A job script created a <code>boto3</code> client.	Use <code>spark.readS3</code> , <code>spark.writeS3</code> or <code>getClient()</code> .
<code>ValueError: No executable code found.</code>	The job has no code.	Save the script and run again.
<code>ValueError: siteName is required for GZLAKEHOUSE connection 'x'.</code>	Groundzero could not work out the site.	Pass <code>siteName='<site>'</code> to the method or to <code>sparkSession</code> .
<code>ValueError: REST catalog writes only support mode='append' or mode='overwrite'; got 'x'.</code>	The write mode is not allowed for lakehouse tables.	Change mode to <code>append</code> or <code>overwrite</code> .
<code>ValueError: database is required for REST catalog writes</code>	No database was given.	Pass <code>database=</code> or set a default database on the connection.
<code>PermissionError: GZLAKEHOUSE connection 'x' is external and read-only; ...</code>	External lakehouse connections only allow <code>SELECT</code> , <code>SHOW</code> , <code>DESCRIBE</code> and <code>EXPLAIN</code> .	Write to an internal connection.
<code>ValueError: partitionBy is not supported for streaming GZLAKEHOUSE writes; ...</code>	<code>partitionBy</code> was passed to a streaming write.	Create the table with its partition spec, then stream into it.
<code>RuntimeError: GZLAKEHOUSE catalog 'x' belongs to another execution.</code>	You used a catalog name saved from an earlier run.	Get the current one with <code>spark.getLakehouseCatalog(...)</code> or use <code>executeQuery</code> .
<code>ValueError: Connection x not found or is not a JDBC type.</code>	The name is wrong or the connection is not a JDBC type.	Check the connection name and type.
<code>TypeError: Use writeSnowflake() for connection x.</code>	<code>writeJDBC</code> was used on a Snowflake connection.	Call <code>writeSnowflake</code> .
<code>Exception: Error reading file from S3: ...</code>	The text after the colon is the cause (missing key, permission, malformed file).	Check the path and the connection's bucket.
<code>ValueError: Missing columns: [...]. Available: [...]</code>	A column named in <code>promptColumn</code> , <code>promptTemplate</code> or <code>systemPromptColumn</code> is not in the <code>DataFrame</code> .	Fix the column name.
<code>ValueError: AWS Bedrock model must be set in the connection or call.</code>	No model was given.	Pass <code>model=</code> (the model ID) or set a model on the connection.

Symptom	Cause	Fix
The dataset session stopped responding before it took the run. Run the dataset again to start a new session.	The session ended before your run started.	Run the dataset again. A new session starts.
<code>TimeoutError</code> from <code>invokeAgent</code>	The wait for the Agent ended after 15 minutes. The Agent run is not cancelled.	Look up the run ID in the error and inspect that run before you submit again.
<code>ErrorType: message at line N</code>	An error in your script. <code>N</code> is the line in your code, and the failing line is printed below.	Fix that line.
A job is slow to start	A new session starts a driver and at least one worker.	Keep a notebook or dataset warm, raise the minimum workers for large jobs, or schedule heavy work ahead of time.

2.6.5 FAQ

Do I need to create the SparkSession myself? Call `sparkSession(appName)`. It returns the shared session with all connectors attached. A bare PySpark session has no connectors, credentials or governance.

How do I see the result of my job? Assign a DataFrame to `outputDF`. The preview and its schema appear in the UI. Anything you `print` appears in the log.

Can I use pandas? Yes. Assign a pandas DataFrame to `outputDF`. `toPandas()` works for small results. For large data, stay in Spark.

Where do my credentials come from? From the connection. Code uses only the connection name, and secret values are hidden from your script.

How do I choose how many workers I get? Set Minimum Workers in the runtime picker. Groundzero adds workers, up to 10, when CPU or memory use passes 70 percent.

Why did my Spark settings not apply? The session already existed. Settings passed to `sparkSession` are used only when the session is created.

Can two users share a dataset session? Yes. Runs on the same dataset share a pool of sessions, one run per session at a time. Lakehouse permissions apply to each user, and the session clears the previous user's views.

How do I add a library? Use `%pip install` in a notebook for a trial. For permanent packages, ask an administrator to create a custom runtime based on a Spark runtime.

Which variant do I need for machine learning? For scikit-learn or boosting libraries, the scikit-learn variant. For deep learning with PyTorch, the PyTorch variant. For TensorFlow or spaCy, the TensorFlow variant.

Can I read a table I cannot write? Yes. External lakehouse connections are read-only, and per-table permissions decide who can query or write a table.

What happens to my notebook files? Your notebook is saved each time you save it.

How are runs recorded? Each run records status, logs, Spark metrics, audit entries for lakehouse access and lineage from every connector read and write.

2.7 Examples gallery

All examples use fictional names and data. Replace connection names (`acme-*`) with your own. Each example is a complete job script.

2.7.1 Daily revenue rollup in the lakehouse

Reads yesterday's orders from a lakehouse table, aggregates them by region and appends the result to a summary table. The aggregate is also shown as the job preview.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("DailyRevenueRollup", siteName="acme")

orders = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="""
        SELECT region, product, quantity, unit_price
        FROM sales.orders
        WHERE order_date = date_sub(current_date(), 1)
    """,
)

daily = (
    orders.withColumn("revenue", F.col("quantity") * F.col("unit_price"))
    .groupBy("region")
    .agg(
        F.sum("revenue").alias("revenue"),
        F.countDistinct("product").alias("products"),
    )
    .withColumn("report_date", F.date_sub(F.current_date(), 1))
)

spark.writeGzLakehouse(
    df=daily,
    connName="acme-lakehouse",
    tableName="revenue_daily",
    database="sales",
    mode="append",
)

outputDF = daily.orderBy(F.desc("revenue"))
```

2.7.2 Incremental load from PostgreSQL to Parquet

Pulls rows changed in the last day, stamps them with the run's name and start time through placeholders, and appends them to Parquet in S3.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("IncrementalCustomers")

run_job = {JOBNM}.strip("'")
run_started = {STRTTM}.strip("'")

changes = spark.readJDBC(
    connName="acme-postgres",
    query="SELECT customer_id, email, country, updated_at "
    "FROM crm.customers WHERE updated_at ≥ now() - interval '1 day'",
)

landed = (
    changes.withColumn("loaded_by", F.lit(run_job))
    .withColumn("loaded_at", F.lit(run_started))
)

spark.writeS3(
    df=landed,
    connName="acme-s3",
    filePath="landing/crm/customers",
    format="parquet",
    mode="append",
)

print("Loaded", landed.count(), "changed customers")
outputDF = landed.groupBy("country").count()
```

The placeholders are substituted as quoted string literals, so `.strip("'")` removes the quotes that are meant for SQL.

2.7.3 Deduplicate and enrich with window functions

Keeps the latest record per customer and joins a country reference table read from CSV.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import Window
from pyspark.sql import functions as F

spark = sparkSession("DedupeCustomers")

customers = spark.readS3(
    connName="acme-s3",
    fileName="landing/crm/customers_raw/",
    format="csv",
    options={"header": "true", "inferSchema": "true"},
)
countries = spark.readS3(
    connName="acme-s3",
    fileName="reference/countries.csv",
    format="csv",
    options={"header": "true"},
)

latest = Window.partitionBy("customer_id").orderBy(F.desc("updated_at"))
deduped = (
    customers.withColumn("rank", F.row_number().over(latest))
    .where(F.col("rank") == 1)
    .drop("rank")
    .join(F.broadcast(countries), on="country_code", how="left")
)

spark.writeS3(
    df=deduped,
    connName="acme-s3",
    filePath="curated/customers",
    format="parquet",
    mode="overwrite",
)
outputDF = deduped
```

2.7.4 Stream Kafka events into an Iceberg table

Consumes a topic and appends micro-batches to a lakehouse table. With the `availableNow` trigger the job processes everything that has arrived and then finishes, which suits scheduled runs.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("OrderEventsStream", siteName="acme")

raw = spark.readKafka(
    connName="acme-kafka",
    topic="orders.created",
    startingOffsets="earliest",
)

events = raw.select(
    F.col("key").cast("string").alias("order_key"),
    F.get_json_object(F.col("value").cast("string"), "$.region").alias("region"),
    F.get_json_object(F.col("value").cast("string"), "$.total")
    .cast("double")
    .alias("total"),
    F.col("timestamp").alias("event_time"),
)

query = spark.writeGzLakehouse(
    df=events,
    connName="acme-lakehouse",
    tableName="order_events",
    database="sales",
    streaming=True,
    checkpointLocation="s3://acme-data/checkpoints/order_events",
    outputMode="append",
    trigger={"availableNow": True},
)

query.awaitTermination()
```

CREATE THE TABLE FIRST

The target `sales.order_events` table must already exist with the partition specification you want, because the streaming writer does not apply `partitionBy`.

2.7.5 Classify support tickets with an LLM

Calls an LLM connection once per row, keeps going when individual rows fail, and saves successes and failures separately.

PYTHON

```

from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("TicketTriage")

tickets = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT ticket_id, subject, body FROM support.tickets LIMIT 500",
    siteName="acme",
)

scored = spark.complete(
    connName="acme-openai",
    df=tickets,
    promptTemplate=(
        "Classify this support ticket as billing, technical or other. "
        "Reply with one word.\nSubject: {{subject}}\nBody: {{body}}"
    ),
    systemPrompt="You are a precise ticket router.",
    outputColumn="category",
    maxTokens=5,
    continueOnError=True,
)

good = scored.where(F.col("category_status") == "success")
bad = scored.where(F.col("category_status") == "error")

spark.writeS3(good, "acme-s3", "triage/ok", "parquet", "overwrite")
spark.writeS3(bad, "acme-s3", "triage/errors", "parquet", "overwrite")

outputDF = good.groupBy(F.lower(F.trim("category")).alias("category")).count()

```

2.7.6 Train and apply a scikit-learn model

Trains on a sample collected to the driver, then scores the full DataFrame in parallel with a pandas UDF. Run it on the scikit-learn (or another ML) variant, or on the base runtime, since scikit-learn is part of the base runtime.

PYTHON

```

import pandas as pd
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import pandas_udf
from sklearn.ensemble import GradientBoostingClassifier

spark = sparkSession("ChurnModel")

features = ["tenure_months", "monthly_spend", "support_tickets"]
customers = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT customer_id, tenure_months, monthly_spend, "
    "support_tickets, churned FROM crm.customer_features",
    siteName="acme",
)

sample = customers.sample(fraction=0.2, seed=7).limit(50_000).toPandas()
model = GradientBoostingClassifier(random_state=7)
model.fit(sample[features], sample["churned"])

@pandas_udf("double")
def churn_probability(*cols: pd.Series) → pd.Series:
    frame = pd.concat(cols, axis=1)
    frame.columns = features
    return pd.Series(model.predict_proba(frame)[: , 1])

scored = customers.withColumn(
    "churn_probability", churn_probability(*[F.col(c) for c in features])
)
outputDF = scored.select("customer_id", "churn_probability").orderBy(
    F.desc("churn_probability")
)

```

2.7.7 Load a warehouse query into the lakehouse in bulk

Verifies the lakehouse route, lets Snowflake unload the rows directly into the table, then counts them.

PYTHON

```

from GZ.SparkUtils import sparkSession

spark = sparkSession("OrdersBulkLoad", siteName="acme")

check = spark.verifyLakehouseConnection("acme-lakehouse", siteName="acme")
print("Routing to warehouse", check["warehouse"], "at site", check["site_used"])

summary = spark.bulkLoadGzLakehouseFromSnowflake(
    connName="acme-snowflake",
    query="SELECT * FROM analytics.orders WHERE order_date ≥ '2026-01-01'",
    lakehouseConn="acme-lakehouse",
    tableName="orders_history",
    database="sales",
    mode="overwrite",
    warehouseSize="X-Large",
    siteName="acme",
)
print("Bulk load summary:", summary)

outputDF = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT COUNT(*) AS loaded_rows FROM sales.orders_history",
)

```

2.7.8 Extract structured data from a folder of PDFs with an Agent

Runs an approved Groundzero Agent over every PDF in a prefix and reads the results back as a DataFrame.

PYTHON

```

from GZ.SparkUtils import sparkSession

spark = sparkSession("InvoiceExtraction")

result = spark.invokeAgentPDFCorpus(
    connName="acme-agents",
    agentId="123e4567-e89b-42d3-a456-426614174000",
    message="Extract the invoice number, vendor and total into the schema.",
    pdfSourceConnName="acme-s3",
    pdfSourcePrefix="incoming/invoices/",
    pdfSinkConnName="acme-s3",
    pdfSinkPrefix="agent-pdf-results/invoices",
    summaryWorkers=8,
)
print(result["status"], result["outputMode"])
print("Processed", result["processedDocumentCount"], "documents")

outputDF = spark.readS3(
    connName="acme-s3",
    fileName="agent-pdf-results/invoices/",
    format="json",
)

```

2.7.9 Quality gate with notification

Runs checks on a table, writes the HTML report, and mails the team when a column is out of range.

PYTHON

```

from GZ.SendEmail import sendEmail
from GZ.SparkUtils import sparkSession
from qualipilot import CheckConfig, QualipilotConfig
from qualipilot.models.config import ColumnRange

spark = sparkSession("OrdersQualityGate")

orders = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT order_id, customer_email, amount FROM sales.orders",
    siteName="acme",
)
config = QualipilotConfig(
    checks=CheckConfig(
        required_columns=["order_id", "customer_email", "amount"],
        column_ranges={"amount": ColumnRange(min=0, max=100_000)},
    )
)
report = spark.runDataQualityChecks(
    df=orders, connectionName="acme-openai", config=config
)
spark.saveDataQualityReport(report, "reports/orders_quality.html", "html")

negative = orders.where("amount < 0").count()
if negative:
    sendEmail(
        subject="Orders quality gate failed",
        body=f"{negative} orders have a negative amount.",
        to_email="data-team@example.com",
        from_email="alerts@example.com",
    )
outputDF = orders.where("amount < 0")

```

2.7.10 Orchestrate sub-jobs

A parent job runs two sub-jobs in order and then reports their outputs.

PYTHON

```

from GZ.JobUtils import executeJob
from GZ.SparkUtils import sparkSession

spark = sparkSession("NightlyPipeline")

results = executeJob(["load_orders", "build_marts"])
for name, output in results.items():
    print("Finished", name, "with output", output)

outputDF = spark.createDataFrame(
    [(name,) for name in results], ["completed_job"]
)

```

PART 3

DuckDB Runtime

Fast single-node analytics with DuckDB and Iceberg on Groundzero.

- [3.1 Overview](#)
- [3.2 Quickstart](#)
- [3.3 How-to guides](#)
- [3.4 API reference](#)
- [3.5 Installed packages](#)
- [3.6 Limits and troubleshooting](#)
- [3.7 Examples gallery](#)

The Groundzero DuckDB runtime runs your Python code next to the DuckDB analytics engine. It is the fastest way to join, filter and aggregate data that fits on one machine. You read from any Groundzero connection into DuckDB relations, combine them with SQL, and either return the result (datasets, previews) or write it back to storage and databases (jobs).

This guide shows how to run your first job, dataset and notebook, how to read and write data, and how to use the Groundzero helper functions. It also covers the installed packages, limits, troubleshooting and a gallery of runnable examples.

WHO THIS GUIDE IS FOR

Developers writing datasets, ETL jobs and notebooks on Groundzero. You should be comfortable with Python and SQL. All examples use fictional data and connection names such as `sales_s3` and `orders_pg`; replace them with the names of your own connections.

3.1 Overview

3.1.1 What the DuckDB runtime gives you

A Groundzero *runtime* is a software environment plus a compute size. When you run a dataset, job or notebook, Groundzero starts the runtime you selected with the connections your code needs, runs your code, and returns the output or logs.

The DuckDB runtime includes:

- Python 3.12 with pandas, NumPy and PyArrow.
- DuckDB 1.5.5 with the `iceberg`, `httpfs`, `aws`, `postgres`, `mysql`, `azure`, `json`, `parquet`, `spatial` and `motherduck` extensions already installed, so nothing is downloaded when your code runs.
- The Groundzero helper library (`GZ`). `duckSession()` returns a session object that wraps a DuckDB connection and adds one read and write method per connector (S3, JDBC databases, Snowflake, BigQuery, the Groundzero Lakehouse, REST APIs, vector stores, LLM providers and more).
- PDF text extraction (Docling with RapidOCR), the Qualipilot data quality library, and clients for Groundzero Agents and MCP servers.
- JupyterLab for notebooks, with `numpy`, `pandas`, `boto3`, `getClient` and `duckSession` already imported.

DuckDB is a columnar engine that runs inside your Python process. There is no cluster to size and no copying between Python and the engine. Relations, pandas DataFrames and Arrow tables are queried in place, and work larger than memory spills to local disk.

3.1.2 Choose the DuckDB runtime

Groundzero offers three runtime choices. Pick the one that matches the shape of your workload.

	Spark	DuckDB	Standard
Engine	Apache Spark 3.5.6	DuckDB 1.5.5	Platform default
Topology	Driver plus workers	One machine, one process	No compute to choose
Scale	Distributed	Single node, spills to disk	Single SQL connection
Lakehouse	Read and write	Read only	See the Standard guide
Code	PySpark and SQL	Python, SQL, pandas, Arrow	SQL
Best for	Very large or write-heavy ETL	Fast analytics and joins	Simple SQL datasets

Choose DuckDB when:

- the data you actually process fits one machine's memory and local disk;
- you join several sources (for example a lakehouse table, an S3 file and a PostgreSQL table) in one SQL statement;
- you want a quick start for datasets and previews, because DuckDB needs no cluster;
- you want to mix SQL with pandas, scikit-learn or Arrow code.

Choose another runtime when:

- you must write to the Groundzero Lakehouse. DuckDB reads the lakehouse only, so use the Spark runtime for lakehouse writes;
- the data does not fit on one machine even after filtering and projection;
- the dataset is a single SQL query against one SQL connection and needs no code, which the Standard option covers.

3.1.3 How a run works

Every dataset, job or notebook that uses the DuckDB runtime runs in its own isolated environment. Its size comes from the compute you choose in the runtime selector (see the Quickstart).

Workload	What starts	When it stops
ETL job	One run of your job file	When the job code finishes
Dataset run or preview	A session that serves runs	After idle time, see below
Notebook	A JupyterLab session	When you stop the session

A session handles one run at a time. For datasets and previews, Groundzero keeps idle sessions ready for a short while. A new run uses an idle session if one exists and otherwise starts another, so runs never queue behind each other. An idle session is released after about five minutes. Do not keep state in memory between runs, because any run may land on a new session.

Credentials

Your code only names a connection (`connName="sales_s3"`). It never handles keys or passwords. Secret values are hidden from your code: `os.environ` shows them as `****` .

Resources

The compute you pick sets the CPU and memory of the run. DuckDB is sized from it automatically:

- threads equal the CPU count of the compute (minimum two);
- the DuckDB memory limit is 80 percent of the compute's memory (minimum 512 MB), which leaves the rest for Python, Arrow and client libraries;
- work that does not fit in memory spills to disk under `/tmp/duckdb-spill` .

To override threads and memory for one session, call `duckSession(threads=..., memory_limit=...)` .

3.1.4 Versions

The runtime is updated continuously and has no numbered releases.

Component	Version
Python	3.12
DuckDB	1.5.5
NumPy	2.3.5

Component	Version
pandas	2.2 or newer, below 3
scikit-learn	1.5.2
SciPy	1.17.1
Qualipilot (<code>gz</code> , <code>linking</code>)	3.5.0
Docling / RapidOCR	2.116.0 / 3.9.2
MLflow	2.20.4
PyTorch (CPU build)	2.14.0, torchvision 0.29.0

Packages without a version in these tables install at their latest release when the runtime is built. The complete list is in Installed packages.

3.1.5 What changed recently

Recent changes that affect your code, newest first:

- Lakehouse reads in long-running sessions no longer fail with expired storage tokens.
- Lakehouse access policies, including column masking, apply to the user who runs the code. Cached rows are kept per user policy.
- Lineage records schemas for connector reads and writes.
- DuckDB threads and memory are sized from your compute, and each dataset session uses the full compute for one run at a time.
- Time zone data is included, so time-zone-aware timestamps (including `UTC`) work.
- Python 3.12 is the base for every runtime, including the ML variants.

3.1.6 Machine-learning variants

Three variants add machine learning libraries on top of the DuckDB runtime. They share the same `gz` library, extensions and behavior.

Variant	Adds
DuckDB scikit-learn	OpenCV (headless), XGBoost, LightGBM, CatBoost
DuckDB PyTorch	PyTorch 2.5.1 and torchvision 0.20.1 (CUDA 12.1), plus the scikit-learn additions, SHAP, LIME, Alumentations, Hugging Face Hub, Transformers, simsimd and fastai
DuckDB TensorFlow	TensorFlow, OpenCV, XGBoost, LightGBM, CatBoost, SHAP, LIME, NLTK, spaCy, h5py

scikit-learn, NumPy, SciPy and pandas are in every variant. The PyTorch variant uses the CUDA 12.1 build of PyTorch, so it can use an NVIDIA GPU when you set a GPU size.

If a library is in none of the variants, ask an administrator to create a custom runtime (see Use extra packages).

3.2 Quickstart

This section takes you from nothing to a working job, dataset and notebook. You need a Groundzero site with at least one connection to a data source. The examples use an Amazon S3 connection named `sales_s3` whose bucket, `acme-retail-data`, holds Parquet files of orders.

The orders files in these examples have the following columns.

Column	Type	Meaning
<code>order_id</code>	BIGINT	Unique order number
<code>customer_id</code>	BIGINT	Customer who placed the order
<code>region</code>	VARCHAR	Sales region, for example <code>EMEA</code>
<code>amount</code>	DOUBLE	Order value
<code>status</code>	VARCHAR	<code>open</code> , <code>shipped</code> or <code>returned</code>
<code>ordered_at</code>	TIMESTAMP	When the order was placed

3.2.1 Your first job

An ETL job is a Python file that runs once. This job reads one day of orders from S3, computes revenue per region and writes the result back to S3 as Parquet.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()

orders = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/orders/2026-09-30.parquet",
    fileType="parquet",
)
session.register("orders", orders)

revenue_sql = """
    SELECT region,
           count(*)           AS orders,
           round(sum(amount), 2) AS revenue
    FROM orders
    WHERE status <> 'returned'
    GROUP BY region
    ORDER BY revenue DESC
"""

print(session.query(revenue_sql, returnFormat="pandas"))

session.writeS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="curated/daily_revenue/2026-09-30.parquet",
    fileType="parquet",
    sourceQuery=revenue_sql,
)
session.close()

```

What happened:

1. `duckSession()` created a DuckDB connection and attached every connector method to the returned session object.
2. `readS3File` configured S3 access from the `sales_s3` connection and returned a lazy DuckDB relation. No data was downloaded yet.
3. `register` exposed the relation to SQL under the name `orders`.
4. `query` ran the aggregation. With `returnFormat="pandas"` the result is materialized as a DataFrame. The `print` output appears in the job's log.
5. `writeS3File` ran a `COPY ... TO` that streamed the result to S3.

To check it worked, open the job log for the printed table, then look for the new Parquet file under `curated/daily_revenue/` in the bucket.

KEEP DATA IN DUCKDB FOR AS LONG AS POSSIBLE

Relations are lazy. Convert to pandas or Arrow only for the final step or when a library requires it. Doing joins and filters in SQL avoids copying data into Python memory.

3.2.2 Your first dataset

A dataset runs your code and returns rows to the Groundzero UI, dashboards and APIs. Instead of writing files, you leave the rows you want to return in a variable called `outputDF` (or pass them to `gz_output()`).

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

orders = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/orders/2026-09-30.parquet",
    fileType="parquet",
)
session.register("orders", orders)

outputDF = session.query(
    """
    SELECT region, status, count(*) AS orders, sum(amount) AS revenue
    FROM orders
    GROUP BY region, status
    ORDER BY region, status
    """
)
```

`outputDF` can be a DuckDB relation, a PyArrow table, record batch or reader, a pandas DataFrame, or a list of dictionaries. Groundzero converts it to JSON rows plus a schema, returns at most 100,000 rows, and pages the result for the UI.

3.2.3 Your first notebook

In a notebook the runtime pre-imports `numpy as np`, `pandas as pd`, `boto3`, `getClient` and `duckSession`, so the first cell can start working immediately:

PYTHON

```
session = duckSession()
orders = session.readS3File("sales_s3", "acme-retail-data",
                           "raw/orders/2026-09-30.parquet")

orders.limit(5).df()
```

The notebook session stays alive until you stop it, and you cannot change the runtime while it runs.

In the UI

You choose the runtime in the same control everywhere it appears. The control is a single-line field that shows the selected runtime name and, for non-Standard runtimes, a small badge with the compute name. Clicking it opens a panel with:

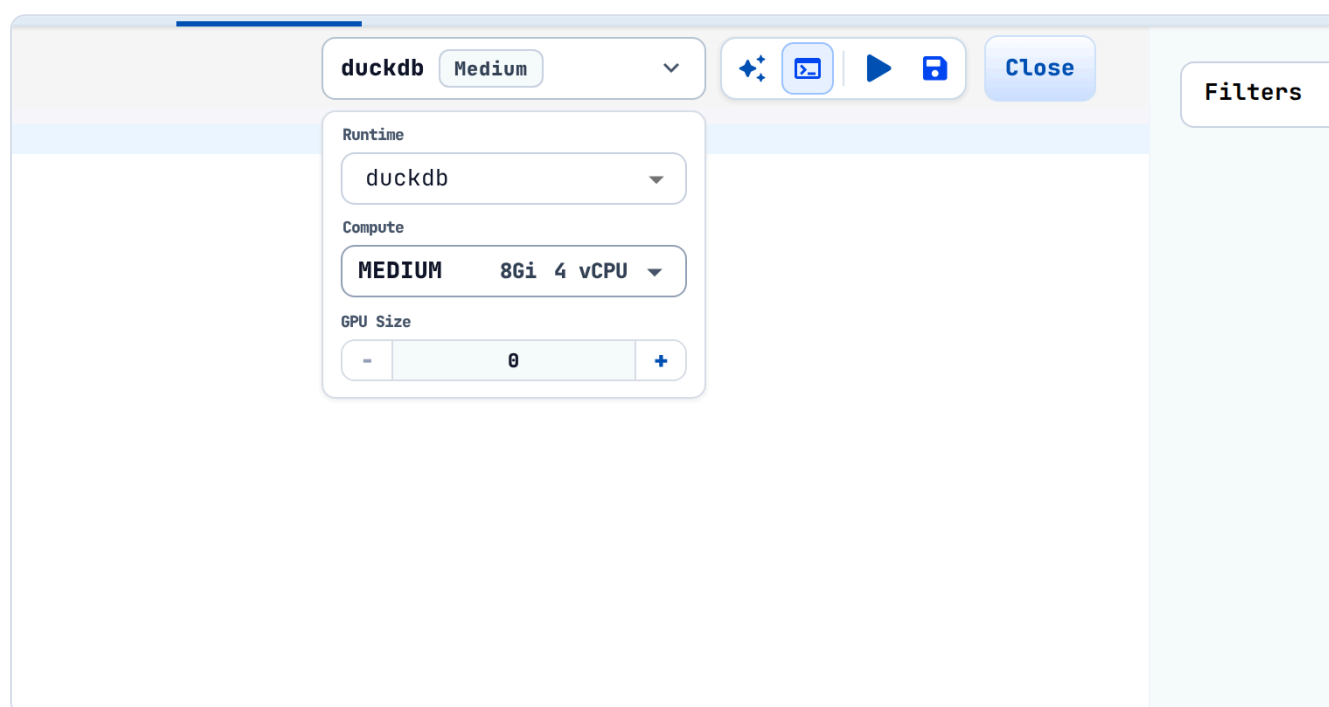
- **Runtime:** a dropdown listing the runtimes your administrators have enabled for the site. It shows `No RTEs Available` if none are enabled.
- **Compute:** a dropdown whose entries read `Name | memory | cpu`, one per compute size defined for the site. It shows `Insufficient Permission` if you cannot describe compute resources.

