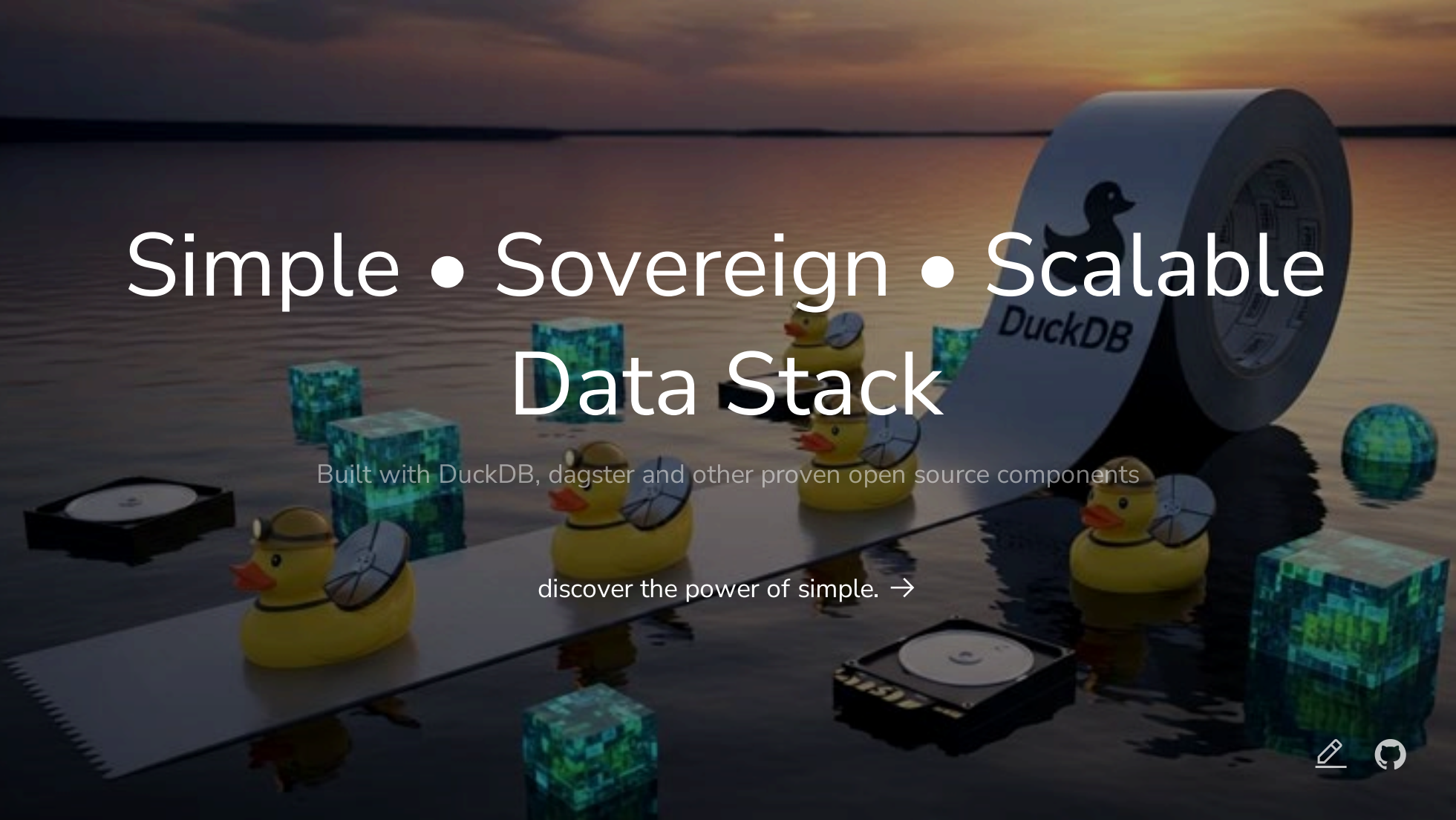


Simple • Sovereign • Scalable Data Stack

Built with DuckDB, dagster and other proven open source components

discover the power of simple. →

Georg Heiler

Solving challenges with data.

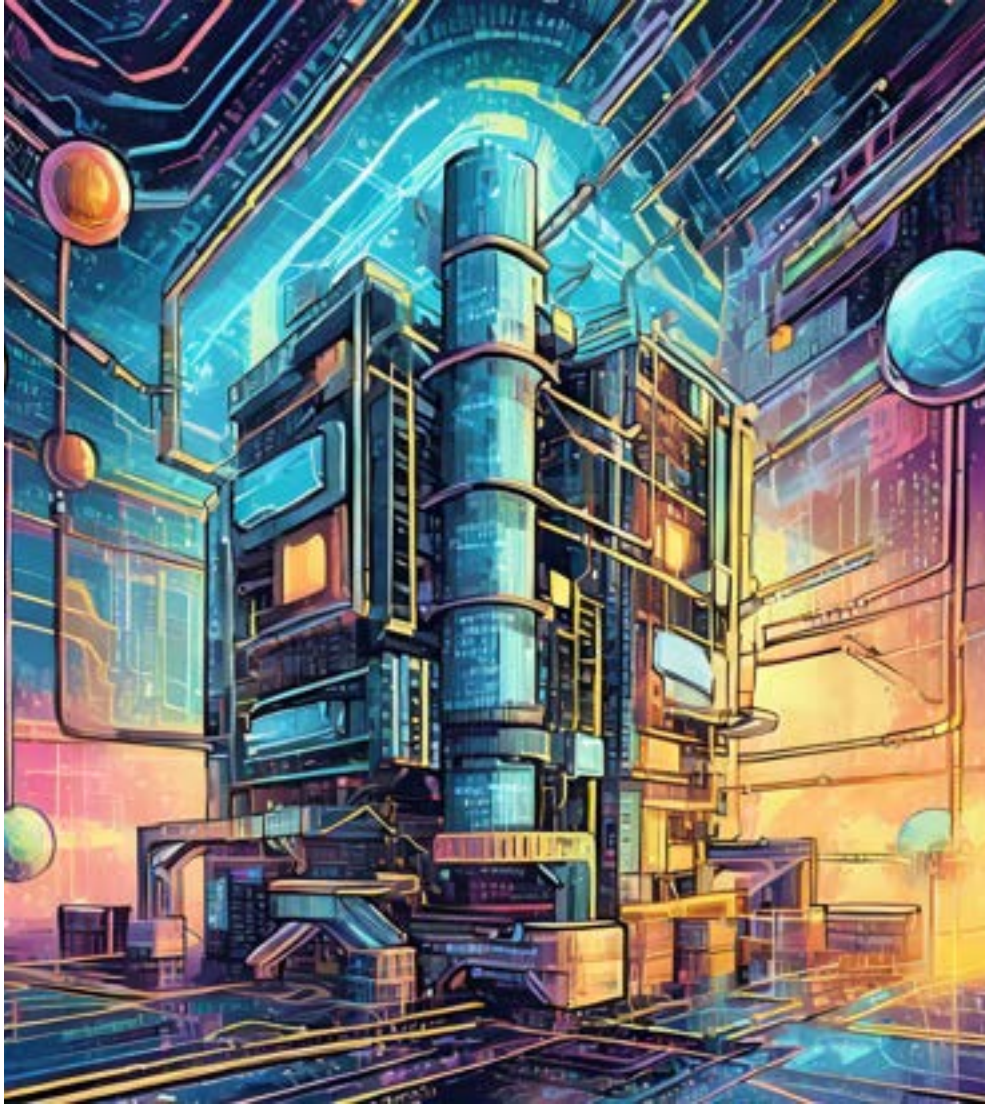
- Senior data expert @Magenta
- Research Software Engineer @ASCII & CSH
- Meetup organizer & frequent Speaker

geoheil.com, linkedin.com/in/geoheil

History

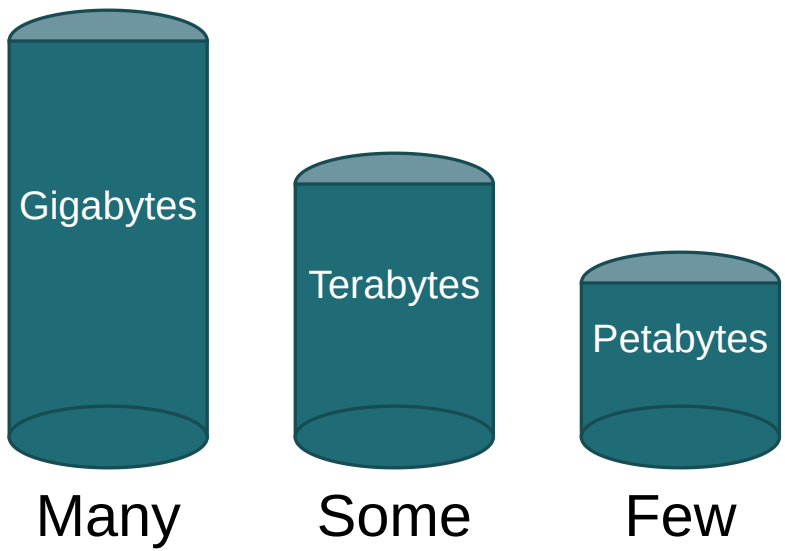
- Mainframe
- Data warehouse
- Big Data (Hadoop)
- SQL on large data (Hive, Spark)
- Cloud DWH (Snowflake, BigQuery)

- Cloud exit?
- SQL is there to stay!



