DuckDB:

An embedded database for data science

Pedro Holanda
holanda@cwi.nl

About Us

- Developed @ CWI in Amsterdam;
- Database Architectures group;
- We are supervised by Hannes Mühleisen; Stefan Manegold and Peter Boncz.
- Mark & Hannes created and are the main maintainers of DuckDB.

CWI

Why we care about data science?

- Most PhD Projects at CWI are sponsored by private companies;
- Most of our projects are related to **data** science (Particularly, ML pipelines);
- Main goal is to optimize/facilitate the management of data in data science projects.

TATA

PROMIMOOC

HRI

Honda Research Institute EU

DAMTOSO

Outline

Outline (Today)

- Why Database Systems.
- How our data science projects were managing data before us?
- Our failed attempts of integrating RDBMSs and ML pipelines.
- DuckDB: The light in the end of the tunnel.
- **Hands-on ~ 45 min. [Using DuckDB]**

Why Database Systems?

Motivation

- Why **should** people use relational database systems?

- This is a strange question in our field (DBMS research)
- **Obviously** everyone should use RDBMSs!

- But for many people it is not so obvious
- So *why should you actually use a RDBMS?*

Database Example

- Database that models a digital music store to keep track of artists and albums.

- Things we need to store:
  - Information about artists.
  - What albums those artists released.

This example is originally from:
https://15445.courses.cs.cmu.edu/fall2019/slides/01-introduction.pdf

Flat File

- Store database as comma-separated value (CSV) files that we manage in our own code
  - Use separate file per entity
  - The application has to parse files each time they want to read/update records

Flat File Example

- Database that models a digital music store

Artist (name, year, country)

```
“Backstreet Boys”,1994,"USA"
“Ice Cube”, 1992,"USA
“Notorious BIG”,1989,USA
```

Album (name,artist,year)

```
“Millenium", "Backstreet Boys", 1999
“DNA”, “Backstreet Boys”, 2019
“AmeriKKKa's Most Wanted”, “Ice Cube”, 1990
```

Flat File Example

- Get the year that Ice Cube went solo

Artist (name, year, country)

“Backstreet Boys”,1994,"USA"

“Ice Cube”, 1992,"USA

“Notorious BIG”,1989,USA

```
for line in file
    record = parse(line)
    if 'Ice Cube' == record[0]
        print int(record[1])
```

Data Integrity

- How do we ensure that the artist is the same for each album entry?
- What if someone overwrites the album year with an invalid string?
- How do we store that there are multiple artists on an album?

Implementation

- How do we find a particular record?
- What if we now want to create a new application that uses the same database?
- What if two threads try to write to the same file at the same time?

Durability

- What if the machine crashes while our program is updating a record?
- What if we want to replicate the database on multiple machines for high availability?

Database Management System

- Software that allows application to store and analyse information in a database.

- A general-purpose DBMS is designed to allow the definition, creation, querying, update and administration of databases.

How our data science projects were managing data before us?

Motivation

- Many data scientists do not use relational databases
  - Despite requiring many of the things they offer!
  - Data management, data wrangling…
- Instead: Engineer their own solutions
  - **Flat files** to store data (CSV, Binary, HDF5, etc).
  - **dplyr/pandas** as query execution engines.

Flat File Management - what is the problem?

- Manually managing files is cumbersome.
- Loading and parsing e.g. CSV files is inefficient.
- File writers typically do not offer resiliency.
  - Files can be corrupted;
  - Difficult to change/update.
- It does not scale!
  - The reason people use it:

```r
# load a CSV file into a DataFrame
df <- read.csv("input.csv", sep="|")
# write a CSV file to a DataFrame
write.csv(df, sep="|")
```

Dplyr; Pandas; DataFrames

- For those unfamiliar: these libraries are basically query execution engines

```
SELECT SUM(l_quantity)
FROM lineitem
GROUP BY l_returnflag, l_linestatus;
```

dplyr →

```
lineitem %>% group_by(l_returnflag, l_linestatus) %>%
    summarise(sum_qty=sum(l_quantity))
```

```
SELECT *
FROM part JOIN partsupp ON (p_partkey=ps_partkey)
WHERE p_size=15 AND p_type LIKE '%BRASS';
```

dplyr →

```
part %>%  filter(p_size == 15, grepl(".*BRASS$", p_type)) %>%
    inner_join(partsupp, by=c("p_partkey" = "ps_partkey"))
```

Dplyr; Pandas; DataFrames - What is the problem?

