Special topics: DuckDB at scale

Alán F. Muñoz

2026/01/22

Outline

Introduction

Case study: The MBTA open data

Tools for large-scale data

Conclusions

Introduction

What is Duckdb

DuckDB is an open source in-process SQL engine optimised for analytics queries.

- 'In-process' means it runs within your application. You don't need to start a separate service (e.g., Postgres).
- 'Optimised for analytics queries' means that it's designed joins and aggregations involving large numbers of rows.

Key features

- Speed ("Ridiculously fast")
- No dependencies!
- SQL (+ improvements)
- Supports all important filetypes (e.g., csv, parquet, sql)
- User-Defined Functions (UDF) + community extensions
- LLM-friendly documentation

In which cases is it useful?

- Local data exploration
- Large datasets
- Query remote data without copying

It can be 100-1000 times faster than SQLite/Postgres.

The quintessential use-case

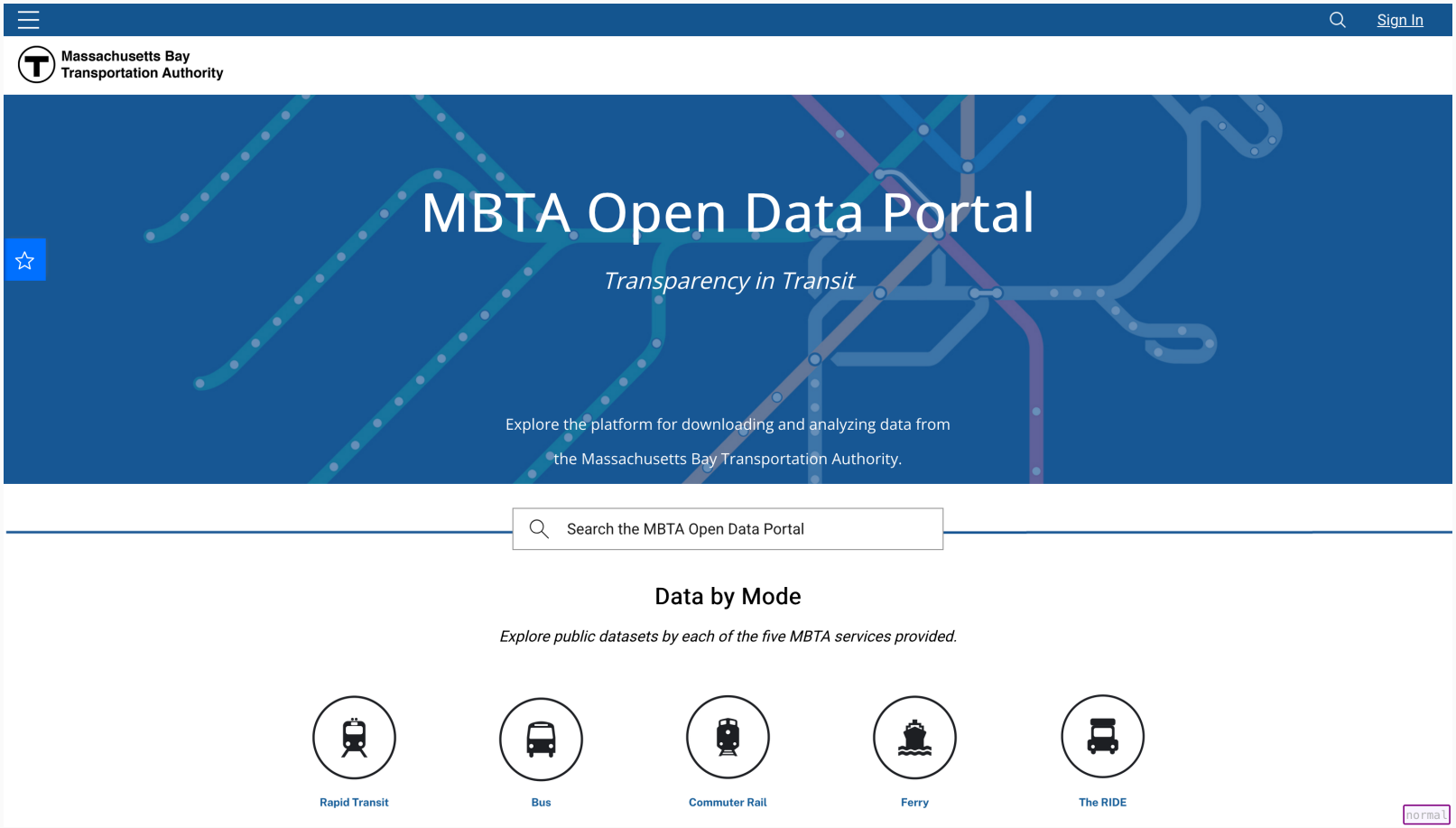
1. Read `csv`, `parquet`, `json`.
2. Clean, join, aggregate, math operations, create new columns

