In Process Analytical Data Management with DuckDB

Alex Monahan / alex@duckdblabs.com / alex@motherduck.com

Howdy, I’m Alex Monahan!

- Industrial and Systems Engineer from Virginia Tech
- 9 years at Intel
    - Industrial Engineer -> Tech. Analyst -> Data Scientist
    - SQL -> & Python -> & JavaScript
- 2020: Found DuckDB
    - Integrated it into self-service analytics platform @Intel
    - Used it in multiple data projects
    - Became a big DuckDB fan on Twitter!
- 2021: Part time Docs and Blogs for DuckDB Foundation
- 1 Month ago: Joined MotherDuck!
    - DuckDB both in the cloud and on your laptop

Outline

- Motivation
  - Data Science Workflow
- DuckDB
  - What is DuckDB?
  - Alternatives
  - Performance
  - Design
  - Integrations
- Demo
- Summary

What I want you to remember about DuckDB!

1. It’s Fast

2. It’s Easy

3. Works great with Python

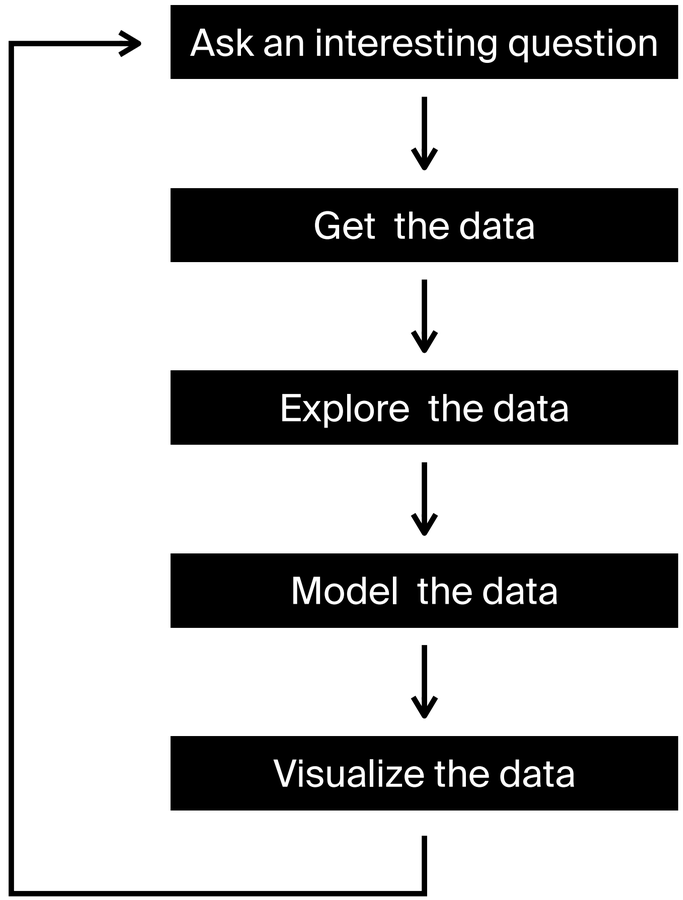
4. Handles larger than RAM data

Motivation

Requirements for a Data Science Workflow

- Data processing is an important piece
- Must:
  - Scan different file formats.
    - CSV, JSON, Parquet, Postgres, SQLite
  - Integrate with Ecosystem Tools.
    - Plot Libraries, ML Libraries.
  - Be efficient analytical execution engines
    - Beyond Memory Execution
    - Complex Query Optimization
  - Support SQL and Relational API.