- **GPU Size** and, for Spark only, **Minimum Workers**: numeric steppers.

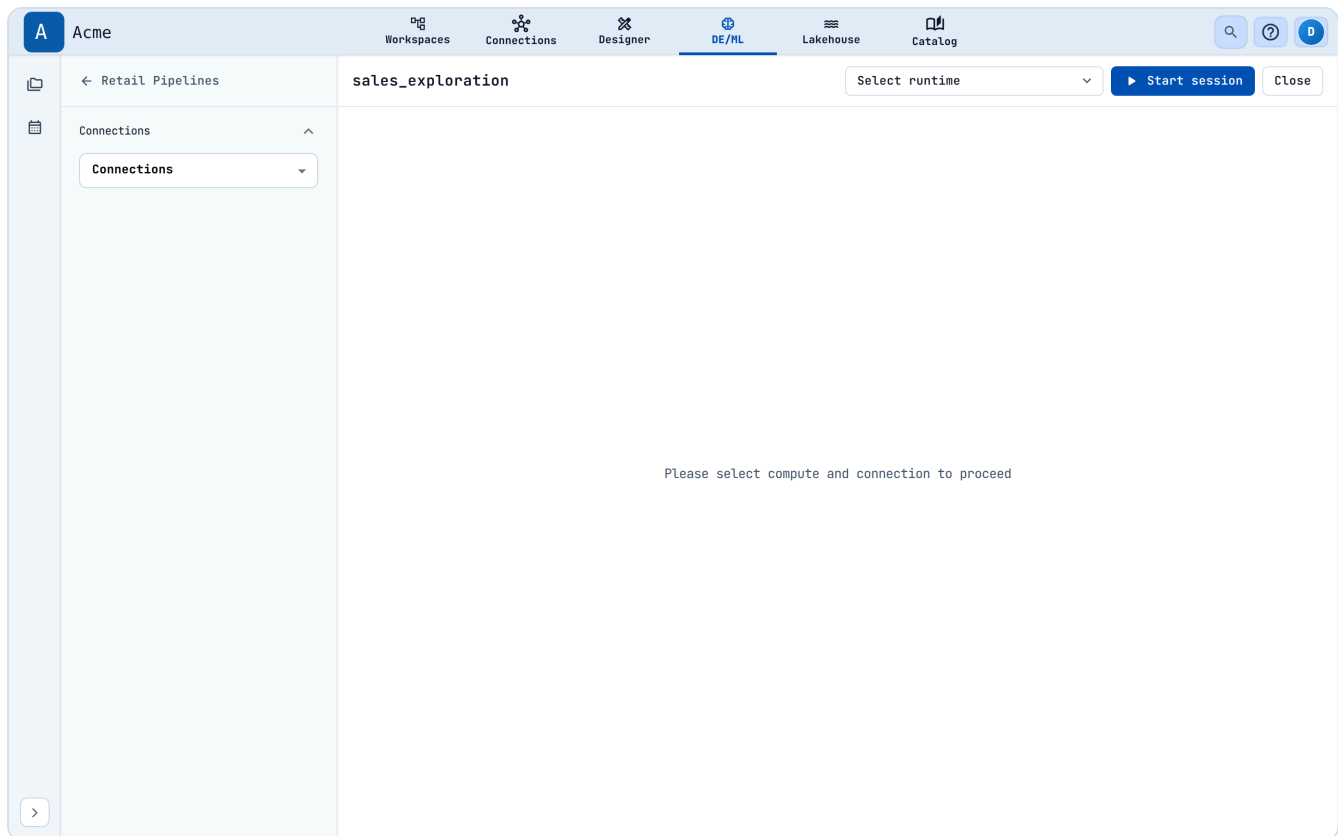
Where the control appears:

Place	Notes
Dataset designer	In the editor toolbar. Standard is offered only for no connection or a single SQL connection; DuckDB and Spark need a compute.
ETL job editor	In the toolbar next to Code . Choose DuckDB or Spark.
Notebook editor	Locked while the session runs; hovering shows <code>Stop the session to change compute</code> .
Job scheduler	A field labelled RTE that opens a Runtime and Compute menu; Save stays disabled until both are chosen.

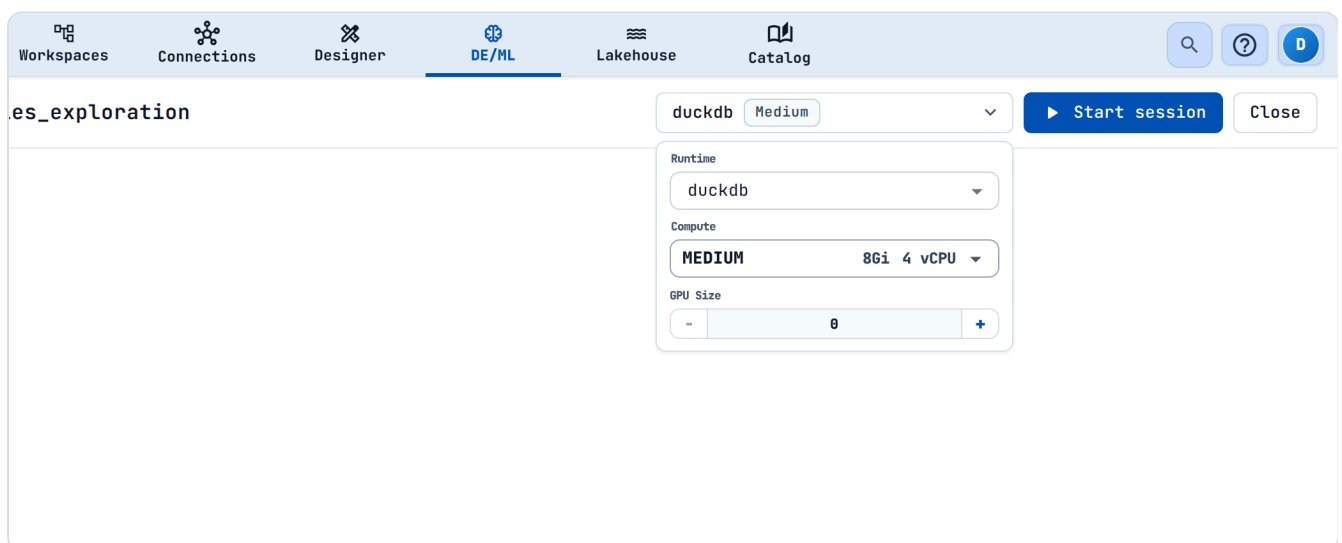
If you run a dataset without a compute, the designer shows `Please select Compute` or `Please select RTE and Compute`.



DuckDB selected: Compute and GPU Size, with no Minimum Workers control.



A notebook before its session starts: choose a runtime, then Start session.



Notebook runtime picker with duckdb and Medium compute.

3.3 How-to guides

3.3.1 Create and configure a session

`duckSession()` is the single entry point. It returns a `DuckSession`: a DuckDB connection plus every connector method (`readS3File`, `executeJdbc`, `readGzLakehouse` and so on).

PYTHON

```
from GZ.DuckDBUtils import duckSession

# Defaults: in-memory database, threads and memory sized from your compute.
session = duckSession()

# Persist the database to a file, pin resources, set DuckDB options.
session = duckSession(
    duckdbPath="/tmp/work.duckdb",
    logLevel="WARNING",
    threads=4,
    memory_limit="6GB",
    http_timeout="120",
)
```

Any extra keyword argument becomes a DuckDB `SET name='value'` statement, so every documented DuckDB setting is available. Settings that fail are logged as warnings and skipped.

A session is also a context manager, which closes the connection and releases connector clients when the block ends:

PYTHON

```
from GZ.DuckDBUtils import duckSession

with duckSession(logLevel="WARNING") as session:
    session.query("CREATE TABLE t AS SELECT range AS n FROM range(5)")
    print(session.query("SELECT sum(n) AS total FROM t", returnFormat="pandas"))
```

ONE SESSION PER RUN

Create the session once at the top of your code and reuse it. Each call to `duckSession()` opens a new DuckDB connection with its own tables, secrets and memory budget.

3.3.2 Read from connections

Every connector method follows the same conventions:

- the first argument is `connName`, the name of a Groundzero connection;
- `returnFormat` is `"duckdb"` (default, a lazy relation), `"pandas"` or `"arrow"`;
- `targetDuckdbTable`, when given, also registers the result as a table or view with that name so SQL can refer to it.

Object storage (S3, GCS, Azure Blob)

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

# One Parquet file
orders = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/orders/2026-09-30.parquet",
)

# Many files with a glob, registered under a name
session.readS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders_all",
)

# CSV with reader options; keys are DuckDB's own option names
customers = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/customers.csv",
    fileType="csv",
    readOptions={"delim": ",", "header": True},
)

print(session.query("SELECT count(*) AS n FROM orders_all",
    returnFormat="pandas"))
```

`readS3File` retries a failing read in two more modes before giving up: first with every column read as text (`all_varchar`), then with malformed rows ignored (`ignore_errors`). CSV reads default to a header row, a double quote character, null padding, relaxed strictness and a 100,000-row type-detection sample unless you override those options.

`readGcsFile(connName, bucketName, filePath, fileType, ...)` and `readAzureBlob(connName, containerName, blobName, fileType, ...)` take the same `fileType`, `readOptions`, `targetDuckdbTable` and `returnFormat` arguments.

SQL databases (JDBC)

`executeJdbc` runs a query on PostgreSQL, MySQL, Oracle, SQL Server, Db2, Redshift, CockroachDB, SAP HANA, TimescaleDB, Exasol and SQLite connections and loads the result into DuckDB. PostgreSQL, SQLite, Redshift and CockroachDB results stream in Arrow batches.

PYTHON

```
customers = session.executeJdbc(
    connName="orders_pg",
    sqlQuery="""
        SELECT customer_id, segment, signup_date
        FROM public.customers
        WHERE signup_date ≥ DATE '2025-01-01'
    """,
    targetDuckdbTable="customers",
)
```

Cloud warehouses

PYTHON

```
# Snowflake: results stream into a DuckDB table in Arrow batches.
n_rows = session.readSnowflake(
    conn_name="wh_snowflake",
    sql_query="SELECT * FROM ANALYTICS.PUBLIC.ORDERS",
    target_duckdb_table="orders_sf",
    returnFormat="row_count",
)

# BigQuery
events = session.readBigQuery(
    connName="analytics_bq",
    query="SELECT user_id, event FROM `acme.web.events` LIMIT 1000",
)
```

Note that `readSnowflake` uses snake_case argument names and requires the target table name. Because it streams one Arrow batch at a time, it can load tables far larger than Python memory.

APIs and other sources

`executeREST` calls an HTTP API configured as a REST connection and flattens the JSON records into a relation. `readKafka`, `readMongo`, `readDynamoDB`, `readSalesforce`, `readCassandra` and the rest of the connector table in the API reference follow the same pattern.

PYTHON

```
tickets = session.executeREST(
    connName="support_api",
    endpoint="/v2/tickets",
    params={"status": "open"},
    resultPath="data.items",
    maxPages=5,
)
```

3.3.3 Query across sources

Because every read returns a DuckDB relation, you can join data from different systems in one statement. Register each relation under a name, then use SQL.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
orders = session.readS3File(
    "sales_s3", "acme-retail-data", "raw/orders/2026-09-30.parquet"
)
customers = session.readS3File(
    "sales_s3", "acme-retail-data", "raw/customers.csv", fileType="csv"
)
session.register("orders", orders)
session.register("customers", customers)

top = session.query(
    """
    SELECT c.segment, count(*) AS orders, round(sum(o.amount), 2) AS revenue
    FROM orders o
    JOIN customers c USING (customer_id)
    GROUP BY c.segment
    ORDER BY revenue DESC
    """,
    returnFormat="pandas",
)
print(top)

```

`register` also accepts pandas DataFrames and Arrow tables, so you can mix DuckDB with pandas code:

PYTHON

```

import pandas as pd
from GZ.DuckDBUtils import duckSession

session = duckSession()
rates = pd.DataFrame({"region": ["EMEA", "APAC"], "fx": [1.08, 0.65]})
session.register("rates", rates)
print(session.query("SELECT * FROM rates ORDER BY region", returnFormat="pandas"))

```

3.3.4 Query the Groundzero Lakehouse

The Groundzero Lakehouse is an Apache Iceberg lakehouse. From the DuckDB runtime you read it through the lakehouse catalog. Table-level permissions, column masking and row policies are applied for the user running the code.

READ ONLY

Lakehouse access from DuckDB is read only. `writeGzLakehouse` raises `PermissionError`. To write lakehouse tables, use the Spark runtime.

Read a table or run SQL

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession(siteName="acme")

# Whole table; the name is database.table
orders = session.readGzLakehouse(
    connName="lakehouse_main",
    tableName="retail.orders",
)

# SQL: always qualify tables as database.table
top = session.readGzLakehouse(
    connName="lakehouse_main",
    query="""
        SELECT region, sum(amount) AS revenue
        FROM retail.orders
        WHERE ordered_at ≥ DATE '2026-09-01'
        GROUP BY region
    """,
    returnFormat="pandas",
)
```

Rules the runtime enforces:

- Provide `tableName` or `query`. If you give neither, the call fails with `Either 'query' or 'tableName' must be provided`.
- Only read-only statements run (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`). Anything else fails with `Only read-only queries are allowed`.
- Table references in a query must be fully qualified. An unqualified table raises `Use fully qualified table names`.
- A connection that points at an external lakehouse needs a site name: pass `siteName` to `duckSession()` (one site for all lakehouse connections) or `siteNames={"lakehouse_main": "acme"}` for a mapping, or pass `siteName` to `readGzLakehouse`. One session can use only one site per connection; to use another site, create a new session.

Filters from the UI

`readGzLakehouse` accepts structured `filters` that are added to the query as extra predicates, which keeps the query text stable while values change:

PYTHON

```
sales = session.readGzLakehouse(
    connName="lakehouse_main",
    query="SELECT * FROM retail.orders",
    filters=[
        {"column": "region", "op": "in", "values": ["EMEA", "APAC"]},
        {"column": "amount", "op": "gte", "values": [100]},
    ],
)
```

Supported operators are `in`, `eq`, `between`, `gte`, `gt`, `lte` and `lt`. A malformed filter raises an error; the query never runs unfiltered.

Keep a lakehouse table in the session

`registerLakehouseTable` reads a table once and keeps it for repeated queries. The table is named `lakehouse_<alias>`:

PYTHON

```
session.registerLakehouseTable(
    connName="lakehouse_main",
    tableName="retail.customers",
    alias="customers",
)
vip = session.query(
    "SELECT * FROM lakehouse_customers WHERE segment = 'vip' LIMIT 10"
)
```

By default the table is materialized as a physical DuckDB table. Pass `materialize=False` to register it as a view over the Arrow data instead.

Time travel and snapshots

Groundzero Lakehouse tables are Iceberg tables. Use DuckDB's Iceberg syntax in your `query` where the extension supports it.

3.3.5 Write results

DuckDB writes with `COPY`, so results stream to the destination without materializing in Python.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", "acme-retail-data", "raw/orders/2026-09-30.parquet",
    targetDuckdbTable="orders",
)
session.query("""
    CREATE TABLE summary AS
    SELECT region, count(*) AS orders FROM orders GROUP BY region
""")

# Write a DuckDB table
session.writeS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="curated/summary.parquet",
    fileType="parquet",
    sourceTable="summary",
)

# Write a query result as CSV with options
session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/curated/summary.csv",
    fileType="csv",
    sourceQuery="SELECT * FROM summary ORDER BY region",
    writeOptions={"header": True},
)
```

For databases use `writeJdbc(connName, targetTable, data=None, sourceTable=None, sourceQuery=None, mode="append")`, where `mode` is `append` or `replace`:

PYTHON

```
session.writeJdbc(
    connName="orders_pg",
    targetTable="reporting.region_summary",
    sourceTable="summary",
    mode="replace",
)
```

Most other connectors have a matching `write ...` method; see the connector table in the API reference. Supported file types for `writeS3File`, `writeGcs` and `writeAzureBlob` are `parquet`, `csv` and `json`; any other value raises `Unsupported file type`.

3.3.6 Return output and previews

When code runs as a dataset (or any run that returns rows), Groundzero looks for output in two places, in this order of priority:

1. a value passed to `gz_output(value)`; if called more than once, the last value wins;
2. a variable named `outputDF`.

If both exist, `gz_output()` is used and a warning is logged.

The accepted types and what happens to them:

Type	Handling
DuckDB relation	Limited to 100,000 rows, converted through Arrow
<code>pyarrow.Table</code> , <code>RecordBatch</code> , <code>RecordBatchReader</code>	Sliced to 100,000 rows
pandas DataFrame	<code>head(100000)</code> ; datetimes become <code>YYYY-MM-DD HH:MM:SS</code> strings, numeric NaN becomes <code>"NaN"</code>
list	First 100,000 items; dict items become rows

Any other type fails with `outputDF` must be `DuckDBPyRelation`, `pa.Table`, `pa.RecordBatch`, `pa.RecordBatchReader`, `DataFrame`, or `list`.

The schema returned with the rows uses these type names: `integer`, `long`, `short`, `byte`, `string`, `boolean`, `double`, `float`, `date`, `timestamp`, `time`, `binary`, `decimal`, `interval`, `array`, `json` and `null`.

Previews and dashboards page through the result: the response carries `totalRecords`, `totalPages`, `pageNumber`, `hasMore` and the `data` for the requested page. Asking for a page past the end fails with `Page N is out of range. Total pages: X for page size Y.`

IDENTICAL CODE RETURNS CACHED ROWS

Dataset output is cached by the exact text of your code. Re-running unchanged code returns the cached rows, without reading the source again, for up to 15 minutes. Results that read governed lakehouse tables are cached per user policy. Change the code (for example a comment) or wait for the cache to expire to force a refresh.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
orders = session.readS3File(
    "sales_s3", "acme-retail-data", "raw/orders/2026-09-30.parquet"
)
session.register("orders", orders)

# Either of these returns rows; do not use both.
gz_output(session.query("SELECT region, count(*) AS n FROM orders GROUP BY 1"))
```

3.3.7 Use parameters and placeholders

Groundzero substitutes values into your code text before it runs. There are two kinds.

Runtime placeholders

These describe the run and are available in every runtime. Write them as `${NAME}` or `{NAME}`.

Placeholder	Value
STRTTM	Run start time
ENDTM	Time the placeholder is evaluated
CLNTNM	Client name
SITENM	Site name
PRJNM	Project name
JOBNM	Job name
RTE	Runtime name
RGN	Region
USR	User running the code
USRGRPS	The user's groups, comma separated, each quoted

Each placeholder expands to a Python string literal that already contains single quotes, so it can be dropped straight into SQL text. For a project named `Sales`, the line

PYTHON

```
project = ${PRJNM}
```

becomes `project = "'Sales'"`, a string whose value is `'Sales'` including the quotes. Use it like this:

PYTHON

```
audit = session.query(f"SELECT * FROM audit WHERE project = {project}")
```

BRACE NAMES ARE REPLACED EVERYWHERE

`{PRJNM}` is replaced even inside an f-string or a SQL snippet, and the placeholder names are case sensitive. Do not name your own variables with these upper-case names.

Job parameters

When you start a job through the external job API you can pass your own parameters as `placeholders`. Each key is `$` followed by a name made of letters, digits and underscores. Values may be a string, number, boolean, null or a list of those. Every `$NAME` in the job's code is replaced by the matching Python literal before the run:

PYTHON

```
cutoff = $CUTOFF_DATE
regions = $REGIONS
orders = session.query(
    f"SELECT * FROM orders WHERE ordered_at ≥ DATE '{cutoff}'"
)
```

Request body (POST to the external job endpoint):

JSON

```
{
  "projectName": "Retail Analytics",
  "jobName": "daily_orders",
  "rteName": "duckdb",
  "computeName": "Medium",
  "placeholders": {
    "$CUTOFF_DATE": "2026-09-01",
    "$REGIONS": ["EMEA", "APAC"]
  }
}
```

Placeholder names that are not valid or values of unsupported types are rejected before anything runs (`Invalid placeholder name: ...`, `Invalid value for placeholder: ...`).

3.3.8 Run jobs and schedule them

An ETL job is a Python file saved in a project. To run it:

1. Open the job in the ETL job editor and choose the runtime and compute in the toolbar (see the Quickstart).
2. Run it once from the editor to check the output pane and logs.
3. Open the scheduler, pick the project and job, and set:
 - **Schedule type:** `Batch` or `Streaming`. A batch schedule adds a recurrence (cron-style) and an **Ends** choice of `Never` or `On` a date and time.

- **RTE:** the runtime and compute to use (both are required).
- **Name and Description.**

When a schedule has an end time, the run stops at that time. A job ends when your code finishes.

To chain jobs, call `executeJob` from a parent job. It runs the named sub-jobs of the current multi-job run in order and returns their results:

PYTHON

```
from GZ.JobUtils import executeJob

results = executeJob(["load_orders", "build_summary"])
```

`executeJob` raises `ValueError` if none of the names match a sub-job of the current multi-job run.

3.3.9 Use extra packages

Use this when your code needs a library the runtime does not include. You have three options, from lightest to heaviest.

1. **An ML variant:** choose the DuckDB scikit-learn, PyTorch or TensorFlow runtime when the library you need is listed there (see Machine-learning variants).
2. **A custom runtime:** administrators can build a runtime from any base runtime plus your own `requirements.txt`. Open **Custom RTE** in the admin area and choose **Add custom RTE:**
 - **Base Image:** pick the runtime to extend, for example the DuckDB scikit-learn runtime.
 - **Name:** a short name. The runtime kind is added as a prefix, so a name of `forecasting` on a DuckDB runtime is shown as `duckdb-forecasting`.
 - **Choose File:** a `.txt` file in `requirements.txt` format.

The page lists each custom runtime with its creator, creation time and status. The build can take a while, and the status changes when it finishes. Once it succeeds, choose the new runtime like any other.

3. **Install in a notebook:** `pip install` works for the current session. Packages installed this way last only until you stop the session, so move them to a custom runtime for jobs.

To check a package is available, import it in a notebook cell.

LEAVE THE PINNED PACKAGES ALONE

The runtime pins NumPy, pandas, scikit-learn and DuckDB. A custom `requirements.txt` that upgrades them can break the `GZ` library, so add new packages and leave those alone.

3.3.10 Work with files

Your code can use local scratch space. The sandbox allows reading and writing under `/tmp`, `/home/workspace` and `/home`, and rejects other locations:

PYTHON

```
import pandas as pd

df = session.query("SELECT 1 AS a, 2 AS b", returnFormat="pandas")
df.to_csv("/tmp/scratch.csv", index=False)
back = session.query("SELECT * FROM read_csv('/tmp/scratch.csv')")
```

Files written locally disappear when the run ends. Write anything you want to keep to an object store or database connection.

Job and dataset code runs with restrictions. Modules that give network or process access (`socket` , `urllib` , `http` and others) cannot be imported, `subprocess` is blocked, `eval` , `exec` and `compile` are not allowed, and secret values in `os.environ` read as `****` . Use connections for network access.

3.3.11 Tune performance

DuckDB is fast by default. These are the levers that matter.

- 1. Select columns and filter early.** Parquet is columnar and DuckDB reads only the columns and row groups a query needs. Prefer `SELECT col_a, col_b` and `WHERE` clauses to `SELECT *` .
- 2. Stay lazy.** Relations and SQL do not copy data into Python. Calling `.df()` or `returnFormat="pandas"` on a large table pulls it into memory.
- 3. Read many files with a glob.** `s3FullUrl="s3://bucket/path/*.parquet"` lets DuckDB parallelize across files. Pass `readOptions={"hive_partitioning": True}` for partitioned layouts.
- 4. Aggregate before returning.** A dataset returns at most 100,000 rows; do the grouping in SQL.
- 5. Right-size the compute.** The DuckDB memory limit is 80 percent of the compute's memory. When a query needs more it spills to disk under `/tmp/duckdb-spill` , which is slower. If a run is slow and logs show spilling, choose a larger compute.
- 6. Control threads and memory per session** when several heavy queries share a run:


```
duckSession(threads=4, memory_limit="6GB").
```
- 7. Convert to Arrow, not pandas.** `relation.to_arrow_table()` is zero-copy friendly and works with Parquet writers and Polars.
- 8. Materialize reused data.** If a relation is queried many times, create a table once with `CREATE TABLE x AS ...` (or `targetDuckdbTable`) so DuckDB does not re-read the source for every query.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession(threads=2, memory_limit="1GB")

# Materialize once, query many times
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders_remote",
)
session.query("""
    CREATE TABLE orders_local AS
    SELECT order_id, region, amount FROM orders_remote
""")
print(session.query("SELECT region, avg(amount) AS avg_amount "
    "FROM orders_local GROUP BY region ORDER BY 1",
    returnFormat="pandas"))
print(session.query("SELECT current_setting('threads') AS threads",
    returnFormat="pandas"))
```

3.3.12 Log and debug

Anything your code prints is captured and shown in the job's log view with the job identifier prefix. The standard `logging` module writes to the same stream. Very long lines (over 200,000 characters) are split into numbered `LARGE_OUTPUT_PART_n_OF_m` chunks so the UI can reassemble them.