Medium-sized data

- Much of the time we work with medium-sized data
- Too big for traditional data processing applications, too small to justify complex data infrastructure



data accumulates over time

Partitions

local data

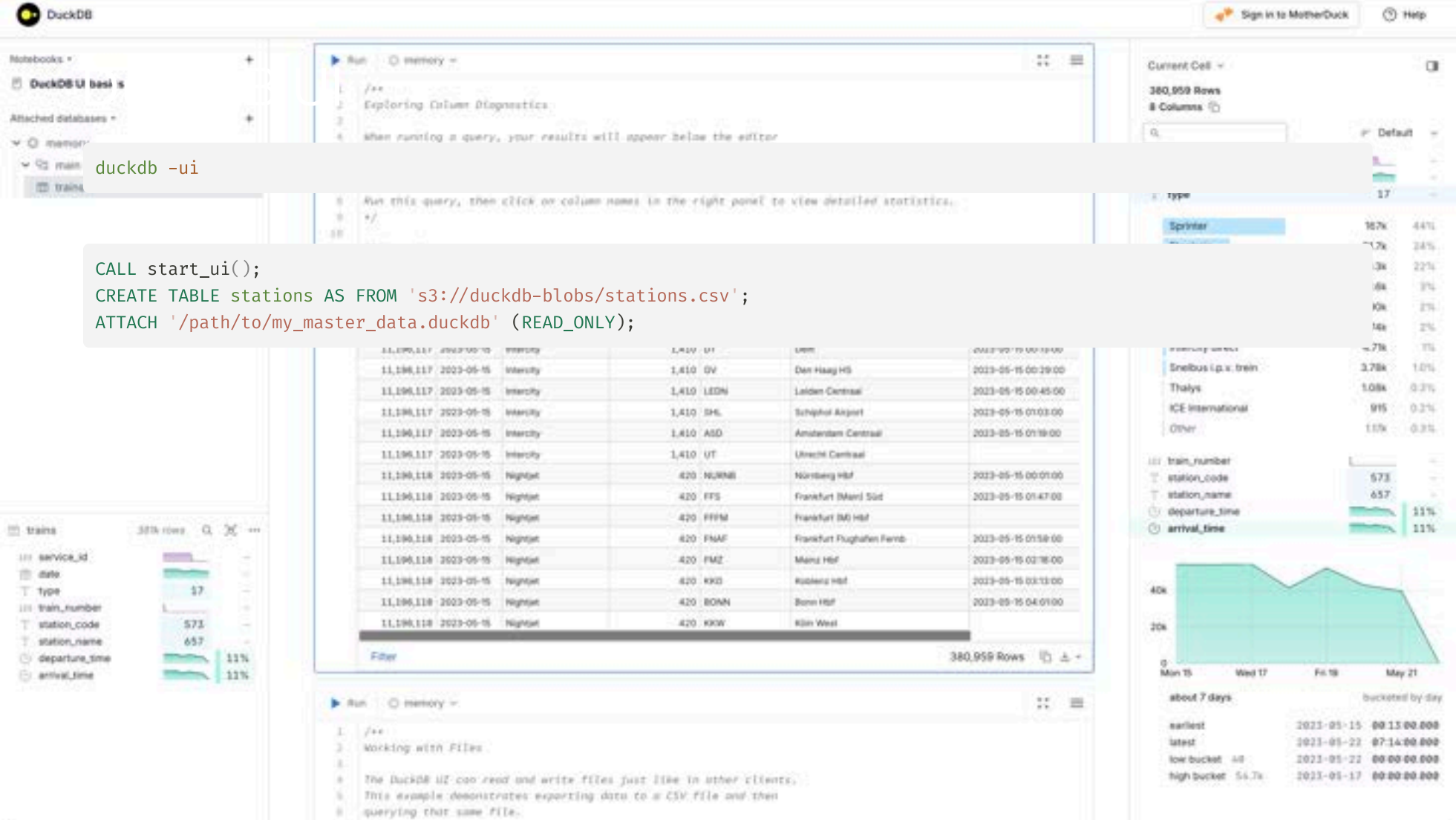
Reduced latency
Enhanced control (privacy, quality)
Not just laptop; k8s is fine

Where does DuckDB fit?

	Transactional	Analytical
Embedded	SQLite RocksDB	**DuckDB**
Stand-alone	Oracle Postgres MongoDB	BigQuery Snowflake Starrocks

DuckDB

- Fast, in-process SQL OLAP database
- Vectorized query engine
- Real database (ACID, memory manager)
- Flexible IO: No ingest mandatory (virtualization?)
- Extension ecosystem
  - `spatial`
  - `duckpgq` (graphs)
  - `vectors` (embeddings)
  - `fts` (full text search)
- Innovative and convenient SQL syntax
- Query plan visualizer


DuckDB
Sign in to MotherDuck
Help
Notebooks
DuckDB UI basics
Attached databases
memory
main
trains
Run
memory
/**
Exploring Column Diagnostics
When running a query, your results will appear below the editor
Run this query, then click on column names in the right panel to view detailed statistics.
*/
duckdb -ui
CALL start_ui();
CREATE TABLE stations AS FROM 's3://duckdb-blobs/stations.csv';
ATTACH '/path/to/my_master_data.duckdb' (READ_ONLY);
11,196,117 2023-05-15 Intercity 1,410 GV Den Haag HS 2023-05-15 00:29:00
11,196,117 2023-05-15 Intercity 1,410 LEDN Leiden Centraal 2023-05-15 00:45:00
11,196,117 2023-05-15 Intercity 1,410 SHL Schiphol Airport 2023-05-15 01:03:00
11,196,117 2023-05-15 Intercity 1,410 ASD Amsterdam Centraal 2023-05-15 01:19:00
11,196,117 2023-05-15 Intercity 1,410 UT Utrecht Centraal
11,196,118 2023-05-15 Nightjet 420 NURNB Nürnberg Hbf 2023-05-15 00:01:00
11,196,118 2023-05-15 Nightjet 420 FFS Frankfurt (Main) Süd 2023-05-15 01:47:00
11,196,118 2023-05-15 Nightjet 420 FFFM Frankfurt (M) Hbf
11,196,118 2023-05-15 Nightjet 420 FNAF Frankfurt Flughafen Fernb. 2023-05-15 01:59:00
11,196,118 2023-05-15 Nightjet 420 FMZ Mainz Hbf 2023-05-15 02:18:00
11,196,118 2023-05-15 Nightjet 420 KKD Koblenz Hbf 2023-05-15 03:13:00
11,196,118 2023-05-15 Nightjet 420 BONN Bonn Hbf 2023-05-15 04:01:00
11,196,118 2023-05-15 Nightjet 420 KKW Köln West
Filter
380,959 Rows
trains
service_id
date
type
train_number
station_code
station_name
departure_time
arrival_time
Run
memory
/**
Working with Files
The DuckDB UI can read and write files just like in other clients.
This example demonstrates exporting data to a CSV file and then
querying that same file.
Current Cell
380,959 Rows
8 Columns
Default
Sprinter
Snelbus i.p.v. trein
Thalys
ICE International
Other
train_number
station_code
station_name
departure_time
arrival_time
40k
20k
0
Mon 15
Wed 17
Fri 19
May 21
about 7 days
bucketed by day
earliest
latest
low bucket
high bucket

DuckDB basics

```
duckdb
```

DuckDB basics

```
CREATE TABLE t1 (
  grp VARCHAR,
  val INT
);

INSERT INTO t1 (grp, val) VALUES
  ('a', 2),
  ('a', 1),
  ('b', 5),
  ('b', 4),
  ('a', 3),
  ('b', 6);

FROM t1;
```

grp	val
varchar	int32
a	2
a	1
b	5
b	4

DuckDB basics

```
-- find the tok-k largest values per group

-- traditional window-function based approach
SELECT array_agg(rs.val), rs.grp
FROM (SELECT val, grp, row_number() OVER (PARTITION BY grp ORDER BY val DESC) AS rid
      FROM t1
      ORDER BY val DESC) AS rs
WHERE rid < 4
GROUP BY rs.grp;

┌──────────────────┬─────────┐
│ array_agg(rs.val) │   grp   │
│     int32[]      │ varchar │
├──────────────────┼─────────┤
│ [3, 2, 1]        │ a       │
│ [6, 5, 4]        │ b       │
└──────────────────┴─────────┘
```

DuckDB basics

```
-- duckdb: shorter + more efficient
SELECT grp, max(val, 3) FROM t1 GROUP BY ALL;

┌─────────┬──────────────┐
│   grp   │ max(val, 3)  │
│ varchar │   int32[]    │
├─────────┼──────────────┤
│ a       │ [3, 2, 1]    │
│ b       │ [6, 5, 4]    │
└─────────┴──────────────┘

-- simpler
-- faster
```