Common processing tools in data science

FEATURES	pandas	Polars	DuckDB	Spark
Language	Python (built on Numpy)	Rust (Python bindings)	C++ (Python bindings)	Scala/Java (Python API)
Data Structure	Row-based	Columnar-based (Apache Arrow)	column-based	Row-based but can leverage column-based
Execution Model	Eager execution (operations are computed immediately)	Eager & Lazy execution (operations can be optimised before execution)	Lazy execution	Lazy execution
Parallelism	Single-threaded	Multi-threaded	Multi-threaded	Distributed multi-threaded
Memory Usage	High (copies data often)	Low (zero-copy Arrow support, efficient memory use)	Efficient (uses columnar storage, disk spill)	Distributed across cluster, can be memory-hungry
Best for	Small to medium datasets, interactive analysis	Large-scale data processing	OLAP, analytics, embedded	Very large datasets, distributed computing

Case study: The MBTA open data

The Open Data portal for the Massachusets Bay Transport Authority



First we adjust the outputs to be slide-friendly

```
-- Remote calls
INSTALL httpfs;
LOAD httpfs;

.maxrows 10
.maxwidth 20

SET VARIABLE line_data = 'https://hub.arcgis.com/api/v3/datasets/
a2d15ddd86b34867a31cd4b8e0a83932_0/
downloads/data?format=csv&spatialRefId=4326&where=1%3D1'
```

Overview of one of the tables

There are multiple available (tabular) datasets:

- Ridership by Trip, Route line and stop
- Montly ridership by month
- Gated station entries
- Passenger surveys

Let's explore the ridership table

```
CREATE OR REPLACE TABLE monthly_ridership AS (
FROM read_csv(getvariable('line_data')));
SELECT #1,#2 FROM (DESCRIBE monthly_ridership);
```

column_name	column_type
service_date	TIMESTAMP WITH TIME ZONE
mode	VARCHAR
route_or_line	VARCHAR
total_monthly_weekday_ridership	BIGINT
average_monthly_weekday_ridersh	BIGINT
countofdates_weekday	BIGINT
total_monthly_ridership	DOUBLE
average_monthly_ridership	BIGINT
countofdates	BIGINT
ObjectId	BIGINT

List the unique route or lines

```
FROM monthly_ridership
SELECT DISTINCT route_or_line
ORDER BY route_or_line
```

route_or_line varchar
Blue Line
Boat-F1
Boat-F3
Boat-F4
Bus
Commuter Rail
Green Line
Orange Line
Red Line
Silver Line
The RIDE
11 rows

Which route or line is the most frequented?

Which route or line is the most frequented?

Buses (as a whole) move the most people amongst underground trains the Red Line is the winner

```
SELECT route_or_line, CAST(MEAN(total_monthly_weekday_ridership) AS INTEGER)
AS mean_weekly_ridership FROM monthly_ridership GROUP BY route_or_line
ORDER BY mean_weekly_ridership DESC;
```

route_or_line varchar	mean_weekly_ridership int32
Bus	7395469
Red Line	5326706
Orange Line	4410355
Green Line	3893135
Commuter Rail	2684943
Blue Line	1392658
Silver Line	724622
The RIDE	138221
Boat-F1	66248
Boat-F3	23922
Boat-F4	19975
11 rows	2 columns

Let us look at a table with per-station entries

This other dataset contains per-station information

```
SET VARIABLE gates_data = 'https://hub.arcgis.com
/api/v3/datasets /001c177f07594e7c99f193dde32284c9_0
/downloads/data?format=csv&spatialRefId=4326&where=1%3D1';
```

Information of specific entrances

```
CREATE OR REPLACE TABLE gated_entries AS (
FROM read_csv(getvariable('gates_data')));
SELECT #1,#2 FROM (DESCRIBE gated_entries);
```

column_name varchar	column_type varchar
service_date	TIMESTAMP WITH TIME ZONE
time_period	VARCHAR
stop_id	VARCHAR
station_name	VARCHAR
route_or_line	VARCHAR
gated_entries	DOUBLE
ObjectId	BIGINT

Which station has the most recorded gate entries?

Which station has the most recorded gate entries?

- Q1: Why Harvard and not another downtown station?
- Q2: Why Magoun/East Somerville if the green line is relatively popular?

```
SELECT station_name,route_or_line,CAST(SUM(gated_entries) AS INT)
AS gated_entries FROM gated_entries
GROUP BY station_name, route_or_line
ORDER BY gated_entries DESC
```

station_name varchar	route_or_line varchar	gated_entries int32
Harvard	Red Line	4802562
Back Bay	Orange Line	4253244
Copley	Green Line	3768772
North Station	Orange Line	3686858
Central	Red Line	3640045
.	.	.
.	.	.
.	.	.
Suffolk Downs	Blue Line	234871
Union Square	Green Line	181843
Ball Square	Green Line	169984
Magoun Square	Green Line	148869
East Somerville	Green Line	89718
78 rows (10 shown)		3 columns

Duckdb makes the data wrangling fast and straightforward

In a few lines you can answer very explicit questions about datasets hosted anywhere (both public and private.)

In general, all this data wrangling can be summarised in a small set of steps.

The skeleton of any analysis: Group, change, save.

Most data processing can be reduced to:

1. Group relevant categories
    - Treatment vs controls
    - Treatments vs each other
2. Math/Logical operation on groups
    - Basic operations (e.g., $+ - */$)
    - Vector operations (e.g., cosine similarity)
    - Logical operations (e.g., if-else)
    - A combination of the above
3. Save into a new dataset for further processing/viz

Tools for large-scale data

What do we consider large-scale?

