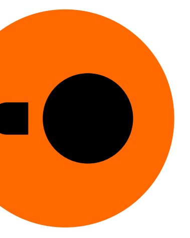
DuckDB: Bringing analytical SQL directly to your Python shell

Pedro Holanda
pedro@duckdblabs.com
pedroholanda.com

DuckDB Labs

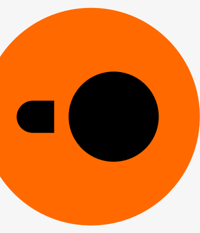
Outline

- What is DuckDB?
  - Motivation
  - Main Characteristics
- DuckDB in the Python-Land
- Demo 7~10 minutes.
  - Estimating NYC taxi fare costs with DuckDB,Pandas and PySpark.
- Summary

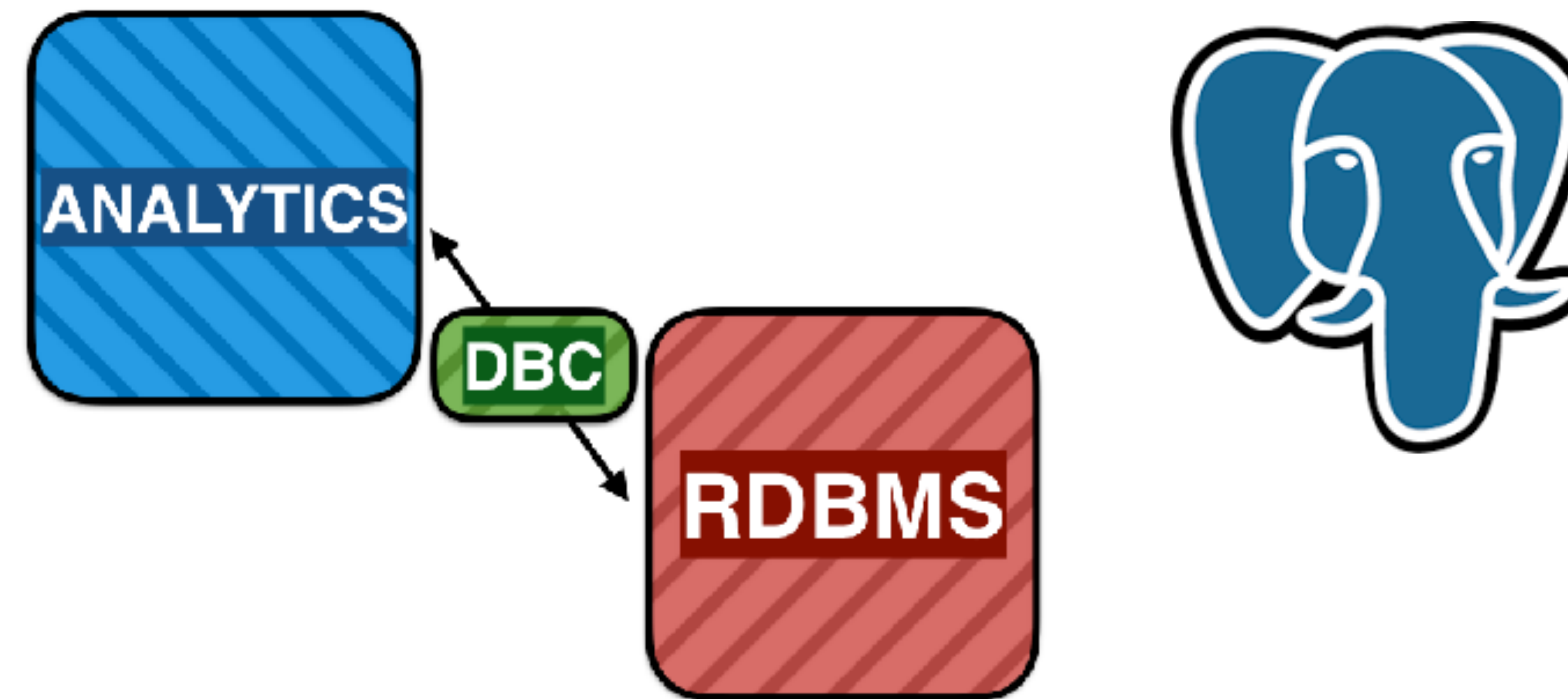


What is DuckDB

Motivation



- **Combining Database Management Systems with Data Science**
  - **DB Connection**



- **Embedded**

Main Characteristics

- **DuckDB: The SQLite for Analytics**

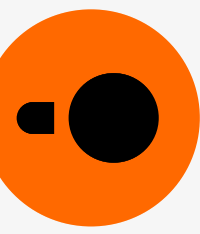
- **Simple installation**

```
$ pip install duckdb
```

- **Embedded:** no server management
- **Fast analytical processing**
- **Fast transfer between R/Python and RDBMS**

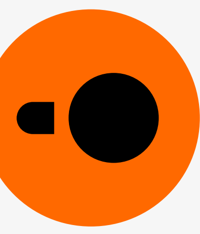
- **DuckDB** is currently in **pre-release (V0.6)**
  - Check duckdb.org for more details.