DuckDB basics

```sql
-- direct IO, no prior ingestion needed
SELECT
    station_name,
    count(*) AS num_services
FROM 's3://duckdb-blobs/train_services.parquet'
GROUP BY ALL
ORDER BY num_services DESC
LIMIT 10;
```

DuckDB basics

```
INSTALL tributary FROM community;
LOAD tributary;

-- Read all messages on a topic
SELECT * FROM tributary_scan_topic('test-topic', "bootstrap.servers" := 'kafka-cluster.query.farm:9092');

-- string similarity (fuzzy matching)
INSTALL rapidfuzz FROM community;
LOAD rapidfuzz;

SELECT rapidfuzz_ratio('hello world', 'helo wrld');
┌──────────────────────────────────────────────┐
│ rapidfuzz_ratio('hello world', 'helo wrld')  │
│                    double                    │
├──────────────────────────────────────────────┤
│                     90.0                     │
└──────────────────────────────────────────────┘

```

DuckDB basics

```
ATTACH 'https://github.com/Dtenwolde/duckpgq-docs/raw/refs/heads/main/datasets/snb.duckdb';

USE snb;
install duckpgq FROM community;
LOAD
duckpgq;

CREATE OR REPLACE
PROPERTY GRAPH snb
VERTEX TABLES (
  Person, Forum
)
EDGE TABLES (
  Person_knows_person     SOURCE KEY (Person1Id) REFERENCES Person (id)
                          DESTINATION KEY (Person2Id) REFERENCES Person (id)
                          LABEL knows,
  Forum_hasMember_Person  SOURCE KEY (ForumId) REFERENCES Forum (id)
                          DESTINATION KEY (PersonId) REFERENCES Person (id)
                          LABEL hasMember
);
```

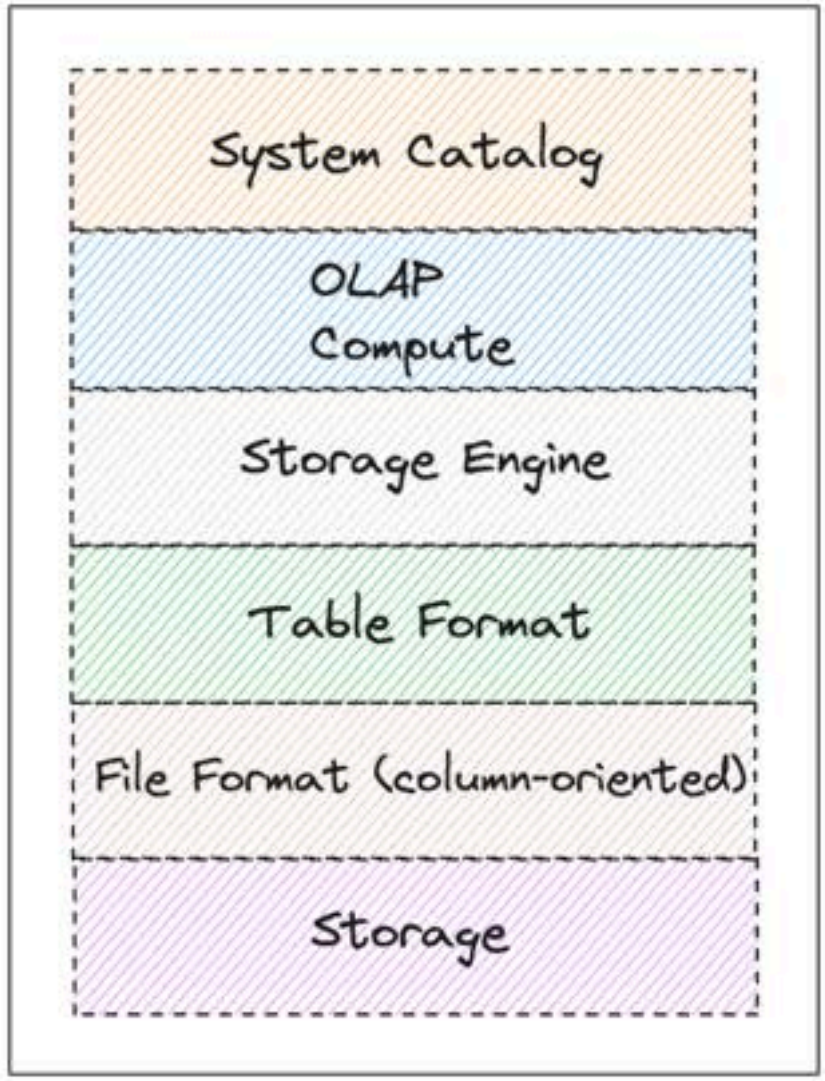
Paradigm shift & limitations

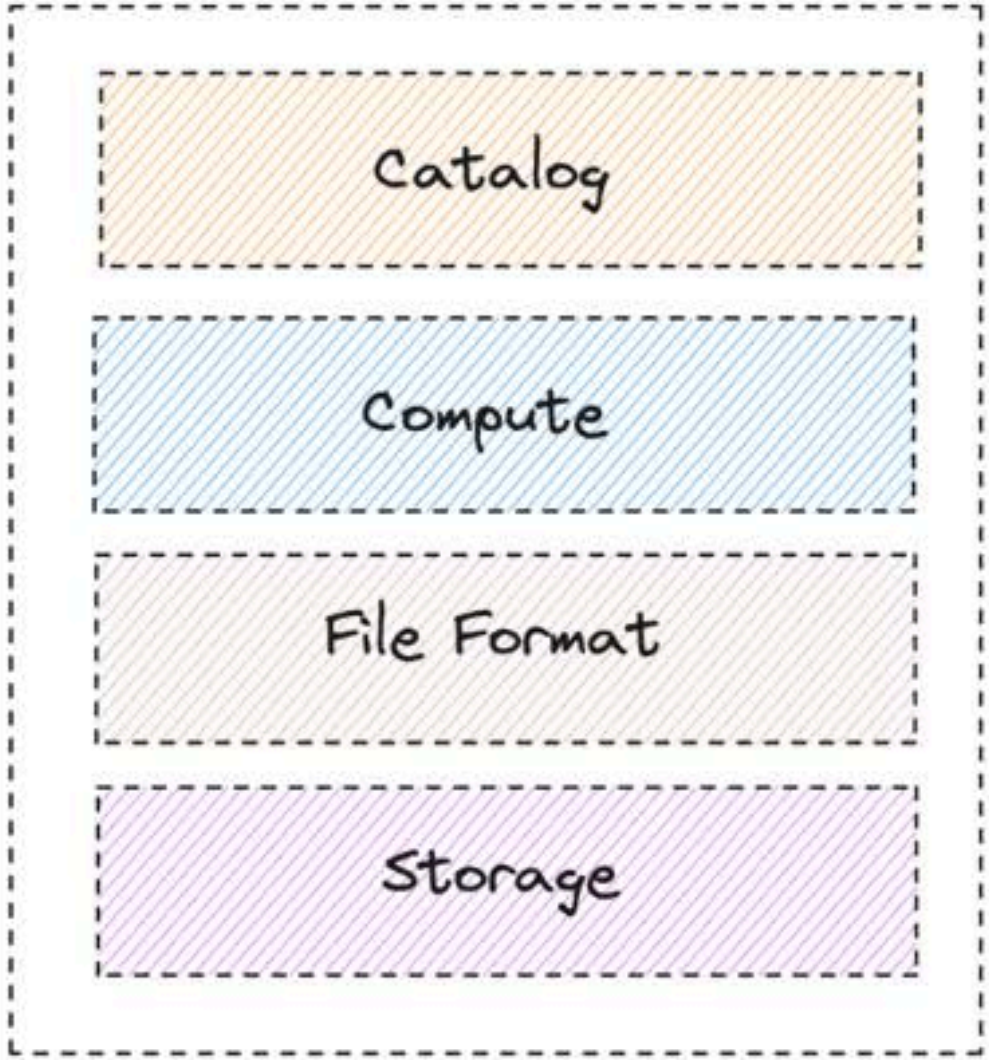
- In-process - No server: Warehouse per user; no mandatory persistent resource allocation
  - No user management
  - Access control via storage layer
  - No dynamic data masking
  - Easily hand a fully hydated database (with masked data) to 3rd parties
  - Possibility of AWS Lambda/Cloudflare Worker massive scale-out deployments
  - Not only a database - dynamic in-process transformation engine
- Arrow zero-copy integration with (data-science/AI) ecosystem + ADBC for fast BI
  - Arrow Flight (columnar streaming) -> first integration: Boilstream streaming ingestion
  - Columnar layout in memory & SIMD
- Latency reduction - client-side analytics (Rill data); even WASM support
- Single writer multiple readers (locking)