- Data larger than an off-the-shelf server
- Processing takes time requires memory/storage considerations
- Traditionally we would use cloud compute

Problem: Many different groups and subgroups

Most of our data is highly structured:

- By Metadata: Source -> Batch -> Plate -> Well -> Site
- By annotation: Perturbation id, Control vs perturbation

Abstraction: Tables vs Trees

Relational databases are a battle-proven technology, but are not always optimal for heavily structured data.

```
SET VARIABLE plates = 'https://github.com/jump-cellpainting/datasets/
raw/refs/heads/duckdb/metadata/plate.csv.gz';
FROM getvariable(plates) LIMIT 3;
```

Metadata_Source varchar	Metadata_Batch varchar	Metadata_Plate varchar	Metadata_PlateType varchar
source_1	Batch1_20221004	UL000109	COMPOUND_EMPTY
source_1	Batch1_20221004	UL001641	COMPOUND
source_1	Batch1_20221004	UL001643	COMPOUND

We can express it as a nested tree

The simplest way to transform table to nested I could come up with.

```
COPY (SELECT json_object(
    Metadata_Source, json_group_array(sub_data)
) AS root
FROM (
    SELECT Metadata_Source, Metadata_Batch,
    json_object(Metadata_Batch,
    json_group_array(json_object(Metadata_Plate, Metadata_PlateType))
    ) AS sub_data
    FROM getvariables(plates)
    GROUP BY Metadata_Source, Metadata_Batch, Metadata_Plate
)
GROUP BY Metadata_Source) TO 'output.json';
```

The nested tree makes one type of (key-based) access more efficient

At the cost basically all others.

```
jq '.root | walk(if type == "array" then add else . end) |
with_entries(select(.key == "source_1")) | select(. != {})'
output.json | head -n 11
```

```
{
  "source_1": {
    "Batch1_20221004": {
      "UL000109": "COMPOUND_EMPTY"
    },
    "Batch2_20221006": {
      "UL001675": "COMPOUND"
    },
    "Batch6_20221102": {
      "UL000599": "COMPOUND"
    },
```

Using multiple tables and schemas can help

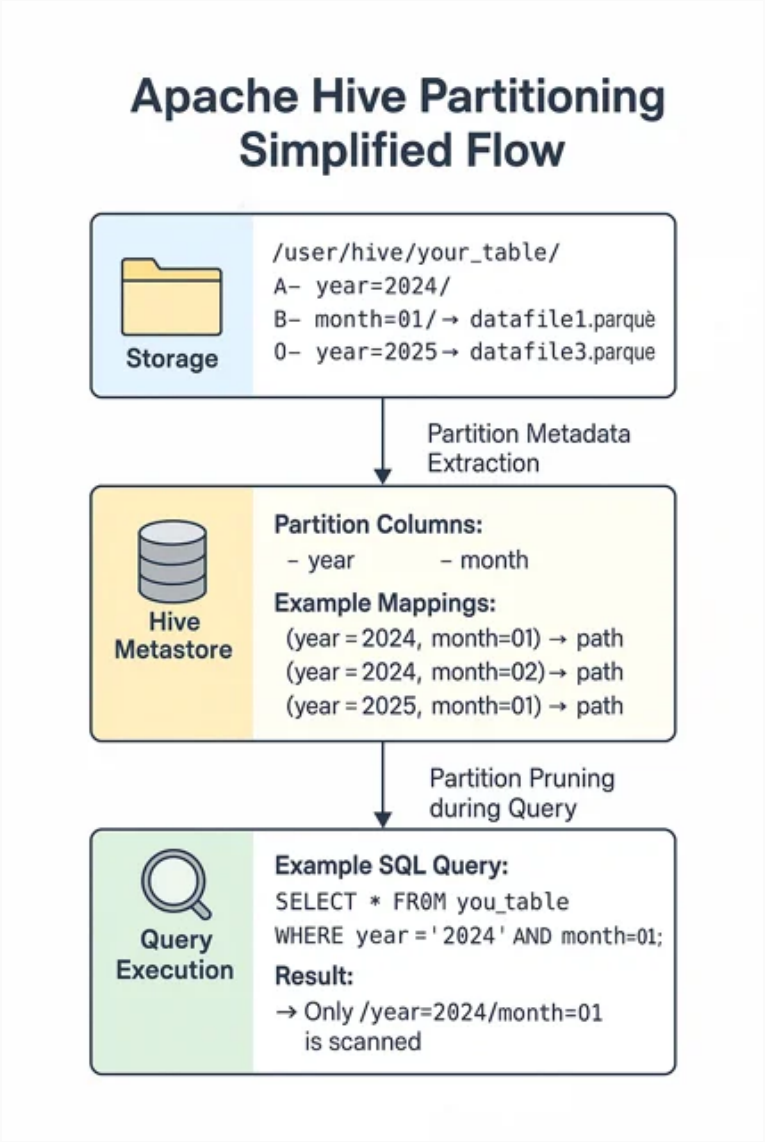
But it is untracktable when the database size exceeds 10s of GBs (compressed) and must be loaded into memory all at once for queries.

Problem: Massive hierarchical datasets

There is no clear correct solution: One big database vs a myriad of files.

Potential solution: Parquet hive partitioning

These prioritise keys that optimise the most common queries.