Main Characteristics

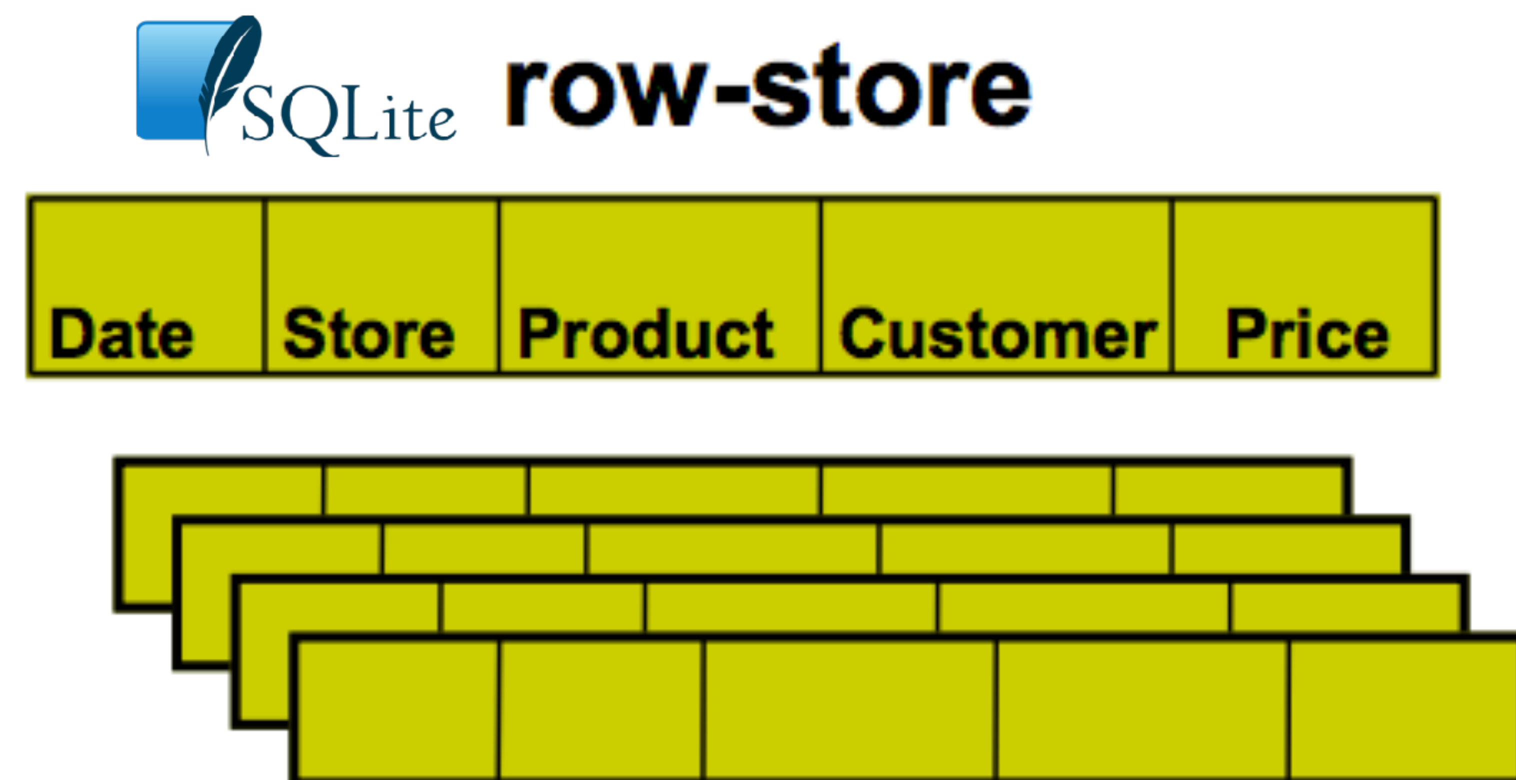


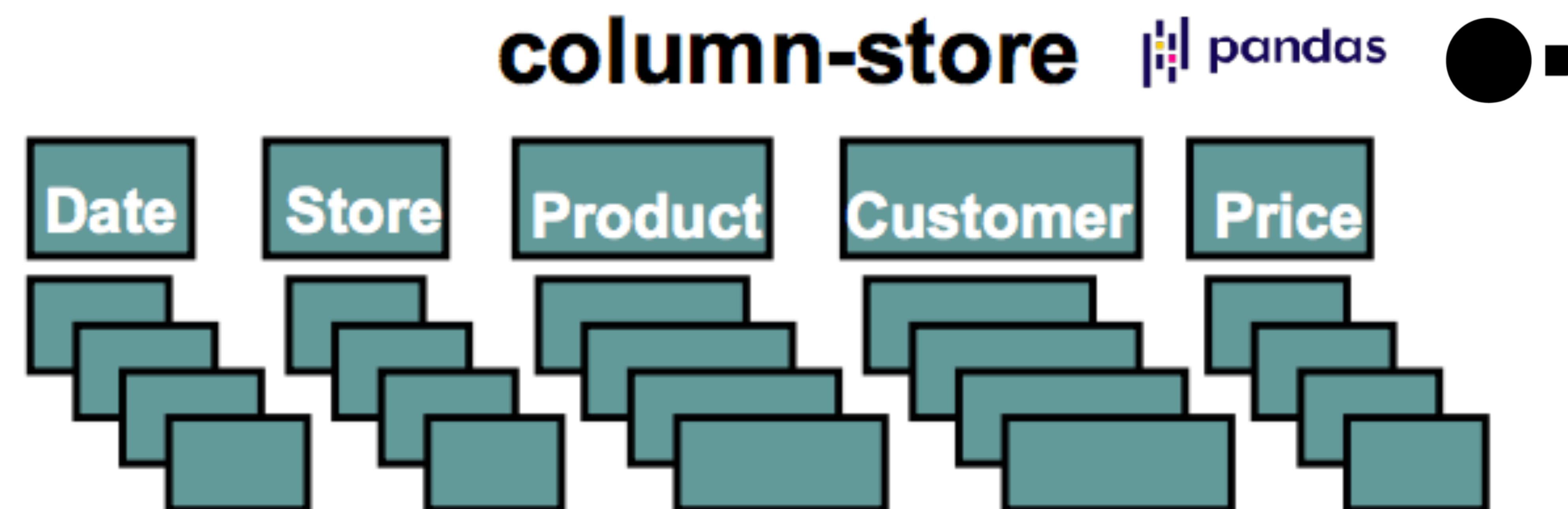
- **Columnar Data Storage**
- **Vectorized Execution Engine**
- **End-to-end Query Optimization**
- **Automatic Parallelism**
- **Data Compression**
- **Beyond Memory Execution**