- The problem is that they are *very poor query engines*!
- Materialize huge intermediates
- **No** query optimizer
  - Not even for basics like filter pushdown
- No support for out of memory computation
- No support for parallelization
- Unoptimized implementations for joins/aggregations

Data Science

- Data scientists **need** the functionality RDBMSs offer
- But they opt not to use RDBMSs
- Often this leads to problems down the road
  - When the data gets bigger…
  - When a power outage corrupts their data…

Can we save these lost souls and unite them with the word of codd?

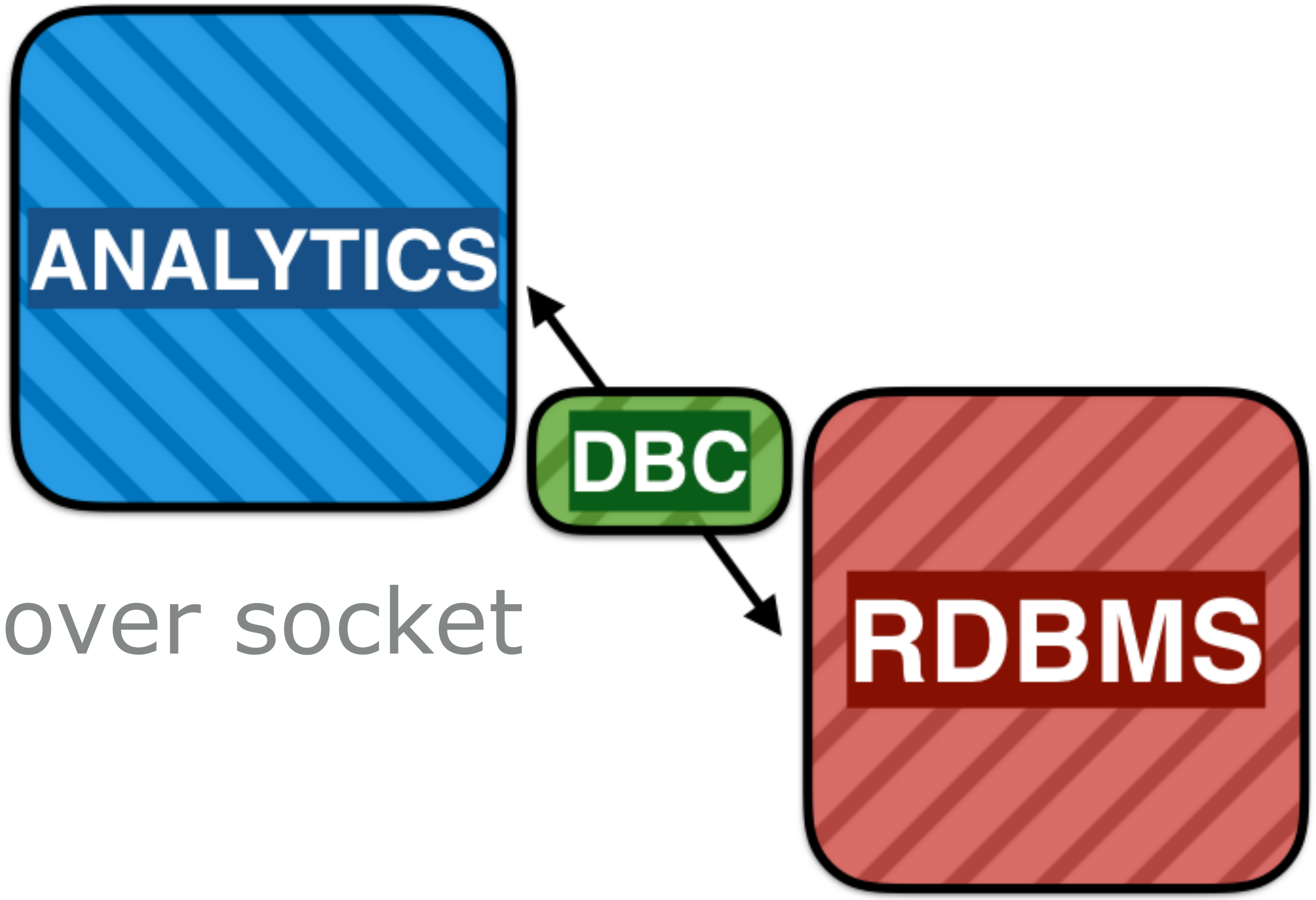
Combining DBMSs with analytical tools

Our failed attempts of integrating RDBMSs and data science.