Hive partitioning structure

In my GSK project I produce profiles of single-object (e.g., cell, nuclei) time series.

```
DATA_DIR="/datastore/alan/gsk/aliby_output/ELN201687/
H00DJKJread1BF48hrs_20230926_095825hive_profiles_tracked"

 tree $DATA_DIR | sed -n "2,10p"
```

```
├── site=A01_001
│   ├── object=cell
│   │   └── data_0.parquet
│   └── object=nuclei
│       └── data_0.parquet
├── site=A01_002
│   ├── object=cell
│   │   └── data_0.parquet
│   └── object=nuclei
```

The profiles are big enough to become problematic

The experiment contains 384 wells · 5 sites · 2 objects · 20 time points ≈ 77k files with single-cell profiles.

```
du -sh ${DATA_DIR} | cut -f 1
```

```
20G
```

We treat key columns like any other column

The full table shows a "tidy"-like structure for morphological features. In reality

```
SET VARIABLE hive = '/datastore/alan/gsk/aliby_output/ ELN201687/
H00DJKJread1BF48hrs_20230926_095825hive_profiles_tracked/
*/*/*.parquet';

SELECT #1,#2 FROM (DESCRIBE FROM
read_parquet(getvariable('hive'), hive_partitioning = true));
```

Hive partitioning

Duckdb will only load the file(s) and column(s) relevant to our query, saving cycles, time memory.

```
SET VARIABLE hive = '/datastore/alan/gsk/aliby_output/ELN201687/
H00DJKJread1BF48hrs_20230926_095825hive_profiles_tracked/*/*/*.parquet';
SELECT branch,site,metric,value FROM (
FROM read_parquet(getvariable('hive'), hive_partitioning = true)
WHERE site LIKE 'B0%' AND object='cell' ORDER BY random())
USING SAMPLE 3;
```

branch varchar	site varchar	metric varchar	value double
1/max/texture	B04_001	SumAverage_3_02_256	119.70689655172414
1/max/radial_zerni…	B01_002	RadialDistribution_…	0.43083862789046273
0/max/radial_zerni…	B04_003	RadialDistribution_…	0.6837160487262536

Memory profiling experiments: Data selection and shuffling

We also do it fromm bash

Site B01_001 (1 file, directly read): 56MB of peak usage.

```
HIVE="${DATA_DIR}/\*\*/\*.parquet"
command time --format='%e seconds; %M max memory (KB)' duckdb -c "
SELECT branch,site,metric,value FROM (
FROM read_parquet('${DATA_DIR}/site=B01_001/object=cell/data_0.parquet')
WHERE site = 'B01_001' AND object = 'cell'
ORDER BY random()) USING SAMPLE 3;"
```

branch varchar	site varchar	metric varchar	value double
1/max/zernike	B01_001	Zernike_6_4	0.0038834561835829843
0/max/sizeshape	B01_001	Center_X	693.9240674955595
1/max/sizeshape	B01_001	FormFactor	0.7671706171485779

0.05 seconds; 56956 max memory (KB)

Memory profiling experiments: Data selection and shuffling

Site B01_001 (1 file): 720MB peak usage

```
command time --format='%e seconds; %M max memory (KB)' duckdb -c "
SELECT branch,site,metric,value FROM (
FROM read_parquet('${HIVE}', hive_partitioning = true)
WHERE site = 'B01_001' AND object='cell'
ORDER BY random()) USING SAMPLE 3;"
```

branch varchar	site varchar	metric varchar	value double
1/max/zernike	B01_001	Zernike_5_3	0.018311454640108134
1/max/sizeshape	B01_001	CentralMoment_2_2	287848640.24825424
0/max/texture	B01_001	InfoMeas1_3_00_256	-0.008348726106163183

0.44 seconds; 723624 max memory (KB)

Memory profiling experiments: Data selection and shuffling

Site B01_00% (5 files): ~1.7 GB peak memory usage.

```
command time --format='%e seconds; %M max memory (KB)' duckdb -c "
SELECT branch,site,metric,value FROM (
FROM read_parquet('${HIVE}', hive_partitioning = true)
WHERE site LIKE 'B01_00%' AND object='cell'
ORDER BY random()) USING SAMPLE 3;"
```

branch varchar	site varchar	metric varchar	value double
1/max/sizeshape	B01_001	SpatialMoment_2_0	4084235.0
0/max/sizeshape	B01_003	NormalizedMoment_2_3	0.2893527115553125
0/max/sizeshape	B01_005	InertiaTensor_1_1	38.509375

0.55 seconds; 1711220 max memory (KB)

Memory profiling experiments: Data selection and shuffling

Site B0% (45 files): ~14GB peak memory

```
command time --format='%e seconds; %M max memory (KB)' duckdb -c "
SELECT branch,site,metric,value FROM (
FROM read_parquet('${HIVE}', hive_partitioning = true)
WHERE site LIKE 'B0%' AND object='cell'
ORDER BY random()) USING SAMPLE 3;"
```

branch varchar	site varchar	metric varchar	value double
0/max/zernike	B01_001	Zernike_7_5	0.0024592469381906 99
1/max/sizeshape	B07_003	SpatialMoment_2_0	23596393.0
0/max/sizeshape	B01_001	BoundingBoxMaximum_Y	642.0

1.70 seconds; 14132836 max memory (KB)

Disadvantages of Hive partitioning

Advantages

- Faster, enables concurrent write
  Partitions can be written without
- Lower memory footprint
  Only relevant partitions are read into memory
- Faster queries
  Optimisation occurs at the IO level