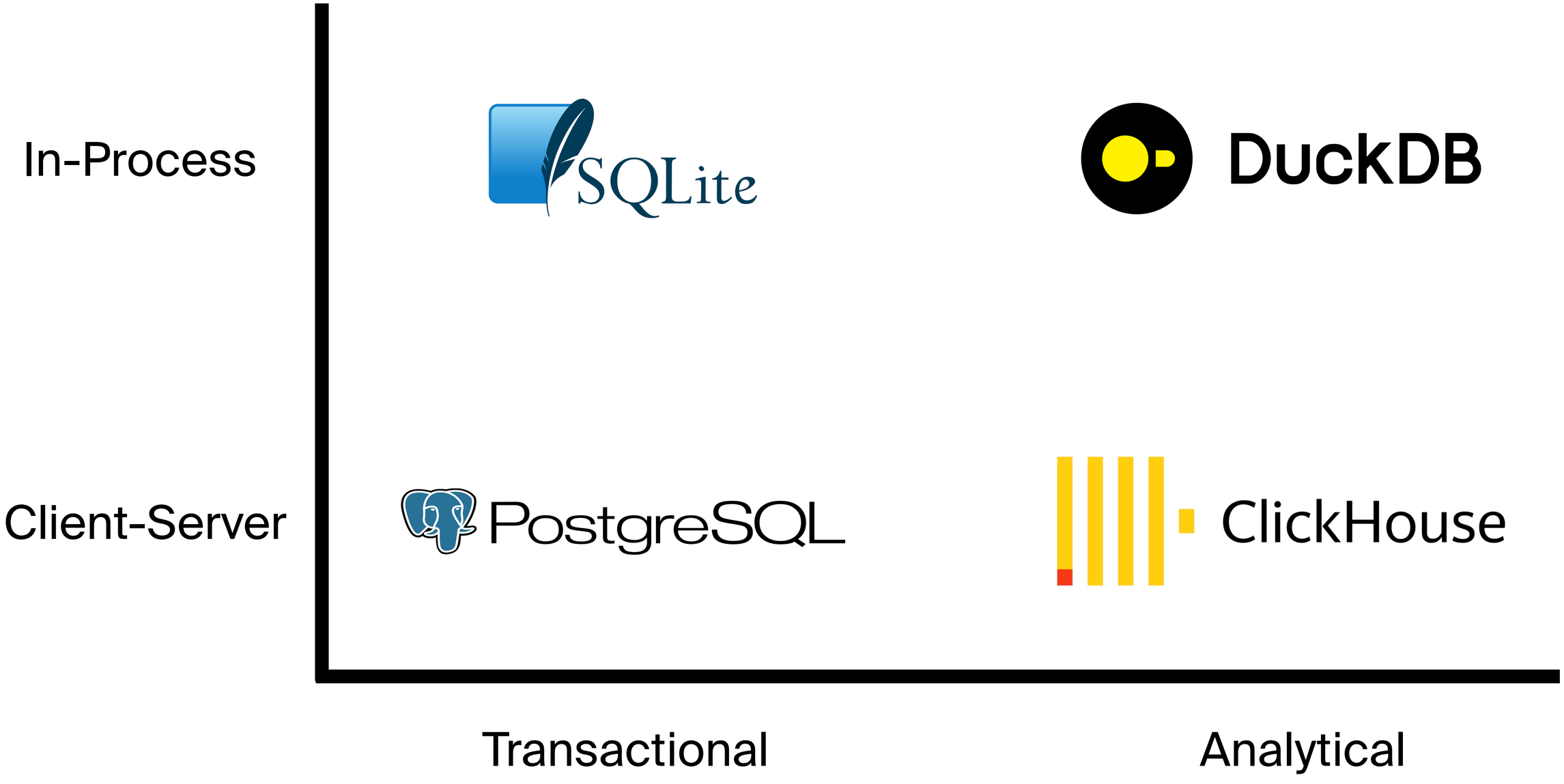
Blitzstein & Pfister's workflow



DuckDB

DuckDB is a new category of database

DuckDB is a new category of database



What is DuckDB?

- DuckDB
- In-Process OLAP DBMS
  - “The SQLite for Analytics”

- 1.3 million PyPI downloads / month
- Free and Open Source (MIT)
- duckdb.org

What is DuckDB? The SQLite for Analytics

- Simple installation `$ pip install duckdb`
    - Pre-compiled and cross platform

- Embedded: no server management
- Fast analytical processing
- Fast transfer between R/Python and RDBMS
- Rich SQL Dialect
- Single File Format

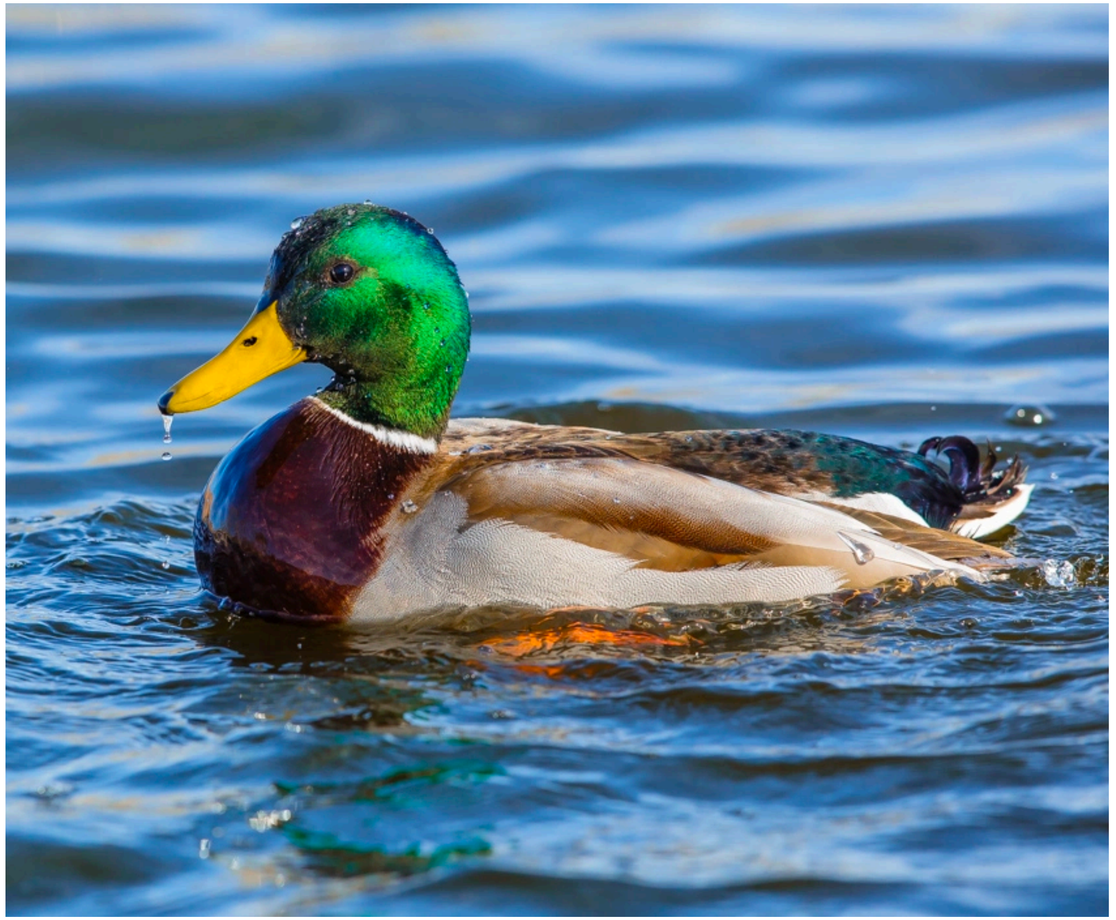
- DuckDB is currently in pre-release (V0.8.1)
    - Check duckdb.org for more details.