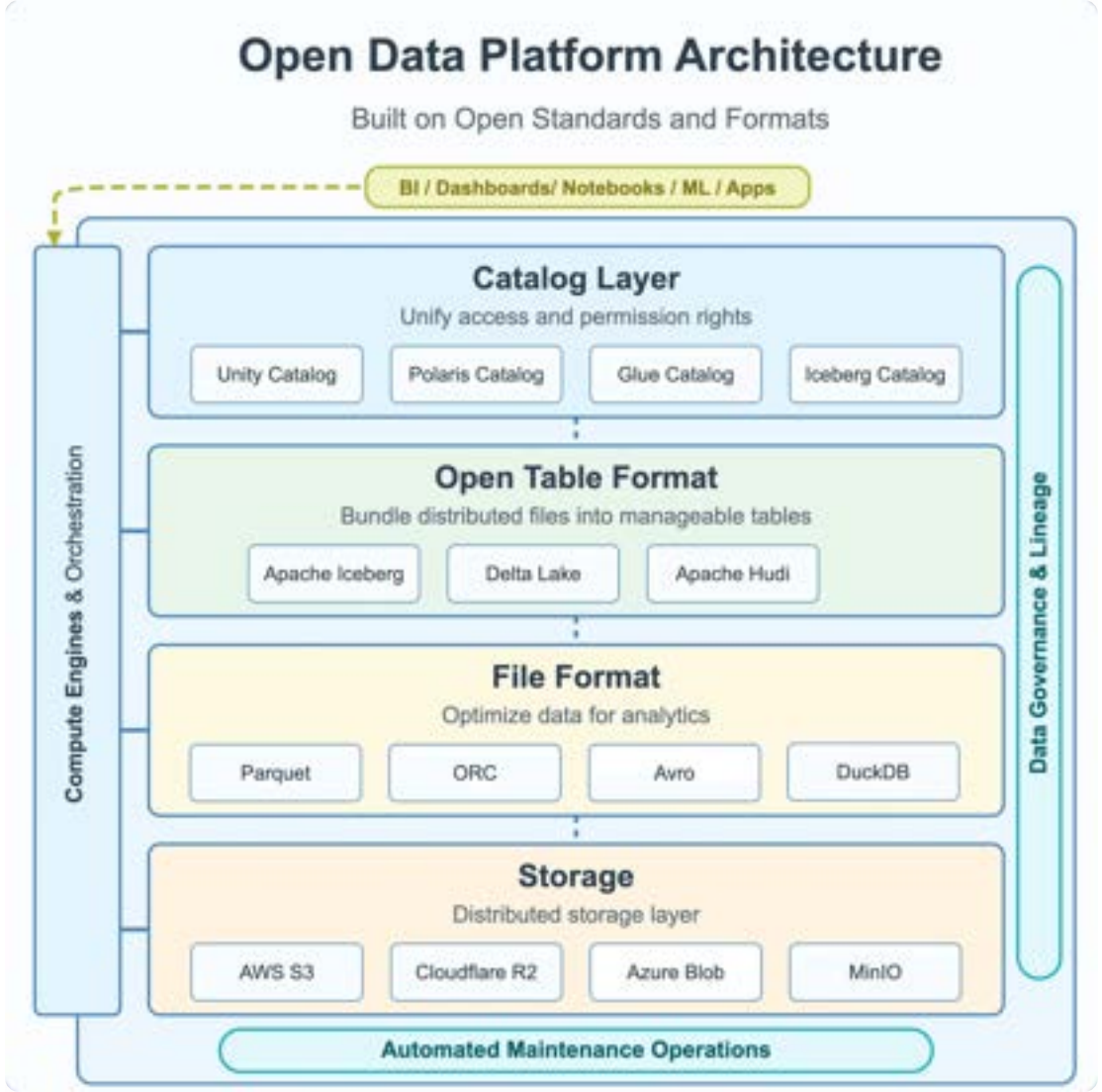
pg_duckdb

- Remember: powerful in-process transformation engine but limitation of single writer
- Combine the strengths of postgres (proven, IAM) as a serving layer with the power of duckdb

```
-- remember DuckDB solo
-- CREATE TABLE stations AS FROM 's3://duckdb-blobs/s

-- combination
CREATE TABLE pg_stations AS
select *
from duckdb.query('SELECT * FROM ''s3://duckdb-blobs/

-- postgres
select *
from pg_stations
limit 10;
```


System Catalog
OLAP
Compute
Storage Engine
Table Format
File Format (column-oriented)
Storage


Catalog
Compute
File Format
Storage


Open Data Platform Architecture
Built on Open Standards and Formats
BI / Dashboards/ Notebooks / ML / Apps
Compute Engines & Orchestration
Catalog Layer
Unify access and permission rights
Unity Catalog
Polaris Catalog
Glue Catalog
Iceberg Catalog
Open Table Format
Bundle distributed files into manageable tables
Apache Iceberg
Delta Lake
Apache Hudi
File Format
Optimize data for analytics
Parquet
ORC
Avro
DuckDB
Storage
Distributed storage layer
AWS S3
Cloudflare R2
Azure Blob
MinIO
Data Governance & Lineage
Automated Maintenance Operations

Iceberg catalog

- S3 until recent - no strong consistency
- Read-after-write consistency was missing
- Impact: Add layer to handle transactions

> TLDR: Add a database in disguise

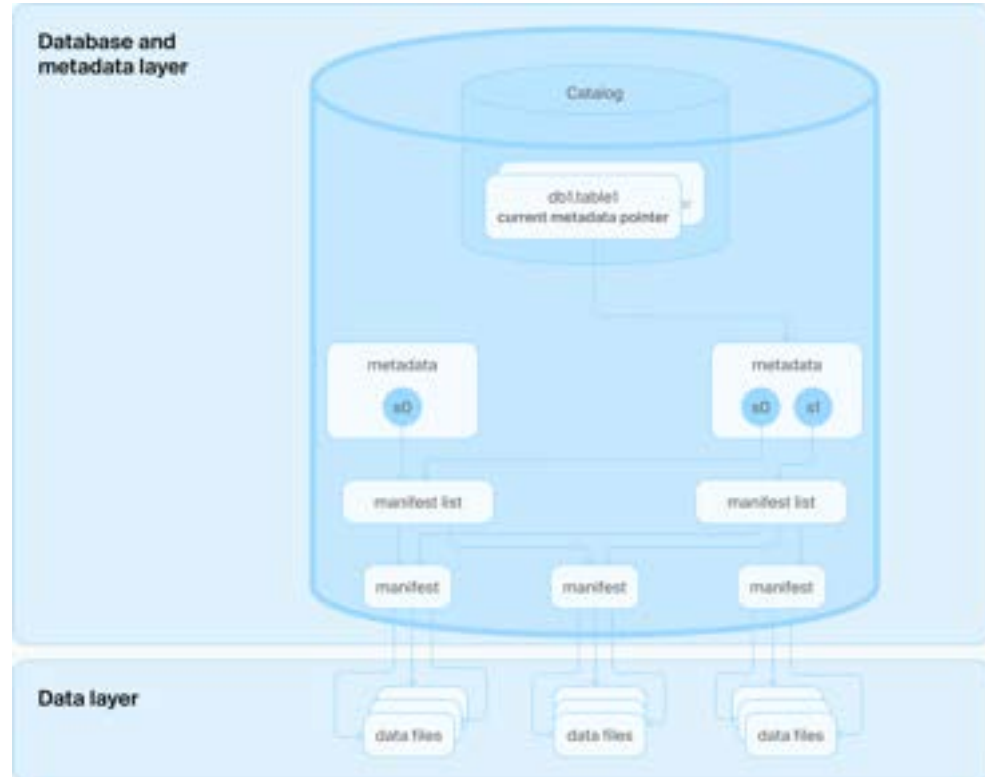
Less features + custom API; no SQL.