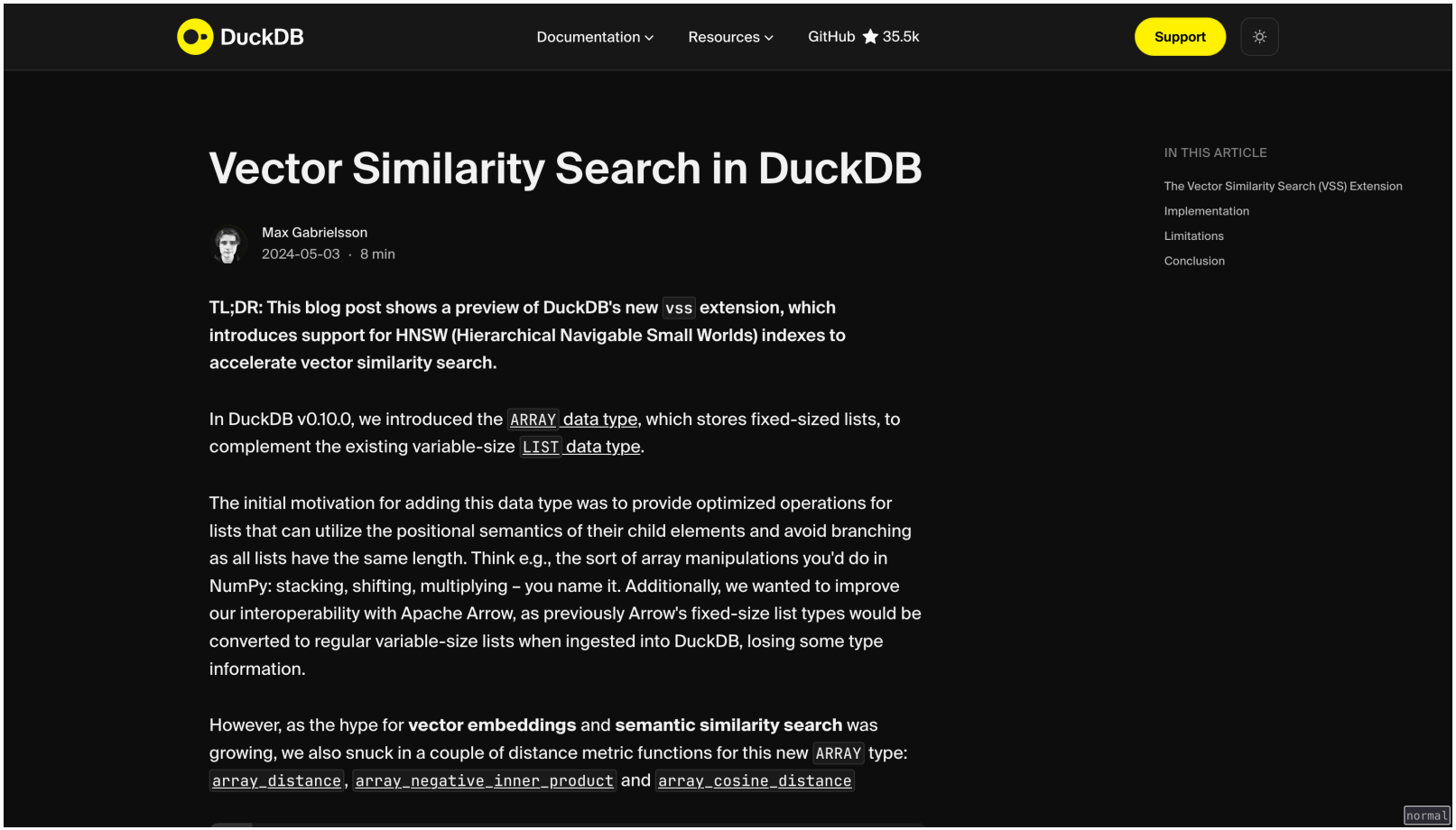
Disadvantages

- Many small files may impact the performance of very small and very large queries.
- Compression penalty: Compression occurs at the most granular level.