Columnar Data Storage

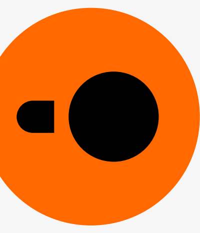


- **Row-Storage:**
  - Individual rows can be fetched cheaply
  - However, all columns must always be fetched!
- What if we only use a few columns?
- e.g.: What if we are only interested in the price of a product, not the stores in which it is sold?

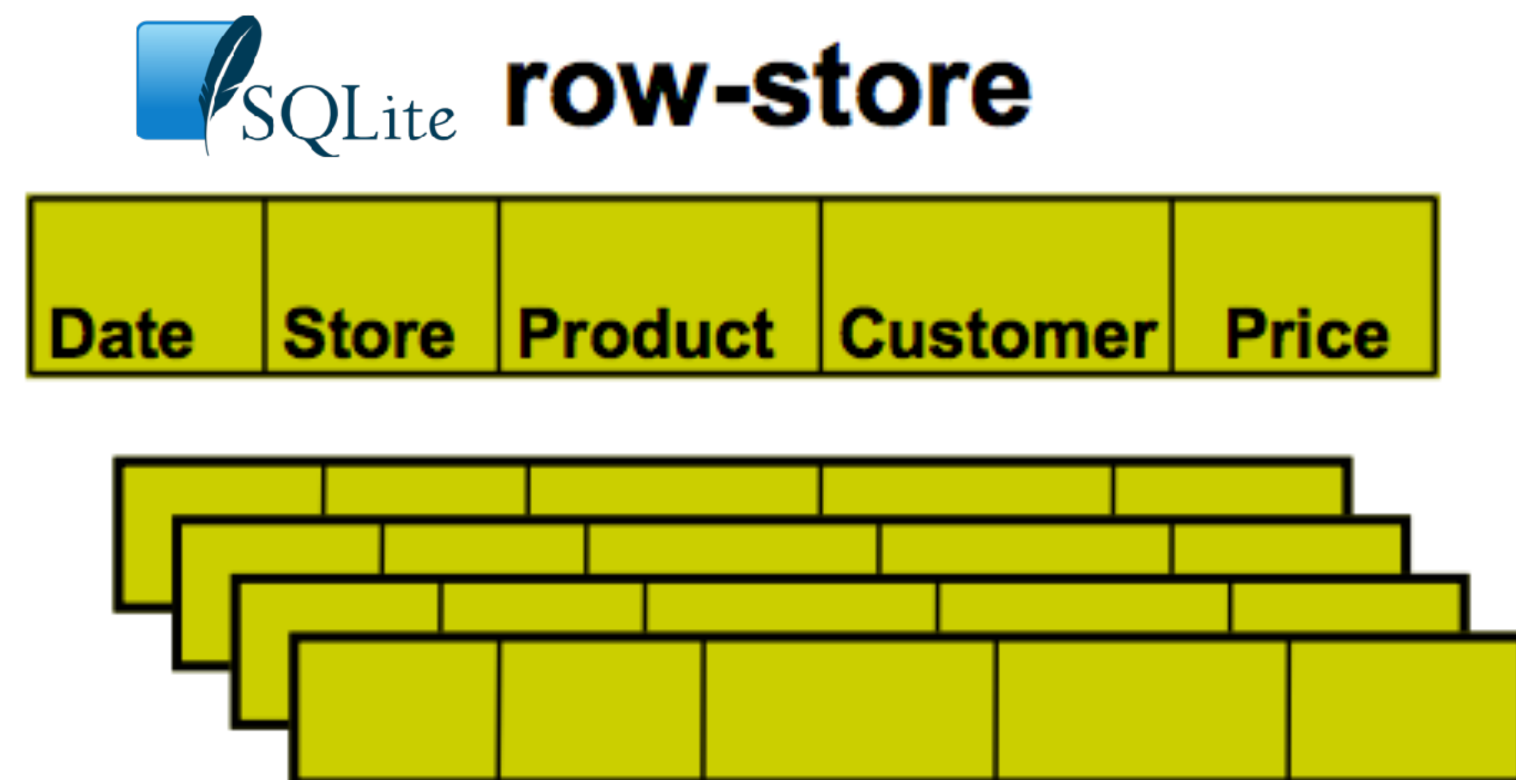


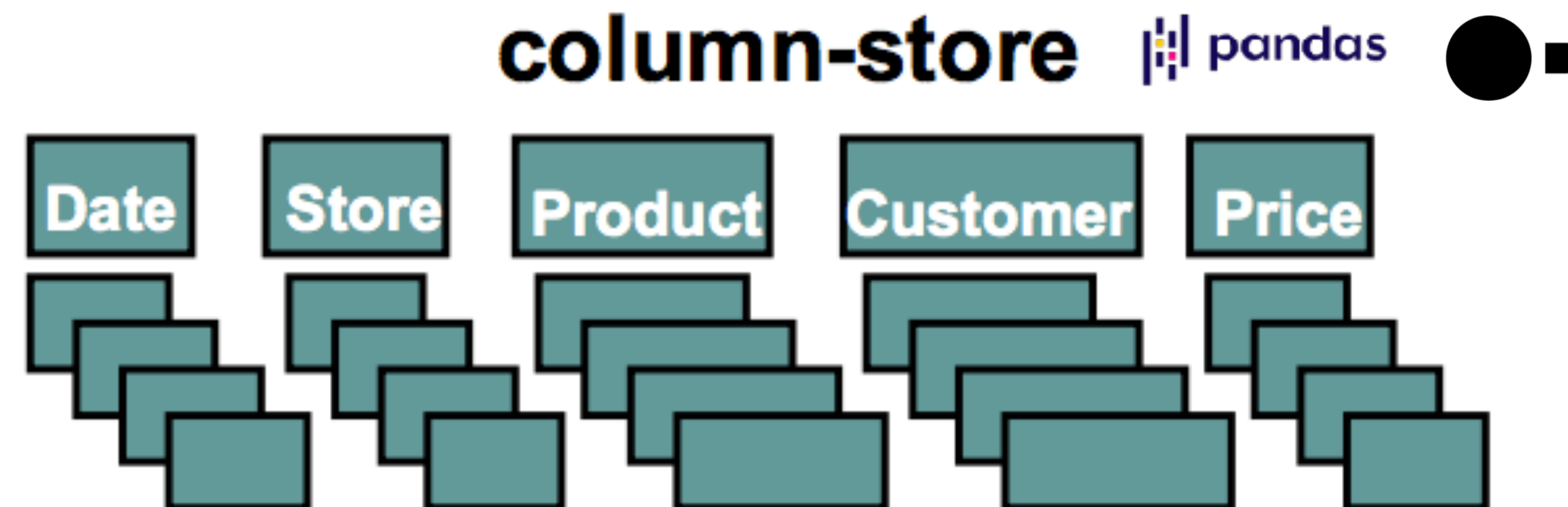


Columnar Data Storage

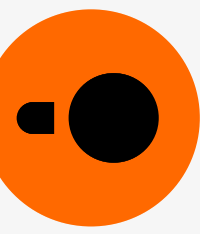


- **Column-Storage:**
  - We can fetch individual columns