DuckDB combines the best of databases and dataframes

	Dataframes	Client-Server Databases	SQLite	DuckDB
Fast analytical queries	Yes	Yes	No	Yes
Fast data transfer	Yes	No	No	Yes
Easy to use	Yes	No	Yes	Yes
Data Science Integrations	Yes	No	No	Yes
Query Optimization	Some	Yes	Yes	Yes
Built-in Storage	No	Yes	Yes	Yes
Larger than RAM Execution	No	Yes	Yes	Yes
Relational API	Yes	No	No	Yes
SQL Support	No	Yes	Yes	Yes

Why is it called DuckDB?

- Ducks are versatile. They can:
  - Fly
  - Walk
  - Swim
- It sits on top of the data lake



Why is it called DuckDB?

- Ducks are versatile. They can:
  - Fly
  - Walk
  - Swim

- It sits on top of the data lake

Performance

DuckDB is #1 in H2O.ai Analytics Benchmarks

50 GB Group Bys

basic questions

Input table: 1,000,000,000 rows x 9 columns (50 GB)

duckdb-latest*	0.8.0	2023-04-13	76s
Polars	0.16.18	2023-04-05	127s
DuckDB*	0.7.1	2023-04-05	143s
ClickHouse	22.12.1.1752	2023-03-24	189s
data.table	1.14.9	2023-03-24	191s
DataFrames.jl	1.5.0	2023-04-21	286s
spark	3.3.2	2023-03-24	389s
Arrow	11.0.0.3	2023-04-12	624s
(py)datatable	1.1.0a0	2023-03-24	870s
dask	2023.3.2	2023-04-07	3990s
pandas	2.0.0	2023-04-12	out of memory
Modin		see README	pending

advanced questions

Input table: 1,000,000,000 rows x 9 columns (50 GB)

duckdb-latest*	0.8.0	2023-04-13	563s
DuckDB*	0.7.1	2023-04-05	652s
Polars	0.16.18	2023-04-05	705s
data.table	1.14.9	2023-03-24	1323s
ClickHouse	22.12.1.1752	2023-03-24	2256s
(py)datatable	1.1.0a0	2023-03-24	2843s
pandas	2.0.0	2023-04-12	out of memory
spark	3.3.2	2023-03-24	not yet implemented
dask	2023.3.2	2023-04-07	out of memory
Arrow	11.0.0.3	2023-04-12	internal error
DataFrames.jl	1.5.0	2023-04-21	out of memory
Modin		see README	pending

First time
Second time

5 GB Joins

basic questions

Input table: 100,000,000 rows x 7 columns (5 GB)

duckdb-latest	0.8.0	2023-04-12	22s
DuckDB	0.7.1	2023-04-05	23s
Polars	0.16.18	2023-04-05	42s
data.table	1.14.9	2023-03-24	112s
DataFrames.jl	1.5.0	2023-04-21	123s
ClickHouse	22.12.1.1752	2023-03-24	187s
spark	3.3.2	2023-03-24	469s
dplyr	1.0.10	2022-12-30	498s
pandas	2.0.0	2023-04-12	666s
(py)datatable	1.1.0a0	2023-03-24	10903s
dask	2023.3.2	2023-04-07	internal error
Arrow	11.0.0.3	2023-04-12	out of memory
Modin		see README	pending

First time
Second time

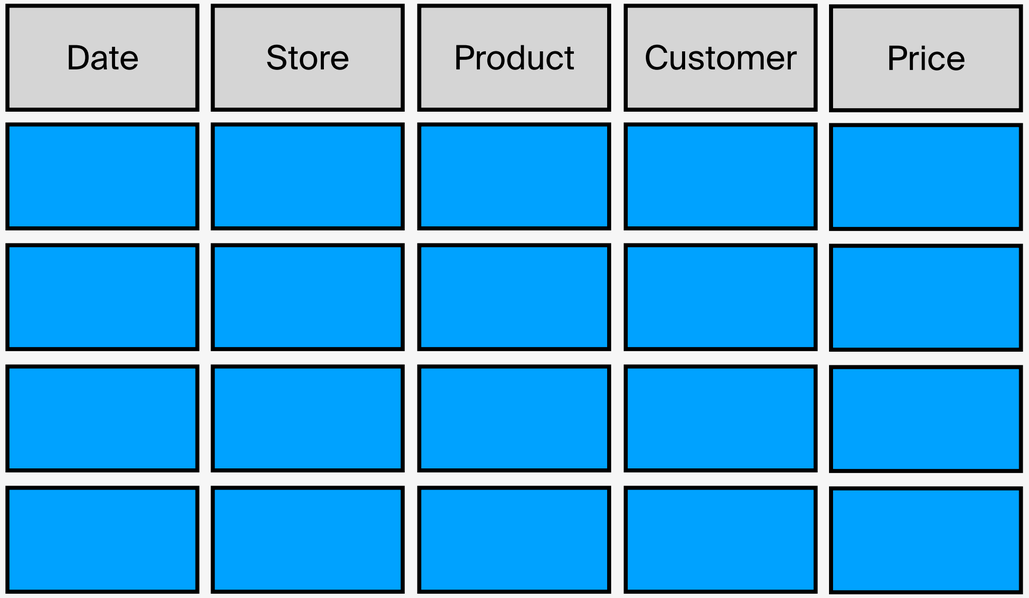
Design

Design

- Data Layout (Column Store)
- Data Compression
- Vectorized Execution Engine
- End-to-end Query Optimization
- Automatic Parallelism
- Beyond Memory Execution
- Python Ecosystem Integrations