- No global transaction.
- No neat SQL filters.
- Lots of **complexity**

Problem: Table format was specifically designed to not require a database

Ducklake

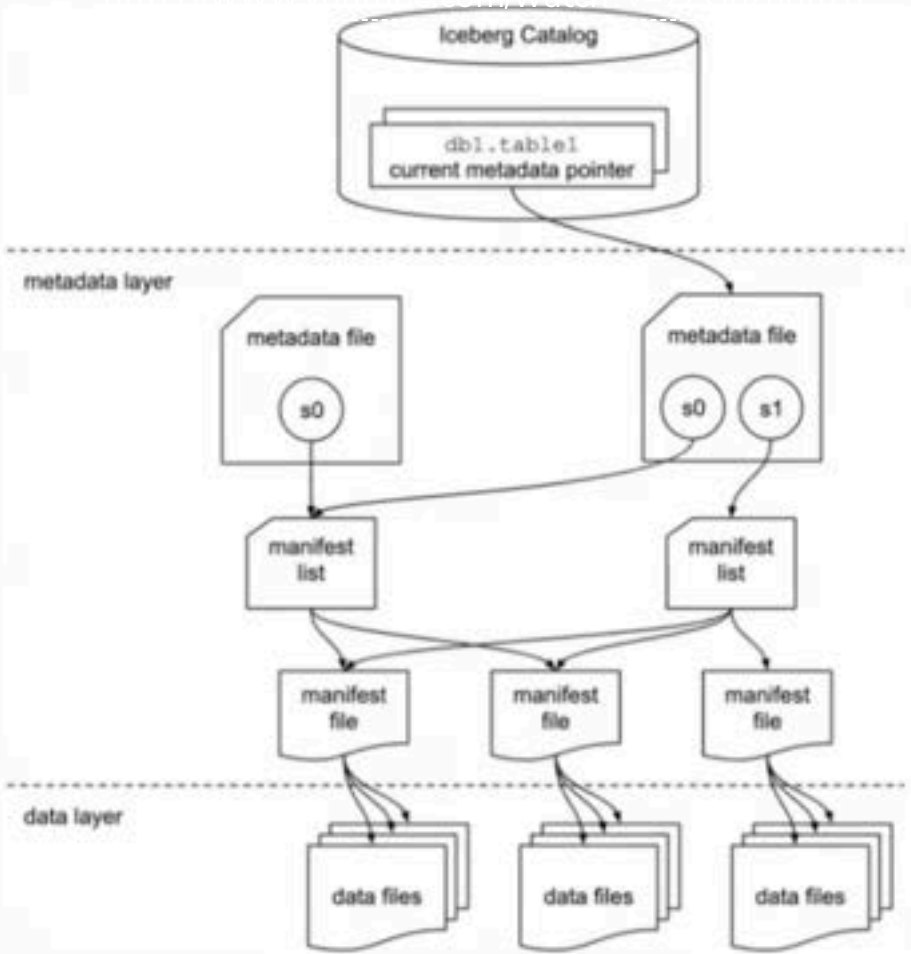
- Database and SQL first design
- Open standard
- Flexible backends
  - Catalog (postgres, sqlite, duckdb, mysql)
  - Storage (local fs, s3 compatible, ...)
- Real ACID transactions (not just for a single table)
- Reduce complexity (increase performance - inlining)
- Bridge gap to table formats
- Simple encryption of data

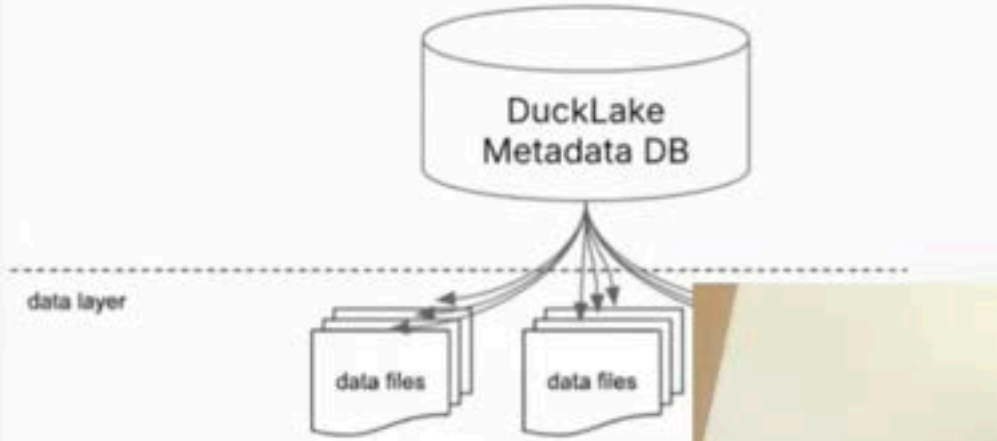


> Most cloud data warehouses internally have a similar architecture!

Replacing a whole lot of stuff with a database

https://www.youtube.com/watch?v=z2Gbznqjl20





Ducklake basics

```
duckdb
```

Ducklake basics

```
INSTALL ducklake;
```

Ducklake basics

```
ATTACH 'ducklake:metadata.ducklake' AS my_ducklake;
CREATE TABLE my_ducklake.demo
(
    i INTEGER
);
INSERT INTO my_ducklake.demo
VALUES (42),
       (43);
FROM my_ducklake.demo;

┌─────────┐
│    i    │
│  int32  │
├─────────┤
│      42 │
│      43 │
└─────────┘

```

Ducklake basics

```
DELETE
FROM my_ducklake.demo
WHERE i = 43;
FROM my_ducklake.demo;

```

i
int32
42

Ducklake basics

```
BEGIN TRANSACTION;
DELETE
FROM my_ducklake.demo;

FROM my_ducklake.demo;
┌─────────┐
│    i    │
│  int32  │
├─────────┤
│ 0 rows  │
└─────────┘

ROLLBACK;
FROM my_ducklake.demo;
┌─────────┐
│    i    │
│  int32  │
├─────────┤
│   42    │
└─────────┘

```

Ducklake basics

```
FROM ducklake_snapshots('my_ducklake');
```

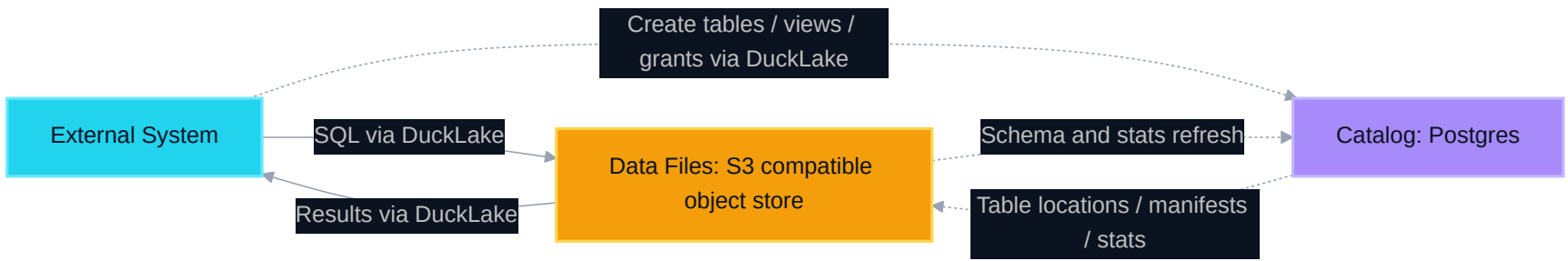
snapshot_id int64	snapshot_time timestamp with time zone	schema_version int64	changes map(varchar, varchar[])
0	2025-08-31 13:01:19.451+02	0	{schemas_created=[main]}
1	2025-08-31 13:01:24.263+02	1	{tables_created=[main.demo]}
2	2025-08-31 13:01:24.511+02	1	{tables_inserted_into=[1]}
3	2025-08-31 13:01:33.954+02	1	{tables_deleted_from=[1]}

```
FROM my_ducklake.demo AT (VERSION ⇒ 2);
```

i int32
42
43

Ducklake: Example architecture

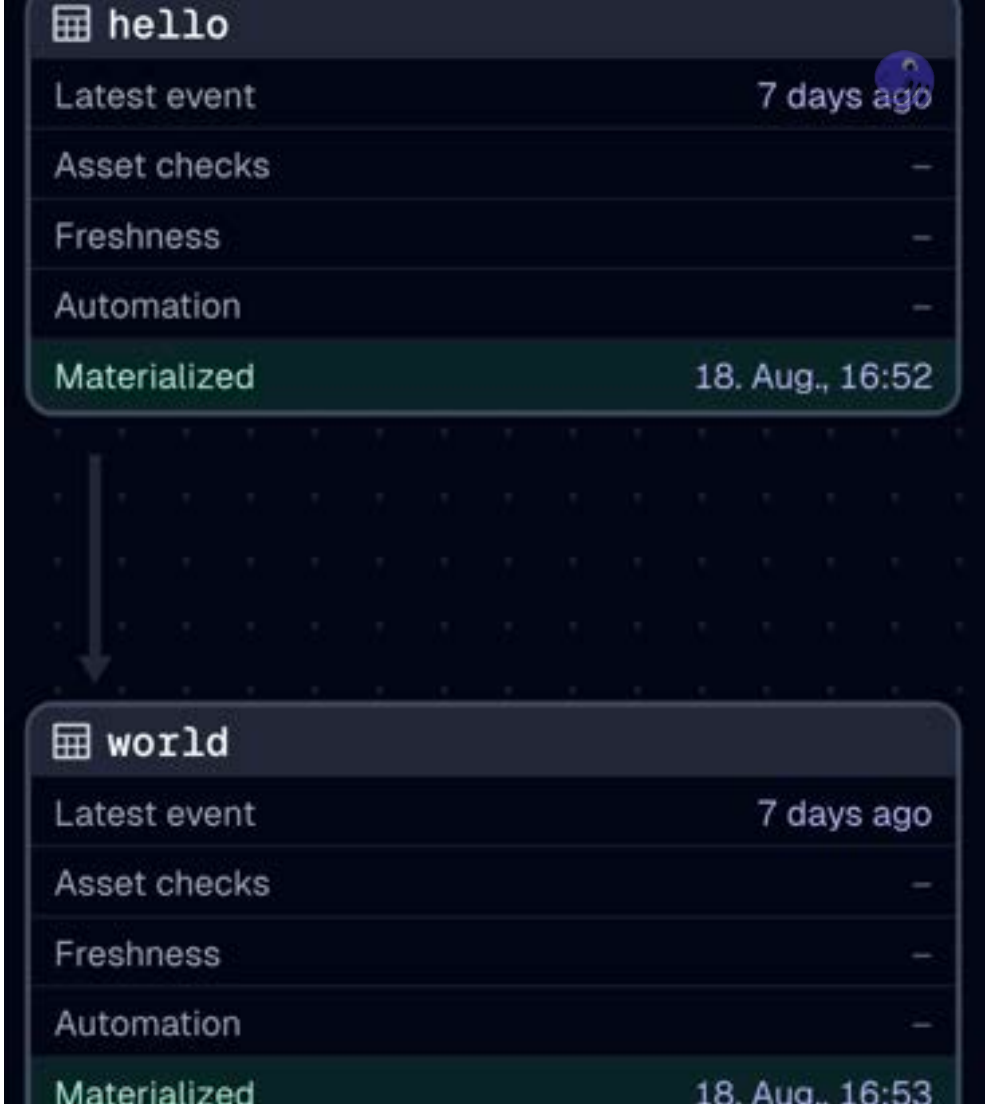
- Catalog: Postgres
- Storage: S3 compatible (MinIO) for parquet files
- All locally in docker with SSL + DNS