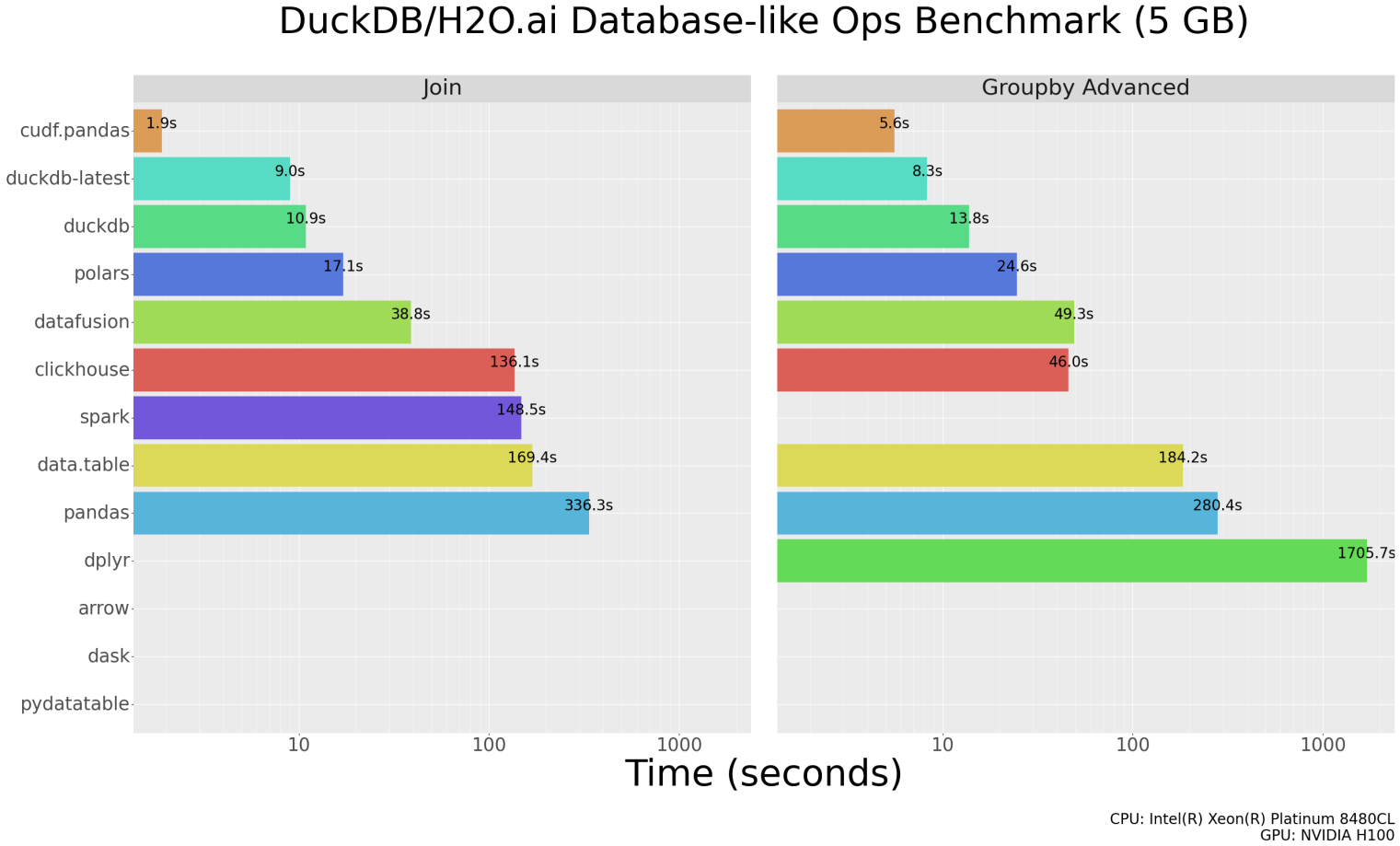
Problem: But my vectors! :(

SQL was invented before vector operations became the pillar of civilization.

Solution: Vector Similarity extension



Caveat: GPU processing will usually be faster for round-robin comparisons



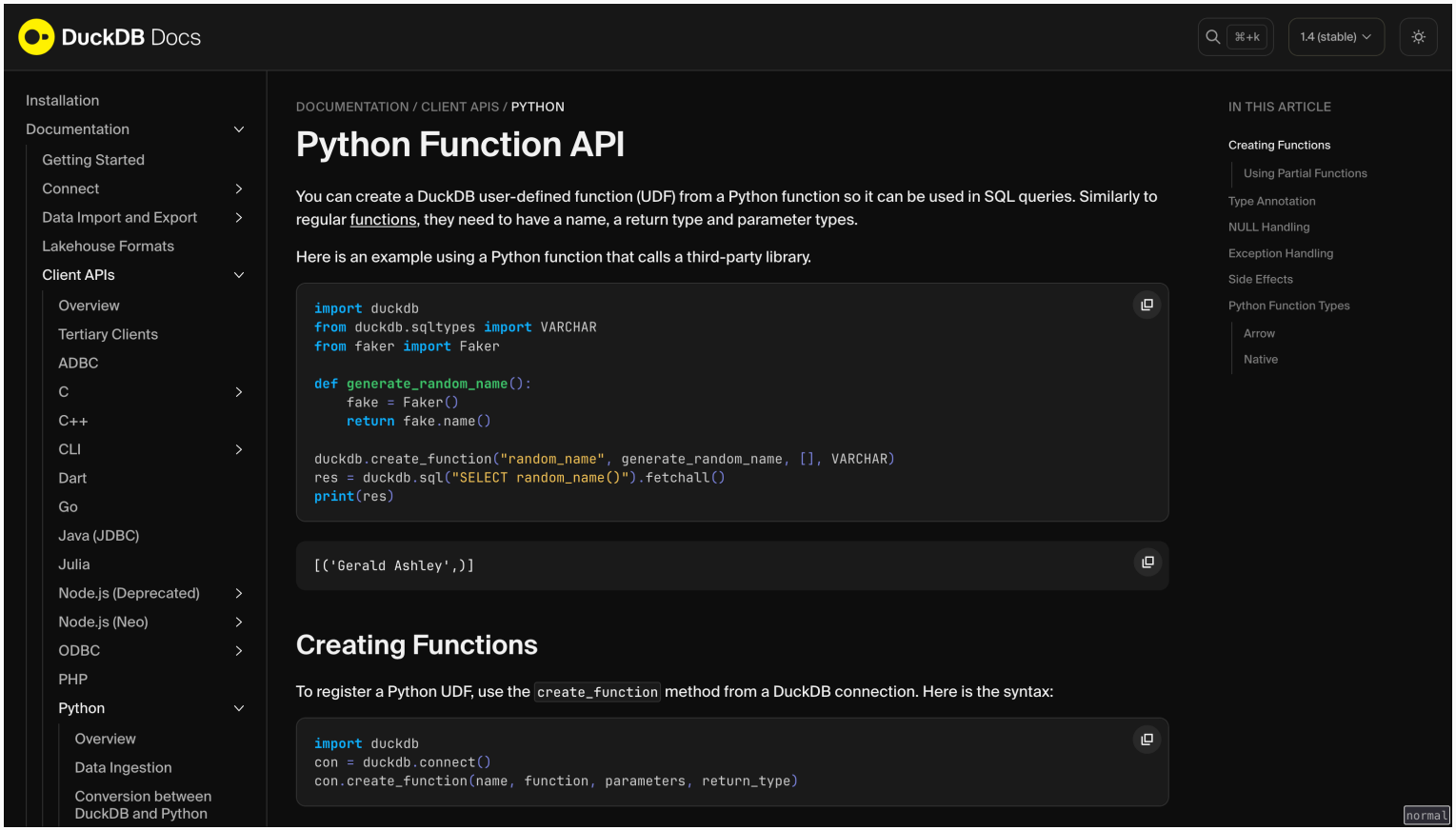
Problem: I need a function to process data, but it's not available in duckdb