When your code raises an exception, the run fails and the error is reduced to the line in your code that raised it, for example:

TEXT

```
ValueError: Either s3FullUrl or both bucketName and filePath must be provided at line 7
```

For syntax errors the message ends with the offending line number. Use `print(session.query("EXPLAIN ...", returnFormat="pandas"))` to inspect query plans and `session.listTables()` to see what is registered.

To raise the verbosity of the helper library, pass `logLevel="DEBUG"` to `duckSession`.

3.3.13 Handle secrets

Never put keys or passwords in code. Create a connection in Groundzero and refer to it by name. Groundzero supplies the credentials to the helper library, and your code never sees them. In jobs and datasets, `os.environ` shows secret values as `****`. Reading Groundzero's own lakehouse storage buckets (names starting with `gz-lakehouse-`) directly is refused with `Direct access to gz-lakehouse-* buckets is restricted; use the GZLAKEHOUSE connector`.

If you need a native client for a connection, ask the helper library for one:

PYTHON

```
from GZ.Client import getClient

s3 = getClient().getS3Client("sales_s3")
resp = s3.list_objects_v2(Bucket="acme-retail-data", Prefix="raw/", MaxKeys=5)
```

3.3.14 Extract text from PDFs

`parsePDF` and `parsePDFCorpus` extract text and tables from PDFs (Docling with RapidOCR) and write results to S3.

PYTHON

```
result = session.parsePDFCorpus(
    s3ConnName="sales_s3",
    sourcePrefix="incoming/contracts/",
    outputPrefix="parsed/contracts",
    pageRangeSize=64,
)
print(result["status"], result["documentCount"], result["manifestUri"])
```

To check it worked, `result["status"]` reports the outcome and `manifestUri` points to the manifest of parsed documents.

Discovery is recursive and case insensitive. Results are restartable: running again with the same source and output skips page ranges that were already committed. OCR processes one document model at a time, so throughput is bounded by that. For distributed OCR over very large corpora use the Spark runtime.

3.3.15 Run Groundzero Agents over documents

`invokeAgent` calls an approved Groundzero Agent and waits for its result; `invokeAgentPDFCorpus` runs an Agent over every document in an S3 prefix and writes one result per document.

PYTHON

```
result = session.invokeAgentPDFCorpus(
    connName="agents",
    agentId="00000000-0000-4000-8000-000000000001",
    message="List each policy's effective date and exceptions.",
    pdfSourceConnName="sales_s3",
    pdfSourcePrefix="incoming/policies/",
    pdfSinkConnName="sales_s3",
    pdfSinkPrefix="agent-results/policies",
    summaryWorkers=4,
)
print(result["status"], result["resultsUri"])
```

The corpus run resolves the approved Agent version and its response schema before any OCR starts. With a JSON response schema configured the output mode is structured; without one the Agent returns cited facts. `summaryWorkers` runs between 1 and 32 documents concurrently (default 4), the document limit defaults to 10,000 and the Agent-call budget to 50,000. Unchanged documents are skipped on later runs.

3.3.16 Check data quality

`runDataQualityChecks`, `runTableDataQualityChecks` and `runQueryDataQualityChecks` run Qualipilot against a relation, a table or a query using the session's own DuckDB connection, and can add an LLM-written narrative through a Groundzero LLM connection:

PYTHON

```
report = session.runTableDataQualityChecks(
    tableName="orders_local",
    connectionName="llm_openai",
    includeLlm=True,
)
session.saveDataQualityReport(report, "/tmp/orders_quality.html", "html")
```

Set `includeLlm=False` to run only the deterministic checks.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
session.query("""
    CREATE TABLE orders_local AS
    SELECT range AS order_id, 'EMEA' AS region, range * 1.5 AS amount
    FROM range(100)
""")

report = session.runTableDataQualityChecks(
    tableName="orders_local",
    includeLlm=False,
)
session.saveDataQualityReport(report, "/tmp/orders_quality.json")
```

3.4 API reference

This reference documents the helper library available inside the DuckDB runtime. Everything below is importable in jobs, datasets and notebooks without installing anything.

3.4.1 Conventions

All connector methods are called on the session object returned by `duckSession()`. They share these conventions.

Convention	Meaning
<code>connName</code>	Name of a Groundzero connection. Credentials never appear in code.
<code>returnFormat</code>	"duckdb" (default), "pandas" or "arrow".
<code>targetDuckdbTable</code>	Optional name; the result is also registered under it so SQL can use it.
Lazy relations	A "duckdb" result is a <code>DuckDBPyRelation</code> ; it is evaluated when consumed.

WHAT ARROW RETURNS

`returnFormat="arrow"` returns a `pyarrow.RecordBatchReader` from `query()` and from most connector reads, but a `pyarrow.Table` from `readGzLakehouse`. Call `.read_all()` on a reader to get a table, or use `relation.to_arrow_table()`.

3.4.2 Session

duckSession

PYTHON

```
def duckSession(
    duckdbPath=":memory:", logLevel="INFO", threads=None, memory_limit=None,
    siteName=None, siteNames=None, **kwargs
) → DuckSession: ...
```

Creates a configured session: opens a DuckDB connection, applies performance settings, attaches Groundzero Lakehouse catalogs for lakehouse connections and returns the session.

Parameter	Type	Default	Description
<code>duckdbPath</code>	str	<code>":memory:"</code>	Database file path, or <code>:memory:</code>
<code>logLevel</code>	str	<code>"INFO"</code>	Python log level name
<code>threads</code>	int	from compute	Pins DuckDB threads for this session
<code>memory_limit</code>	str	from compute	Pins DuckDB memory, e.g. <code>"8GB"</code>
<code>siteName</code>	str or list	<code>None</code>	Lakehouse site for REST-backed connections

Parameter	Type	Default	Description
<code>siteNames</code>	dict or list	<code>None</code>	Per-connection lakehouse sites
<code>**kwargs</code>	any		Extra DuckDB <code>SET name='value'</code> options

`siteNames` is preferably a mapping `{"connectionName": "siteName"}`. A list of `{"connName": ..., "siteName": ...}` entries is also accepted. A single site name applies to every REST-backed lakehouse connection; a list of site names must have one entry per connection or the mapping is ignored with a warning.

The session sets `preserve_insertion_order=false`, turns off the progress bar, turns on metadata caches for objects, HTTP and Parquet, and spills to `/tmp/duckdb-spill`.

Returns a `DuckSession`.

Raises `AttributeError` if `logLevel` is not a valid level name. Failures applying individual settings are logged as warnings, not raised.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession(logLevel="WARNING", threads=2, memory_limit="2GB")
print(session.query("SELECT current_setting('memory_limit') AS m",
                    returnFormat="pandas"))
session.close()
```

query

PYTHON

```
def query(sqlQuery, params=None, returnFormat="duckdb"): ...
```

Runs SQL against everything registered in the session.

Parameter	Type	Default	Description
<code>sqlQuery</code>	str	required	SQL text
<code>params</code>	list or dict	<code>None</code>	Values for prepared-statement parameters
<code>returnFormat</code>	str	<code>"duckdb"</code>	<code>"duckdb"</code> , <code>"pandas"</code> or <code>"arrow"</code>

Returns a relation, a pandas DataFrame or an Arrow reader. A statement that returns no result set (for example `CREATE TABLE`) returns the underlying DuckDB connection object.

Raises whatever DuckDB raises for invalid SQL (`duckdb.ParserException`, `duckdb.CatalogException`, ...); the error and the query text are logged first.

PYTHON

```
rows = session.query(
    "SELECT * FROM orders WHERE region = $r AND amount > $min",
    params={"r": "EMEA", "min": 100},
    returnFormat="pandas",
)
```

register and unregister

PYTHON

```
def register(name, data): ...
def unregister(name): ...
```

`register` exposes a pandas DataFrame, Arrow table, relation or other object DuckDB can scan as a view called `name`. `unregister` removes it. Both return `None` and re-raise DuckDB errors.

listTables and describeTable

PYTHON

```
def listTables() → DataFrame: ...
def describeTable(tableName) → DataFrame: ...
```

`listTables` returns a DataFrame with a `name` column. `describeTable` returns DuckDB's `DESCRIBE` output (`column_name`, `column_type`, `null`, `key`, `default`, `extra`). It raises `ValueError("Invalid table name: ...")` if the name contains characters other than letters, digits, underscores and dots.

registerLakehouseTable

PYTHON

```
def registerLakehouseTable(connName, tableName, alias=None, materialize=True): ...
```

Reads a Groundzero Lakehouse table and keeps it in the session as `lakehouse_<alias>`.

Parameter	Type	Default	Description
<code>connName</code>	str	required	Lakehouse connection
<code>tableName</code>	str	required	<code>database.table</code> name
<code>alias</code>	str	table name	Suffix; dots and dashes become underscores
<code>materialize</code>	bool	True	True copies into a table; False registers a view

Returns `None`. **Raises** the errors of `readGzLakehouse`.

close

PYTHON

```
def close(): ...
```

Closes connector clients, unregisters session views and closes the DuckDB connection. Using the session as a `with` block calls it for you. Errors while closing are logged, not raised.

3.4.3 Groundzero Lakehouse

readGzLakehouse

PYTHON

```
def readGzLakehouse(
    connName, query=None, returnFormat="duckdb", tableName=None, siteName=None,
    filters=None
): ...
```

Aliases: `executeGzLakehouse`, `readGZLAKEHOUSE`.

Runs a read-only query on a lakehouse connection and returns the result.

Parameter	Type	Default	Description
<code>connName</code>	str	required	Groundzero Lakehouse connection
<code>query</code>	str	None	Read-only SQL; tables as <code>database.table</code>
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas" or "arrow" (a Table)
<code>tableName</code>	str	None	Reads <code>SELECT *</code> from this table when no <code>query</code>
<code>siteName</code>	str	None	Site for this connection
<code>filters</code>	list of dict	None	Structured predicates, see below

Each filter is `{"column": str, "op": str, "values": list}` with an optional `"table"`. Operators: `in`, `eq`, `between`, `gte`, `gt`, `lte`, `lt`.

The call authorizes every referenced table for the caller, applies the caller's column masks and row policies by rewriting the query, runs it, and records an audit entry.

Returns a relation, DataFrame or Arrow table.

Raises

Exception	When
<code>ValueError</code>	Neither <code>query</code> nor <code>tableName</code> ; non-read-only SQL; unparsable SQL; missing site name; unknown warehouse; site already bound
<code>PermissionError</code>	Authentication fails, or the caller lacks <code>Select</code> on a table, or tables are not fully qualified
<code>ConnectionError</code>	The lakehouse cannot be reached
<code>RuntimeError</code>	Other lakehouse failures

PYTHON

```
df = session.readGzLakehouse(
    connName="lakehouse_main",
    query="SELECT region, count(*) AS n FROM retail.orders GROUP BY region",
    returnFormat="pandas",
)
```

Writing to the lakehouse

DuckDB reads the lakehouse only. `writeGzLakehouse` (alias `writeGZLAKEHOUSE`) raises `PermissionError` so code written for another runtime fails clearly. Use the Spark runtime to write lakehouse tables, or write results to S3 or a database from DuckDB.

getLakehouseCatalog

PYTHON

```
def getLakehouseCatalog(connName, siteName=None) → str: ...
```

Attaches the connection's REST catalog if needed and returns its DuckDB catalog alias (of the form `gzlakehouse__<connection>`), so you can write fully qualified SQL against `session.query`:

PYTHON

```
catalog = session.getLakehouseCatalog("lakehouse_main", siteName="acme")
rel = session.query(f'SELECT count(*) FROM "{catalog}".retail.orders')
```

Raises `ValueError` if the connection has no lakehouse URL (it is not REST-backed) or no site can be determined.

NOTE

Queries issued through `session.query` against an attached catalog do not go through the authorization, masking and audit steps of `readGzLakehouse`. Use `readGzLakehouse` for governed access.

setLakehouseSite and configureGzLakehouseCatalogs

PYTHON

```
def setLakehouseSite(connName, siteName): ...
def configureGzLakehouseCatalogs(connectionSiteNames=None, failFast=False): ...
```

`setLakehouseSite` binds a REST-backed connection to a site and attaches its catalog (`ValueError` if `siteName` is empty or the connection is not REST-backed). `configureGzLakehouseCatalogs` attaches the catalogs of every REST-backed connection; `duckSession` calls it for you. With `failFast=False` (default) an attach failure is deferred to first use.

3.4.4 Object storage

readS3File

PYTHON

```
def readS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
): ...
```

Reads Parquet, CSV or JSON from Amazon S3 with DuckDB's native reader.

Parameter	Type	Default	Description
<code>connName</code>	str	required	S3 connection
<code>bucketName</code>	str	None	Bucket; use with <code>filePath</code>
<code>filePath</code>	str	""	Key or glob inside the bucket
<code>fileType</code>	str	"parquet"	parquet, csv or json (other values read as CSV)
<code>s3FullUrl</code>	str	None	Full <code>s3://</code> URL or glob; replaces bucket and path
<code>targetDuckdbTable</code>	str	None	Register the result under this name
<code>readOptions</code>	dict	None	Reader options, e.g. <code>{"delim": ";"}</code>
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas" or "arrow"

`readOptions` keys are passed to the DuckDB reader as named arguments, so they must be DuckDB's own option names (for CSV: `delim`, `header`, `quote`, `dateformat`, `skip`, and so on; `delimiter` is not valid). Booleans, numbers and strings are accepted. The call retries failed reads with `all_varchar` and then `ignore_errors` before raising the last error.

Raises `ValueError` for Either `s3FullUrl` or both `bucketName` and `filePath` must be provided or incomplete connection details, and DuckDB errors (for example `duckdb.IOException`) when the object cannot be read.

writeS3File

PYTHON

```
def writeS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    sourceTable=None, sourceQuery=None, writeOptions=None
): ...
```

Alias: `writeS3`. Writes a table or query result with `COPY ... TO`.

Parameter	Type	Default	Description
<code>connName</code>	str	required	S3 connection
<code>bucketName</code> , <code>filePath</code>	str	None	Destination, or use <code>s3FullUrl</code>

Parameter	Type	Default	Description
<code>fileType</code>	str	"parquet"	parquet, csv or json
<code>sourceTable</code>	str	None	Name of a table or view in the session
<code>sourceQuery</code>	str	None	SQL whose result is written; wins over <code>sourceTable</code>
<code>writeOptions</code>	dict	None	COPY options, e.g. {"compression": "zstd"}

Returns `None`. **Raises** `ValueError` for a missing destination (Either `s3FullUrl` or both `bucketName` and `filePath` must be provided), a missing source (Either `sourceTable` or `sourceQuery` must be provided) or an unsupported type (`Unsupported file type: ...`).

3.4.5 SQL databases

executeJdbc

PYTHON

```
def executeJdbc(
    connName, sqlQuery, params=None, targetDuckdbTable=None, returnFormat="duckdb"
): ...
```

Runs a query on a SQL database connection and loads the result.

Parameter	Type	Default	Description
<code>connName</code>	str	required	A PostgreSQL, MySQL, Oracle, SQL Server, Db2, Redshift, CockroachDB, SAP, Exasol, TimescaleDB or SQLite connection
<code>sqlQuery</code>	str	required	Query in the database's own dialect
<code>params</code>	list or dict	None	Bound parameters
<code>targetDuckdbTable</code>	str	None	Register the result under this name
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas" or "arrow"

Returns the result in the requested format. **Raises** `ValueError` (`Unsupported database type: ...`) for connection types it cannot map, and the driver's errors for failing SQL. If the ADBC driver for PostgreSQL-compatible databases fails, the call falls back to SQLAlchemy and pandas automatically.

writeJdbc

PYTHON

```
def writeJdbc(
    connName, targetTable, data=None, sourceTable=None, sourceQuery=None,
    mode="append"
): ...
```

Parameter	Type	Default	Description
<code>connName</code>	str	required	SQL database connection
<code>targetTable</code>	str	required	Destination table
<code>data</code>	relation, DataFrame or Table	None	Data to write
<code>sourceTable</code>	str	None	Session table to write
<code>sourceQuery</code>	str	None	Query to write
<code>mode</code>	str	"append"	append or replace

Provide one of `data`, `sourceTable` or `sourceQuery`. PostgreSQL-compatible and MySQL targets are written through DuckDB's `postgres` and `mysql` extensions; other databases go through SQLAlchemy in chunks of 10,000 rows.

Returns `True`. **Raises** `ValueError` (Either `data`, `sourceTable`, or `sourceQuery` must be provided) or the database's errors. `replace` drops and recreates the table.

getSqlEngine

PYTHON

```
def getSqlEngine(connName) → sqlalchemy.Engine: ...
```

Returns a SQLAlchemy engine for a JDBC connection, for code that needs the native API. Dispose of it when you are done.

3.4.6 Cloud warehouses

readSnowflake

PYTHON

```
def readSnowflake(
    conn_name, sql_query, target_duckdb_table, warehouse=None,
    force_microsecond_precision=False, client_prefetch_threads=4,
    returnFormat="duckdb"
): ...
```

Streams a Snowflake query into a DuckDB table in Arrow batches, so Python holds only one batch at a time.

Parameter	Type	Default	Description
<code>conn_name</code>	str	required	Snowflake connection
<code>sql_query</code>	str	required	Query to run
<code>target_duckdb_table</code>	str	required	Table to create or replace
<code>warehouse</code>	str	None	Overrides the connection's warehouse
<code>force_microsecond_precision</code>	bool	False	Cast timestamps to microseconds; use for dates outside 1677-2262

Parameter	Type	Default	Description
<code>client_prefetch_threads</code>	int	4	Download threads
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas", "arrow" or "row_count"

Raises `ValueError` for an unsupported `returnFormat` (Unsupported `returnFormat`. Use 'duckdb', 'pandas', 'arrow', or 'row_count'.) and Snowflake connector errors for failing SQL.

3.4.7 Outputs and run context

gz_output and outputDF

PYTHON

```
def gz_output(value): ...
```

Available as a global in datasets and jobs run by the runtime (it is not defined in notebooks). Records `value` as the run's output. If called repeatedly the last value is used. A variable named `outputDF` is the alternative; when both exist `gz_output` wins. See How-to guides, Return output and previews, for the accepted types and limits.

Placeholders

`${NAME}` or `{NAME}` for `STRTTM`, `ENDTM`, `CLNTNM`, `SITENM`, `PRJNM`, `JOBNM`, `RTE`, `RGN`, `USR` and `USRGRPS`, and `$NAME` for job parameters passed through the external job API. See How-to guides, Use parameters and placeholders.

executeJob

PYTHON

```
def executeJob(jobNames: list[str]) → dict: ...
```

Import: `from GZ.JobUtils import executeJob`. Runs the named sub-jobs of the current multi-job run in sequence and returns `{jobName: result}`.

Parameter	Type	Description
<code>jobNames</code>	list of str	Names of sub-jobs to run

Raises `ValueError` if none of the names match the run's sub-jobs, and re-raises a sub-job's failure after recording it as failed.

sendEmail

PYTHON

```
def sendEmail(subject, body, to_email, from_email, **kwargs): ...
```

Import: `from GZ.SendEmail import sendEmail`. Sends an email through the Groundzero notification service.

Parameter	Type	Description
<code>subject</code>	<code>str</code>	Subject line
<code>body</code>	<code>str</code>	Message body
<code>to_email</code>	<code>str</code>	Recipient address
<code>from_email</code>	<code>str</code>	Sender address
<code>**kwargs</code>	<code>any</code>	Extra fields added to the message

Returns `None` . **Raises** the error from the notification service.

PYTHON

```
from GZ.SendEmail import sendEmail

sendEmail(
    subject="Daily revenue ready",
    body="The 2026-09-30 revenue table was written.",
    to_email="data-team@example.com",
    from_email="noreply@example.com",
)
```

3.4.8 Clients and connection details

getClient

PYTHON

```
def getClient() → ClientSession: ...
```

Import: `from GZ.Client import getClient` . Returns an object with one factory per connector that builds the vendor's native client from a connection, for the cases where the session methods are not enough.

Method	Returns
<code>getS3Client(connName)</code>	<code>boto3</code> S3 client
<code>getS3FSCClient(connName)</code>	<code>s3fs.S3FileSystem</code>
<code>getSQLEngine(connName)</code>	SQLAlchemy engine
<code>getSnowflakeClient(connName)</code>	Snowflake connection
<code>getMongoClient(connName)</code>	<code>pymongo</code> client
<code>getBigQueryClient(connName)</code>	BigQuery client
<code>getAzureBlobClient(connName)</code>	Azure Blob service client
<code>getGCSCClient(connName)</code>	Google Cloud Storage client
<code>getDynamoDBClient(connName)</code>	DynamoDB client
<code>getKinesisClient(connName)</code>	Kinesis client

The class also provides `getSalesforceConnection`, `getCassandraSession`, `getSFTPClient`, `getRedshiftClient`, `getS3SelectClient`, `getCosmosDBClient`, `getElasticClient`, `getExasolConnection`, `getTimescaleClient`, `getPineconeClient`, `getTabularCatalog`, `getFireboltClient`, `getTeradataClient`, `getNeo4JClient`, `getInfluxDBClient`, `getDuckDBClient`, `getBusinessCentralClient`, `getShopifyClient`, `getMarketoClient`, `getAthenaClient`, `getCouchbaseClient`, `getClickhouseClient`, `getGoogleDriveClient`, `getOnedriveClient`, `getSplunkClient`, `getKDBAIClient`, `getDatabricksSQLClient`, `getAzureSynapseClient`, `getEventHubsClient`, `getSharepointClient`, `getScyllaDBClient`, `getAwsOpenSearchClient`, `getWeaviateClient`, `getFirestoreClient`, `getFirebaseDBClient`, `getOpenAIClient`, `getXAIClient`, `getAzureOpenAIClient`, `getAwsBedrockClient`, `getCohereClient`, `getTogetherAIClient`, `getFireworksAIClient`, `getHuggingFaceClient`, `getAnthropicClient`, `getGeminiClient` and `getMcpClient`, each taking `connName`.

Raises `ValueError` when the named connection does not exist or has the wrong type, for example `s3` connection details for 'x' not found or the connection type is incorrect.

3.4.9 Connectors

The signatures below list every connector method on the session object. session object. Reads return the result in `returnFormat`; most connectors register the result as `targetDuckdbTable` when given. Writes return `None`, a status value or a count depending on the system, and raise the vendor's errors. Many connectors cache a client per connection; `closeConnection(connName)` releases it and `session.close()` releases all of them.