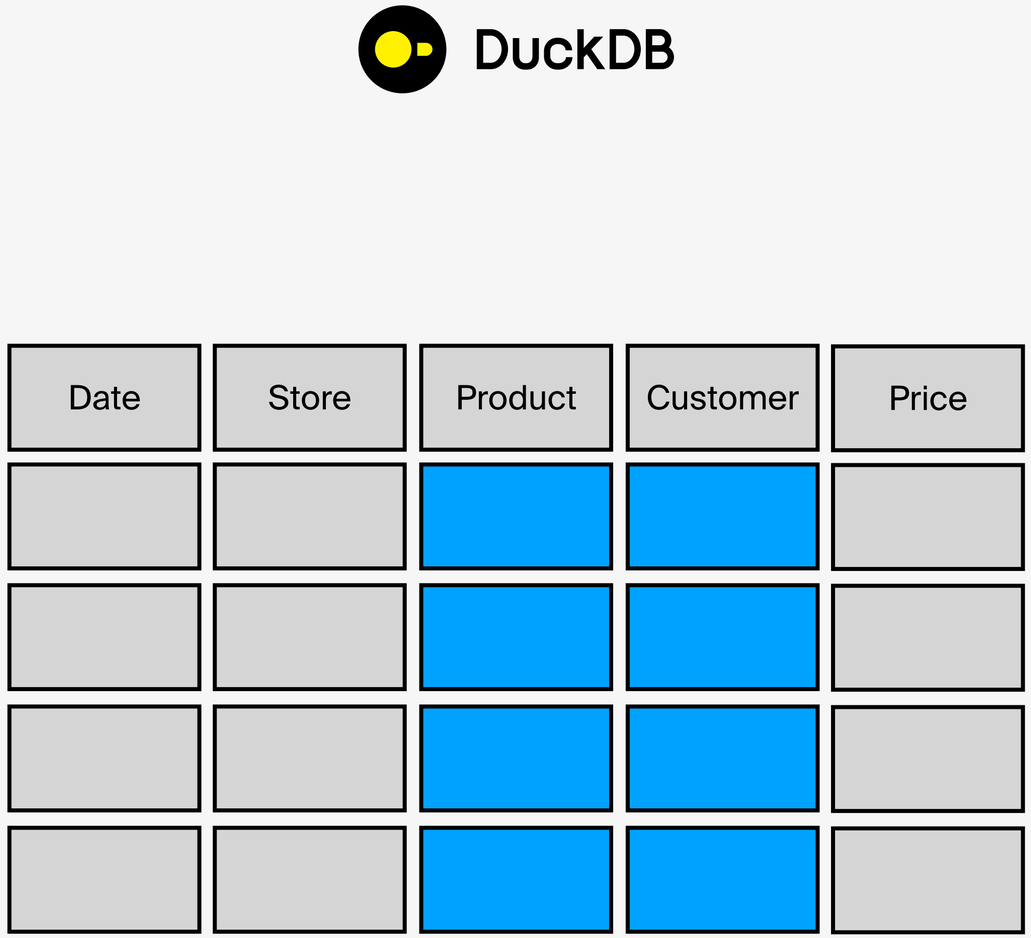
Data Layout

- Row-Storage:
  - Individual rows can be fetched cheaply
  - However, all columns must be fetched!
- What if we only use a few columns?
- e.g.: What if we are only interested in the price of a product, not the stores in which it is sold?



Columnar Data Storage

- Column-Storage:
  - We can fetch individual columns
  - Immense savings on disk IO/memory bandwidth when only using few columns
- e.g.: What if we are only interested in the price of a product, not the date and stores in which it is sold?



Columnar Data Storage

- Example:
  1TB table with 100 columns. A query requires 5 columns from the table.
  - Row-store:
    Read entire 1TB of data from disk at 100MB/s ≅ 3 hours
  - Column-store:
    Read 5 columns (50GB) from disk ≅ 8 minutes

Compression

- Individual columns often have similar values, e.g. dates are usually increasing
  - Save ~3-5X on storage
    (depending on compression algorithms used and data)

DuckDB Version	Taxi	Ratio	Lineitem	Ratio	Compression	Date
0.2.8	15.3 GB	1	0.85 GB	1	None	07/21
0.2.9	11.2 GB	1.36x	0.79 GB	1.07x	RLE + Constant	09/21
0.3.2	10.8 GB	1.41x	0.56 GB	1.51x	Bitpacking	02/22
0.3.3	6.9 GB	2.21x	0.32 GB	2.64x	Dictionary	24/22
0.5.0	6.6 GB	2.31x	0.29 GB	2.93x	For	09/22
0.6	4.8GB	3.18x	0.17 GB	5x	FSST + CHIMP	11/22

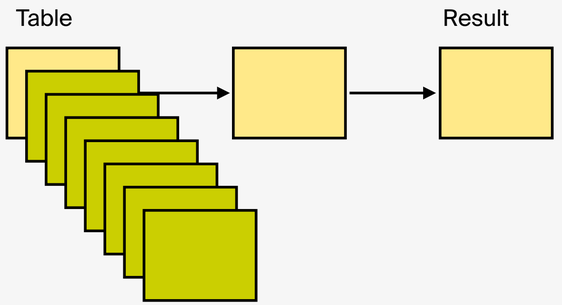
Compression

- Example:
  1TB table with 100 columns. A query requires 5 columns from the table.
  - No compression:
    Read 5 columns (50GB) from disk ≅ 8 minutes
  - Compression:
    Read 5 columns compressed by 5x (10GB) from disk ≅ 1:40 minutes