Normally this would entail:

1. Loading data into PythonLand and doing slow processing
2. Moving it back to duckdb and resuming data wrangling

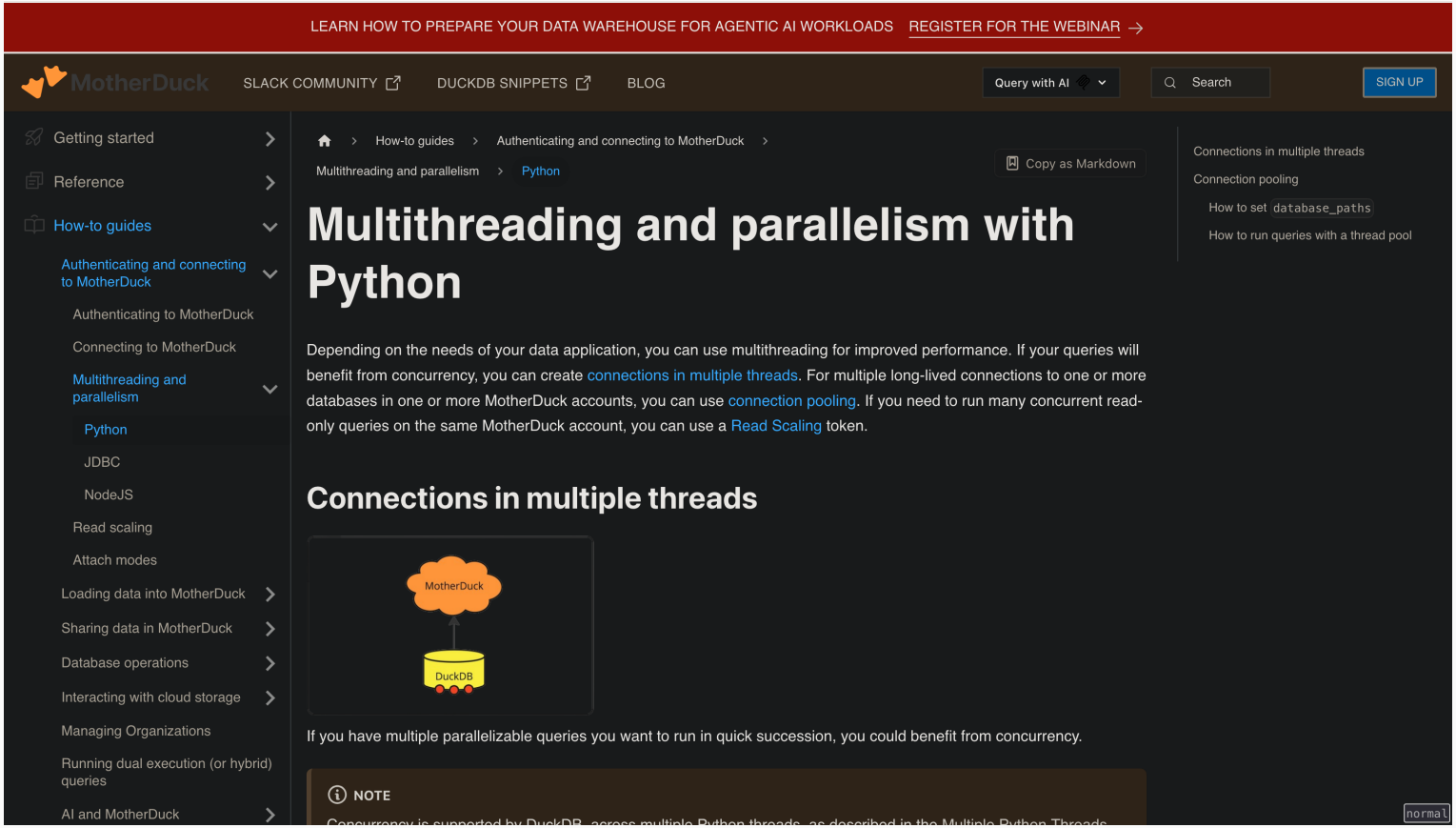
Solution 1: User-Defined functions

source



Solution 2: Duckdb-friendly multithreading

source



Some previous use-cases at the CS-Lab

- Dealing with tens/hundreds of GB of data processing for GSK project
- `copairs` 10-100x speedup
- On-browser cosine similarity between chemical embeddings
- *Claudetanu* uses it extensively to structure JUMP metadata