  - Immense savings on disk IO/memory bandwidth when only using few columns





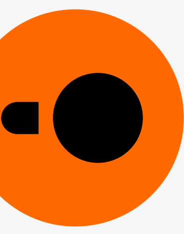
Compression



- 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

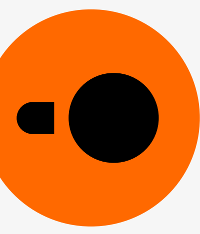


- **Example:**
  We have a query that requires 5 columns of the table.

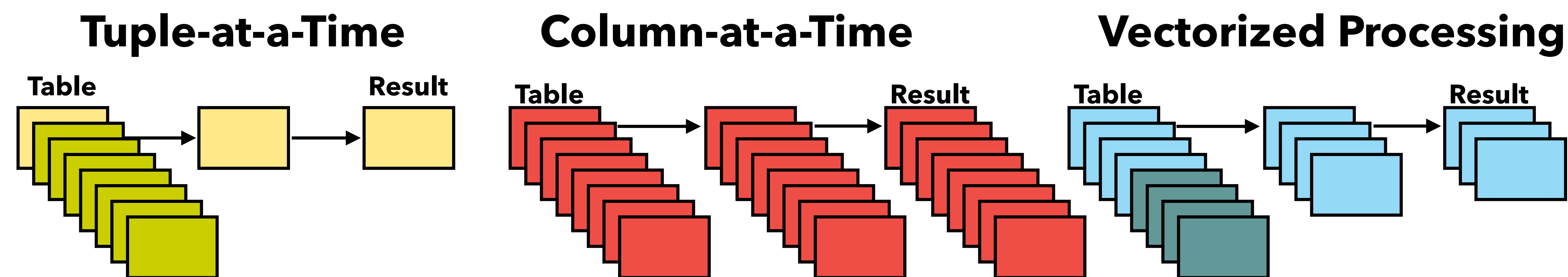
  - **No compression:**
    Read 5 columns (50GB) from disk ≅ 8 minutes

  - **Compression:**
    Read 5 compressed columns (5x = 10GB) from disk ≅ 1:40 minutes