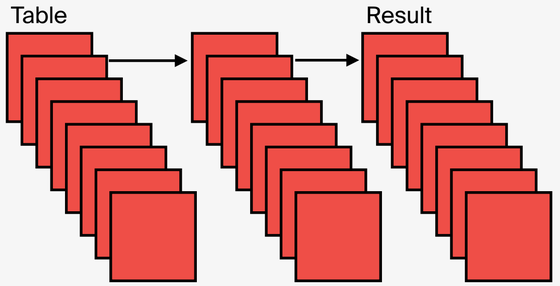
Execution

- SQLite uses tuple-at-a-time processing
  - Low RAM use, but slow
- Pandas uses column-at-a-time processing
  - Faster (SIMD), but lots of RAM needed
- DuckDB uses vectorized processing
  - Best of both!
  - Moderate RAM
  - Fast processing using SIMD
  - Use CPU's L1 cache

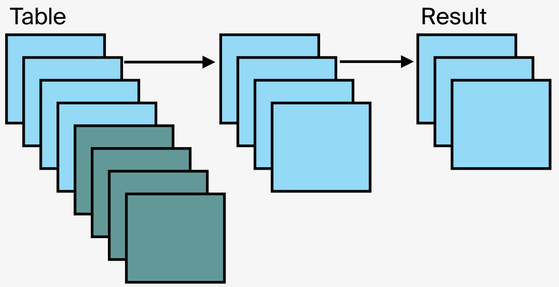
Tuple-at-a-Time



Column-at-a-Time



Vectorized Processing

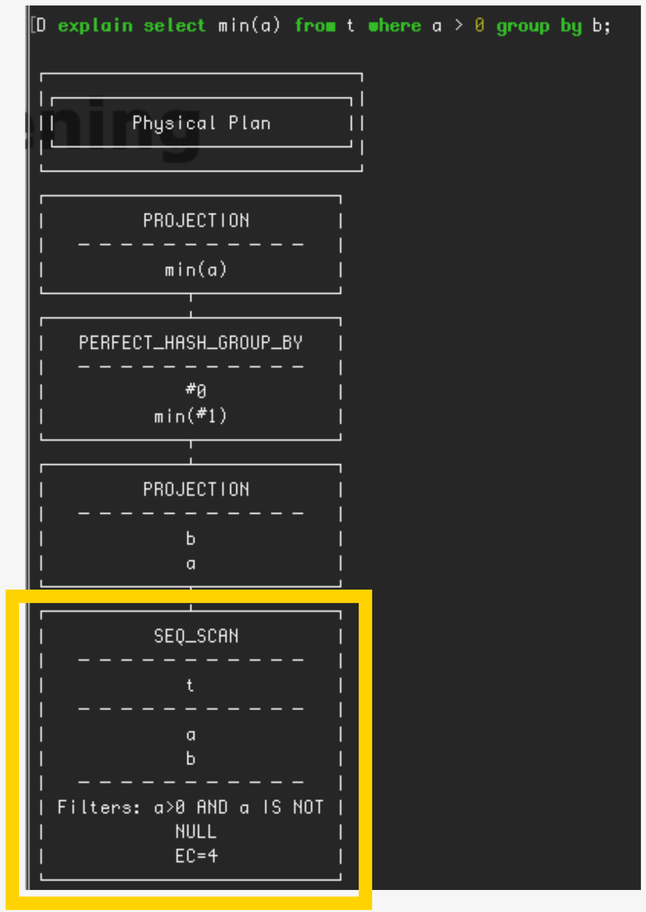


End-To-End Query Optimization

- Expression rewriting
- Join Ordering
- Subquery Flattening
- Filter/Projection Pushdown
  - Automatic in DuckDB
  - Manual in Pandas

DuckDB

```
D explain select min(a) from t where a > 0 group by b;
```

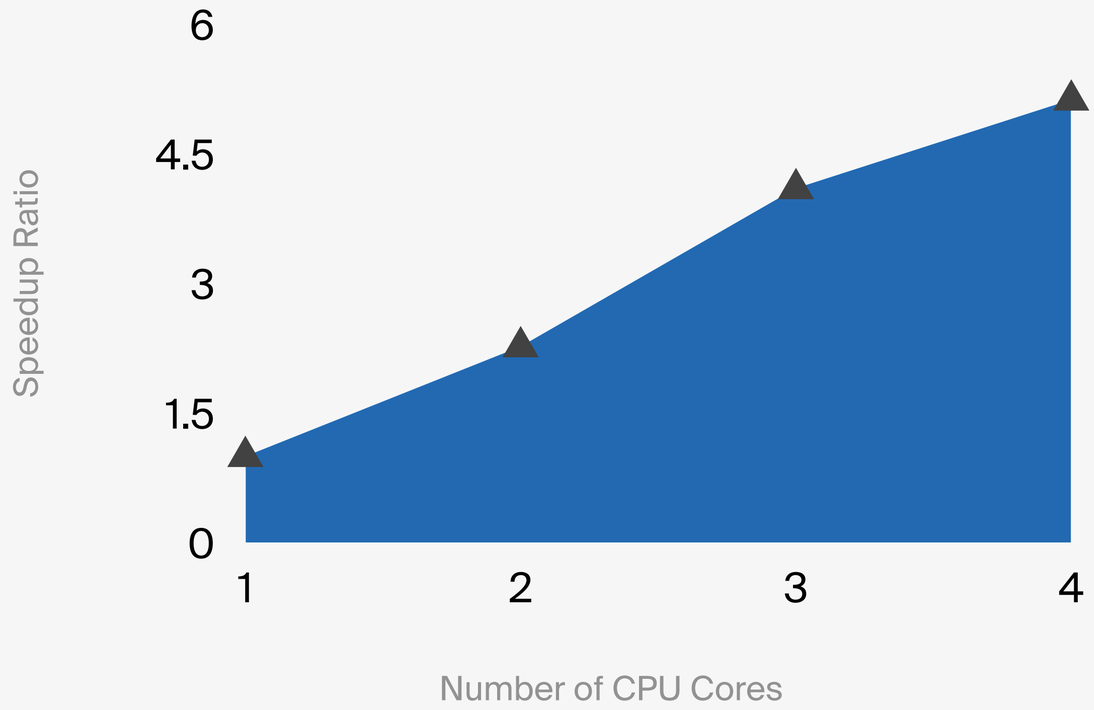


pandas

```python
# filter out the rows
filtered_df = t[ t['a'] > 0]
# perform the aggregate
result = filtered_df.groupby(['b']).agg(
  Min=('a', 'min')
)
```