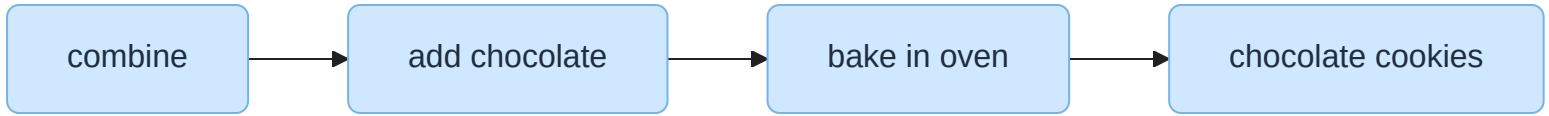
Dagster & asset graph

- Like a calculator for crunching numbers
- Graph allows computer to reason about data dependencies
- Rapid iteration: Just edit code. No need to wait for XYZ SaaS service
- Break down tool and department silos

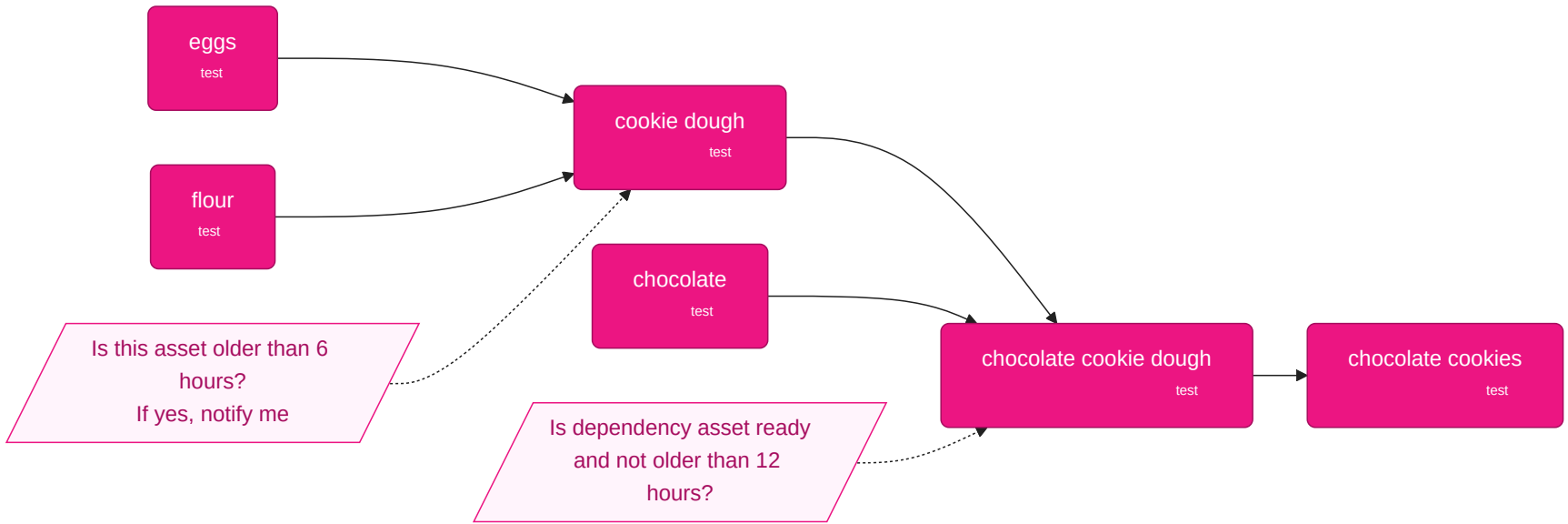
```python
import dagster as dg

@dg.asset
def hello(context: dg.AssetExecutionContext):
    context.log.info("Hello!")

@dg.asset(deps=[hello])
def world(context: dg.AssetExecutionContext):
    context.log.info("World!")
```



Task-based orchestrator



Asset-based orchestration



Process automation: Small Language Models

- No Human to chat and correct
- Cost and energy efficient
- Minimal latency
- Many small AI enabled steps with context engineering for repeatable reliable results

Ray

- Easily scaling arbitrary computation graphs
- Minimal devops support needed
- Seamless scaling: From laptop to cluster, no code rewrite.
- Elastic & cost-efficient: Scale up/down with demand, minimize idle GPUs.
- Batch and API serving support

Combined showcase

Ducklake (postgres catalog, MinIO as object storage)

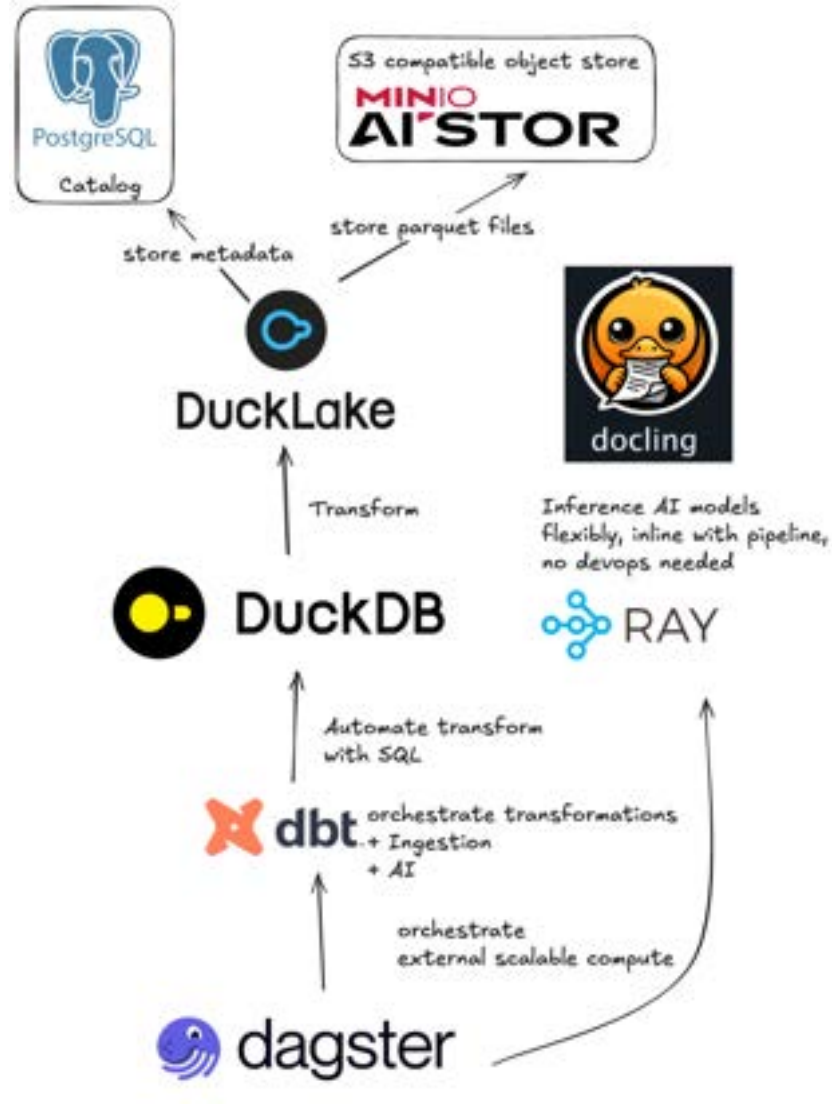
Case 1: `dlt` ducklake

- REST API
- Ingestion to ducklake
- Dagster integration

Case 2: AI - Small language models

> For ASR and document processing via docling

- V1: Plain python
- V2: Scaled with ray
- V3: Ray integration with dagster



Demo 1: dlt ducklake ingestion

```python
# OSS dagster ducklake integration to follow along
# https://github.com/dagster-io/community-integrations/tree/main/libraries/dagster-ducklake

import dagster as dg

@dg.asset
def my_ducklake_asset(ddb: DuckDBConnectionProvider):
    with ddb.duckdb_connect() as con:
        query = "select * from table"

        df = con.query(query).df()
        df.head()
```