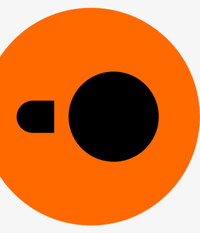
Execution

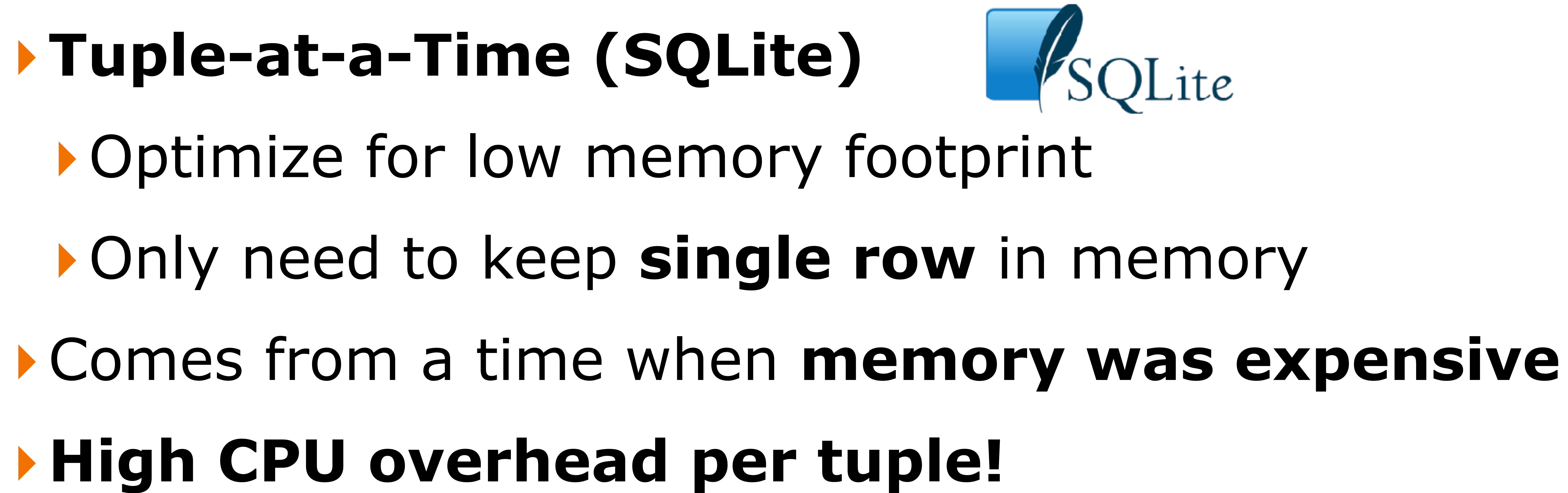


- **SQLite use tuple-at-a-time processing**
  - Process one row at a time
- **Pandas use column-at-a-time processing**
  - Process entire columns at once
- **DuckDB uses vectorized processing**
  - Process batches of columns at a time

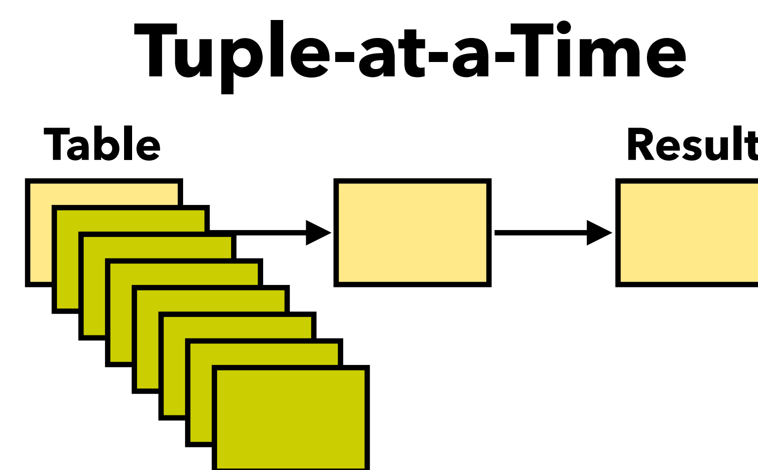


Execution

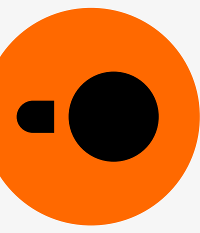




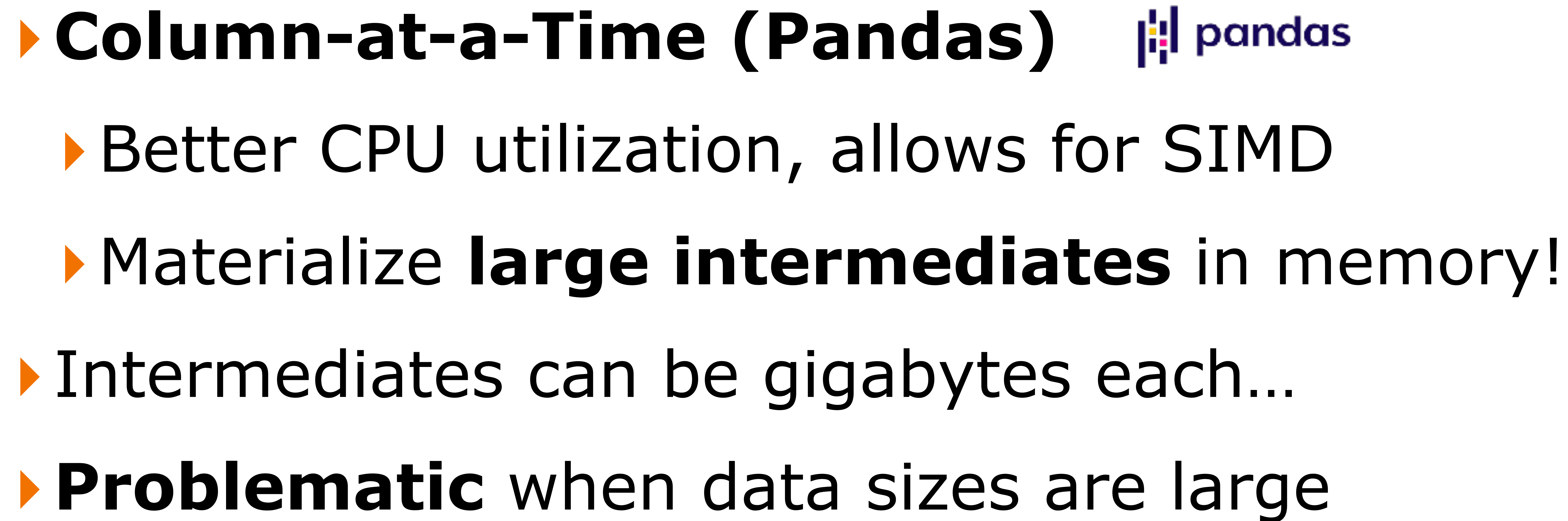
- **Tuple-at-a-Time (SQLite)**
  - Optimize for low memory footprint
  - Only need to keep **single row** in memory
- Comes from a time when **memory was expensive**
- **High CPU overhead per tuple!**