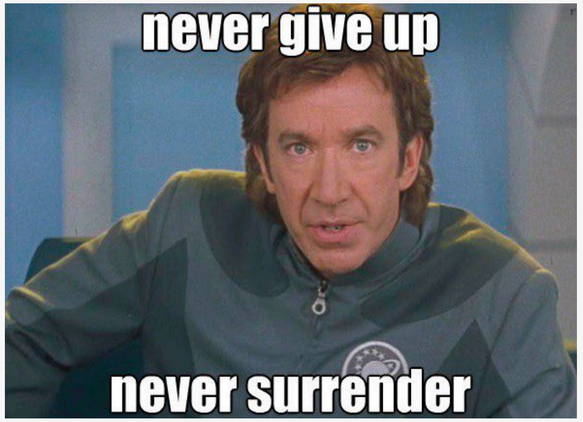
Automatic Parallelism

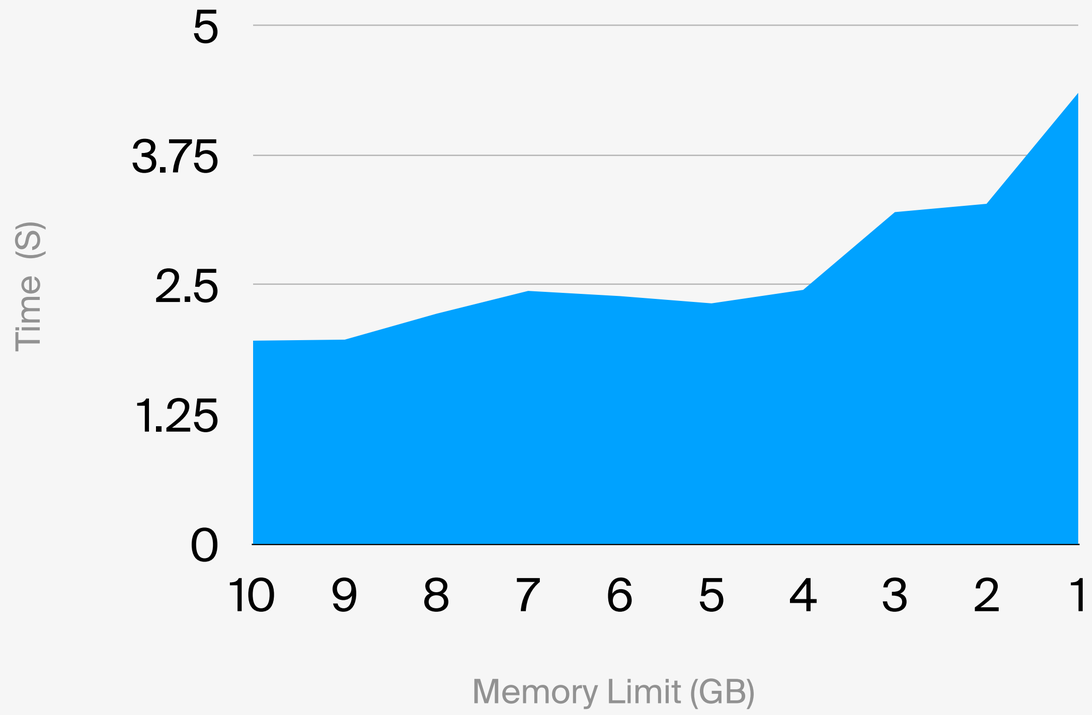
- DuckDB has parallel versions of most operators
  - Scanners (Insertion Order Preservation)
  - Aggregations
  - Joins
- e.g., Aggregations TPC-H Q06 SF 10



Beyond Memory Execution - Hash Join

- DuckDB supports larger-than-RAM data
  - Graceful degradation
  - Never crash - always execute the query
    - e.g., Hash-Join