How?

- Database Client Connections
- User Defined Functions
- Embedded Databases

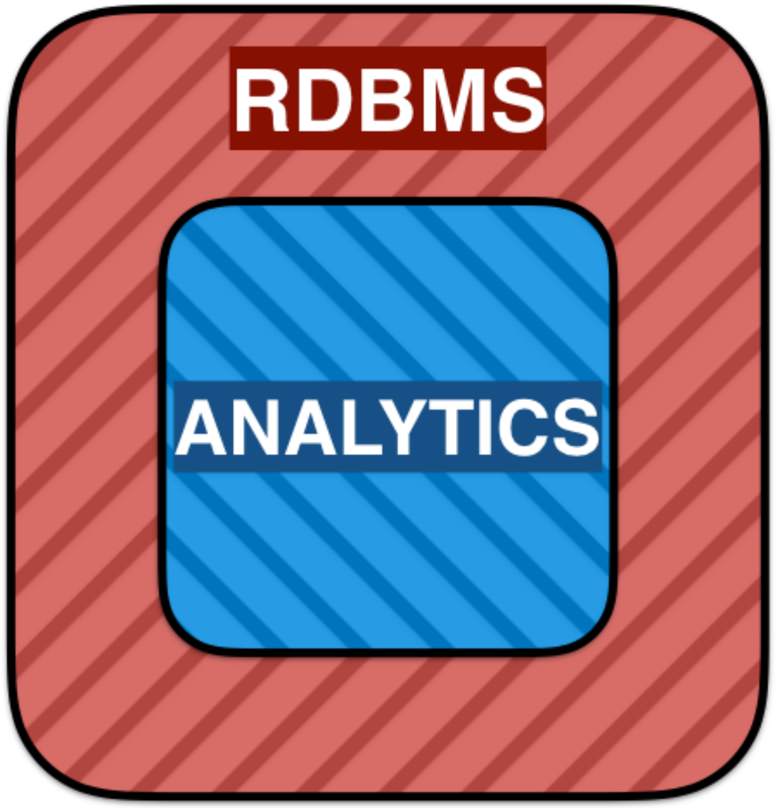
Database Client Connections



- **1: DB Connection**
- DBMS is separate process
- Queries & Data transferred over socket
- Problems:
  - Data transfer is very slow (both directions)
  - Requires setup & management of DBMS server

User-Defined Functions

- How can we combine analytical tools (R/Python) with relational databases?



- **2: User-Defined Functions (UDFs)**
- Analytics is run inside DBMS server
- No separate analytics program!
- Problems:
  - Difficult to implement and debug
  - DBMS-specific, requires knowledge of DBMS internals
  - Also requires setup & management of DBMS server…

SQLite

- **3: Embedded Databases**
- It runs inside the analytics applications;
- Has same low-transfer cost advantages as UDFs;
- Are easy to install;
- The clients are easy to use, require almost no change from original source code;
- Binds to almost every language.
- What the most famous embeddable DBMS?
  - SQLite

SQLite

- SQLite is an embedded database
  - No external server management
- It has bindings for every language
- Database is stored in a single **file** (not directory)
- **Easy to install!!!**

* https://www.sqlite.org/famous.html

SQLite

- SQLite is great
- It is public domain and very easy to use
- It is secretly the most used RDBMS in the world
  - Runs on every phone, browser and OS*
  - It even runs inside airplanes!

* https://www.sqlite.org/famous.html

SQLite

- SQLite has one problem: designed for OLTP

- Row store (basically a giant B-tree)
- Tuple-at-a-time processing model
- Does not utilise memory to speed up computation
- Query optimizer is very limited

- **Great for OLTP, not so good for analytics**

CWI

DuckDB
an Embeddable Analytical RDBMS

DuckDB

DuckDB

- DuckDB: The SQLite for Analytics

- **Core Features:**
  - Simple installation
  - Embedded: no server management
  - Single file storage format
  - Fast analytical processing
  - Fast transfer between R/Python and RDBMS

CWI

DuckDB

- Why "Duck" DB?