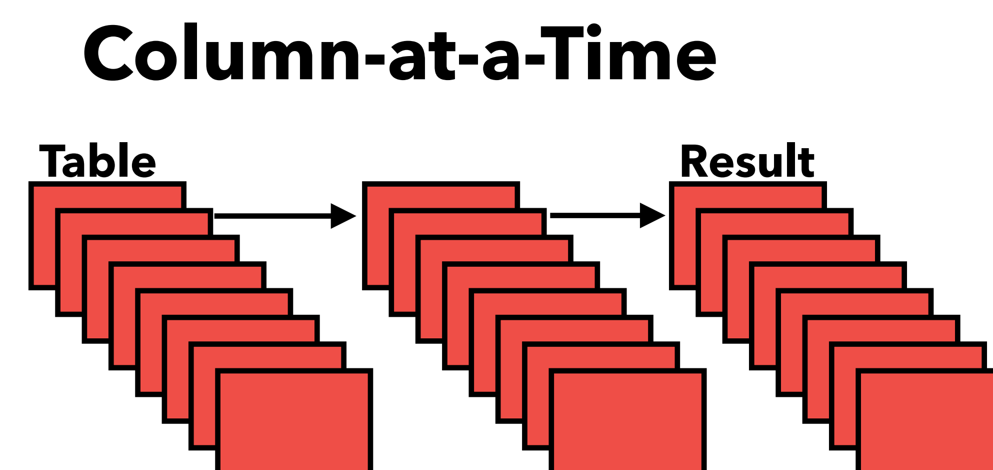
Execution



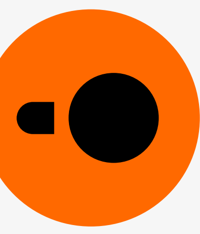
- **Column-at-a-Time (Pandas)**



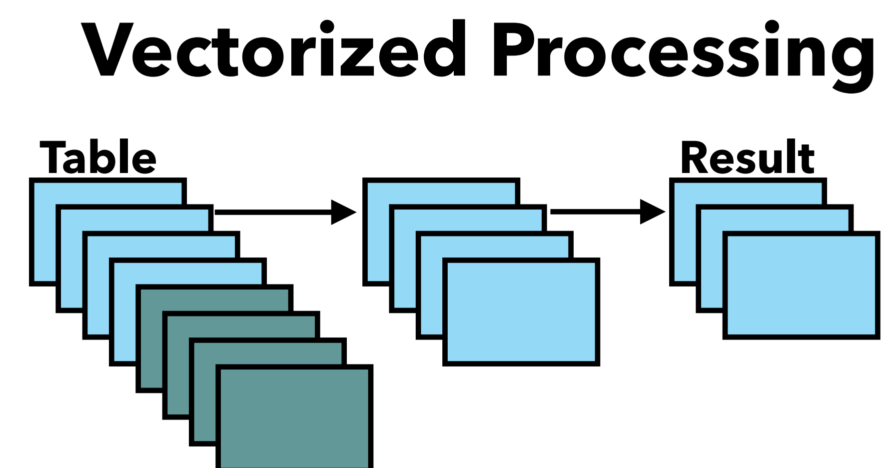
  - Better CPU utilization, allows for SIMD
  - Materialize **large intermediates** in memory!
- Intermediates can be gigabytes each…
- **Problematic** when data sizes are large