Demo 1: dlt ducklake ingestion

```
# define resources
DuckLakeResource(
  metadata_backend=PostgresConfig(
      host="db.duckpond.data-engineering.expert",
      port=5433,
      database="ducklake_catalog",
      user=dg.EnvVar("POSTGRES_USER"),
      password=dg.EnvVar("POSTGRES_PASSWORD"),
  ),
  storage_backend=S3Config(
      endpoint_url="objectstore.duckpond.data-engineering.expert",
      bucket="duckpond-dev",
      prefix="stage",
      aws_access_key_id=dg.EnvVar("OBJECT_STORE_ROOT_USER"),
      aws_secret_access_key=dg.EnvVar("OBJECT_STORE_ROOT_PASSWORD"),
      region=dg.EnvVar("OBJECT_STORE_REGION"),
      use_ssl=True,
      url_style="path",
  ),
  alias="stage",
  plugins=["postgres", "httpfs", "ducklake"],
)
```

Demo 1: dlt ducklake ingestion

```python
# prepare sqlalchemy connection

@contextmanager
  def duckdb_connect(self) -> Generator[DuckDBPyConnection, None, None]:
      """Yields a pre-configured native DuckDB connection for safe use."""
      conn = self.get_duckdb_connection()
      # ...
      # CREATE OR REPLACE SECRET {ducklake_secret_name} (TYPE DUCKLAKE ...
      # ATTACH 'ducklake
      # ...
      try:
          yield conn
      finally:
          conn.close()
```

Demo 1: dlt ducklake ingestion

```
# use with dlt
pokemon_source = rest_api_source({
        "client": {"base_url": "https://pokeapi.co/api/v2/"},
        ...

POKE_RESOURCES = {"pokemon": "General Pokemon data, loaded to Duck Lake.",}

poke_asset_specs = [
    dg.AssetSpec(
        key=["poke_api_ducklake", name],
        description=desc,
        skippable=True,
        kinds={"duckdb", "dlt"},
    )
    for name, desc in POKE_RESOURCES.items()
]
```

Demo 1: dlt ducklake ingestion

```python
@dg.multi_asset(
    specs=poke_asset_specs,
    group_name="pokemons",
    can_subset=True,
)
def load_pokemon_to_ducklake(
    context: dg.AssetExecutionContext, ducklake: DuckLakeResource
):
    selected_resource_names = {key.path[-1] for key in context.selected_asset_keys}
    source_to_run = pokemon_source.with_resources(*selected_resource_names)
    pipeline = dlt.pipeline(
        pipeline_name="rest_api_pokemon_to_ducklake",
        dataset_name="poke_rest_api",
        destination=dlt.destinations.sqlalchemy(ducklake.get_engine()),
    )
    load_info = pipeline.run(source_to_run)
```

Demo 2: AI small language models + ray (ASR)

```python
# huggingface whisper large 3 turbo
# https://huggingface.co/openai/whisper-large-v3-turbo

class UniversalASRTranscriber:
    def __init__(self, model_id: str):
        self.model_id = model_id
        self._init_whisper()
    def _init_whisper(self):
        self.model = AutoModelForSpeechSeq2Seq.from_pretrained(
            self.model_id, dtype=self.dtype, low_cpu_mem_usage=True
        )
        self.model.to(self.device)
        self.processor = AutoProcessor.from_pretrained(self.model_id)
        self.pipe = pipeline(
            "automatic-speech-recognition",
            model=self.model,
            ...
        )
```

Demo 2: AI small language models + ray (ASR)

```
    def transcribe_one(
        self, audio_path: str,
    ):
      audio, sr = load_audio_16k_mono(audio_path, target_sr=16000)
      out = self.pipe(audio)
      text = out.get("text", str(out)) if isinstance(out, dict) else str(out)
        return {
            "path": audio_path,
            "text": text,
         }
```

Demo 2: AI small language models + ray (ASR)

```
model_id = "openai/whisper-large-v3-turbo"
transcriber = UniversalASRTranscriber(model_id=model_id)

transcriber.transcribe_one("path/to/audio/file.mp3")
```

Demo 2: AI small language models + ray (ASR)

```python
class RayASRWorker:
    def __init__(self, model_id: str):
        self.transcriber = UniversalASRTranscriber(model_id=model_id)

    def transcribe(self, audio_path: str, **kwargs) -> Dict[str, Any]:
        return self.transcriber.transcribe_one(audio_path, **kwargs)

def build_asr_mapper(*, model_id: str):
    class _ASRMapper:
        def __init__(self):
            self.worker = RayASRWorker(model_id=model_id)

        def __call__(self, batch):
            results = []
            for audio_path in batch["path"]:

                result = self.worker.transcribe(audio_path, **kwargs)
                results.append(result)

            return {"result": results}

    return _ASRMapper
```

Demo 2: AI small language models + ray (ASR)

```python
def run_asr_on_ray_cluster(
    model_id: str,
    inputs: List[str],
    num_workers: int,
    logger,
):
    ds = rd.from_items([{"path": p} for p in inputs])

    result_ds = ds.map_batches(
        build_asr_mapper(model_id=model_id),
        batch_size=4,
        num_cpus=1, num_gpus=1, resources={"MY_AI_NODE": 1},
    )

    materialized = result_ds.materialize()
```

Demo 2: AI small language models + ray (ASR)

```
ray.init(resources={"MY_AI_NODE": 1})

all_results = []
for model_id in ["openai/whisper-large-v3-turbo"]:
    results = run_asr_on_ray_cluster(
        model_id=model_id,
        inputs=["data/obama.mp3",],
    )
    all_results.extend(results)
```

Demo 2: AI small language models + ray (ASR)

```python
@dg.asset
def asr_ray_integrated(
    context: dg.AssetExecutionContext,
    ray_cluster: RayResource,
):
    inputs = ["./ai_example/data/obama.mp3"]
    model_id = "openai/whisper-large-v3-turbo"

    results = run_asr_on_ray_cluster(
        model_id=model_id,
        inputs=inputs,
    )

    total_duration = sum(r["duration_seconds"] for r in results)
    context.add_output_metadata(
        {
            "num_files_processed": len(results),
            "total_transcription_time": format_duration(total_duration),
            "model_id": model_id,
            "first_result": results[0]["text"] if results else "N/A",
        }
    )
    return results
```

Demo 3: Sovereign document intelligence

```python
from docling.datamodel.base_models import InputFormat
from docling.document_converter import DocumentConverter

source = "data/WO2021041671-eval-small.pdf"

class MyDocumentConverter:
    def __init__(self, document_format: InputFormat = InputFormat.PDF):
        self.converter = DocumentConverter()
        self.converter.initialize_pipeline(document_format)

    def convert_one(self, uri: str):
        return self.converter.convert(uri).document

converter = MyDocumentConverter()
doc = converter.convert_one(source)

print(doc.export_to_markdown())
```

Demo 3: Sovereign document intelligence

```python
class DoclingActor:
    def __init__(self):
        self.converter = MyDocumentConverter()

    def convert_one(
        self, path: str, out_dir: str,
    ) -> ConvertSummary:
        res: ConversionResult = self.converter.convert_one(src_path)

        # handle persistence from distributed actor
        handler = markdown_referenced_only_handler
        info = handler(
            src_path=src_path,
            result=res,
            out_dir=Path(out_dir),
        )
```

Demo 3: Sovereign document intelligence

```python
def build_converter(actor_kwargs: Dict[str, Any], out_dir: str):
    class _DocConverter:
        def __init__(self):
            self.worker = DoclingActor(**actor_kwargs)
            self.out_dir = out_dir

        def __call__(self, batch: Dict[str, List[str]]) -> Dict[str, List[Any]]:
            rows: List[Dict[str, Any]] = []
            for p in batch["path"]:
                s = self.worker.convert_one(
                    src_path=p,
                    out_dir=self.out_dir,
                )
                rows.append( ... result statistics ... )

            cols = {k: [r[k] for r in rows] for k in rows[0].keys()}
            return cols
    return _DocConverter
```

Demo 3: Sovereign document intelligence

```
files = [str(p) for p in Path().glob(INPUT_GLOB)]
ray.init(resources={"DOCL_NODE": 1})

ds = rd.from_items([{"path": p} for p in files])

result = ds.map_batches(
    build_converter(out_dir=OUTPUT_DIR),
    num_cpus=4,
    num_gpus=1,
    resources={"DOCL_NODE": 1},
)

status_ds = result.materialize()
```

Demo 3: Sovereign document intelligence

```python
@dg.asset
def document_ray_integrated(
    context: dg.AssetExecutionContext,
    ray_cluster: RayResource,
) -> Dict[str, Any]:
    data_dir = "/path/to/data"
    pattern = "*.pdf"

    files = [str(p) for p in data_dir.glob(pattern)]
    ds = rd.from_items([{"path": p} for p in files])

    mapper_document = build_converter(out_dir="/path/to/output")
    result = ds.map_batches(
        mapper_document,
        num_cpus=4,
        num_gpus=1,
        resources={"DOCL_NODE": 1},
    )
    status_ds = result.materialize()

    total = status_ds.count()

    return { ... statistics of resulsts, processed records, failures ... }
```

Ray resource configuration

```
from dagster_ray import LocalRay
from dagster_ray.kuberay import in_k8s, KubeRayInteractiveJob

ray_cluster = LocalRay() if not in_k8s else KubeRayInteractiveJob()

definitions = dg.Definitions(
    assets=[my_distributed_computation],
    resources={"ray_cluster": ray_cluster},
)
```

Full ray k8s JSON can be specified as needed.