- Ducks are amazing animals
- They can fly, walk and swim
- They are resilient
- They can live off anything

- Also Hannes used to own a pet duck

DuckDB

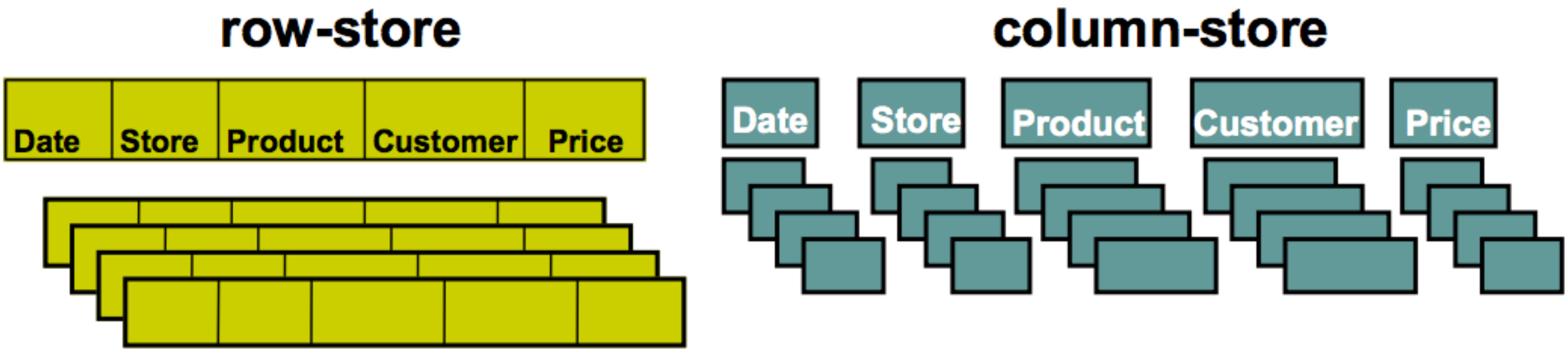
- **DuckDB Internals Quick-Summary**
  - Column-storage database
  - Vectorized processing model
  - MVCC for concurrency control
  - ART index, used also for maintaining key constraints
  - Combination of both cost/rule based optimizer
- We use the PostgreSQL parser
- Bindings for C/C++, Python and R

Why is DuckDB Faster than SQLite for Data Science?

- Data Science is OLAP!
- Storage Model
  - Row-Store vs Column-Store
- Compression
- Query Execution
  - Row-wise vs vector-wise