Object storage

PYTHON

```
session.readS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
)
session.writeS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    sourceTable=None, sourceQuery=None, writeOptions=None
)
session.writeS3(*args, **kwargs)

session.readGcsFile(
    connName, bucketName=None, filePath="", fileType="parquet", gcsFullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
)
session.writeGcs(
    connName, bucketName=None, filePath="", fileType="parquet", gcsFullUrl=None,
    data=None, sourceTable=None, sourceQuery=None, writeOptions=None
)

session.readAzureBlob(
    connName, containerName, blobName, fileType="parquet", azureFullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
)
session.writeAzureBlob(
    connName, containerName, blobName, fileType="parquet", azureFullUrl=None,
    data=None, sourceTable=None, sourceQuery=None, writeOptions=None
)
```

SQL databases and warehouses

PYTHON

```

session.getSqlEngine(connName)
session.executeJdbc(
    connName, sqlQuery, params=None, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeJdbc(
    connName, targetTable, data=None, sourceTable=None, sourceQuery=None,
    mode="append"
)

session.getSnowflakeClient(conn_name, client_prefetch_threads=4)
session.readSnowflake(
    conn_name, sql_query, target_duckdb_table, warehouse=None,
    force_microsecond_precision=False, client_prefetch_threads=4,
    returnFormat="duckdb"
)

session.readBigQuery(
    connName, query, useLegacySql=False, targetDuckdbTable=None,
    returnFormat="duckdb", maxQueueSize=2, maxStreamCount=None
)
session.readBigQueryTable(
    connName, projectId, datasetId, tableId, selectedFields=None, rowFilter=None,
    targetDuckdbTable=None, returnFormat="duckdb", maxQueueSize=2,
    maxStreamCount=None
)
session.writeBigQuery(
    connName, tableName, data, writeDisposition="WRITE_APPEND", **kwargs
)

session.readAthena(
    connName, region, query, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeAthena(connName, region, sqlQuery)

session.executeDatabricks(
    connName, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)
session.writeDatabricks(connName, sqlQuery)

session.readAzureSynapse(
    connName, query, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)

session.executeTeradata(
    connName, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)
session.writeTeradata(connName, tableName, data, **kwargs)

session.getExasolClient(connName)
session.readExasol(
    connName, sqlQuery, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)
session.writeExasol(connName, tableName, data, **kwargs)

```

```

session.executeFirebolt(
    connName, sqlQuery, parameters=None, targetDuckdbTable=None,
    returnFormat="duckdb", fetchBatchSize=10000
)
session.getFireboltSystemInfo(
    connName, infoType="tables", targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeFirebolt(connName, sqlQuery)

session.executeClickHouse(
    connName, sourceQuery, parameters=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.getSystemInfo(
    connName, table="tables", targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeClickhouse(connName, tableName, data, database=None, **kwargs)

session.executeSingleStore(
    connName, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeSingleStore(connName, tableName, data, **kwargs)

session.executeMotherDuck(
    connName, query=None, tableName=None, limit=None, alias="motherduck",
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.readMotherDuck(
    connName, query=None, tableName=None, limit=None, alias="motherduck",
    targetDuckdbTable=None, returnFormat="duckdb"
)

```

NoSQL, search and graph

PYTHON

```

session.getMongoClient(connName)
session.readMongo(
    connName, databaseName, collectionName, mongoQuery=None, projection=None,
    targetDuckdbTable=None, returnFormat="duckdb", **kwargs
)
session.writeMongodb(connName, databaseName, collectionName, data, **kwargs)

session.readDynamoDB(
    connName, tableName, filters=None, targetDuckdbTable=None,
    returnFormat="duckdb", parallelSegments=4
)
session.queryDynamoDB(
    connName, tableName, keyConditionExpression, expressionAttributeValues,
    expressionAttributeNames=None, filterExpression=None, indexName=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeDynamoDb(connName, tableName, data)

session.executeCosmos(
    connName, sourceQuery, enableCrossPartition=True, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeCosmosDb(connName, data, **kwargs)

session.executeCouchbase(

```

```

        connName, sourceQuery, parameters=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.writeCouchbase(connName, data, idField, **kwargs)

    session.getCassandraSession(connName)
    session.readCassandra(connName, cqlQuery, targetDuckdbTable=None, returnFormat="duckdb")

    session.executeScyllaDB(
        connName, keyspace, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb"
    )

    session.readFirestore(
        connName, collection, filters=None, orderBy=None, limit=None,
        targetDuckdbTable=None, returnFormat="duckdb"
    )
    session.writeFirestore(connName, collection, data, idField=None)

    session.readRealtimeDatabase(
        connName, path, orderBy=None, limitToFirst=None, limitToLast=None,
        targetDuckdbTable=None, returnFormat="duckdb"
    )

    session.executeElasticsearch(
        connName, indexName, query=None, size=10000, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.executeESQL(connName, esqlQuery, targetDuckdbTable=None, returnFormat="duckdb")
    session.writeElasticsearch(connName, indexName, data, idField=None, **kwargs)

    session.searchOpenSearch(
        connName, index, query=None, size=1000, scroll=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.vectorSearch(
        connName, index, vector, vectorField, k=10, size=100, filter=None,
        targetDuckdbTable=None, returnFormat="duckdb"
    )
    session.getIndexMapping(connName, index, targetDuckdbTable=None, returnFormat="duckdb")
    session.writeAwsOpenSearch(connName, index, data, idField=None, **kwargs)

    session.executeSplunk(
        connName, searchQuery, earliestTime=None, latestTime=None, maxCount=0,
        targetDuckdbTable=None, returnFormat="duckdb"
    )

    session.streamGraphData(
        connName, graphName, nodeLabels=None, relationshipTypes=None,
        nodeProperties=None, relationshipProperties=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.executeCypherQuery(
        connName, cypherQuery, parameters=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.getGraphSchema(connName, targetDuckdbTable=None, returnFormat="duckdb")
    session.getNodeLabels(connName, targetDuckdbTable=None, returnFormat="duckdb")
    session.getRelationshipTypes(connName, targetDuckdbTable=None, returnFormat="duckdb")
    session.writeNeo4j(connName, label, data, idField, properties=None, **kwargs)

    session.executeInfluxQuery(
        connName, query, database=None, targetDuckdbTable=None, returnFormat="duckdb"
    )

```

```

)
session.writeToInflux(
    connName, dataframe, measurement, database=None, tagColumns=None,
    timeColumn="_time"
)

```

Streaming and lake formats

PYTHON

```

session.readKafka(
    connName, topic, maxMessages=1000, autoOffsetReset="earliest",
    consumerTimeout=10000, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeKafka(connName, topic, data, keyField=None, **kwargs)

session.readEventHub(
    connName, consumerGroup="$Default", maxEvents=1000, startingPosition="-1",
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeEventHub(connName, data, partitionKey=None, **kwargs)

session.readKinesis(
    connName, streamName=None, shardIteratorType="LATEST", limit=1000,
    maxShards=None, targetDuckdbTable=None, returnFormat="duckdb"
)
session.readKinesisStream(
    connName, streamName=None, shardIteratorType="LATEST", limit=1000,
    maxShards=None, targetDuckdbTable=None, returnFormat="duckdb"
)

session.readHudi(
    connName, tablePath, filters=None, columns=None, asOfTimestamp=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.readHudiIncremental(
    connName, tablePath, startCommitTime, endCommitTime=None, columns=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)

```

SaaS, files and APIs

PYTHON

```

session.executeREST(
    connName, endpoint=None, method=None, headers=None, params=None, body=None,
    resultPath=None, targetDuckdbTable=None, returnFormat="duckdb", timeout=60,
    maxPages=None, nextPath=None, pageParam=None, pageStart=1, maxRecords=None
)

session.getSalesforceConnection(connName)
session.readSalesforce(
    connName, sqlQuery, targetDuckdbTable=None, returnFormat="duckdb",
    useBulkApi=True, batchSize=10000, pkChunking=False, chunkSize=100000
)

session.readHubSpot(
    connName, objectType, properties=None, limit=100, after=None, filters=None,
    searchTerm=None, maxPages=None, rateLimit=10, targetDuckdbTable=None,
    returnFormat="duckdb"
)

```

```

session.readServiceNow(
    connName, table, query=None, limit=1000, displayValue=True, options=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.executeServiceNow(
    connName, sourceQuery, limit=1000, options=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)

session.readZohoCRM(
    connName, module, perPage=200, maxPages=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.executeZohoCRM(connName, module, targetDuckdbTable=None, returnFormat="duckdb")

session.readShopify(
    connName, resource, limit=None, sinceId=None, fields=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeShopify(connName, resource, data, **kwargs)

session.readM365BC(
    connName, entity, filter=None, select=None, top=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeM365BC(connName, entity, data, **kwargs)

session.readGoogleDrive(
    connName, fileId, fileType=None, targetDuckdbTable=None, returnFormat="duckdb",
    **kwargs
)
session.listFiles(
    connName, folderId=None, query=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeGoogleDriveFile(
    connName, fileName, data, folderId=None, fileType="csv", **writeOptions
)

session.readOneDriveFile(
    connName, driveId, filePath, fileType="csv", targetDuckdbTable=None,
    returnFormat="duckdb", **readOptions
)
session.listOneDriveFiles(
    connName, driveId, folderPath=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeOneDriveFile(
    connName, driveId, filePath, data, fileType="csv", **writeOptions
)

session.readSharePointList(
    connName, siteUrl, listName, filter=None, select=None, top=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeSharePointList(connName, siteUrl, listName, data, idField=None)

session.readSFTPFile(
    connName, filePath, fileType="csv", targetDuckdbTable=None,
    returnFormat="duckdb", **readOptions
)

```

```
session.listSFTPFiles(
    connName, directory=".", targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeSFTPFile(connName, filePath, data, fileType="csv", **writeOptions)
```

Vector stores

PYTHON

```
session.queryVectors(
    connName, vector, topK=10, includeMetadata=True, includeValues=False,
    namespace=None, filter=None, targetDuckdbTable=None, returnFormat="duckdb"
)
session.upsertVectors(connName, vectors, namespace=None)
session.deleteVectors(connName, ids, namespace=None)
session.getIndexStats(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.writePinecone(connName, data, namespace=None, **kwargs)

session.queryWeaviate(
    connName, className, properties=None, nearVector=None, nearText=None,
    where=None, limit=100, targetDuckdbTable=None, returnFormat="duckdb"
)
session.hybridSearch(
    connName, className, query, alpha=0.5, properties=None, where=None, limit=100,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.getSchema(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.writeWeaviate(connName, className, data, batchSize=100, **kwargs)

session.readKDBAI(
    connName, tableName, filter=None, limit=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.searchKDBAI(
    connName, tableName, vectors, nResults=10, filter=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeKdbai(connName, tableName, data, **kwargs)
```

3.4.10 LLM providers

Groundzero connections exist for Anthropic, Amazon Bedrock, Azure OpenAI, Cohere, Fireworks AI, Google Gemini, Hugging Face, OpenAI, Together AI and xAI. The generic methods below pick the provider from the connection's type, so code does not change when you switch providers.

complete

PYTHON

```
def complete(
    connName, df, model=None, promptColumn="prompt", promptTemplate=None,
    systemPrompt=None, systemPromptColumn=None, outputColumn="llm_response",
    temperature=0.0, maxTokens=None, responseFormat=None, continueOnError=False,
    timeout=120, llmOptions=None, targetDuckdbTable=None, returnFormat="duckdb"
): ...
```

Adds an LLM completion to every row of a DataFrame or relation.

Parameter	Type	Default	Description
<code>connName</code>	str	required	LLM connection
<code>df</code>	DataFrame or relation	required	Input rows
<code>model</code>	str	None	Model name (optional)
<code>promptColumn</code>	str	"prompt"	Column holding the prompt
<code>promptTemplate</code>	str	None	Template using <code>{column}</code> fields instead of a prompt column
<code>systemPrompt</code> , <code>systemPromptColumn</code>	str	None	Fixed or per-row system prompt
<code>outputColumn</code>	str	"llm_response"	Name of the response column
<code>temperature</code> , <code>maxTokens</code>	number	0.0, None	Sampling controls
<code>responseFormat</code>	str	None	Provider response format hint
<code>continueOnError</code>	bool	False	Record row errors instead of raising
<code>timeout</code>	int	120	Seconds per call
<code>llmOptions</code>	dict	None	Provider-specific options

Besides `outputColumn` the result has `<outputColumn>_status`, `_error`, `_provider`, `_model`, `_prompt_tokens`, `_completion_tokens`, `_total_tokens` and `_latency_ms` columns.

Raises `ValueError` for an unknown connection (`Connection 'x' was not found.`), an unsupported connection type or missing input columns, and the provider's error for a failed call unless `continueOnError=True`.

PYTHON

```
import pandas as pd

tickets = pd.DataFrame({
    "ticket_id": [1, 2],
    "text": ["App crashes on login", "Love the new dashboard"],
})

scored = session.complete(
    connName="llm_openai",
    df=tickets,
    promptTemplate="Reply with one word, positive or negative: {text}",
    outputColumn="sentiment",
    continueOnError=True,
    returnFormat="pandas",
)
```

Fine-tuning and model management

These generic methods route to the provider-specific method for the connection type (for example `fineTuneOpenAI`):

PYTHON

```
session.fineTune(connName, df, **kwargs)
session.distill(connName, df, **kwargs)
session.listFineTuneJobs(connName, **kwargs)
session.getFineTuneJob(connName, **kwargs)
session.cancelFineTuneJob(connName, **kwargs)
session.listModels(connName, **kwargs)
```

If the provider has no such operation the call raises `NotImplementedError: <operation> is not available for connection type '<type>'`.

Provider-specific methods exist for each operation. The suffix is the provider name: `OpenAI`, `AzureOpenAI`, `AwsBedrock`, `Cohere`, `TogetherAI`, `HuggingFace`, `FireworksAI`, `Gemini`, `Anthropic` or `XAI`.

Operation	Methods (with provider suffix)
Fine-tune	<code>fineTune<P></code> (not XAI)
Distill	<code>distillOpenAI</code> , <code>distillAzureOpenAI</code> , <code>distillAwsBedrock</code>
Jobs	<code>listFineTuneJobs<P></code> , <code>getFineTuneJob<P></code> , <code>cancelFineTuneJob<P></code> (Bedrock: <code>stopFineTuneJobAwsBedrock</code>)
Models	<code>listModels<P></code> (not Azure OpenAI)
Batch	<code>createBatchJob<P></code> , <code>listBatchJobs<P></code> , <code>getBatchJob<P></code> , <code>cancelBatchJob<P></code> , <code>getBatchResults<P></code>
Deployments	<code>createDeployment<P></code> , <code>listDeployments<P></code> , <code>getDeployment<P></code> , <code>deleteDeployment<P></code>

Not every provider supports every operation: a method that does not apply to a provider fails with an explicit error (for example OpenAI has no deployment lifecycle). Fine-tuning methods take a pandas DataFrame with prompt and response columns and return a DataFrame describing the job:

PYTHON

```
job = session.fineTuneOpenAI(
    connName="llm_openai",
    df=training_df,
    model="gpt-4o-mini-2024-07-18",
    promptColumn="prompt",
    responseColumn="response",
)
```

3.4.11 MCP servers

MCP connections let your code call tools, read resources and fetch prompts from a Model Context Protocol server. Every method returns the result as a relation (or the requested `returnFormat`).

PYTHON

```

session.listMcpTools(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.callMcpTool(
    connName, toolName, arguments=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.listMcpResources(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.readMcpResource(
    connName, uri, targetDuckdbTable=None, returnFormat="duckdb"
)
session.listMcpPrompts(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.getMcpPrompt(
    connName, promptName, arguments=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.getMcpLlmConfig(
    connName, provider="openai", serverName=None, requireApproval="always",
    includeSecrets=False, targetDuckdbTable=None, returnFormat="duckdb"
)

```

`completeWithMcpTool` calls an MCP tool for each row and then asks an LLM to complete a prompt that includes the tool result:

PYTHON

```

session.completeWithMcpTool(
    llmProvider, llmConnName, mcpConnName, df, toolName, arguments=None,
    argumentsColumn=None, promptColumn="prompt", promptTemplate=None,
    mcpPromptTemplate=None, systemPrompt=None, systemPromptColumn=None,
    outputColumn="llm_response", mcpResultColumn="mcp_tool_result",
    mcpErrorColumn="mcp_tool_error", model=None, temperature=0.0,
    maxTokens=None, responseFormat=None, continueOnError=False, timeout=120,
    llmOptions=None, targetDuckdbTable=None, returnFormat="duckdb"
)

```

MCP CONNECTION SETUP

Create the connection of type MCP Server with transport `streamable_http`, a server URL that includes `http://` or `https://`, an authentication type (`bearer` with a token, or `none` for public servers), optional headers and allowed tools, and a timeout. A URL without a scheme fails before tool discovery.

3.4.12 PDF extraction

PYTHON

```

session.parsePDF(
    s3ConnName=None, inputUri=None, outputPrefix=None, outputFormat="csv",
    outputFormats=None, inputFile=None, connName=None, hybrid=False,
    ocrMode="auto", ocrStack="ppocrv6", strict=False, password=None,
    language="eng", maxPages=None, maxFileBytes=None, llmProvider=None,
    llmConnName=None, llmModel=None, llmBaseUrl=None, llmApiKey=None,
    outputS3ConnName=None, returnAgentPages=False, contentAddressed=False,
    inlineAgentPages=True, pageRange=None
)

```

Alias: `extractPDF`. Parses one document from S3 and writes the results (CSV, JSON, XLSX or Markdown) under `outputPrefix`.

Parameter	Type	Default	Description
<code>s3ConnName</code>	str	required*	S3 connection holding the input (<code>connName</code> is an alias)
<code>inputUri</code>	str	required*	Document key or <code>s3://</code> URI (<code>inputFile</code> is an alias)
<code>outputPrefix</code>	str	generated	Destination prefix
<code>outputFormat</code> , <code>outputFormats</code>	str, list	"csv"	One or several output formats
<code>hybrid</code>	bool	False	Hybrid extraction; pairs with the <code>llm...</code> options
<code>ocrMode</code>	str	"auto"	OCR policy
<code>ocrStack</code>	str	"ppocrv6"	OCR stack
<code>strict</code>	bool	False	Strict extraction mode
<code>password</code>	str	None	Password for encrypted PDFs
<code>maxPages</code> , <code>maxFileBytes</code>	int	None	Safety limits
<code>returnAgentPages</code>	bool	False	Produce page JSON or Markdown for Agents
<code>pageRange</code>	tuple	None	(first, last) page; requires <code>returnAgentPages=True</code>

*One of `inputUri` or `inputFile` is required, and an S3 connection name.

Returns a dictionary that includes `result_uri`. **Raises** `ValueError` (`s3ConnName` is required , `inputUri` or `inputFile` is required , `pageRange` requires `returnAgentPages=True`) and `PdfLimitError` when a source is larger than the configured limit.

PYTHON

```
session.readPDF(
    s3ConnName, inputUri=None, outputPrefix=None, outputFormat="csv",
    outputFormats=None, targetDuckdbTable=None, readOptions=None,
    returnFormat="duckdb", outputS3ConnName=None, **options
)
session.readPDFResult(
    resultUri, outputFormat="csv", s3ConnName=None, targetDuckdbTable=None,
    readOptions=None, returnFormat="duckdb"
)
session.parsePDFCorpus(
    s3ConnName, sourcePrefix=None, inputUris=None, outputPrefix=None,
    outputS3ConnName=None, pageRangeSize=64, partitions=None, ocrMode="auto",
    language="eng", strict=False, password=None, maxFileBytes=None,
    maxDocuments=10000
)
```

`readPDF` parses a document and returns its table in one call (`pdfParser` , `readPDFExtraction` are aliases); `readPDFResult` (alias `readPDFExtractionResult`) loads an already parsed `csv` or `json` result and raises `ValueError` for other formats. `parsePDFCorpus` (alias `processPDFCorpus`) parses every PDF under `sourcePrefix` , or the listed `inputUris` , with restartable, content addressed output. `partitions` sets the number of I/O workers (1 to 8); OCR still processes one document model at a time.

3.4.13 Groundzero Agents

PYTHON

```
def listAgents(connName) → List[dict]: ...

def invokeAgent(
    connName, agentId, message, sessionId=None, *, requestId=None,
    expectedAgentVersion=None, disableTools=False, pdfSourceConnName=None,
    pdfSourceObject=None, pdfSinkConnName=None, pdfSinkPrefix=None
) → dict: ...

def invokeAgentPDFCorpus(
    connName, agentId, message, pdfSourceConnName, pdfSinkPrefix,
    pdfSourcePrefix=None, pdfSourceObjects=None, pdfSinkConnName=None,
    pageRangeSize=64, partitions=None, summaryWorkers=4,
    summaryInputChars=100000, maxDocuments=10000, maxAgentCalls=50000,
    ocrMode="auto", language="eng", strict=False, password=None,
    maxFileBytes=None
) → dict: ...
```

Method	Purpose
<code>listAgents</code>	Agents visible to the connection's service account
<code>invokeAgent</code>	Run an Agent in the background and wait for the final result (polls up to 15 minutes)
<code>invokeAgentPDFCorpus</code>	Extract JSON or cited facts from every complete PDF

Parameters worth noting:

Parameter	Description
<code>agentId</code>	Agent identifier (a UUID)
<code>message</code>	Instruction for the Agent; must be a non-empty string
<code>expectedAgentVersion</code>	Positive integer; the call fails if the approved version differs
<code>disableTools</code>	True runs without tools; it cannot be combined with PDF arguments
<code>pdfSourcePrefix</code> / <code>pdfSourceObjects</code>	Supply exactly one
<code>summaryWorkers</code>	Concurrent document workflows, 1 to 32
<code>maxDocuments</code> , <code>maxAgentCalls</code>	Bounds for one run, defaults 10,000 and 50,000

Raises `ValueError` for invalid arguments (`agentId` is required, `message` is required, `expectedAgentVersion` must be a positive integer), `RuntimeError` when an Agent run returns an invalid run identifier or status, and `TimeoutError` when polling exceeds 15 minutes. A timeout does not cancel the run: inspect it before starting another.

3.4.14 Data quality and record linkage

PYTHON

```
session.createDataQualityChecker(data, connectionName=None, config=None,
                                references=None)
session.runDataQualityChecks(data, connectionName=None, config=None,
                             includeLlm=True, references=None)
session.runTableDataQualityChecks(tableName, connectionName=None, config=None,
                                  includeLlm=True, references=None)
session.runQueryDataQualityChecks(query, connectionName=None, config=None,
                                  includeLlm=True, references=None)
session.saveDataQualityReport(report, path, reportFormat="json")
```

Parameter	Description
<code>data</code>	DuckDB relation, Arrow table or reader, pandas or Polars DataFrame
<code>tableName</code> , <code>query</code>	A session table, or SQL, to check without copying
<code>connectionName</code>	Groundzero LLM connection used for the narrative
<code>config</code>	A <code>qualipilot.QualipilotConfig</code>
<code>includeLlm</code>	<code>False</code> runs only deterministic checks
<code>references</code>	Optional reference data for referential checks
<code>reportFormat</code>	<code>json</code> , <code>html</code> or <code>markdown</code>

The run methods return a Qualipilot `QualityReport` . Checks reuse the session's DuckDB connection, and only a compact aggregate of the deterministic results is sent to the LLM, not row samples.

Local record linkage and consolidation collect inputs into Polars frames, so each requires `maxRows` :

PYTHON

```
def runLocalRecordLinkage(data, config, rightData=None, *, maxRows): ...
def runLocalRecordDeduplication(data, config, consolidation, *, maxRows): ...
def consolidateLocalRecords(data, idColumn, clusters, config, *, maxRows): ...
```

`config` is a `qualipilot.linking.LinkConfig` (mode `dedupe` or `link`) and `consolidation` a `ConsolidationConfig` . **Raises** `TypeError` if `maxRows` is not an integer, `ValueError` if it is not positive, if an input has more than `maxRows` rows (input has more than `maxRows=N` rows), if `link` mode has no `rightData` , if `dedupe` mode has `rightData` , or if deduplication is not given a `dedupe` configuration. `maxRows` bounds rows, not width or candidate pairs: select only the columns you need and keep blocking rules tight.

3.4.15 Talend migration helpers

Import from `GZ.TalendCompilerUtils` . These helpers load an exported Talend project into a DuckDB manifest table, parse it and compile it into Groundzero job code. They take the session as their first argument.

Function	Purpose
<code>loadTalendProjectFromS3ToDuckDB(duck, connName, prefix, ...)</code>	Load project files from S3 into a manifest table
<code>loadTalendProjectFilesToDuckDB(duck, files, ...)</code>	Load files you already hold in memory
<code>parseTalendManifestToIR(duck, ...)</code>	Parse the manifest into an intermediate representation
<code>compileTalendManifestToGroundZeroJobs(duck, ...)</code>	Parse and compile to job code
<code>compileTalendIRToGroundZeroJobs(ir, ...)</code>	Compile an existing representation
<code>writeTalendCompileResultToDuckDB(duck, result, ...)</code>	Persist compiled jobs to a table
<code>renderTalendCompileReport(result)</code>	Render a compact report

3.5 Installed packages

Packages without a version install at their latest release when the runtime is built. To read the exact version in your session, run `pip show <name>` in a notebook.

3.5.1 Python and system

Component	Version or detail
Python	3.12
<code>numpy</code>	2.3.5
<code>pandas</code>	2.2.0 or newer, below 3
<code>pyarrow</code> , <code>SQLAlchemy</code> , <code>boto3</code> , <code>urllib3</code> , <code>matplotlib</code> , <code>pymysql</code>	unpinned
<code>jupyterlab</code> , <code>ipython</code> , <code>ipykernel</code>	unpinned
<code>mlflow</code>	2.20.4
<code>torch</code> , <code>torchvision</code> (CPU)	2.14.0, 0.29.0
<code>scipy</code>	1.17.1
<code>scikit-learn</code>	1.5.2
System tools	AWS CLI v2, <code>unixodbc</code> , PostgreSQL and MySQL client libraries, time-zone data

Time zone data is included, so time-zone-aware timestamps and the `UTC` zone work correctly in PyArrow.

3.5.2 Connector and data libraries

Package	Version	Purpose
<code>duckdb</code>	1.5.5	Query engine
<code>qualipilot[gz,linking]</code>	3.5.0	Data quality and record linkage
<code>pymongoarrow</code> , <code>pymongo</code>	unpinned	MongoDB
<code>google-cloud-bigquery-storage</code> , <code>google-cloud-bigquery</code>	unpinned	BigQuery
<code>pyodbc</code>	unpinned	ODBC databases
<code>psycopg2-binary</code> , <code>cx-Oracle</code>	unpinned	PostgreSQL, Oracle
<code>teradatasql</code>	unpinned	Teradata
<code>snowflake-connector-python</code> , <code>snowflake-sqlalchemy</code>	unpinned	Snowflake
<code>s3fs</code>	unpinned	S3 file system

Package	Version	Purpose
simple-salesforce	unpinned	Salesforce
azure-storage-blob	unpinned	Azure Blob Storage
google-cloud-storage , gcsfs	unpinned	Google Cloud Storage
paramiko	unpinned	SFTP
cassandra-driver	unpinned	Cassandra and ScyllaDB
redshift-connector	unpinned	Amazon Redshift
requests , httpx (>=0.27,<1), httpcore (>=1,<2)		HTTP clients
azure-cosmos	unpinned	Cosmos DB
elasticsearch , opensearch-py	unpinned	Elasticsearch, OpenSearch
pyexasol	unpinned	Exasol
neo4j , graphdatascience	unpinned	Neo4j
pinecone	unpinned	Pinecone
weaviate-client	unpinned	Weaviate
kdbai-client	>=1.3.0	KDB.AI
firebolt-sdk	unpinned	Firebolt
influxdb3-python	unpinned	InfluxDB 3
ms-dynamics-business-central-sdk	unpinned	Business Central
shopifyapi , marketorestorepython	unpinned	Shopify, Marketo
azure-synapse	unpinned	Azure Synapse
google-api-python-client , google-auth , google-auth-http2 , google-auth-oauthlib	unpinned	Google Drive and authentication
msal	unpinned	Microsoft identity
couchbase	unpinned	Couchbase
clickhouse-connect	unpinned	ClickHouse
databricks-sql-connector	unpinned	Databricks SQL
splunklib , splunk-sdk	unpinned	Splunk
firebase-admin	unpinned	Firebase and Firestore
adbc-driver-postgresql , adbc-driver-sqlite , adbc-driver-manager	unpinned	Arrow database drivers
pyiceberg[sql-postgres,s3fs,pyarrow]	unpinned	Iceberg client
hudi	unpinned	Apache Hudi
pyathena	unpinned	Amazon Athena

Package	Version	Purpose
kafka-python	unpinned	Kafka
azure-eventhub	unpinned	Event Hubs
singlestoredb	unpinned	SingleStore
openpyxl	unpinned	Excel files

3.5.3 LLM and AI libraries

Package	Version	Purpose
openai	unpinned	OpenAI and Azure OpenAI
anthropic	unpinned	Anthropic
cohere	unpinned	Cohere
huggingface-hub	unpinned	Hugging Face
google-generativeai	unpinned	Gemini
mcp	>=1.29,<2	Model Context Protocol client

3.5.4 Document extraction

Package	Version	Purpose
docling[rapidocr]	2.116.0	Document conversion with OCR
docling-core	2.88.0	Docling core
docling-ibm-models	3.13.3	Docling models
docling-parse	7.8.1	Docling PDF parser
rapidocr	3.9.2	OCR
onnxruntime	1.28.0	Model inference
pdf2image	>=1.17	PDF rasterizing
pdfplumber	>=0.11	PDF text extraction
pillow	>=10.0	Images
pypdf	>=4.2	PDF utilities
pytesseract	>=0.3	OCR bridge

The layout, table-structure and OCR models come with the runtime, so PDF extraction works without internet access. The Groundzero helper packages (`gz` and others) are already importable.