Conclusions

Duckdb solves multiple problems

- In-memory OR in disk
- Larger-than memory data
- Multithreading/Multiprocessing
- Optimised queries
- Web requests
- Vector operations

Rules of the thumb for DuckDB vs GPU

Prefer matrix operations (ideally over GPUs) when

- You need pairwise mathematical operations AND
- You are already in a Python GPU-enabled environment

Otherwise, duckdb is likely the cost and time-efficient approach.

The ecosystem is still growing

- DuckLake (format): Single repository of data stored in its natural format
- MotherDuck (startup): Duckdb-based data warehouse in SQL or natural language

I'm excited (or not)! How can I try it out?

From within Python: PyPI's duckdb package.

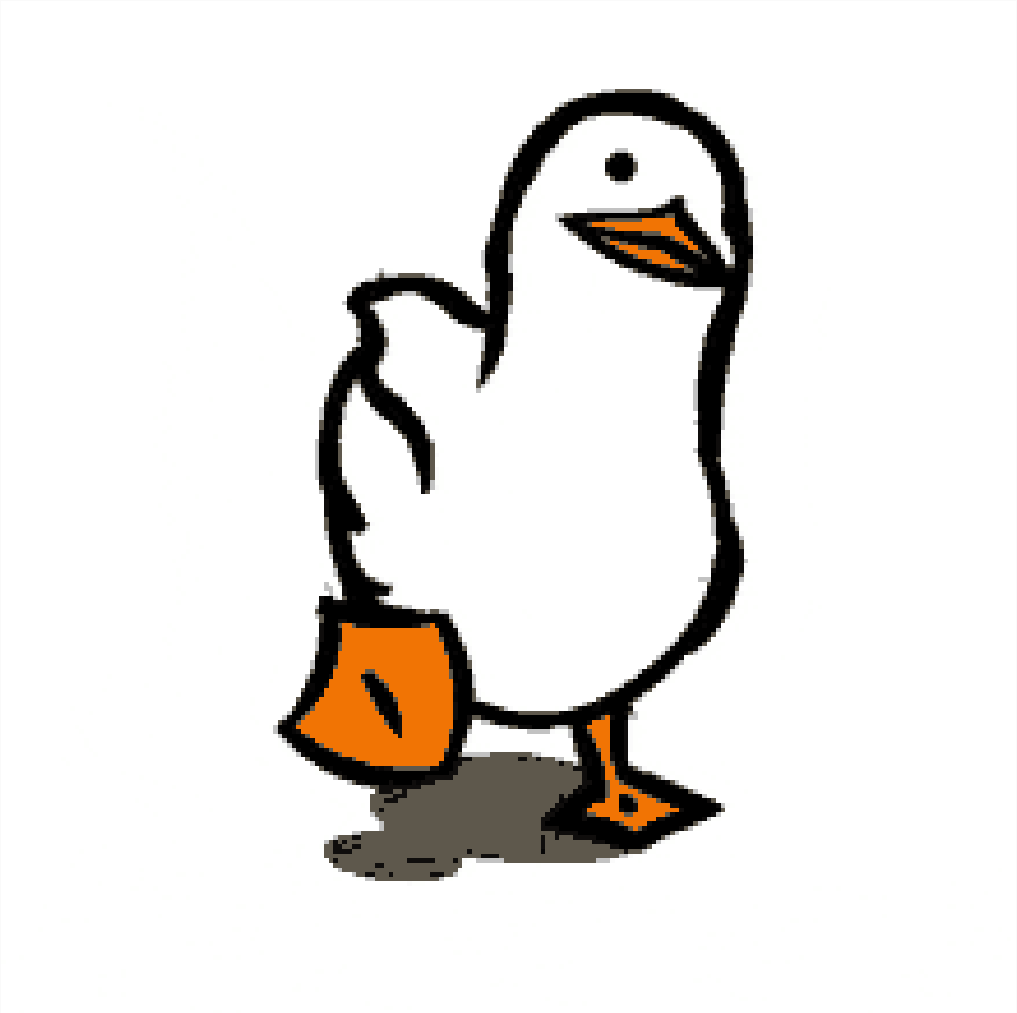
In my computers/servers:

- Already on CS-Lab Nix servers
- MacOS: `brew` or Download from `duckdb.org`
- Windows: `duckdb.org`

Notebook-like user interfaces

- The CLI comes with one: `duckdb --ui`
- `Marimo` supports duckdb out of the box!
- Jupyter notebooks running duckdb

Thanks for your attention



Resources/Blogs

- General
  - Why Duckdb is my first choice for data processing
  - DuckDB is Probably the Most Important Geospatial Software of the Last Decade
  - Practical SQL for Data Analysis
  - Vector Similarity Search in DuckDB
  - The Rise of SQL
  - Against SQL
- Benchmarks
  - cudf-pandas
  - pandas vs polars vs duckdb