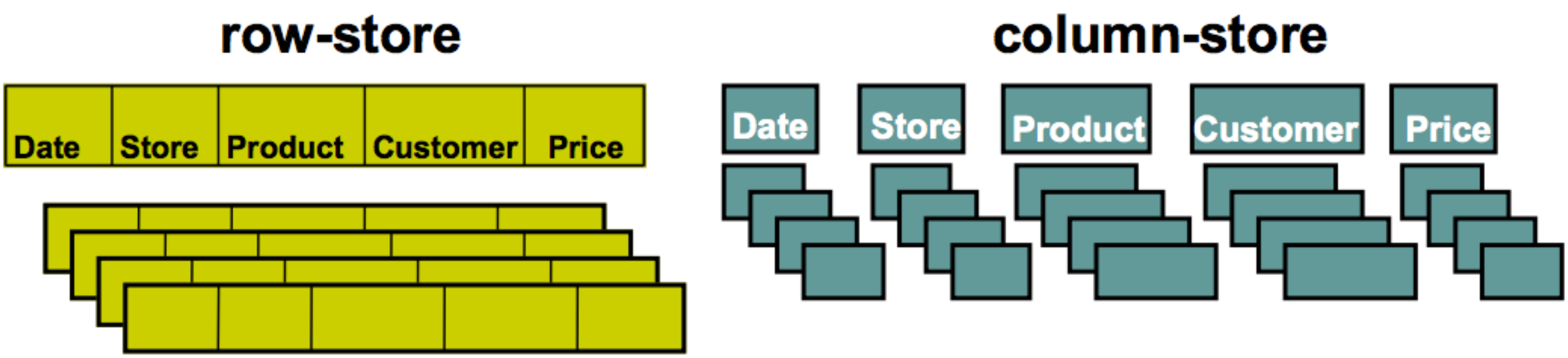
Storage Model

- SQLite use a row-storage model
- DuckDB uses a columnar storage model



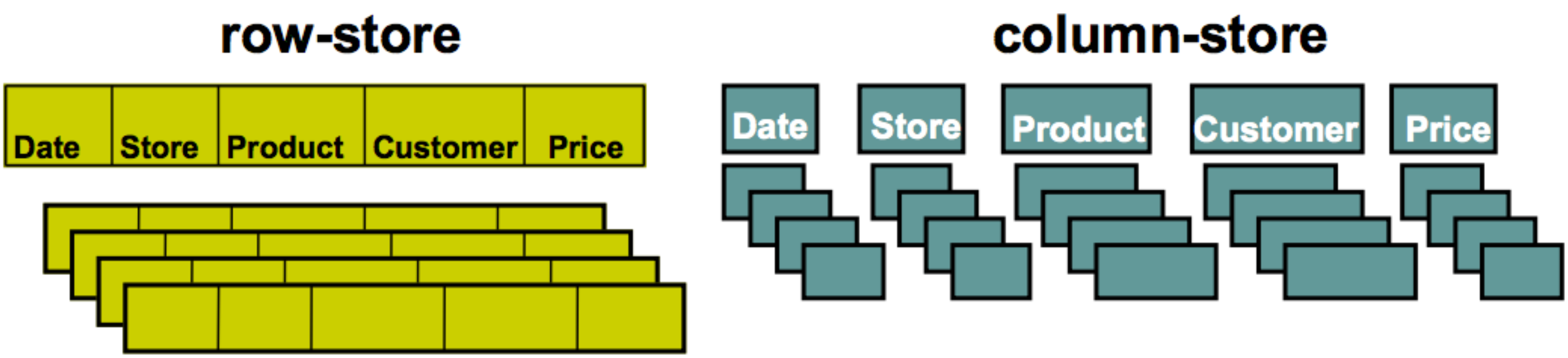
Storage Model

- **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?