3.5.5 DuckDB extensions

These extensions are installed. Connectors load the ones they need the first time you use them. For the others, such as `spatial`, run `session.query("LOAD spatial")`. Nothing is downloaded when your code runs.

Extension	Used for
<code>httpsfs</code>	Reading and writing S3, GCS and HTTP objects
<code>aws</code>	AWS credential handling
<code>iceberg</code>	Groundzero Lakehouse and Iceberg tables
<code>parquet</code> , <code>json</code>	File formats
<code>azure</code>	Azure Blob Storage
<code>postgres</code> , <code>mysql</code>	Attach and write PostgreSQL and MySQL databases
<code>spatial</code>	Geospatial types and functions
<code>motherduck</code>	MotherDuck

3.5.6 Packages added by the ML variants

Variant	Packages
scikit-learn	<code>opencv-python-headless</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code>
PyTorch	<code>torch 2.5.1</code> , <code>torchvision 0.20.1 (CUDA 12.1)</code> , <code>opencv-python-headless</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code> , <code>shap</code> , <code>lime</code> , <code>alumentations</code> , <code>huggingface_hub</code> , <code>transformers</code> , <code>simsimd</code> , <code>fastai</code>
TensorFlow	<code>tensorflow</code> , <code>opencv</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code> , <code>shap</code> , <code>lime</code> , <code>nltk</code> , <code>spacy</code> , <code>h5py</code>

3.6 Limits and troubleshooting

3.6.1 Limits and quotas

Limit	Value
Rows returned by a dataset or preview	100,000
Rows per page through the data API	up to 10,000
Runs per session at the same time	1
Wait for a dataset run	300 seconds
Idle session	released about 5 minutes after its last run
Output cache	up to 15 minutes
DuckDB memory	80 percent of the compute's memory, at least 512 MB
DuckDB threads	the compute's CPU count, at least 2
Spill location	<code>/tmp/duckdb-spill</code>
Agent run wait	15 minutes
Corpus run	10,000 documents, 50,000 Agent calls, 1 to 32 workers
REST connector request timeout	60 seconds by default
Log line length before splitting	200,000 characters

Compute (CPU and memory) comes from the compute sizes defined for your site. A job has no maximum duration. A schedule that has an end time stops the run at that time.

WHERE RESULTS ARE LIMITED

The 100,000 row limit applies to rows *returned* as output. Queries inside the run can process any amount of data that fits the compute's memory and local disk.

3.6.2 Things to know

- **DuckDB reads the lakehouse only.** Use the Spark runtime to write lakehouse tables.
- `returnFormat="arrow"` returns a reader in most places. `query()` and the file and database connectors return a `pyarrow.RecordBatchReader`; `readGzLakehouse` returns a `pyarrow.Table`. Call `.read_all()` on a reader.
- **One site per lakehouse connection per session.** To use a second site, start a new session.
- **Placeholders expand everywhere.** Upper-case names such as `{USR}` are replaced even in f-strings and comments.
- **pandas output turns numeric NaN into the text `NaN`** and datetimes into strings. Return a relation or Arrow table to keep types.
- **PDF OCR processes one document model at a time.** Use the Spark runtime for distributed OCR.

- **Identical dataset code is served from cache** for up to 15 minutes.
- **Some imports are blocked in jobs and datasets.** Modules such as `socket`, `urllib`, `http`, `shutil`, `pathlib`, `pickle`, `sqlite3` and `threading` cannot be imported. Use connections and the DuckDB engine instead.

3.6.3 Troubleshooting

Each row gives the message you see, the cause and the fix. Messages are shown exactly as the runtime prints them.

Symptom	Cause	Fix
Either <code>s3FullUrl</code> or both <code>bucketName</code> and <code>filePath</code> must be provided	A read or write was called without a location. The same form applies to <code>readGcsFile</code> and <code>readAzureBlob</code> .	Pass <code>s3FullUrl="s3:// ..."</code> , or both <code>bucketName</code> and <code>filePath</code> .
S3 connection details for 'x' not found or the connection type is incorrect.	The connection name does not exist in this site, is not an S3 connection, or was not attached to the run.	Check the spelling (names are case sensitive) and that the dataset or job has the connection selected.
S3 connection details for 'x' are incomplete	The S3 connection is missing a region, access key or secret.	Edit the connection in Groundzero.
GZLAKEHOUSE writes are not supported from DuckDB for connection 'x'. DuckDB lakehouse access is read-only.	<code>writeGzLakehouse</code> was called.	Switch the job to the Spark runtime, or write to S3 or a database from DuckDB.
Only read-only queries are allowed	<code>readGzLakehouse</code> accepts <code>SELECT</code> , <code>SHOW</code> , <code>DESCRIBE</code> and <code>EXPLAIN</code> only.	Remove <code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> , <code>DDL</code> and similar statements.
Use fully qualified table names.	A lakehouse query referenced a table without its database.	Write <code>retail.orders</code> , not <code>orders</code> .
Either 'query' or 'tableName' must be provided	<code>readGzLakehouse</code> was called with neither.	Provide one.
<code>databaseName</code> is required when <code>tableName</code> is not qualified	You passed <code>tableName="orders"</code> and the connection has no default database.	Use <code>tableName="retail.orders"</code> .
GZLAKEHOUSE 'x' has <code>lakeHouseUrl</code> but no <code>siteName</code> for REST catalog setup.	The site for the connection cannot be worked out.	Pass <code>siteName="..."</code> or <code>siteNames={"x": "..."} to <code>duckSession</code>, or <code>siteName</code> to <code>readGzLakehouse</code>.</code>
GZLAKEHOUSE connection 'x' is already configured for site 'a'; create a new DuckDB session to use site 'b'.	One session can use one site per connection.	Create another <code>duckSession</code> for the second site.

Symptom	Cause	Fix
Authentication failed or You do not have permission (a <code>PermissionError</code>)	The lakehouse rejected the user.	Ask a site administrator for <code>Select</code> permission on the table or database.
Direct access to <code>gz-lakehouse-*</code> buckets is restricted; use the <code>GZLAKEHOUSE</code> connector	Your code tried to read Groundzero's lakehouse storage directly.	Read the table through <code>readGzLakehouse</code> .
Import of 'socket' is not allowed for security reasons (also <code>eval()</code> is not allowed for security reasons and Access to '/var/data.csv' outside sandbox-safe directories is not allowed)	The module, function or path is blocked in jobs and datasets.	Use connections for network access and <code>/tmp</code> for files.
<code>outputDF</code> must be <code>DuckDBPyRelation</code> , <code>pa.Table</code> , <code>pa.RecordBatch</code> , <code>pa.RecordBatchReader</code> , <code>DataFrame</code> , or <code>list</code> . Got X	The value in <code>outputDF</code> (or passed to <code>gz_output</code>) is not a supported type.	Convert with <code>.df()</code> , <code>.to_arrow_table()</code> or <code>list(...)</code> .
Page 3 is out of range. Total pages: 2 for page size 10.	A preview asked for a page past the end of the result.	Ask for a page within the total.
No executable code found.	The run had no code, or the job file was empty.	Save the code and run again.
The dataset session stopped responding before it took the run. Run the dataset again to start a new session.	The session ended before your run started.	Run the dataset again. A new session starts.
Connection not found: x	A dataset or job references a connection that no longer exists in the site.	Recreate the connection or select another one.
Invalid RTE	The runtime name was not recognised.	Choose the runtime again in the runtime selector.
Binder Error: Invalid named parameter "DELIMITER" for function <code>read_csv_auto</code>	A key in <code>readOptions</code> is not a DuckDB option name. The read is tried three times, so you see the last error.	Use the DuckDB name (<code>delim</code> or <code>sep</code> for the separator) and see the candidate list in the message.
Unsupported returnFormat. Use 'duckdb', 'pandas', 'arrow', or 'row_count'.	<code>readSnowflake</code> got another format. Other connectors accept <code>duckdb</code> , <code>pandas</code> and <code>arrow</code> .	Use one of the listed formats.
Out of Memory Error (<code>duckdb.OutOfMemoryException</code>)	The query needed more memory than the DuckDB limit and could not spill enough.	Filter and project earlier, avoid converting large relations to pandas, or choose a larger compute. Run <code>current_setting('memory_limit')</code> to see the limit.

Symptom	Cause	Fix
Could not parse query: ...	<code>readGzLakehouse</code> could not parse your SQL.	Check for dialect-specific syntax.
input has more than <code>maxRows=100,000</code> rows	A local record-linkage call received more rows than <code>maxRows</code> .	Filter the input, or raise <code>maxRows</code> knowingly.
<code>TimeoutError</code> from an Agent call	The wait for an Agent run exceeded 15 minutes. The run is not cancelled.	Look up the run identifier in the error message before you start another invocation.
Runtime is shutting down	A request reached a session that is closing after being idle.	Retry. A new session starts.

3.6.4 FAQ

Can DuckDB write to the Groundzero Lakehouse? No. DuckDB reads the lakehouse only. Write lakehouse tables with the Spark runtime, or write results to S3 or a database from DuckDB.

How do I choose between DuckDB and Spark? Use DuckDB when the data you process fits one machine and you want fast joins and a quick start. Use Spark for distributed processing and lakehouse writes. See Choose the DuckDB runtime.

Why did my dataset show old data? Dataset output is cached by the exact code text. Wait for the cache to expire or change the code. See Return output and previews.

How many rows can a dataset return? Up to 100,000. Aggregate or filter in SQL before returning.

Does state carry over between runs? Do not rely on it. A run may start in a new session.

Can I use `os.environ` and read credentials? No. In jobs and datasets, secret values in `os.environ` read as `****`. Use connections.

Which time zone does the runtime use? Time zone data is included, so named zones such as `UTC` and `Europe/Berlin` work with PyArrow and pandas.

How do I add a Python package? Choose an ML variant, ask an administrator for a custom runtime built from a `requirements.txt`, or `pip install` in a notebook. See Use extra packages.

How do I see why a job failed? Open the job's log view. The error shows the exception type, message and the line of your code that raised it. Printed output appears in the same view.

Are the method names the same as in the Spark runtime? Many connector methods have matching names, but the session object, return types and write support differ. Check the reference for each runtime.

Where do query results go when I use `returnFormat="pandas"`? Into memory as a DataFrame. For large results keep the relation lazy.

3.7 Examples gallery

The examples use fictional data for a retail company, Acme Retail. Connection names (`sales_s3` , `orders_pg` , `lakehouse_main` , ...) are placeholders for your own.

3.7.1 Example 1: Partitioned daily export

Aggregate all order files and write one Parquet folder per region, ready for downstream readers.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

session.readS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders",
)

# The S3 call above configured access, so plain SQL can COPY to S3 too.
session.query("""
    COPY (
        SELECT region,
               CAST(ordered_at AS DATE) AS order_date,
               count(*)                AS orders,
               round(sum(amount), 2)    AS revenue
        FROM orders
        WHERE status <> 'returned'
        GROUP BY ALL
    )
    TO 's3://acme-retail-data/curated/revenue_by_region'
    (FORMAT PARQUET, PARTITION_BY (region), OVERWRITE_OR_IGNORE)
""")

check = session.query(
    "SELECT count(*) AS files FROM glob("
    "'s3://acme-retail-data/curated/revenue_by_region/*.parquet')",
    returnFormat="pandas",
)

print(check)
```

3.7.2 Example 2: Rolling averages with window functions

DuckDB's analytic SQL makes time-series features a one-liner. This computes a 7-day moving average of revenue and ranks days within each region.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders",
)

outputDF = session.query("""
    WITH daily AS (
        SELECT region,
               CAST(ordered_at AS DATE) AS day,
               sum(amount)                AS revenue
        FROM orders
        GROUP BY region, day
    )
    SELECT region, day, round(revenue, 2) AS revenue,
           round(avg(revenue) OVER (
               PARTITION BY region ORDER BY day
               ROWS BETWEEN 6 PRECEDING AND CURRENT ROW), 2) AS revenue_7d_avg,
           rank() OVER (PARTITION BY region ORDER BY revenue DESC) AS best_day_rank
    FROM daily
    ORDER BY region, day
""")

```

3.7.3 Example 3: Flatten JSON events

Semi-structured JSON is a first-class citizen. This reads newline-delimited event files and extracts nested fields without leaving SQL.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/events/events.json",
    fileType="json",
    targetDuckdbTable="events",
)

outputDF = session.query("""
    SELECT user_id,
           event,
           props.page                AS page,
           CAST(props.duration_ms AS INTEGER) AS duration_ms,
           CAST(ts AS TIMESTAMP)      AS ts
    FROM events
    WHERE event IN ('page_view', 'checkout')
    ORDER BY ts
""")

```

3.7.4 Example 4: Churn scoring with scikit-learn

Train a model on features computed in SQL, score every customer and write the scores back. Use the DuckDB scikit-learn (or PyTorch or TensorFlow) variant, or any runtime that includes scikit-learn.

PYTHON

```
from sklearn.linear_model import LogisticRegression

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders",
)

features = session.query("""
    SELECT customer_id,
           count(*)           AS n_orders,
           avg(amount)        AS avg_amount,
           max(amount)        AS max_amount,
           max(CASE WHEN status = 'returned' THEN 1 ELSE 0 END) AS had_return
    FROM orders
    GROUP BY customer_id
""", returnFormat="pandas")

x = features[["n_orders", "avg_amount", "max_amount"]]
y = features["had_return"]
model = LogisticRegression(max_iter=500).fit(x, y)

features["return_risk"] = model.predict_proba(x)[:, 1].round(4)
session.register("scores", features[["customer_id", "return_risk"]])

session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/curated/return_risk.parquet",
    sourceTable="scores",
)

outputDF = session.query("SELECT * FROM scores ORDER BY return_risk DESC LIMIT 10")
```

3.7.5 Example 5: Deduplicate and validate before loading

A light quality gate that runs entirely in DuckDB: reject malformed rows, keep the latest record per order and report what was dropped.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="raw_orders",
)

session.query("""
    CREATE TABLE clean_orders AS
    SELECT * EXCLUDE (rn)
    FROM (
        SELECT *, row_number() OVER (
            PARTITION BY order_id ORDER BY ordered_at DESC) AS rn
        FROM raw_orders
        WHERE amount IS NOT NULL AND amount ≥ 0 AND region IS NOT NULL
    )
    WHERE rn = 1
""")

stats = session.query("""
    SELECT (SELECT count(*) FROM raw_orders) AS raw_rows,
           (SELECT count(*) FROM clean_orders) AS clean_rows
""", returnFormat="pandas")
print(stats.to_dict("records")[0])

session.writeS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/curated/clean_orders.parquet",
    sourceTable="clean_orders",
)

```

3.7.6 Example 6: Customer 360 across lakehouse, PostgreSQL and S3

One query joins three systems. The lakehouse read is governed for the calling user; the PostgreSQL and S3 reads stream into DuckDB.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession(siteName="acme")

orders = session.readGzLakehouse(
    connName="lakehouse_main",
    query="""
        SELECT customer_id, count(*) AS orders, sum(amount) AS revenue
        FROM retail.orders GROUP BY customer_id
    """,
)
profiles = session.executeJdbc(
    connName="orders_pg",
    sqlQuery="SELECT customer_id, segment, signup_date FROM public.customers",
)
tiers = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="reference/loyalty_tiers.csv",
    fileType="csv",
)

session.register("orders", orders)
session.register("profiles", profiles)
session.register("tiers", tiers)

outputDF = session.query("""
    SELECT p.customer_id, p.segment, t.tier,
           o.orders, round(o.revenue, 2) AS revenue
    FROM profiles p
    LEFT JOIN orders o USING (customer_id)
    LEFT JOIN tiers t ON o.revenue BETWEEN t.min_revenue AND t.max_revenue
    ORDER BY revenue DESC NULLS LAST
""")

```

3.7.7 Example 7: Incremental load from Snowflake

Pull only the rows newer than the last load, append them to a Parquet history file set and report the row count.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()

session.readS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/history/orders/*.parquet",
    targetDuckdbTable="history",
)

last = session.query(
    "SELECT coalesce(max(order_ts), TIMESTAMP '1970-01-01') AS w FROM history",
    returnFormat="pandas",
)["w"].iloc[0]

rows = session.readSnowflake(
    conn_name="wh_snowflake",
    sql_query=f"""
        SELECT order_id, customer_id, amount, order_ts
        FROM ANALYTICS.PUBLIC.ORDERS
        WHERE order_ts > '{last:%Y-%m-%d %H:%M:%S}'
    """,
    target_duckdb_table="orders_new",
    returnFormat="row_count",
)

print(f"loaded {rows} new orders")

if rows:
    session.writeS3File(
        connName="sales_s3",
        s3FullUrl=(
            "s3://acme-retail-data/history/orders/"
            f"batch_{last:%Y%m%d%H%M%S}.parquet"
        ),
        sourceTable="orders_new",
    )

```

3.7.8 Example 8: Classify support tickets with an LLM

Read tickets from PostgreSQL, add a sentiment label from an LLM connection and save the result.

`continueOnError=True` keeps one bad row from failing the run.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

tickets = session.executeJdbc(
    connName="orders_pg",
    sqlQuery="SELECT ticket_id, body FROM support.tickets WHERE status = 'open'",
    returnFormat="pandas",
)

labelled = session.complete(
    connName="llm_openai",
    df=tickets,
    promptTemplate=(
        "Classify the sentiment of this support ticket as positive, "
        "neutral or negative. Reply with one word.\n\n{body}"
    ),
    outputColumn="sentiment",
    continueOnError=True,
    timeout=60,
)
session.register("labelled", labelled)

session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/curated/ticket_sentiment.parquet",
    sourceQuery="""
        SELECT ticket_id, lower(trim(sentiment)) AS sentiment,
               sentiment_status, sentiment_total_tokens
        FROM labelled
    """,
)
```

3.7.9 Example 9: Ingest a REST API

Load paginated JSON from an API connection, keep the fields you need and write them as CSV.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()

session.executeREST(
    connName="support_api",
    endpoint="/v2/tickets",
    params={"status": "open", "per_page": 100},
    resultPath="data.items",
    maxPages=20,
    maxRecords=2000,
    targetDuckdbTable="api_tickets",
)

session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/raw/tickets/open.csv",
    fileType="csv",
    sourceQuery="SELECT id, subject, created_at FROM api_tickets",
    writeOptions={"header": True},
)

```

3.7.10 Example 10: Geospatial distance query

The `spatial` extension is installed. This finds the stores within 50 km of a point.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.query("LOAD spatial")

session.query("""
    CREATE TABLE stores AS
    SELECT * FROM (VALUES
        ('Berlin', 52.5200, 13.4050),
        ('Potsdam', 52.3906, 13.0645),
        ('Hamburg', 53.5511, 9.9937)
    ) AS t(store, lat, lon)
""")

outputDF = session.query("""
    SELECT store,
        round(ST_Distance_Sphere(
            ST_Point(lon, lat), ST_Point(13.4050, 52.5200)) / 1000, 1)
        AS km_from_berlin
    FROM stores
    WHERE ST_Distance_Sphere(
        ST_Point(lon, lat), ST_Point(13.4050, 52.5200)) ≤ 50000
    ORDER BY km_from_berlin
""")

```

3.7.11 Example 11: Extract policy fields from PDFs with an Agent

Parse a folder of PDFs and ask an approved Agent to fill your JSON schema for each one. The Agent's response schema, configured in Agent Builder, decides the output shape.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
try:
    result = session.invokeAgentPDFCorpus(
        connName="agents",
        agentId="00000000-0000-4000-8000-000000000001",
        message="Extract the policy number, holder name and effective date.",
        pdfSourceConnName="sales_s3",
        pdfSourcePrefix="incoming/policies/",
        pdfSinkConnName="sales_s3",
        pdfSinkPrefix="agent-results/policies",
        summaryWorkers=8,
    )
    print(result["status"], result["outputMode"])
    print(result["resultsUri"])
    print(result["manifestUri"])
finally:
    session.close()
```

3.7.12 Example 12: A two-step pipeline with executeJob

Put the heavy preparation in a sub-job and let the parent job orchestrate and notify.

PYTHON

```
from GZ.JobUtils import executeJob
from GZ.SendEmail import sendEmail

results = executeJob(["load_orders", "build_summary"])

sendEmail(
    subject="Retail pipeline finished",
    body=f"Completed sub-jobs: {' '.join(results)}",
    to_email="data-team@example.com",
    from_email="noreply@example.com",
)
```

PART 4

Standard Runtime

The default Python and SQL runtime for Groundzero datasets and jobs.

-
- 4.1 Overview
 - 4.2 Quickstart
 - 4.3 How-to guides
 - 4.4 API reference
 - 4.5 Installed packages
 - 4.6 Limits and troubleshooting
 - 4.7 Examples gallery

The Standard runtime is Groundzero's default, zero-infrastructure way to turn a connection into a dataset. You write one read-only query in the connection's own dialect, optionally add a few lines of pandas "Prep" code, and Groundzero returns pages of rows. You size no cluster, start no session and import no helper library.

WHAT \"STANDARD\" MEANS IN GROUNDZERO

The word covers three related things, and this guide keeps them apart.

- **Standard datasets** (the main subject of this guide): the *Standard* choice in the dataset designer. The query runs in Groundzero's managed query engine, not on a runtime you start.
- **The Python (pandas) job runtime**: a runtime for jobs and code datasets, for plain Python and pandas workloads.
- **The base runtime**: the Python 3.12 foundation that the Spark and DuckDB runtimes, and every custom runtime, are built on.

Spark and DuckDB each ship a Groundzero helper library for use in code. Standard has none: its programming surface is SQL plus the Prep contract described in the API reference.

4.1 Overview

4.1.1 What Standard is

A Standard dataset has two parts that you write in the dataset designer.

Part	Editor tab	Language	Runs
Source query	Query	SQL (or connector query)	At the source
Prep code	Prep	Python with pandas	In the query engine

When you press Run, or when a chart, dashboard or API client reads the saved dataset, the engine does the following.

1. Checks that you hold Execute permission on the dataset.
2. Substitutes server-side placeholders such as `${USR}` into the query.
3. Applies any structured filters to the query.
4. Looks for a cached answer for the same query and the same access policy.
5. Otherwise runs the query on the connection and fetches the page of rows.
6. Runs your Prep code, if it is more than a pass-through.
7. Returns one page of rows with a schema, and records source lineage.

Because the engine is shared and stateless there is nothing to warm up. A Standard run has no compute size. Startup cost is the connection round trip plus the query itself.

4.1.2 How it runs

When you run a Standard dataset, Groundzero looks for a cached answer, then runs your query on the connection, applies your Prep code to the fetched rows and returns a page of rows with its schema.

Key properties:

- **One connection per dataset.** The designer offers Standard only when zero or one connection is selected, and only when that connection is a SQL source or the Groundzero lakehouse.
- **Server-side paging.** For SQL connectors the page window is pushed into the source query, so a page view does not fetch the whole table.
- **Shared caching.** Identical requests are answered from a cache that is checked against the caller's access policy on every hit.
- **Lakehouse reads.** Groundzero lakehouse (Iceberg) tables are read directly from object storage.

4.1.3 Versions

Component	Version	Used for
Python	3.12	Prep code
DuckDB	1.5.5	Lakehouse (Iceberg) reads

Component	Version	Used for
pandas	Installed, version not pinned	Prep code
Python (pandas) job runtime	Python 3.12, NumPy 2.3.5, pandas 2.x	Jobs

4.1.4 Choose the Standard runtime

Choose Standard when:

- the data lives in one SQL-capable connection and the source can do the heavy work (joins, filters, aggregation);
- you want a dataset for charts, dashboards, pivots, exports or API reads;
- the result is small enough to page through, and any reshaping is a few lines of pandas;
- you want an instant first result with no session start.

Choose another runtime when:

You need	Use
Joins across several connections	DuckDB or Spark
Writes to the lakehouse or other sinks	DuckDB or Spark job
Datasets larger than memory, wide shuffles	Spark
Machine learning with scikit-learn, PyTorch or TensorFlow	A DuckDB or Spark ML variant
Extra Python packages	A custom runtime

4.1.5 The Python (pandas) job runtime

Jobs and code datasets can also run on a *pandas* runtime. Groundzero starts the runtime you select with the compute size you choose and runs your job code there.

No Groundzero Python helper package is provided for it; connect to sources with the drivers that come with the runtime (see Installed packages). Use the runtime picker to see which pandas runtimes are available to you.

4.1.6 The base runtime and custom runtimes

Every Groundzero runtime starts from the same base: Python 3.12 with Jupyter, MLflow, the AWS command line and common database client libraries. The Spark and DuckDB runtimes add their engines on top. A *custom runtime* adds your own pip packages on top of any of those runtimes, see the how-to guide Install extra packages.

4.2 Quickstart

This walk-through builds a Standard dataset of shipped orders by region. It uses a sample `sales.orders` table.

4.2.1 1. Pick a connection

You need a connection to a SQL source or to the Groundzero lakehouse. The examples assume a sample connection named `SalesWarehouse` that exposes a `sales.orders` table with these columns.

Column	Type	Example
order_id	integer	1001
customer_id	integer	7
customer	text	Northwind Traders
region	text	EMEA
amount	decimal	1250.00
quantity	integer	5
status	text	shipped
order_date	date	2026-03-14
sales_rep	text	alice

4.2.2 2. Write the source query

SQL

```
SELECT order_id, customer, region, amount, status, order_date
FROM sales.orders
WHERE status = 'shipped'
      AND order_date ≥ DATE '2026-01-01'
```

4.2.3 3. Add Prep code

Prep receives the fetched rows as `queryData` (a list of dictionaries) and must assign its result to `outputDF`.

PYTHON

```
orders = pd.DataFrame(queryData)
outputDF = (
    orders.groupby("region", as_index=False)
    .agg(orders=("order_id", "count"), revenue=("amount", "sum"))
    .sort_values("revenue", ascending=False)
)
```

LEAVE PREP EMPTY WHEN YOU DO NOT NEED IT

An empty Prep tab, or the single line `outputDF = queryData`, is treated as a pass-through. Groundzero then skips running Prep, keeps server-side paging, and trusts the source schema.

4.2.4 4. Run and save

Run the dataset. The result grid shows one page of rows. Save the dataset to use it in visualizations, dashboards, pivots and exports.

4.2.5 5. Read it from code

Any client that can reach the Groundzero data API can run the same request. The body below is what the designer sends.