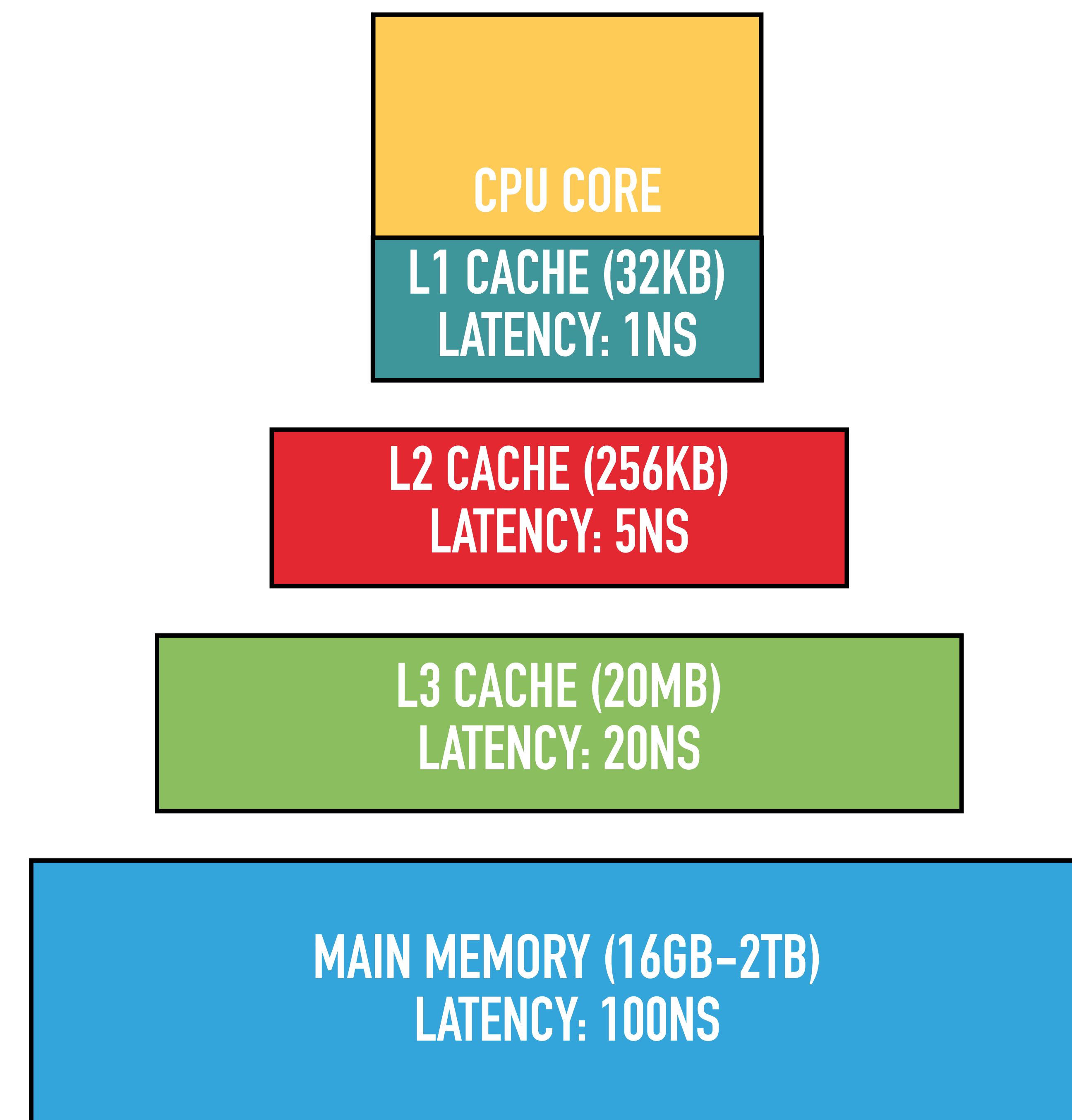


Execution

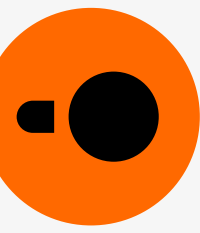


- **Vectorized Processing (DuckDB)**
  - Optimized for CPU Cache locality
  - SIMD instructions, Pipelining
  - **Small** intermediates **(ideally fit in L1 cache)**

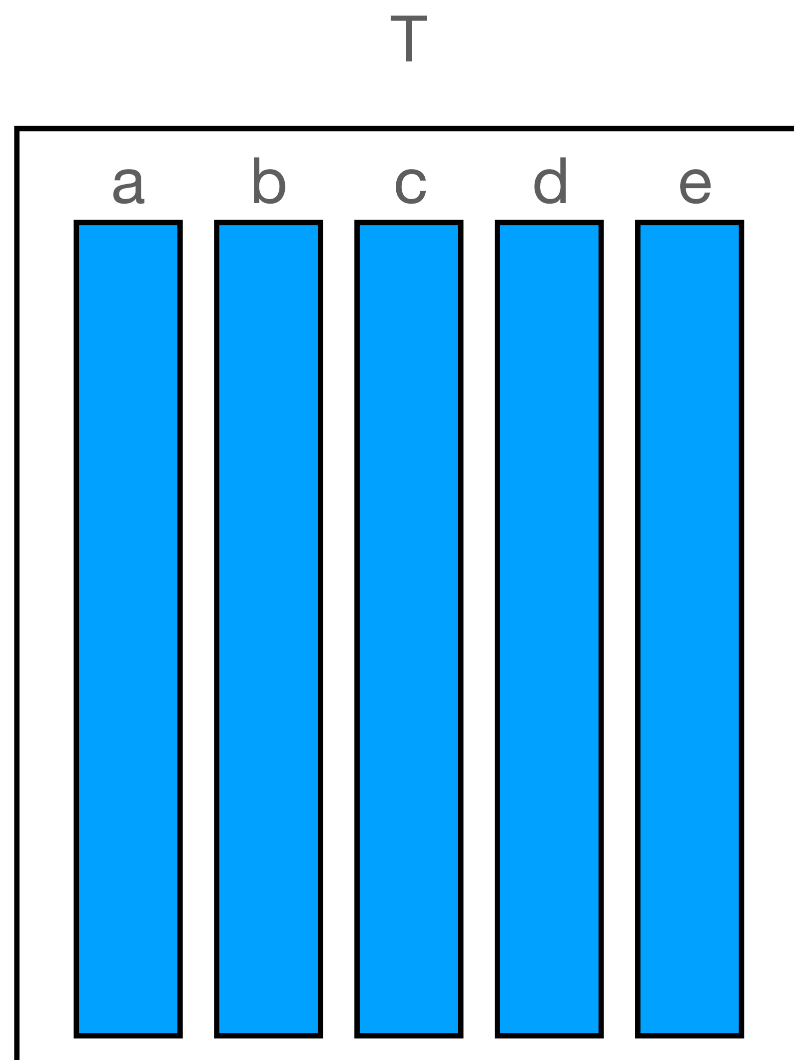




End-To-End Query Optimization



- **Expression rewriting**
- **Join Ordering**
- **Subquery Flattening**
- **Filter/Projection Pushdown**