Futher considerations

Do not return distributed IO to orchestrator

Consider a mutli-modal feature graph engine

Daniel & Georg are developing one

- Row based lineage tracking
- Recompute only what is needed
- Rapid experimentation through memoization

Reach out if you want to learn more (stil early WIP)

Takeaway

- Ducklake is simple, powerful and flexible
- Ecosystem maturity (integrations) needs more time
- Asset based orchestration solves challenges:
  - Inside of datavault for more speed
  - Outside/around DV for AI/ML and ingestion
  - Not stuck on a single execution engine
  - Encapsulate complexity: Easy vibe-coding or onboarding of less technical users
  - Advanced: Environment-re-targeting possible to save money

Local-first data ecosystem is possible, and has advantages

Closing summary

Take back control: Reduce entropy

Systems should scale as needed:

- Down to a single developer`s notebook
- As big as necessary, but with simplicity

You should be in control!

geoheil.com

Dependency handling

- Conda support is a must
- Conda itself is slow and dated (even with mamba)

`pixi` as the solution

- Fast
- Lockfiles
- Multi-environment handling (dev, prod)
- Pip and conda support
- Neatly packaging of environments
- Easy bootstrap
- Task runner

Security

- Sops (multiple backends supported: age, various key stores)
- Age (pgp like without the mess)

Simple: `.env` file with secrets

- Encrypt
- Push to git - or your kms store of choice

Ducklake demo

```
-- setup ducklake
INSTALL ducklake;
INSTALL postgres;
INSTALL httpfs;
LOAD httpfs;
```

Ducklake demo

```
-- define secrets
CREATE OR REPLACE SECRET duckpond_secret_catalog (
    TYPE postgres,
    HOST 'db.duckpond.data-engineering.expert',
    PORT 5433,
    DATABASE ducklake_catalog,
    USER getenv('CATALOG_DB_USER'),
    PASSWORD getenv('CATALOG_DB_PASSWORD')
);

CREATE OR REPLACE SECRET duckpond_secret_storage (
  TYPE S3,
  KEY_ID     getenv('OBJECT_STORE_ROOT_USER'),
  SECRET     getenv('OBJECT_STORE_ROOT_PASSWORD'),
  ENDPOINT   'objectstore.duckpond.data-engineering.expert',
  URL_STYLE  'path',
  REGION getenv('OBJECT_STORE_REGION'),
  USE_SSL    true,
  SCOPE      's3://my-duck-lake'
);
```

Ducklake demo

```
-- conect
CREATE OR REPLACE SECRET duckpond_ducklake (
    TYPE DUCKLAKE,
    METADATA_PATH '',
    DATA_PATH 's3://my-duck-lake/',
    METADATA_PARAMETERS MAP {'TYPE': 'postgres', 'SECRET': 'duckpond_secret_catalog'}
);
ATTACH 'ducklake:duckpond_ducklake' AS duckpond;

 USE duckpond;
```

Ducklake demo

```
CREATE TABLE nl_train_stations AS FROM 'https://blobs.duckdb.org/nl_stations.csv';

SELECT * FROM (SHOW ALL TABLES) WHERE  schema == 'main';
```

database	schema	name	column_names	column_types
varchar	varchar	varchar	varchar[]	varchar[]
duckpond	main	nl_train_stations	[id, code, uic, na…	[BIGINT, VARCHAR, BIGINT, VARCHAR, VARCHAR, VARCHAR, VARCHAR…

Ducklake demo

```
mc ls duckpond/my-duck-lake/main/nl_train_stations
# 1 file

[2025-09-29 19:53:52 CEST]  41KiB STANDARD ducklake-0199969c-0a24-7b90-a7fd-562c7a2e4d69.parquet
```

Ducklake demo

```
UPDATE nl_train_stations SET name_long='Johan Cruijff ArenA' WHERE code = 'ASB';
SELECT name_long FROM nl_train_stations WHERE code = 'ASB';

┌──────────────────┐
│    name_long     │
│     varchar      │
├──────────────────┤
│ Johan Cruijff ArenA │
└──────────────────┘
```

Ducklake demo

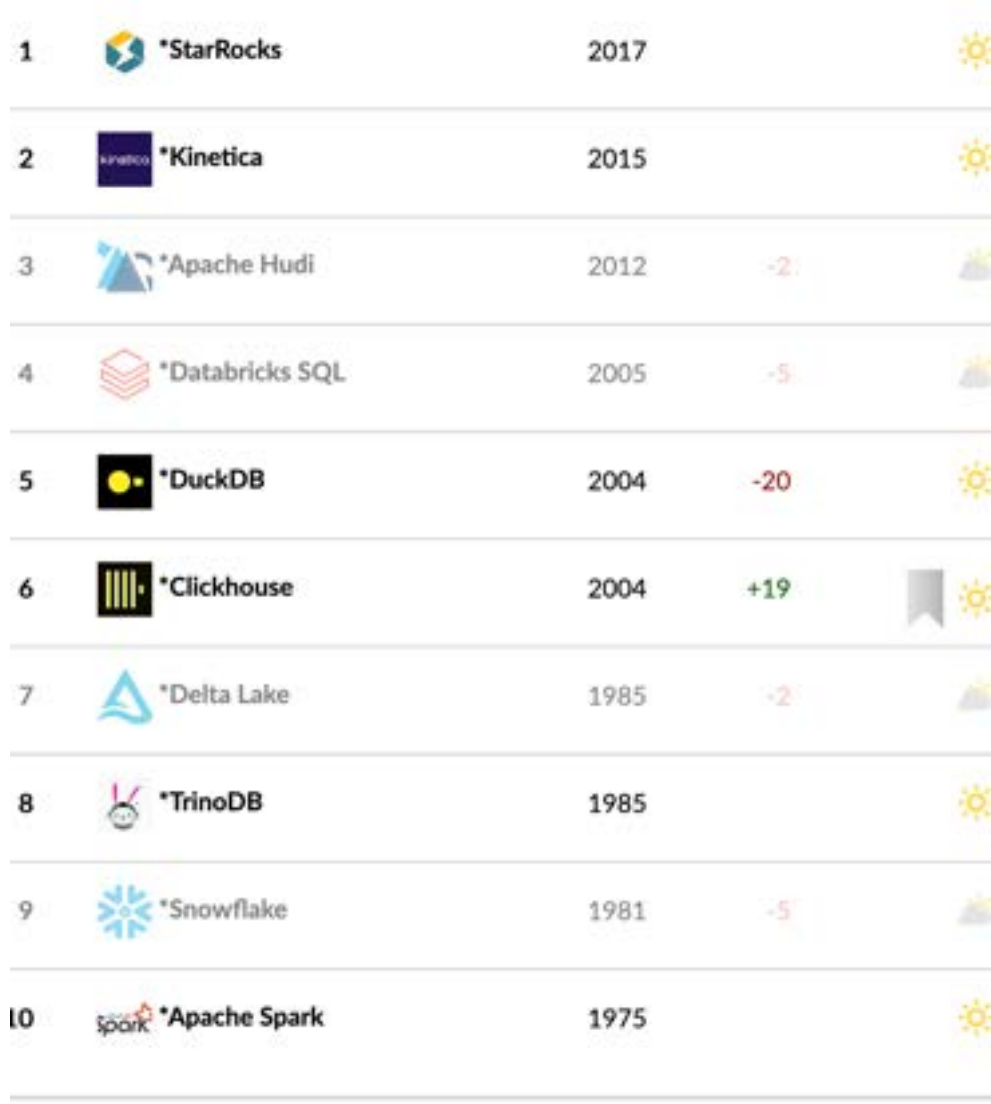
```
mc ls duckpond/my-duck-lake/main/nl_train_stations
# More files current, one delta, one deletion vector

[2025-09-29 19:53:52 CEST]  41KiB STANDARD ducklake-0199969c-0a24-7b90-a7fd-562c7a2e4d69.parquet
[2025-09-29 19:56:58 CEST] 1.8KiB STANDARD ducklake-0199969e-df9a-74ab-9f40-9542612bbc0e.parquet
[2025-09-29 19:56:58 CEST]   790B STANDARD ducklake-0199969e-dfb7-7267-b817-ff61d6edaba4-delete.parquet
```

Relative comparison

of analytical engines

- ELO score: Relative not absolute
- Robustness: Continously improve ranking, robust to cherry picking

See more at: https://georgheiler.com/post/elo-data-challenge/



Useful OSS projects

- local data stack template
- dagster-vertexai integration
- dagster-ducklake integration
- dagster-tableau integration