JSON

```
{
  "ConnectionName": "SalesWarehouse",
  "datasetId": 42,
  "DataSourceQuery": "SELECT region, amount FROM sales.orders",
  "TransformationQuery": "outputDF = queryData",
  "pageSize": 100,
  "pageNumber": 1
}
```

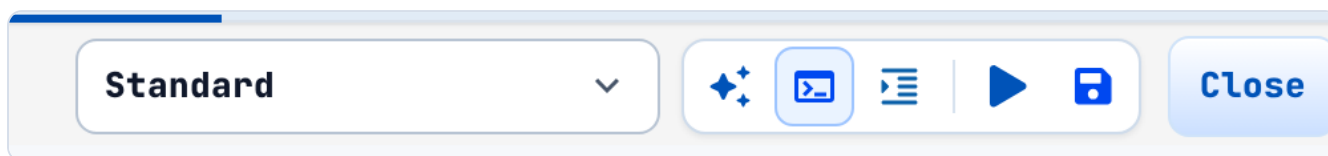
POST it to `/getdata` with your normal `Authorization` header. A successful answer looks like this.

JSON

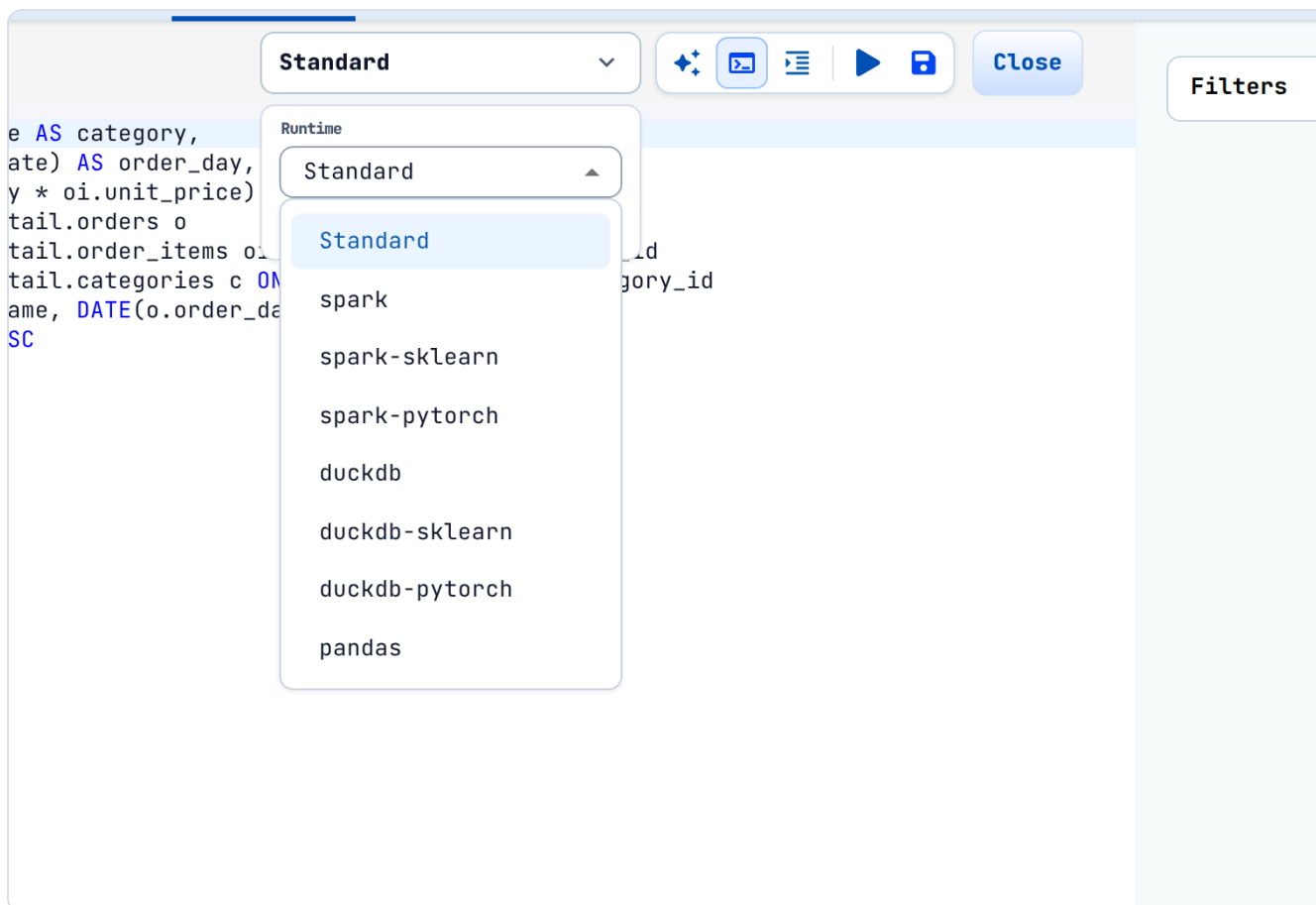
```
{
  "status": "success",
  "data": [{"region": "EMEA", "amount": 1250.0}],
  "totalRecords": 6,
  "hasMore": true,
  "pageStart": 1,
  "pageEnd": 1,
  "pageNumber": 1,
  "schema": [
    {"columnName": "region", "dataType": "text"},
    {"columnName": "amount", "dataType": "decimal"}
  ]
}
```

In the UI

In the dataset designer, the runtime control sits beside the connection picker. **Standard** is the first entry in its Runtime list and is selected by default. When Standard is selected the Compute list is hidden, because Standard has no compute size to choose. Selecting a second connection, or a connection that is not a SQL source or the lakehouse, removes Standard from the list and asks you to choose a runtime and a compute size.



The dataset toolbar with Standard selected.



The Runtime list in the dataset designer: Standard first, then the installed runtimes.

The screenshot displays the Groundzero SQL editor interface. The top navigation bar includes tabs for Workspaces, Connections, Designer (active), DE/NL, Lakehouse, and Catalog. The left sidebar shows a tree view with 'Acme' selected, containing 'Daily Sales by Category', 'Connections', 'AcmeLakehouse', 'Models', and 'System Variables'. The main area is split into 'Query' and 'Prep' tabs. The 'Query' tab shows a SQL query:

```
1 SELECT c.category_name AS category,
2       DATE(o.order_date) AS order_day,
3       SUM(o.quantity * oi.unit_price) AS revenue
4 FROM AcmeLakehouse.retail.orders o
5 JOIN AcmeLakehouse.retail.order_items oi ON oi.order_id = o.order_id
6 JOIN AcmeLakehouse.retail.categories c ON c.category_id = oi.category_id
7 GROUP BY c.category_name, DATE(o.order_date)
8 ORDER BY order_day DESC
```

Below the query editor, the 'Results' tab is active, showing a table with 24 rows and 4 columns. The table headers are 'category' (STRING), 'order_day' (DATE), 'revenue' (DECIMAL), and 'orders' (LONG). The data is as follows:

category	order_day	revenue	orders
Home & Garden	2026-09-30	9000	120
Home & Garden	2026-09-29	9260	121
Home & Garden	2026-09-28	9520	122
Home & Garden	2026-09-27	9780	123
Electronics	2026-09-30	11100	138
Electronics	2026-09-29	11360	139

At the top of the Results tab, it indicates 'RESPONSE 0.167 s', '24 rows', and a dropdown menu set to '500'. The page number 'Page 1 / 1' is also visible.

A Standard dataset run: the SQL editor, the Results grid, row count and response time.

4.3 How-to guides

4.3.1 Read from a connection

Standard supports the SQL sources below. Write the query in the source's own SQL dialect.

Family	Connections
Relational	SQL Server, PostgreSQL, MySQL, Oracle, DB2, SAP HANA, TimescaleDB
Warehouses	Snowflake, Redshift, Google BigQuery, Athena, Teradata, Exasol, Firebolt, SkySQL
Document / search	Cosmos DB, Elasticsearch, Cassandra
Lakehouse	Groundzero lakehouse

SQL

```
SELECT customer, SUM(amount) AS revenue
FROM sales.orders
GROUP BY customer
HAVING SUM(amount) > 1000
ORDER BY revenue DESC
```

Rules of thumb:

- Qualify tables with their schema (`sales.orders`). Lineage uses the qualified name.
- Put filters, joins and aggregations in the SQL. The source is better at them than Prep code.
- Select only the columns you need.

OTHER CONNECTION TYPES

Sources that are not SQL (for example object storage queries) use their own query body. They run in a DuckDB or Spark runtime rather than Standard; the designer will not offer Standard for them.

4.3.2 Query lakehouse tables

Create a connection of type Groundzero lakehouse, then query tables as `schema.table`.

SQL

```
SELECT region, COUNT(*) AS orders, SUM(amount) AS revenue
FROM sales.orders
WHERE order_date ≥ DATE '2026-01-01'
GROUP BY region
```

Reads are scoped to the caller: the lakehouse access policies that apply to you also apply to your result, and cached rows are only reused after the policy check passes.

STANDARD READS, IT DOES NOT WRITE

A Standard dataset runs a read query. To write to lakehouse tables, use a DuckDB or Spark job.

4.3.3 Work with datasets, outputs and previews

A run returns one page. The fields to know:

- `totalRecords` : rows available across all pages.
- `hasMore` : true while more pages remain.
- `pageStart` , `pageEnd` , `pageNumber` : the window you received.
- `schema` : one entry per column with `columnName` and `dataType` .

Request the next page by increasing `pageNumber` . Page size defaults to 1000 and is capped at 1000 rows.

JSON

```
{
  "ConnectionName": "SalesWarehouse",
  "DataSourceQuery": "SELECT * FROM sales.orders ORDER BY order_id",
  "TransformationQuery": "outputDF = queryData",
  "pageSize": 500,
  "pageNumber": 3
}
```

ALWAYS ORDER BY WHEN PAGING

Pages are cut from the result in the order the source returns it. Add an explicit `ORDER BY` so that page 2 never repeats rows from page 1.

Saved datasets can be used as the source of visualizations and dashboards, and appear in the lineage graph with their source tables.

4.3.4 Use parameters and placeholders

Two kinds of placeholder exist.

Server-side placeholders are replaced by the engine before the query runs.

Placeholder	Becomes
<code>\${USR}</code>	The signed-in user name, as a quoted string
<code>\${USRGRPS}</code>	The user's groups, comma-joined, as one quoted string
<code>\${SITENM}</code>	The site name, as a quoted string
<code>\${CLNTNM}</code>	The client name, as a quoted string
<code>\${RGN}</code>	The region, as a quoted string
<code>\${STRTTM}</code>	The request time <code>YYYY-MM-DD HH:MM:SS</code> , wrapped for code (see the warning below)

SQL

```
SELECT order_id, customer, amount
FROM sales.orders
WHERE sales_rep = ${USR}
```

WRITE THE PLACEHOLDER WITHOUT QUOTES

`${USR}`, `${SITENM}`, `${CLNTNM}`, `${RGN}` and `${USRGRPS}` already expand to a single-quoted literal. Writing `'${USR}'` produces doubled quotes and a syntax error. `${STRTTM}` expands to the timestamp wrapped in double and single quotes, which is meant for Prep code (a Python string), not for SQL.

SINGLE BRACES ARE REPLACED TOO

The engine also replaces `{USR}`, `{SITENM}`, `{CLNTNM}`, `{RGN}`, `{USRGRPS}`, `{STRTTM}` and `{ERR}` written with single braces. Avoid those exact names in Python format strings in Prep code.

Filter placeholders `{{filter.prop}}` are filled in by the dashboard or visualization that reads your dataset. If a filter has no value, the run fails with `Filter 'name' has no value` before it reaches the source.

4.3.5 Filter without changing the query text

Dashboards can send filters as data. Groundzero adds them to the query's WHERE clause using parsed expressions, never string concatenation, and keeps cache entries stable across filter changes.

JSON

```
{
  "ConnectionName": "SalesWarehouse",
  "DataSourceQuery": "SELECT o.region, o.amount FROM sales.orders AS o",
  "TransformationQuery": "outputDF = queryData",
  "pageSize": 100,
  "filters": [
    {"column": "region", "table": "o", "op": "in", "values": ["EMEA", "APAC"]},
    {"column": "amount", "table": "o", "op": "gte", "values": [100]}
  ]
}
```

When `filters` is present, send the base query without those filters. Fixed predicates may stay in the query. See the API reference for the operators.

4.3.6 Reshape results with Prep code

Prep is ordinary Python with pandas available as `pd`.

PYTHON

```
orders = pd.DataFrame(queryData)
orders["order_date"] = pd.to_datetime(orders["order_date"])
orders["month"] = orders["order_date"].dt.strftime("%Y-%m")
outputDF = orders.groupby("month", as_index=False)["amount"].sum()
```

`outputDF` may be a DataFrame, a Series, a list of dictionaries, a list of plain values, a dictionary (one row) or a single value. Anything that is not a DataFrame or list of dictionaries is converted to rows (see the API reference).

4.3.7 Schedule with ETL jobs

A Standard dataset answers requests on demand; it is not itself scheduled. To run work on a schedule, create an ETL job on the Python (pandas), DuckDB or Spark runtime and attach a schedule. A scheduled run may be given a stop time; the run then ends at that moment. Job runs go through the statuses Queued, Running, Success, Failed and Aborted, and write logs you can read from the job's run page.

4.3.8 Install extra packages (custom runtime)

Standard datasets cannot import additional packages. For jobs, create a custom runtime.

1. Prepare a `requirements.txt` with one pip requirement per line.
2. Upload it in the custom runtime screen and choose the base runtime to extend and a tag name.
3. Groundzero builds the runtime from that base and installs your packages.
4. Watch the build status move from QUEUED to IN PROGRESS and then SUCCESS or FAILURE.
5. Choose the new runtime when you create the job.

TEXT

```
splink  
sqlglot  
snowflake-connector-python
```

Pin versions (`package=1.2.3`) for repeatable builds.

4.3.9 Tune performance

Order of effect, biggest first.

1. **Push work to the source.** Filter, join and aggregate in SQL.
2. **Keep Prep a pass-through when possible.** Real Prep must see every source row, so Groundzero fetches the full result before it paginates. A pass-through keeps server-side paging.
3. **Select fewer columns and rows.** Cached payloads larger than 8 MiB are not cached.
4. **Use structured filters** so one cached base serves many filter choices.
5. **Prefer the lakehouse connection** for large tables: reads are columnar and go straight to object storage.
6. For results beyond what fits in a page-by-page workflow, move to DuckDB or Spark.

4.3.10 Logging and errors

Standard returns errors in the response body with `status: "error"`, an `errorCode` and a `message`. An exception inside Prep code returns the exception text, with `totalRecords` set to the number of source rows and an empty page. Test Prep logic locally with a small list of dictionaries first (see the last example in the gallery). For jobs, open the run's log view.

4.4 API reference

Standard has no Groundzero Python helper library. Its programming surface is the Prep contract and the data request.

4.4.1 Prep code contract

Prep is executed as a Python program. It sees these names.

Name	Type	Meaning
<code>queryData</code>	list of dict	Rows fetched by the source query, one dictionary per row
<code>pd</code>	module	pandas
<code>json</code>	module	The standard JSON module
<code>outputDF</code>	any	You must assign this. The result

Normalisation of `outputDF` into rows.

You assign	Rows returned
DataFrame	One row per DataFrame row
Series	A one-column frame named after the Series (or <code>value</code>)
List of dictionaries	Unchanged
List of other values	One row per item with column <code>value</code>
Dictionary	A single row
Any other value	A single row with column <code>value</code>

Raises. If Prep throws, or never assigns `outputDF`, the response is `{"status": "error", "message": "<exception text>"}`. A missing `outputDF` reports `'outputDF'`.

Pass-through. Empty Prep and `outputDF= queryData` (whitespace and a trailing semicolon ignored) skip execution entirely.

Example

PYTHON

```
rows = pd.DataFrame(queryData)
outputDF = rows[rows["amount"] > 100]
```

4.4.2 Data request

POST `/getdata` with a JSON object body.

Field	Type	Required	Description
ConnectionName	string	yes	Name of a connection you may use
DataSourceQuery	string	yes	Source query, placeholders allowed
TransformationQuery	string	yes	Prep code, <code>outputDF = queryData</code> for none
datasetId	integer	no	Saved dataset id; enables permission check and lineage
pageSize	integer	no	Rows per page, default 1000, maximum 1000
pageNumber	integer	no	1-based page, default 1
filters	array	no	Structured filters, below

4.4.3 Data response

Field	Type	Description
status	string	success, error or Running
data	array	Rows for the page
totalRecords	integer	Rows across all pages
hasMore	boolean	More pages remain
pageStart, pageEnd	integer	1-based row window
pageNumber	integer	Page returned
schema	array	columnName and dataType per column
errorCode, message		Present when status is error

4.4.4 Structured filters

Each element of `filters`:

Field	Required	Meaning
column	yes	Column name, or dotted path such as <code>o.region</code> (max 128)
table	no	Table alias used in the query; not together with a dotted column
op	yes	in, eq, between, gte, gt, lte, lt
values	yes	Scalars: string, number or boolean

Rules: `between` takes exactly two values; `eq`, `gte`, `gt`, `lte` and `lt` take one; `in` takes 1 to 5000. Strings are limited to 1024 characters. `null`, objects, arrays and non-finite numbers are rejected. Filters combine with AND after any existing WHERE. A malformed filter fails with 400 and never runs the unfiltered query:

JSON

```
{  
  "status": "error",  
  "errorCode": 400,  
  "message": "Invalid filters: filters[0].op must be a filter op"}  
}
```

4.4.5 Error messages

Source	Message
Standard	Request body must be an object (400)
Standard	datasetId must be a positive integer (400)
Standard	Invalid filters: ... (400)
Standard	Filter 'region' has no value (400)
Standard	You do not have "All" or "execute" permission on Dataset.
Standard	Connection type not supported. Please check the dataset.

4.5 Installed packages

4.5.1 Python (pandas) job runtime and base runtime

Python 3.12.

Package	Version
Python	3.12
NumPy	2.3.5
pandas	2.2 or newer, below 3
SQLAlchemy	unpinned
boto3	unpinned
urllib3	unpinned
matplotlib	unpinned
PyMySQL	unpinned
jupyter_server, ipython, ipykernel, jupyterlab	unpinned
MLflow	2.20.4

System tools: the AWS command line, `git`, `curl`, `wget`, `unzip`, `build-essential`, ODBC (`unixodbc`), PostgreSQL and MySQL client libraries, and `tzdata`.

4.5.2 Standard query engine

Package	Version
Python	3.12
DuckDB	1.5.5
pandas	unpinned

4.6 Limits and troubleshooting

4.6.1 Limits and quotas

Limit	Value
Rows per page	1000 (larger values are reduced to 1000)
Connections per Standard dataset	1
Filter values in one <code>in</code>	5000
String filter value	1024 characters
Largest cached result	8 MiB
Cached result lifetime	About 15 minutes (with random jitter up to 20 percent shorter)
Policy and access cache	60 to 120 seconds
Wait for a code dataset's result	300 seconds

4.6.2 Known issues

- A Prep that does real work forces a full fetch before paging, so very large results can be slow or run out of memory. Push work into the SQL.
- Oracle and Teradata queries are fetched in full before paging, because their dialects have no safe way to wrap a page window.

4.6.3 Troubleshooting

Symptom	Cause	Fix
<code>'outputDF'</code>	Prep never assigned <code>outputDF</code>	End Prep with <code>outputDF = ...</code>
Filter <code>'region'</code> has no value	A <code>{{region.prop}}</code> placeholder was left empty	Give the filter a value or remove the placeholder
Syntax error near quotes around a user name	Wrote <code>'\${USR}'</code>	Write <code>\${USR}</code> without quotes
<code>datasetId</code> must be a positive integer	<code>datasetId</code> was text, zero or negative	Send a positive integer or omit it for ad-hoc queries
You do not have "All" or "execute" permission on Dataset.	Missing dataset permission	Ask the dataset owner for Execute
Connection type not supported. Please check the dataset.	The connection type has no Standard connector	Choose DuckDB or Spark
Page 2 repeats rows	No <code>ORDER BY</code>	Add a stable <code>ORDER BY</code>

Symptom	Cause	Fix
Empty frame, <code>KeyError</code> in Prep	Zero rows came back	Guard with <code>if not queryData:</code>
Toast <code>Please select Compute</code>	Chose a DuckDB or Spark runtime without a compute size	Pick a compute size, or choose Standard
Toast <code>Unable to load runtimes</code>	The runtime list request failed	Reload; check your permissions
The dataset session stopped responding before it took the run. Run the dataset again to start a new session.	A dataset session died	Run the dataset again
Query execution timed out.	A DuckDB or Spark run produced no output within 300 seconds	Reduce data, or use a larger compute size

4.6.4 FAQ

Which library does Standard use? SQL and the Prep contract. The Spark and DuckDB runtimes provide helper libraries.

Why is Standard missing from my runtime list? Standard is offered only for datasets, with at most one connection, and only for SQL sources and the lakehouse.

Does Standard use my compute quota? Standard datasets do not use a compute size.

Can two users see each other's cached rows? Only when the access policy permits sharing, and every hit is re-authorized first.

Can Prep call external services or read files? Treat Prep as a pure transformation of `queryData`. For anything else, use a job runtime.

Which timestamp does `$_STRTTM` use? The time the request reached Groundzero.

4.7 Examples gallery

All examples use the sample `sales.orders` table from the Quickstart.

4.7.1 1. Shipped orders in the last quarter

SQL

```
SELECT order_id, customer, region, amount, order_date
FROM sales.orders
WHERE status = 'shipped'
      AND order_date ≥ DATE '2026-01-01'
ORDER BY order_date, order_id
```

4.7.2 2. Revenue by region with a Prep rollup

SQL

```
SELECT region, amount, status
FROM sales.orders
```

PYTHON

```
orders = pd.DataFrame(queryData)
shipped = orders[orders["status"] == "shipped"]
outputDF = (
    shipped.groupby("region", as_index=False)["amount"]
        .sum()
        .rename(columns={"amount": "revenue"})
        .sort_values("revenue", ascending=False)
)
```

4.7.3 3. Monthly trend

SQL

```
SELECT order_date, amount
FROM sales.orders
ORDER BY order_date
```

PYTHON

```
orders = pd.DataFrame(queryData)
orders["month"] = pd.to_datetime(orders["order_date"]).dt.strftime("%Y-%m")
outputDF = orders.groupby("month", as_index=False)["amount"].sum()
```

4.7.4 4. Largest order per region

SQL

```
SELECT order_id, region, customer, amount
FROM (
  SELECT o.*, ROW_NUMBER() OVER (
    PARTITION BY region ORDER BY amount DESC
  ) AS rn
  FROM sales.orders AS o
) ranked
WHERE rn = 1
ORDER BY region
```

4.7.5 5. Pivot status counts by region

SQL

```
SELECT region, status, order_id
FROM sales.orders
```

PYTHON

```
orders = pd.DataFrame(queryData)
outputDF = (
  orders.pivot_table(
    index="region", columns="status", values="order_id",
    aggfunc="count", fill_value=0,
  )
  .reset_index()
)
outputDF.columns.name = None
```

4.7.6 6. Orders owned by the signed-in user

SQL

```
SELECT order_id, customer, amount, status
FROM sales.orders
WHERE sales_rep = ${USR}
ORDER BY order_id
```

4.7.7 7. A single row of headline numbers

SQL

```
SELECT amount, quantity, status
FROM sales.orders
```

PYTHON

```
orders = pd.DataFrame(queryData)
outputDF = {
    "orders": int(len(orders)),
    "revenue": float(orders["amount"].sum()),
    "units": int(orders["quantity"].sum()),
    "shipped_share": float((orders["status"] == "shipped").mean()),
}
```

4.7.8 8. Customers with their segment (join in SQL)

SQL

```
SELECT o.customer, c.segment, SUM(o.amount) AS revenue
FROM sales.orders AS o
JOIN sales.customers AS c ON c.customer_id = o.customer_id
GROUP BY o.customer, c.segment
ORDER BY revenue DESC
```

4.7.9 9. Defensive Prep for empty results

PYTHON

```
orders = pd.DataFrame(queryData)
if orders.empty:
    outputDF = []
else:
    orders["amount"] = orders["amount"].astype(float).round(0)
    outputDF = orders.sort_values("amount", ascending=False).head(5)
```

4.7.10 10. Test Prep locally before you save it

Prep is plain pandas, so you can exercise it with a small list of rows on your own machine.

PYTHON

```
import pandas as pd

queryData = [
    {"order_id": 1, "region": "EMEA", "amount": 120.0},
    {"order_id": 2, "region": "APAC", "amount": 80.0},
    {"order_id": 3, "region": "EMEA", "amount": 40.0},
]

prep = '''
rows = pd.DataFrame(queryData)
outputDF = rows.groupby("region", as_index=False)["amount"].sum()
'''

scope = {"pd": pd, "queryData": queryData}
exec(compile(prepare, "<prep>", "exec"), scope)
print(scope["outputDF"])
```

PART 5

Lakehouse Runtime

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

- 5.1 Overview
- 5.2 Quickstart
- 5.3 How-to guides
- 5.4 SQL reference
- 5.5 Limits and behaviour
- 5.6 Troubleshooting
- 5.7 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.

5.1 Overview

5.1.1 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.

5.1.2 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).

5.1.3 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.

5.1.4 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.