Python Integrations

Python Ecosystem Integrations

Data Formats

NumPy

pandas

APACHE ARROW

polars

TensorFlow

PyTorch

Data Engineering

dbt

Apache Airflow

dagster

Airbyte

Dataframe APIs

The Ibis Project

PONDER

Siuba

Visualization

HEX

Mode

Apache Superset™

Vega Fusion

Other

SQLAlchemy

fsspec

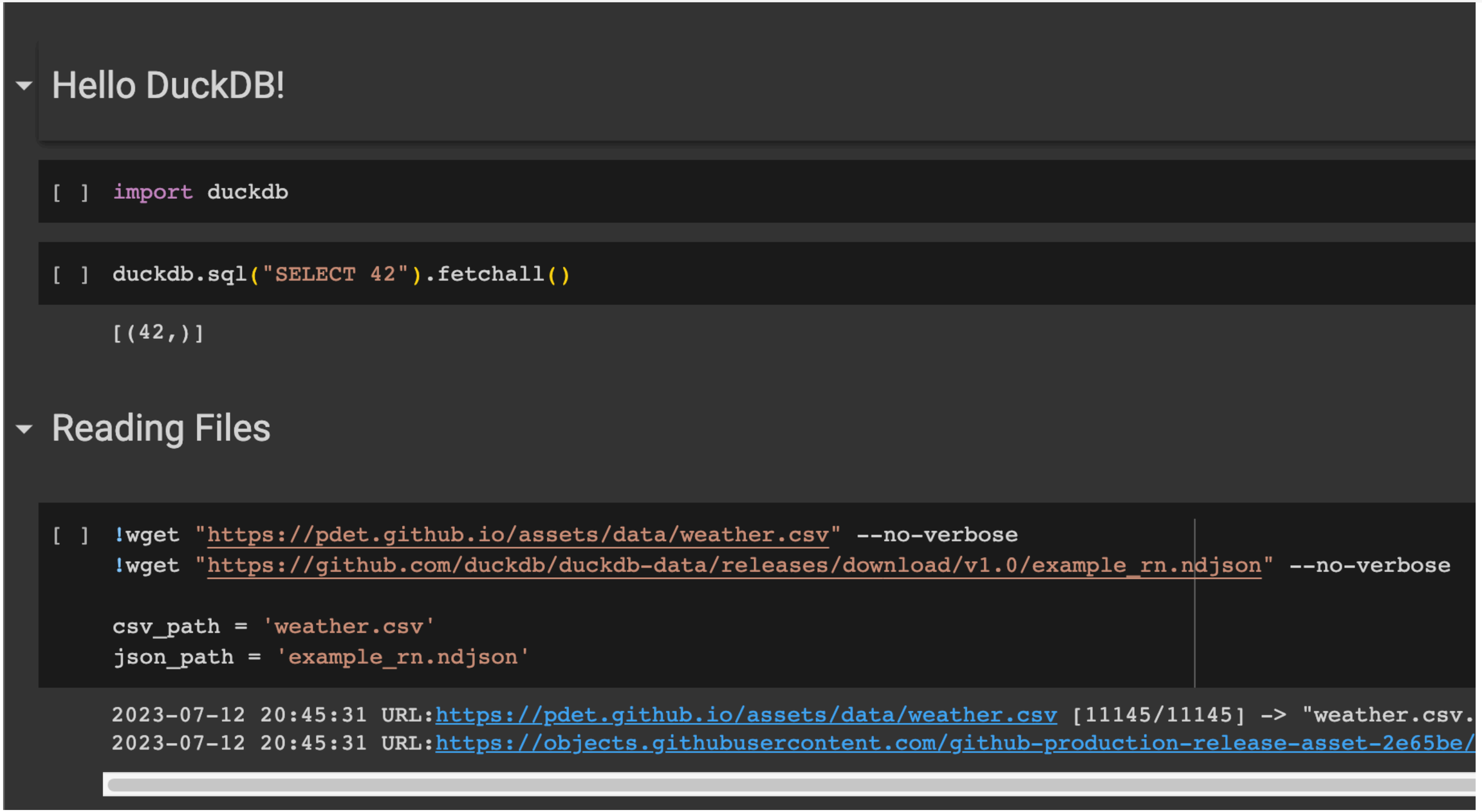
JupySQL

Demo

Google Colab Demo Links

- Hello DuckDB!
  - https://colab.research.google.com/drive/11-kMIcHcLK851EySATPwUhtmUNBWGrPh?usp=sharing
- Hello JupySQL!
  - https://colab.research.google.com/drive/1rlPmusXJmHZgn5ONtCWYj8p1mH70Zvhe?usp=sharing

Hello DuckDB!



Hello DuckDB!

```
[ ]  # Read CSV
     duckdb.from_csv_auto(csv_path).show()
```

date date	maximum_temperature int64	minimum_temperature int64	…	precipitation varchar	snow_fall varchar	snow_depth varchar
2016-01-01	42	34	…	0.00	0.0	0
2016-01-02	40	32	…	0.00	0.0	0
2016-01-03	45	35	…	0.00	0.0	0
2016-01-04	36	14	…	0.00	0.0	0
2016-01-05	29	11	…	0.00	0.0	0
2016-01-06	41	25	…	0.00	0.0	0
2016-01-07	46	31	…	0.00	0.0	0
2016-01-08	46	31	…	0.00	0.0	0
2016-01-09	47	40	…	T	0.0	0
2016-01-10	59	40	…	1.80	0.0	0
·	·	·	·	·	·	·
·	·	·	·	·	·	·
·	·	·	·	·	·	·
2016-12-22	49	37	…	0	0	0
2016-12-23	47	38	…	0	0	0
2016-12-24	47	38	…	0.47	0	0
2016-12-25	50	36	…	0	0	0
2016-12-26	50	33	…	0.02	0	0
2016-12-27	60	40	…	0	0	0
2016-12-28	40	34	…	0	0	0
2016-12-29	46	33	…	0.39	0	0
2016-12-30	40	33	…	0.01	T	0
2016-12-31	44	31	…	0	0	0

366 rows (20 shown) 7 columns (6 shown)

```
     # Read Json
     duckdb.read_json(json_path).show()
```

id int64	name varchar
1	O Brother, Where Art Thou?
2	Home for the Holidays
3	The Firm
4	Broadcast News
5	Raising Arizona

Hello DuckDB!