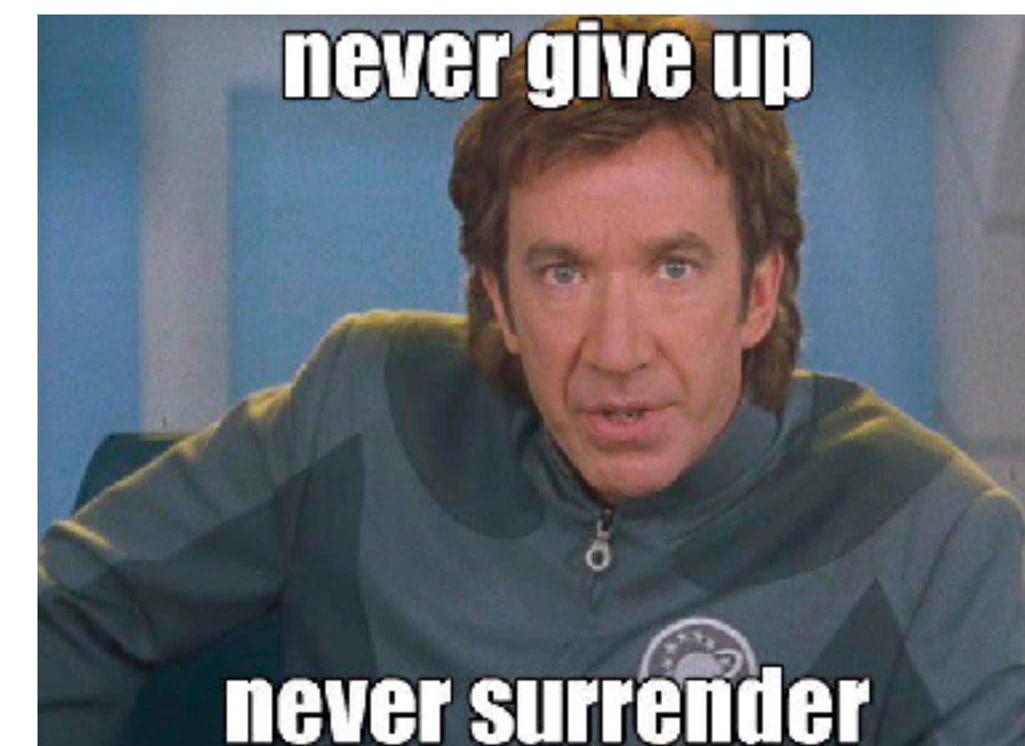


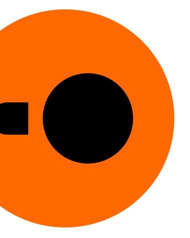
```python
1  # filter out the rows
2  filtered_df = t[ t['a'] > 0]
3  # perform the aggregate
4  result = filtered_df.groupby(['b']).agg(
5    Min=('a', 'min')
6  )
```

Automatic Parallelism & Beyond Memory Execution

- **DuckDB** has **parallel versions of most operators**
  - **Scanners (Insertion Order Preservation)**
  - **Aggregations**
  - **Joins**
- **Pandas** only support **single-threaded** execution**.**

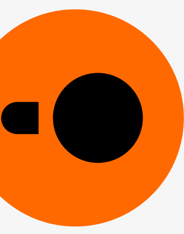
- DuckDB supports execution of **data that does not fit in memory**
  - **Graceful Degradation**
  - **Never Crash always executes query**





DuckDB In the Python Land

APIs



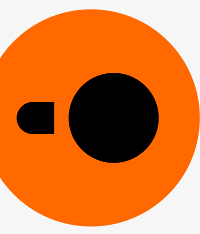
- **Python DB API 2.0 Compliant**

```
import duckdb
con = duckdb.connect("duck.db")
con.execute("SELECT j+1 FROM integers WHERE i=2")
```

- **Relational API**

```
import duckdb
con = duckdb.connect("duck.db")
# Table operator returns a table scan
rel = con.table("integers")
# We can inspect intermediates
rel.show()
# We can chain multiple operators
rel.filter("i=2").project("j+1").show()
```

Integrations



- **Tight Integration - Zero Copy (Input + Output)**
  - **Pandas**

```
1  import pandas as pd
2  import duckdb
3
4  d = {'col1': [1, 2], 'col2': [3, 4]}
5  df = pd.DataFrame(data=d)
6
7  con = duckdb.connect()
8
9  # Consumes Pandas Dataframe
10 res = con.execute("select * from df")
11
12 # Produces Pandas Dataframe
13 result_dataframe = res.df()
```

  - **PyArrow**

```
1  import pyarrow as pa
2  import duckdb
3
4  d = {'col1': [1, 2], 'col2': [3, 4]}
5  arrow = pa.Table.from_pydict(d)
6
7  con = duckdb.connect()
8
9  # Consumes Arrow Object
10 res = con.execute("select * from arrow")
11
12 # Produces Arrow Table
13 result_arrow = res.arrow()
```

  - **NumPy**
  - **SQL Alchemy**
  - **IBIS (Default Backend)**

Usage

Downloads last day: 33,594

Downloads last week: 251,770

Downloads last month: 898,816

Daily Download Quantity of duckdb package - Overall

30d 60d 90d 120d all

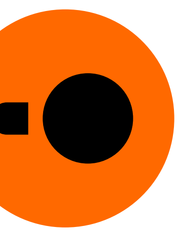
Downloads

60,000
50,000
40,000
30,000
20,000
10,000
0

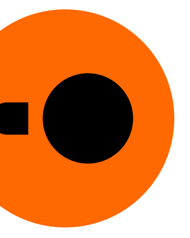
With_Mirrors
Without_Mirrors

08-08 08-15 08-22 08-29 09-05 09-12 09-19 09-26 10-03 10-10 10-17 10-24 10-31 11-07 11-14 11-21 11-28 12-05 12-12 12-19 12-26 01-02 01-09 01-16 01-23 01-30

Date

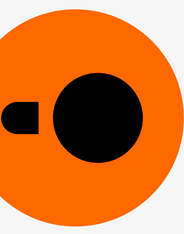


Demo



Summary

Summary



- DuckDB is an embedded database system.
- Designed for **Analytical Queries** (i.e., Data Analysis/Science).
- **Open-Source** (Under MIT license) and free to use!
- Has **binding for many languages** (e.g., Python, R, Java…)
- **Tightly integrated** with the Python Ecosystem.
    - **Zero-Copy** access to Python/NumPy and PyArrow datasets.
- Implements the **DB and Relational APIs.**
- **Full SQL** Support!

Pedro Holanda
pedro@duckdblabs.com
pedroholanda.com