5.1.5 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

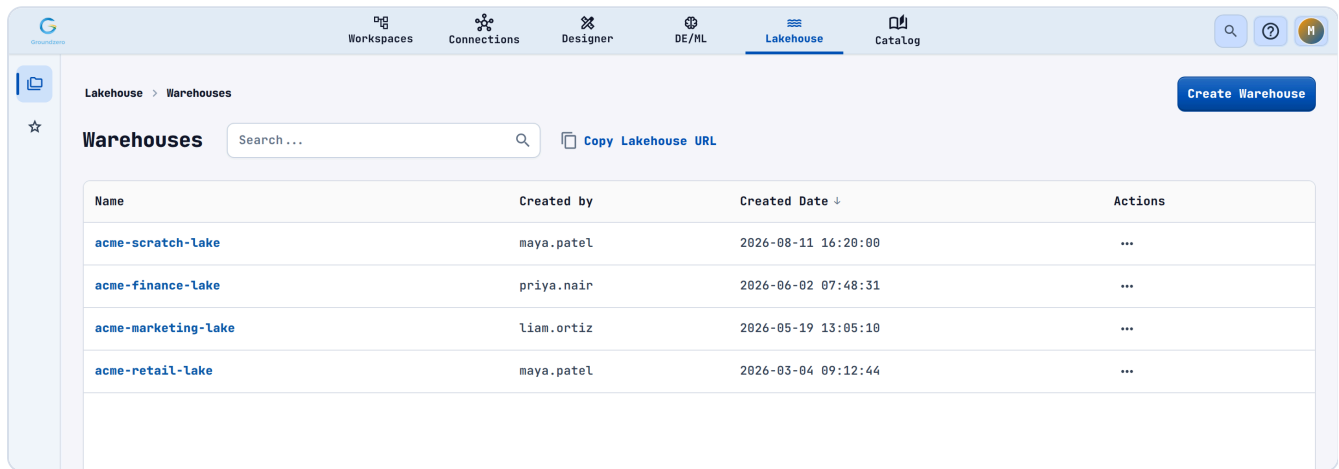
ANSI mode matters for everyday SQL: an invalid cast or a division by zero raises an error instead of returning `NULL`. The functions `try_cast`, `try_divide` and `nullif` are the safe alternatives (see the SQL reference).

5.2 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.

5.2.1 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.



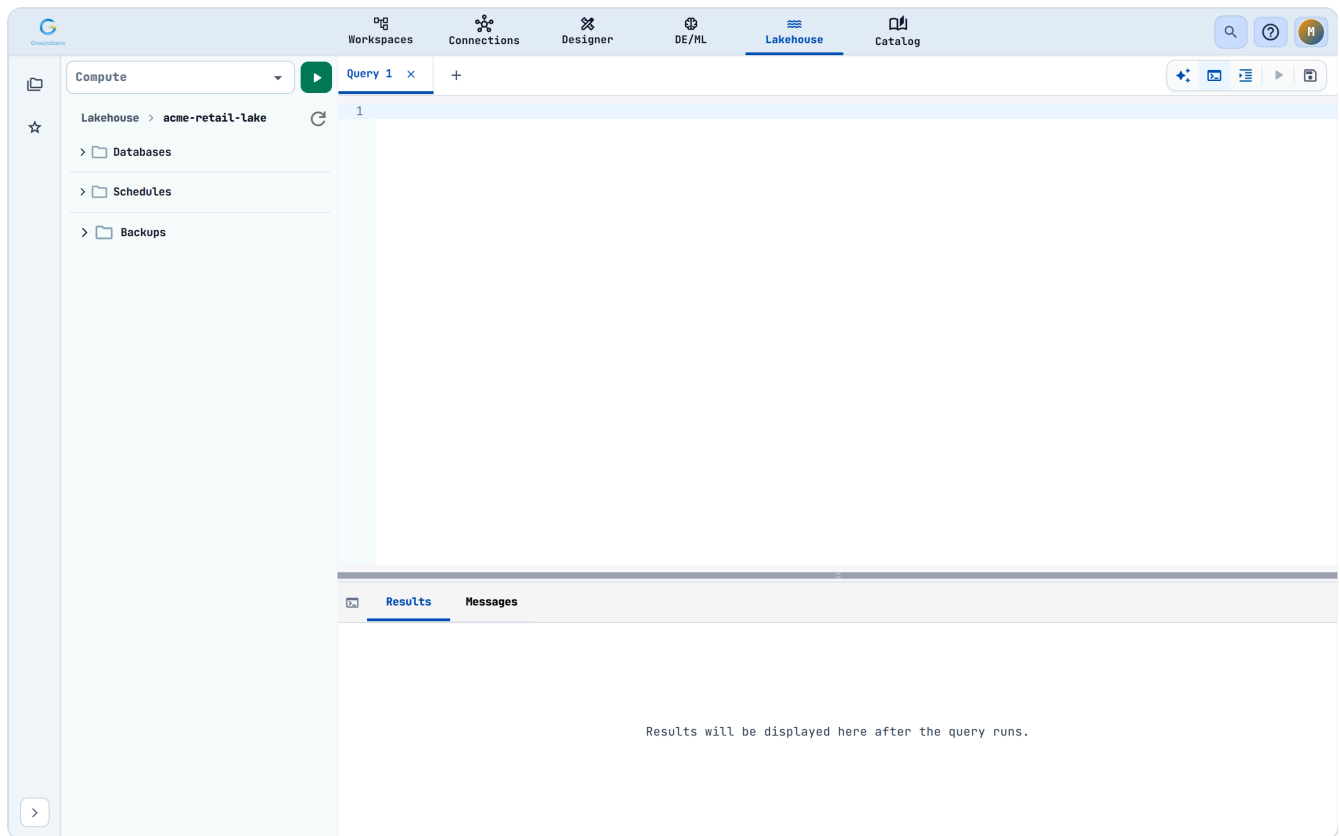
Name	Created by	Created Date ↓	Actions
acme-scratch-lake	maya.patel	2026-08-11 16:28:00	...
acme-finance-lake	priya.nair	2026-06-02 07:48:31	...
acme-marketing-lake	liam.ortiz	2026-05-19 13:05:10	...
acme-retail-lake	maya.patel	2026-03-04 09:12:44	...

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.

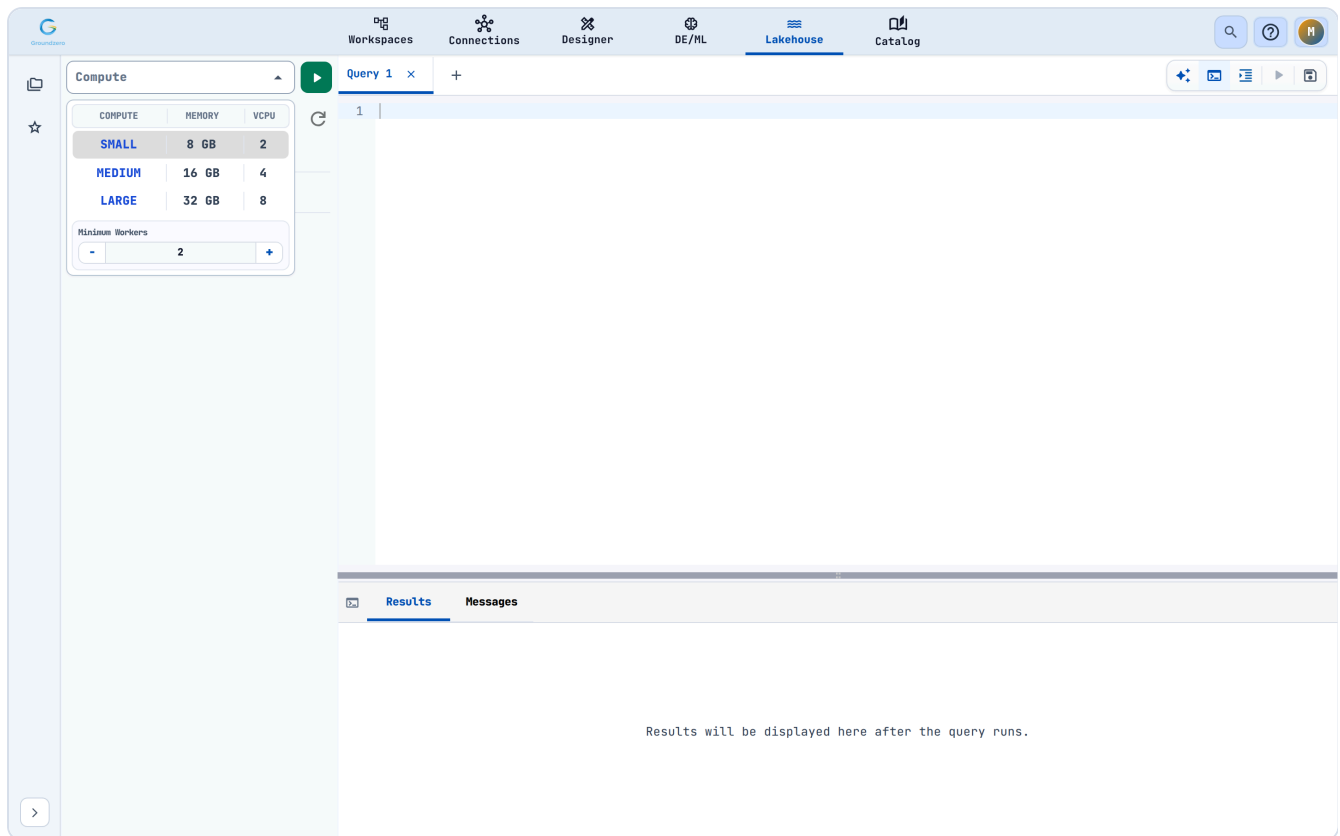
5.2.2 2. Choose a compute and start a session

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.



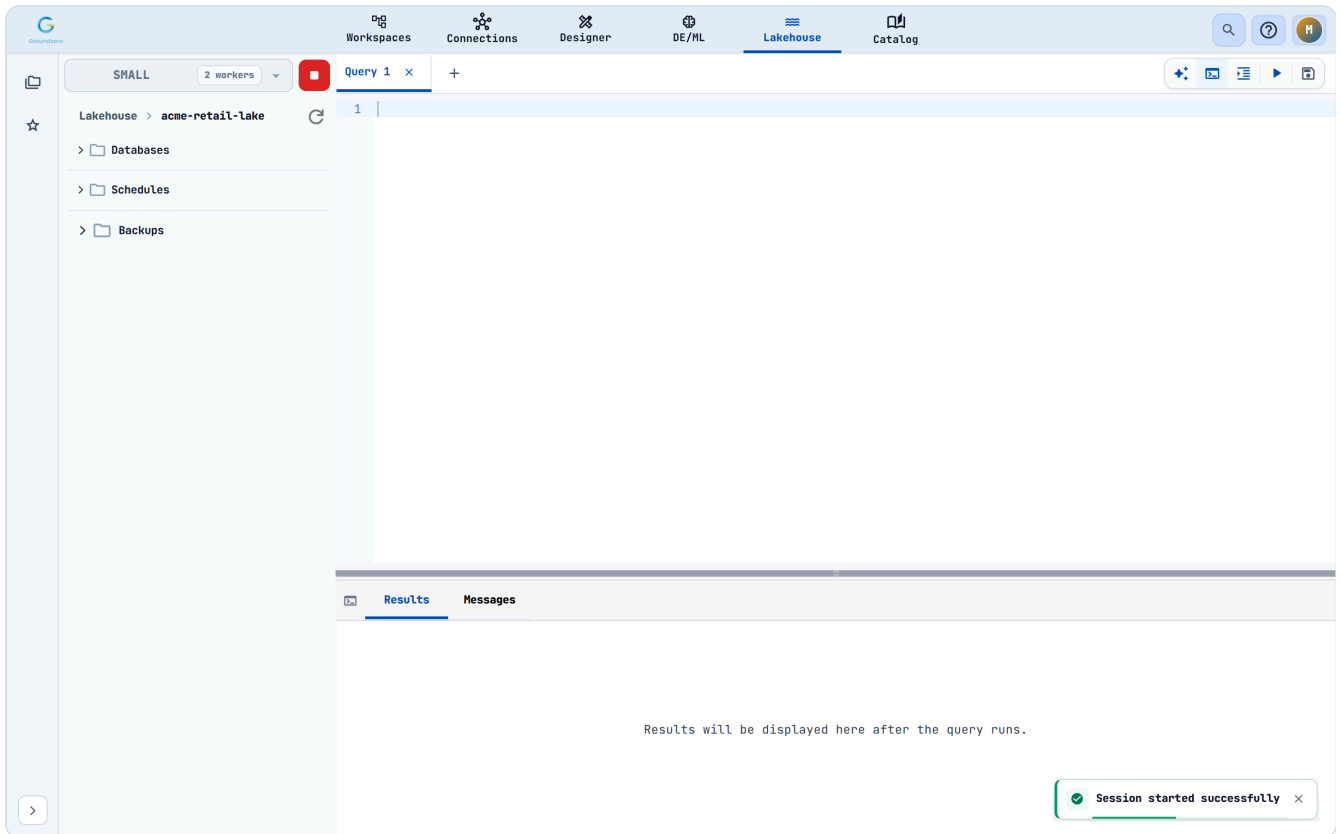
A new workspace before a session starts: compute selector, tree and editor.

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.

5.2.3 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.

5.2.4 4. Insert rows

SQL

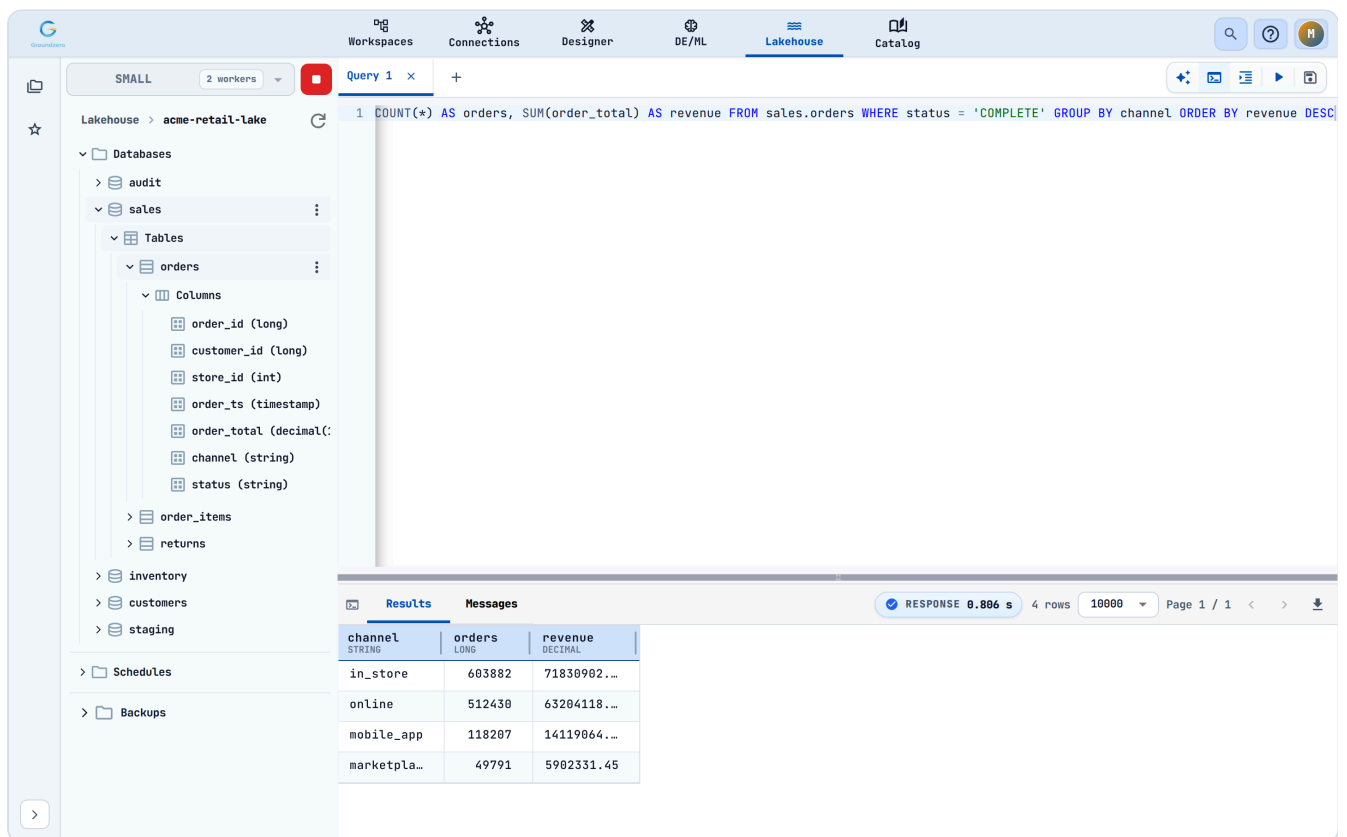
```
INSERT INTO sales.orders VALUES
(1001, 501, 12, TIMESTAMP '2026-09-29 09:14:00', 129.90, 'online', 'COMPLETE'),
(1002, 502, 12, TIMESTAMP '2026-09-29 11:02:00', 45.00, 'in_store', 'COMPLETE'),
(1003, 503, 7, TIMESTAMP '2026-09-30 16:45:00', 310.25, 'mobile_app', 'COMPLETE'),
(1004, 501, 12, TIMESTAMP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

5.2.5 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.

5.2.6 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).

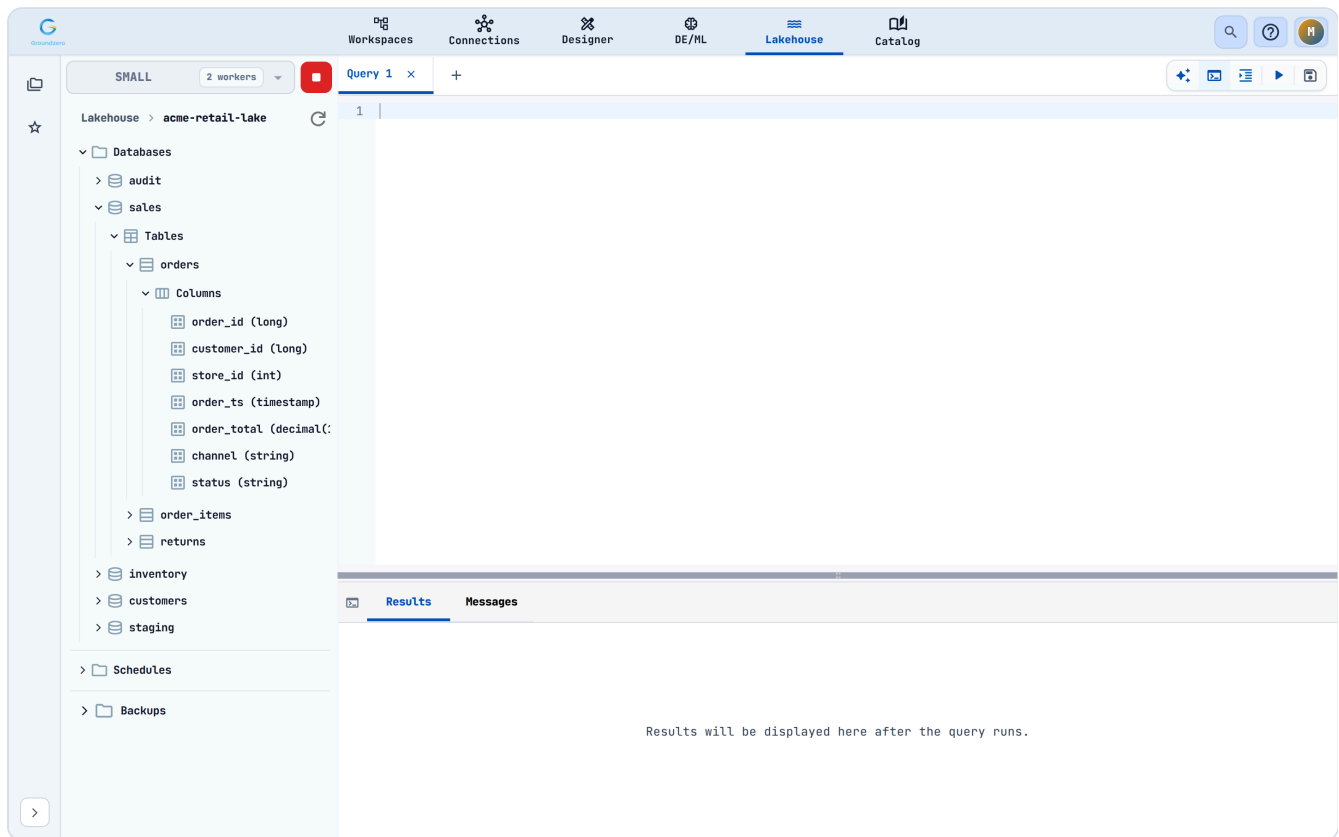
5.2.7 7. Stop the session

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.

5.3 How-to guides

5.3.1 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.



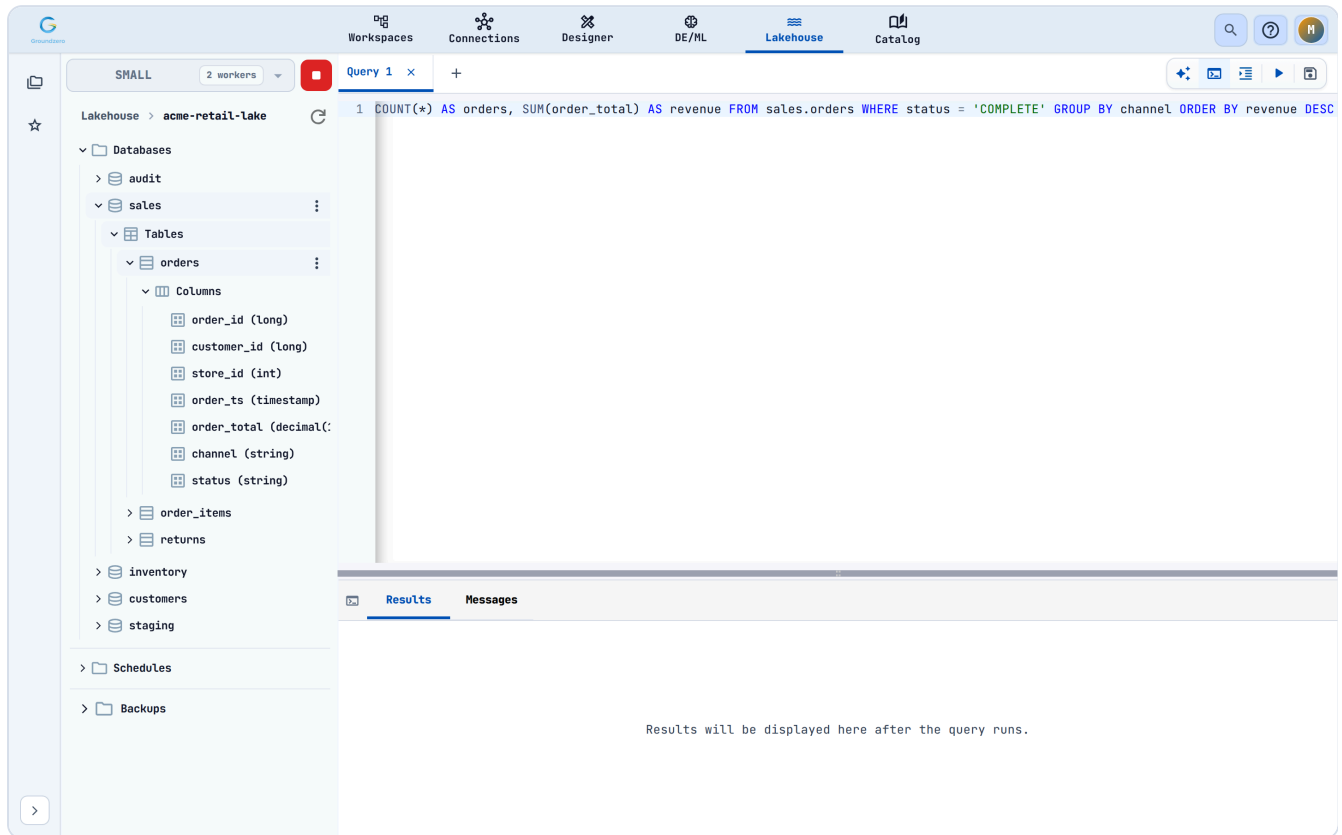
Tables expand to their columns with their types.

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.

5.3.2 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.

5.3.3 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
```

5.3.4 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.

5.3.5 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.

5.3.6 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
```

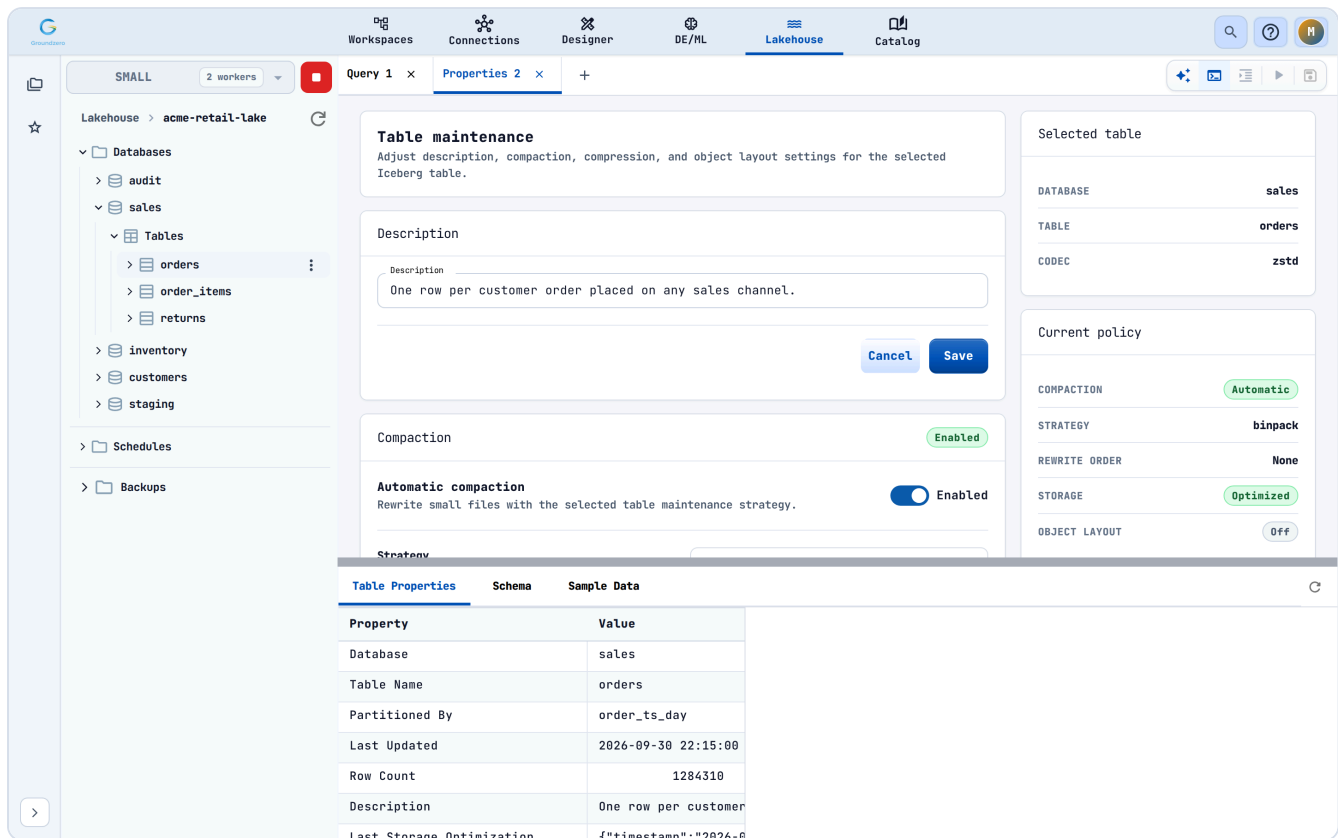
SQL

```
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.