Reading Remote Files

```
[ ] # Install the httpfs extension
    duckdb.sql("install httpfs; load httpfs;")

    # Treat a Parquet file as if it were a table in a SQL query
    remote_parquet_data = duckdb.sql("FROM 'https://shell.duckdb.org/data/tpch/0_01/parquet/lineitem.parquet'").df()
    remote_parquet_data.describe()
```

	l_orderkey	l_partkey	l_suppkey	l_linenumber	l_quantity	l_extendedprice	l_discount	l_tax
count	60175.000000	60175.000000	60175.000000	60175.000000	60175.000000	60175.000000	60175.000000	60175.000000
mean	29958.613594	1002.701321	50.535970	3.004271	25.527661	35765.513261	0.049930	0.040225
std	17299.239820	575.677416	28.862957	1.734026	14.406573	21844.234985	0.031612	0.025774
min	1.000000	1.000000	1.000000	1.000000	1.000000	904.000000	0.000000	0.000000
25%	14981.500000	504.000000	26.000000	2.000000	13.000000	17557.440000	0.020000	0.020000
50%	29888.000000	1004.000000	50.000000	3.000000	25.000000	34245.120000	0.050000	0.040000
75%	44932.000000	1500.500000	76.000000	4.000000	38.000000	51354.910000	0.080000	0.060000
max	60000.000000	2000.000000	100.000000	7.000000	50.000000	94949.500000	0.100000	0.080000

Hello DuckDB!

Reading and Writing DataFrames

```
[ ]  import pandas as pd
```

```
[ ]  my_df = pd.DataFrame([{"column1":"duck","column2":"duck","column3":"goose"}])
     my_df
```

	column1	column2	column3
0	duck	duck	goose

```
[ ]  new_df = duckdb.sql("SELECT * FROM my_df").df()
     new_df
```

	column1	column2	column3
0	duck	duck	goose

```
[ ]  new_df = duckdb.sql("FROM my_df SELECT * REPLACE('GOOSE' as column3)").df()
     new_df
```

	column1	column2	column3
0	duck	duck	GOOSE

Hello DuckDB!

```
[ ]  import pyarrow as pa
```

```
[ ]  arrow_table = duckdb.sql("FROM my_df SELECT * REPLACE('GOOSE' as column3)").arrow()
     arrow_table
```

```
pyarrow.Table
column1: string
column2: string
column3: string
----
column1: [["duck"]]
column2: [["duck"]]
column3: [["GOOSE"]]
```

Hello JupySQL!

Install and Configure JupySQL

```
[ ] # Install JupySQL to enable SQL cells in Jupyter
    !pip install jupysql --quiet
    # Install the DuckDB SQLAlchemy driver
    !pip install duckdb-engine --quiet
```

```
[ ] import duckdb
    import pandas as pd
```

```
[ ] # Load JupySQL
    %load_ext sql
```

```
The sql extension is already loaded. To reload it, use:
  %reload_ext sql
```

```
[ ] # Output directly to Pandas and make output less verbose
    %config SqlMagic.autopandas = True
    %config SqlMagic.feedback = False
    %config SqlMagic.displaycon = False
```

```
[ ] # Connect to an in-memory DuckDB instance
    %sql duckdb:///:memory:
```

Hello JupySQL!

Query DuckDB using SQL Cells!

```
[ ]  %%sql
     SELECT
       42 as my_column
```

	my_column
0	42

```
[ ]  %sql INSTALL httpfs;
     %sql LOAD httpfs;
```

Success

Hello JupySQL!

```
[ ]  %%sql output_df <<
     FROM 'https://shell.duckdb.org/data/tpch/0_01/parquet/lineitem.parquet'
```

```
[ ]  output_df
```

	l_orderkey	l_partkey	l_suppkey	l_linenumber	l_quantity	l_extendedprice	l_discount	l_tax	l_returnflag	l_linestatus	l_shipdate	l_commitdate	l_receiptdate	l_shipinstruct	l_shipmode	l_comment
0	1	1552	93	1	17.0	24710.35	0.04	0.02	N	O	1996-03-13	1996-02-12	1996-03-22	DELIVER IN PERSON	TRUCK	egular courts above the
1	1	674	75	2	36.0	56688.12	0.09	0.06	N	O	1996-04-12	1996-02-28	1996-04-20	TAKE BACK RETURN	MAIL	ly final dependencies: slyly bold
2	1	637	38	3	8.0	12301.04	0.10	0.02	N	O	1996-01-29	1996-03-05	1996-01-31	TAKE BACK RETURN	REG AIR	riously. regular, express dep
3	1	22	48	4	28.0	25816.56	0.09	0.06	N	O	1996-04-21	1996-03-30	1996-05-16	NONE	AIR	lites. fluffily even de
4	1	241	23	5	24.0	27389.76	0.10	0.04	N	O	1996-03-30	1996-03-14	1996-04-01	NONE	FOB	pending foxes. slyly re
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
60170	60000	1843	44	2	23.0	40131.32	0.05	0.03	N	O	1995-08-09	1995-06-08	1995-08-23	COLLECT COD	FOB	ideas about the permanent
60171	60000	1057	63	3	45.0	43112.25	0.02	0.02	R	F	1995-05-15	1995-05-31	1995-06-03	COLLECT COD	SHIP	fully bold pinto beans alongside
60172	60000	271	53	4	29.0	33966.83	0.02	0.01	N	O	1995-07-25	1995-06-07	1995-08-17	COLLECT COD	SHIP	ly final ideas boost s
60173	60000	585	16	5	31.0	46052.98	0.00	0.05	N	O	1995-08-06	1995-07-18	1995-08-19	TAKE BACK RETURN	TRUCK	ly even instr
60174	60000	836	3	6	45.0	78157.35	0.04	0.08	N	O	1995-07-23	1995-07-17	1995-07-24	DELIVER IN PERSON	TRUCK	ke final packages. carefully final fo

60175 rows × 16 columns

Summary

What I want you to remember about DuckDB!

1. It’s Fast
    - Multi-core
    - Vectorized engine
    - In process -> fast data transfer
2. It’s Easy
    - pip install duckdb (MIT License!)
    - No server to manage!
    - Friendly SQL dialect and relational API
3. Works great with Python
    - DuckDB speaks Pandas, PyArrow, & more!
4. Handles larger than RAM data
    - Never give up, never surrender!

Alex Monahan / alex@duckdblabs.com / alex@motherduck.com