5.3.7 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 screenshot shows the Groundzero Lakehouse interface. The top navigation bar includes tabs for Workspaces, Connections, Designer, DE/ML, Lakehouse (selected), and Catalog. The sidebar on the left shows a tree view of the lakehouse structure, including Databases (audit, sales), Tables (orders, order_items, returns), Schedules, and Backups. The main panel displays the 'Properties' tab for the 'orders' table. It includes a 'Table maintenance' section with a description field and 'Save'/'Cancel' buttons. Below that is a 'Compaction' section with an 'Enabled' toggle and an 'Automatic compaction' section with a 'Enabled' toggle. On the right, there is a 'Selected table' section showing the database (sales), table (orders), and codec (zstd). Below that is a 'Current policy' section showing compaction (Automatic), strategy (binpack), rewrite order (None), storage (Optimized), and object layout (Off). At the bottom, there is a 'Table Properties' tab showing a table with columns 'Property' and 'Value'.

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.

5.3.8 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)

The screenshot shows the Groundzero Lakehouse interface. On the left is a sidebar with a tree view of the database structure: Lakehouse > acme-retail-lake > Databases > sales > Tables > orders. The main panel is titled 'Table maintenance' and contains the following sections:

- Description:** A text area with the description 'One row per customer order placed on any sales channel.' and 'Cancel'/'Save' buttons.
- Compaction:** A section with a green 'Enabled' status indicator.
- Automatic compaction:** A toggle switch set to 'Enabled' with the text 'Rewrite small files with the selected table maintenance strategy.'
- Strategy:** A dropdown menu.
- Selected table:** A table showing the selected table's metadata:

Property	Value
DATABASE	sales
TABLE	orders
CODEC	zstd
- Current policy:** A section with settings:

Property	Value
COMPACTION	Automatic
STRATEGY	binpack
REWRITE ORDER	None
STORAGE	Optimized
OBJECT LAYOUT	Off
- Table Properties:** A table showing the table's 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)
```

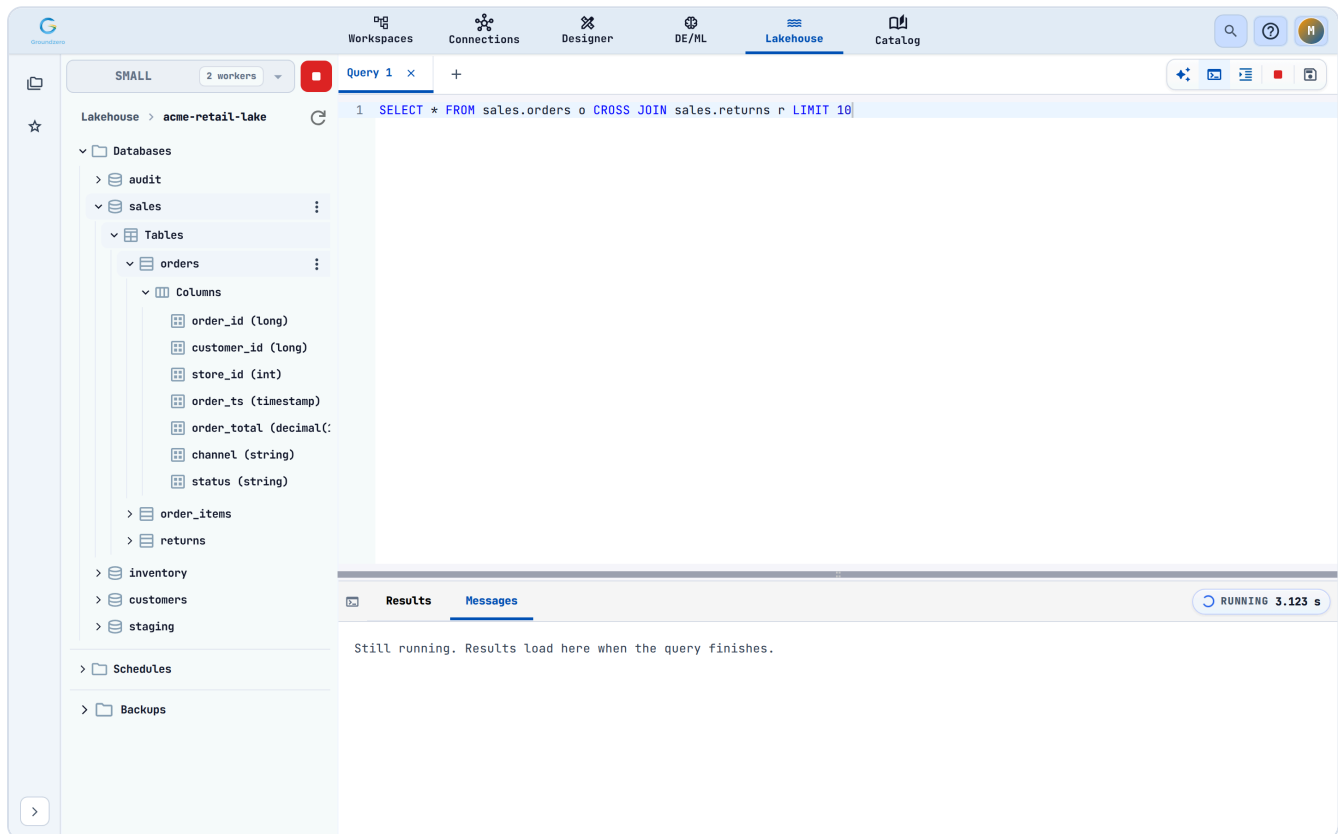
Other table procedures you can call with the AlterTable permission are `rewrite_manifests` (merge manifest files) and `remove_orphan_files` (delete files no snapshot references):

SQL

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

5.3.9 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.

5.3.10 Results, paging and exports

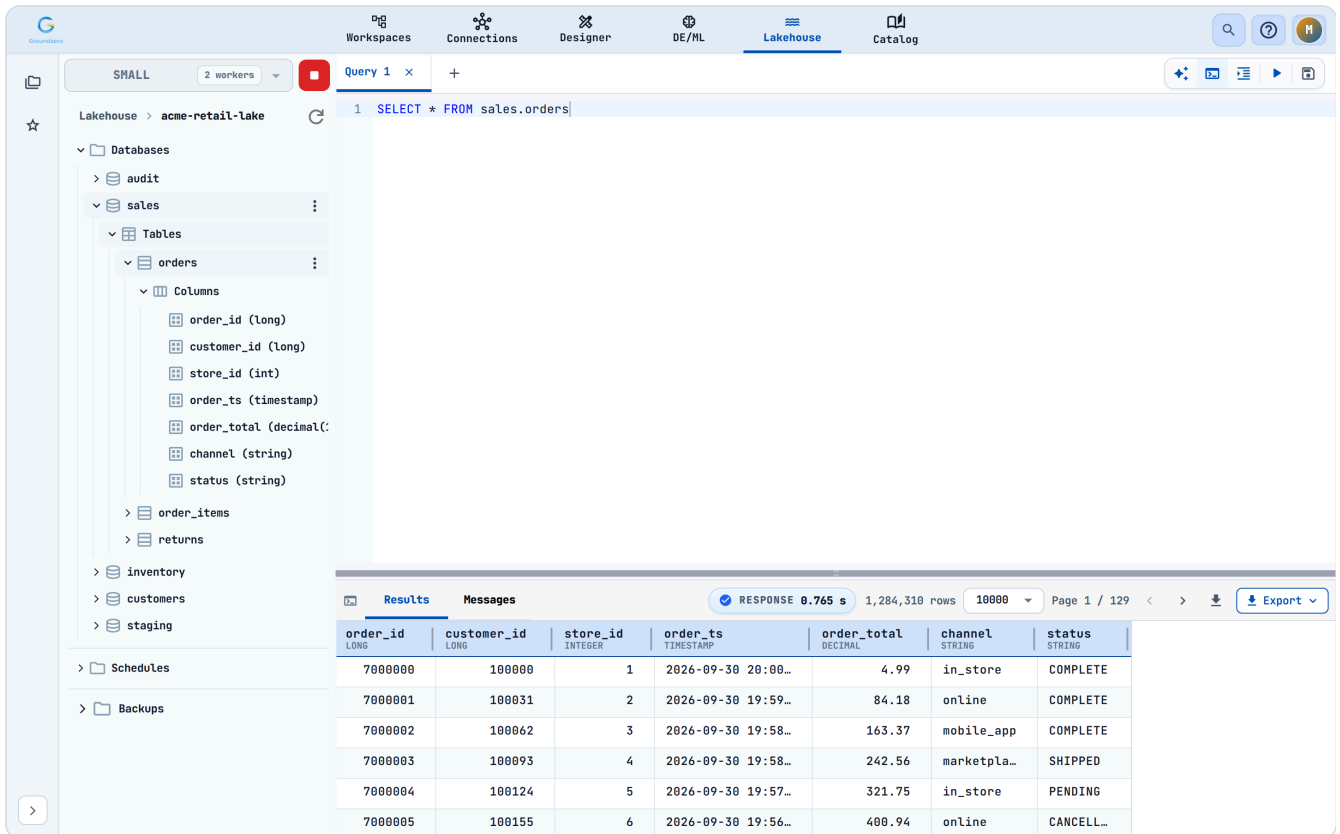
The results grid

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. The top navigation bar includes tabs for Workspaces, Connections, Designer, DE/ML, Lakehouse, and Catalog. The Lakehouse tab is active, showing a query editor with the text: `1 SELECT * FROM sales.orders`. The sidebar on the left displays a tree view of the Lakehouse structure, including Databases (audit, sales), Tables (orders), and Columns (order_id, customer_id, store_id, order_ts, order_total, channel, status). The main area shows the query results in a table format. The table has 7 columns: order_id, customer_id, store_id, order_ts, order_total, channel, and status. The results are paginated, showing Page 1 of 129. The table contains 6 rows of data.

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

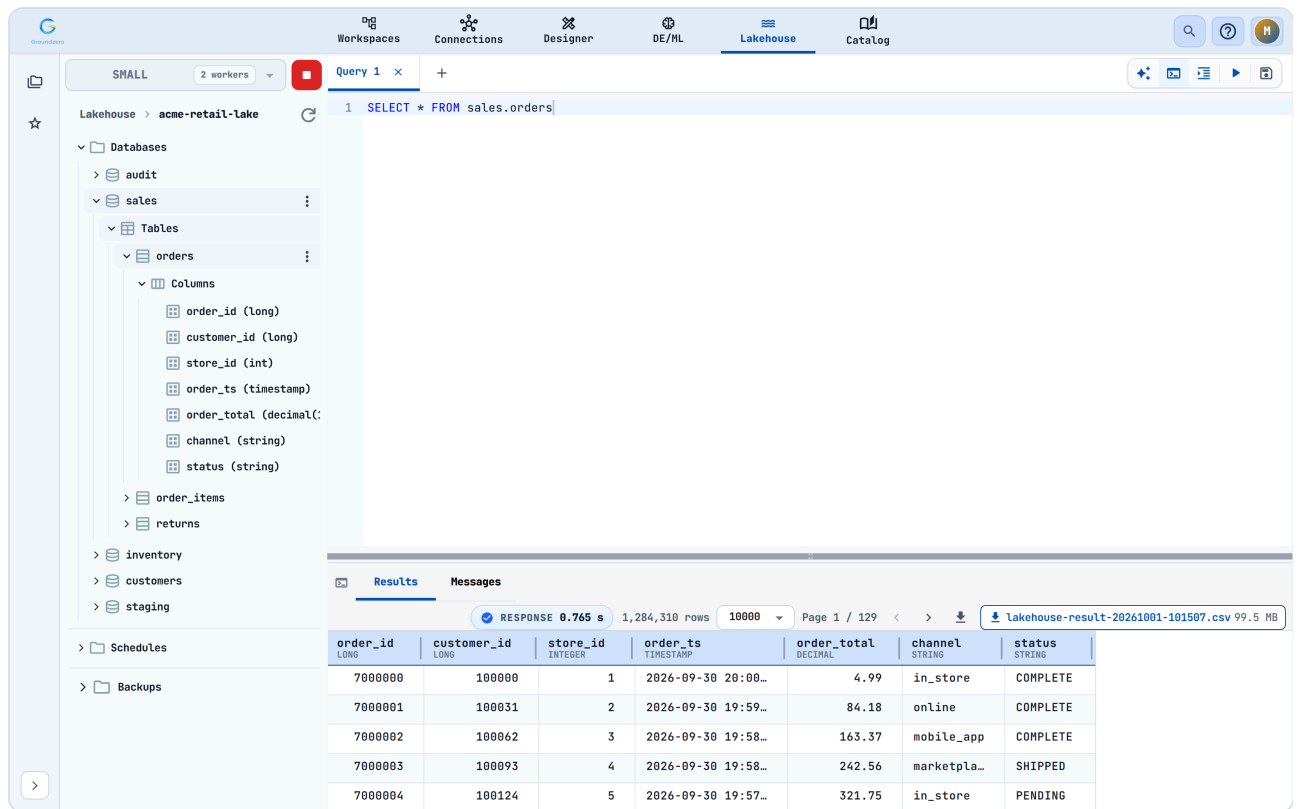
1. Click **Export** and choose **CSV** or **Parquet** in the **Export format** menu.

The screenshot shows the Groundzero Lakehouse interface. On the left, a sidebar displays a tree view of the Lakehouse structure, including Databases (audit, sales), Tables (orders, order_items, returns), Schedules, and 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 displays a query execution result for 'Query 1' with the SQL statement 'SELECT * FROM sales.orders'. Below the query, a table of results is shown with 6 rows and 7 columns. The bottom toolbar includes an 'Export' button with a dropdown menu showing 'CSV' and 'Parquet' options.

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	CANCELLED

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 databases (audit, sales), tables (orders, order_items, returns), and schedules/backups. The 'orders' table is expanded, showing columns: order_id (Long), customer_id (Long), store_id (Int), order_ts (Timestamp), order_total (Decimal), channel (String), and status (String). The main query editor shows a SQL query: `1 SELECT * FROM sales.orders`. Below the editor, the 'Results' tab is active, displaying a table with 5 rows and 7 columns. 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 are:

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

Below the table, a download link is provided: [Lakehouse-result-20261001-101507.csv](#) 99.5 MB.

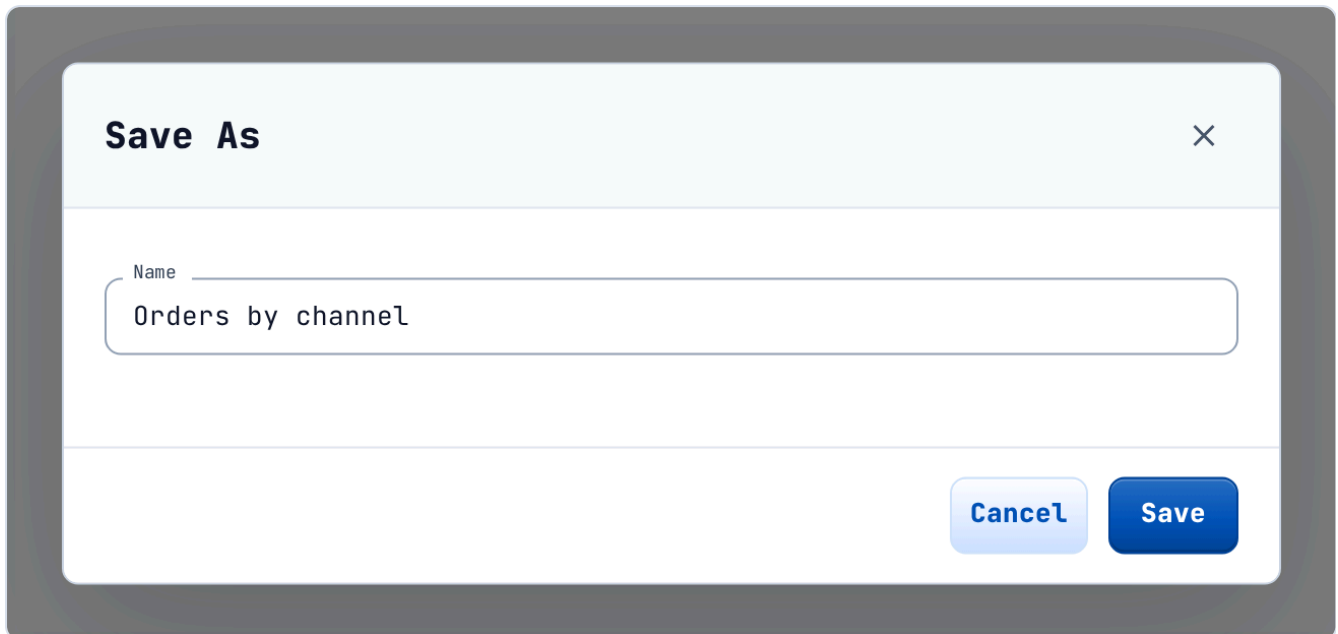
A finished export is a download link.

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.

5.3.11 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.

5.3.12 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".

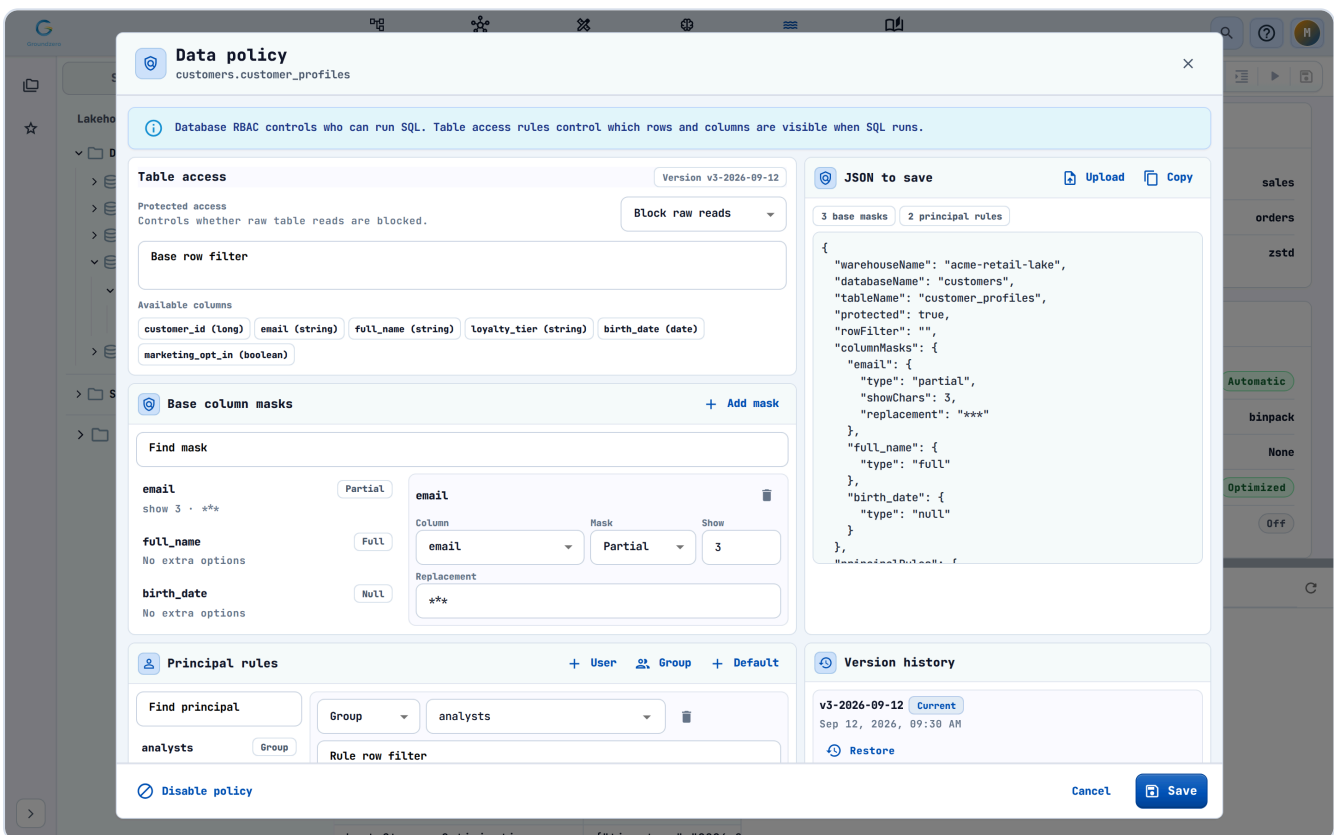
1. **Warehouse:** create, delete and list warehouses, create and drop databases.
2. **Database:** select, insert, update, create table, alter table and drop table on the tables of a database.
3. **Table:** allow or deny rules for single tables, set in the Table Access Manager.

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.

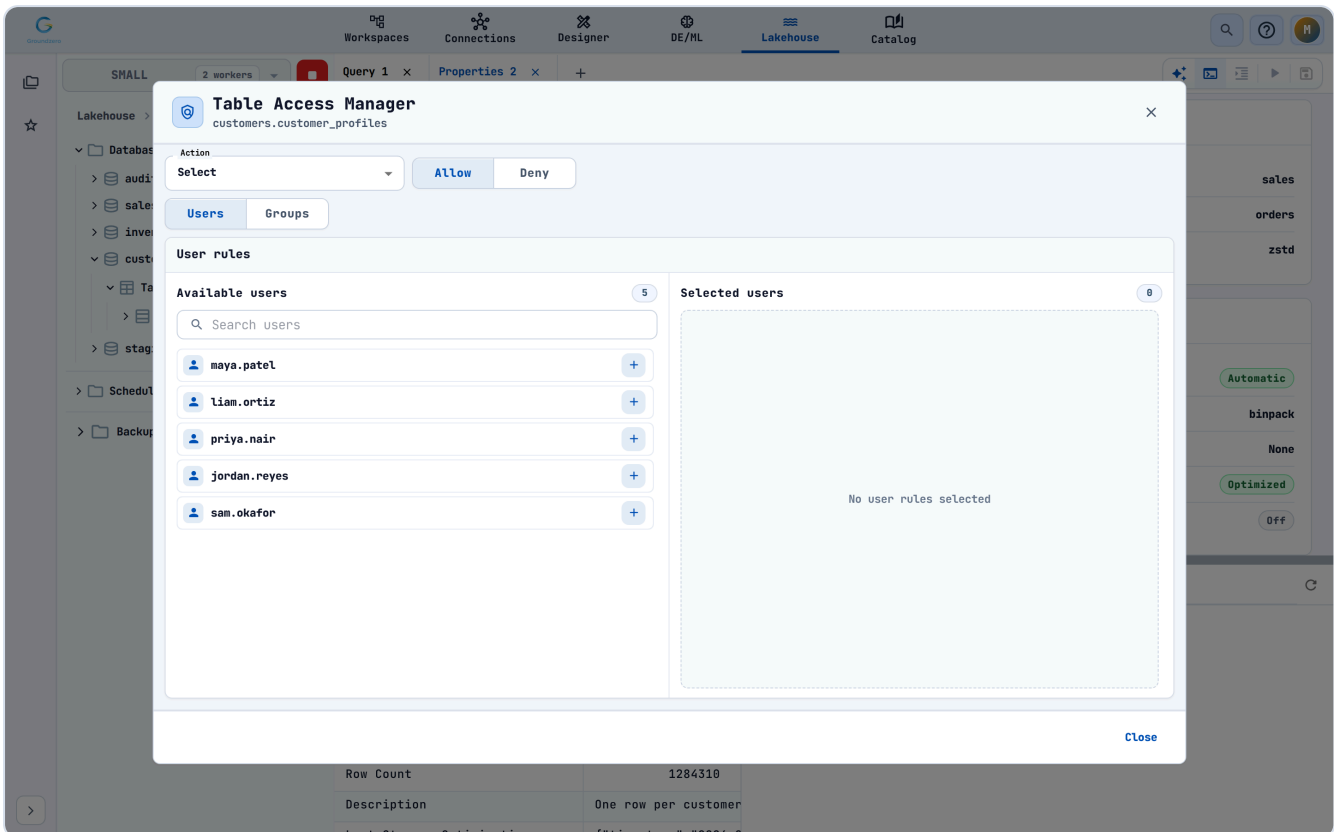


Table Access Manager: action, effect and the users or groups it applies to.

5.3.13 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.

5.3.14 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.

The screenshot shows the Groundzero Lakehouse interface. The top navigation bar includes tabs for Workspaces, Connections, Designer, DE/ML, Lakehouse, and Catalog. The main view is the Lakehouse section, showing a tree view of databases. The 'acme-retail-lake' database is selected, and its 'audit.logs' are displayed. The audit logs table has the following columns: Time, User, Operation, Status, Database, Table, and Duration. The logs show various operations performed by users like maya.patel and liam.ortiz, with statuses ranging from SUCCESS to FAILED. A 'Backups' section is also visible in the sidebar, listing database backups with their status.

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.

5.3.15 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."

5.4 SQL reference

5.4.1 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.

5.4.2 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'))
```

5.4.3 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.

5.4.4 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
```

5.4.5 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 *
```

5.4.6 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.

5.4.7 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)

5.4.8 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.

5.4.9 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.

5.5 Limits and behaviour

5.5.1 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.

5.5.2 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

5.5.3 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.

5.5.4 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

5.6 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

5.7 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>

5.7.1 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
```

5.7.2 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
```

5.7.3 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
```

5.7.4 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
```

5.7.5 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
```

5.7.6 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)
```

5.7.7 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)
```

5.7.8 Example 8: Build a partitioned daily rollout

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
```

5.7.9 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'
```

5.7.10 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
```

5.7.11 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
```

5.7.12 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)
```

5.7.13 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
```

5.7.14 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
```

5.7.15 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)
```

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
```