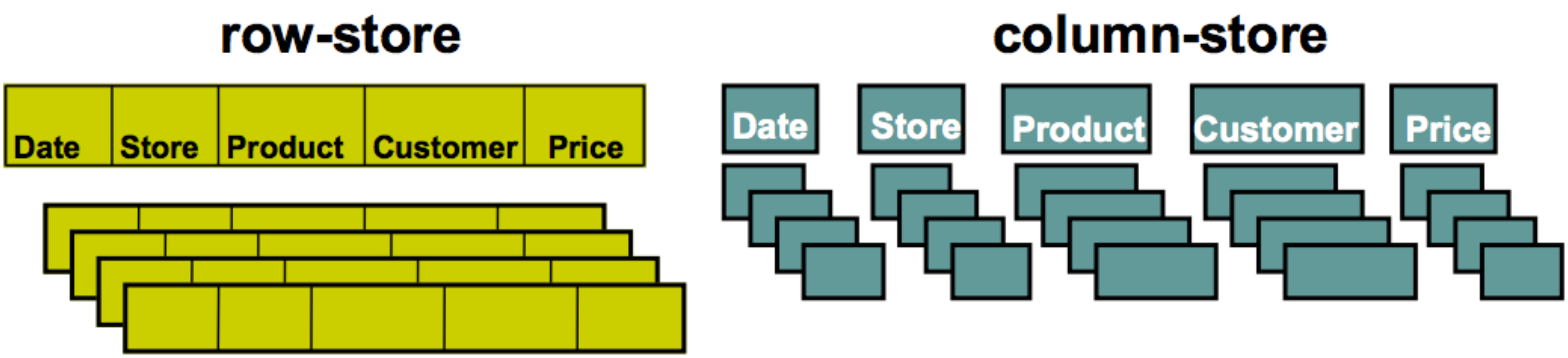
Storage Model

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



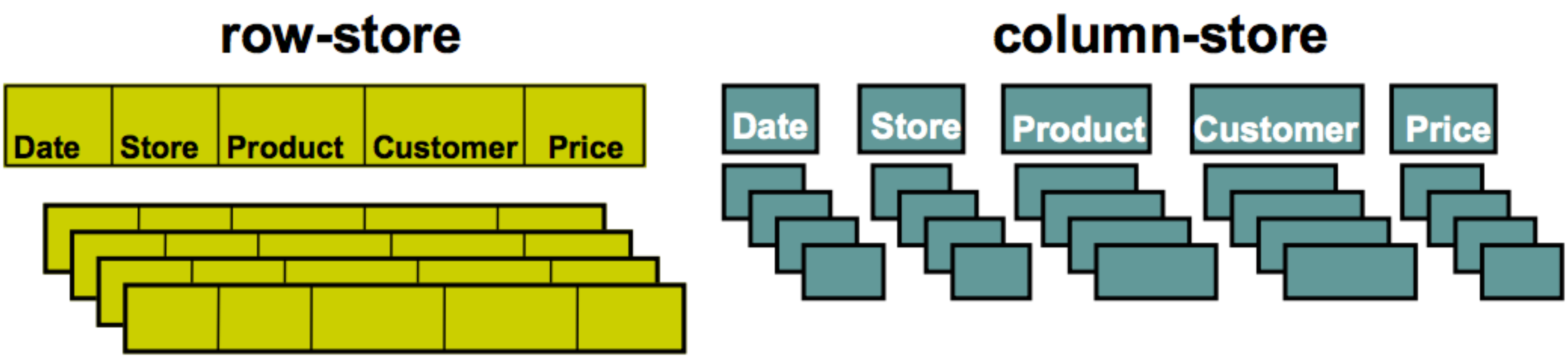
Storage Model

- **Example:** Suppose we have a 1TB table with 100 columns. We have a query that requires 5 columns of 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



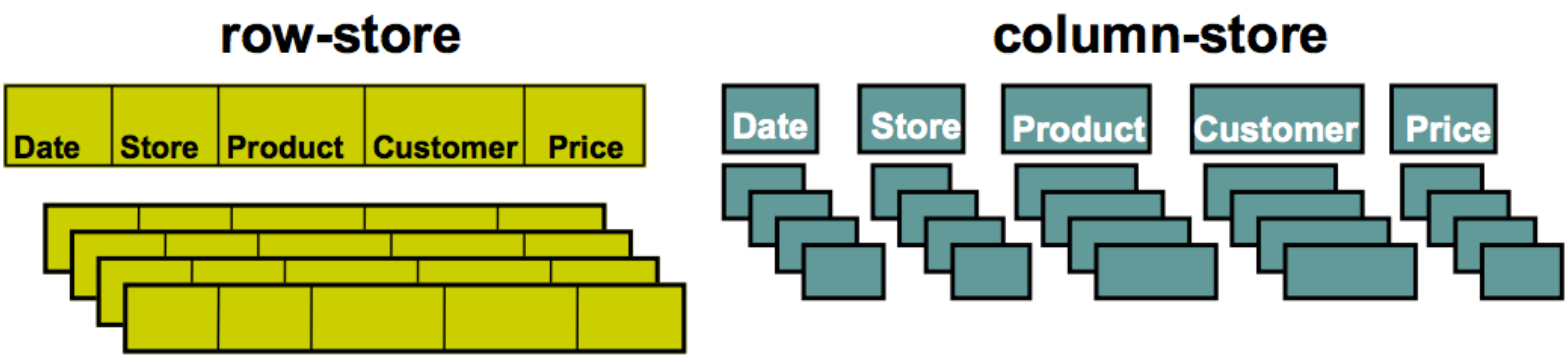
Why DuckDB?

- **Compressibility** is another advantage of column-storage
- Individual columns often have similar values, e.g. dates are usually increasing
- Save ~**2-10X** on storage (depending on compression algorithms used and data)



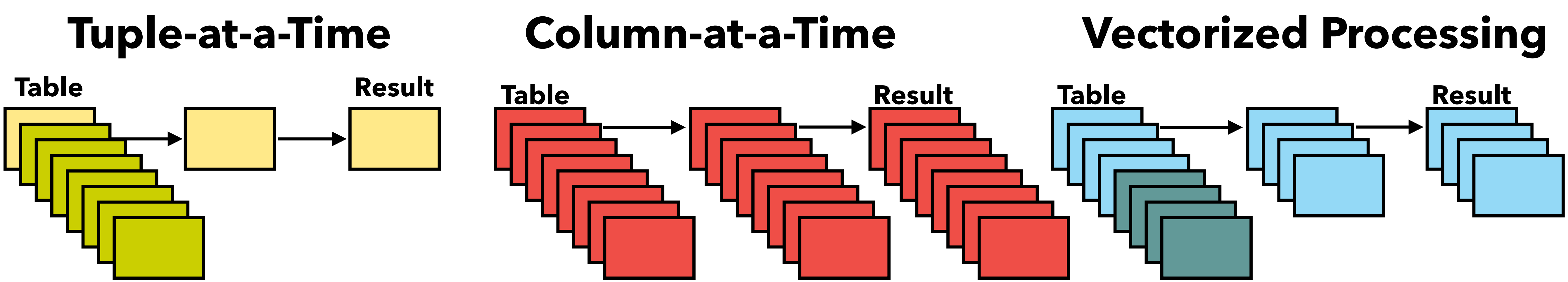
Why DuckDB?

- **Example:** Suppose we have a 1TB table with 100 columns. 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 (5GB) from disk ≅ 50 seconds



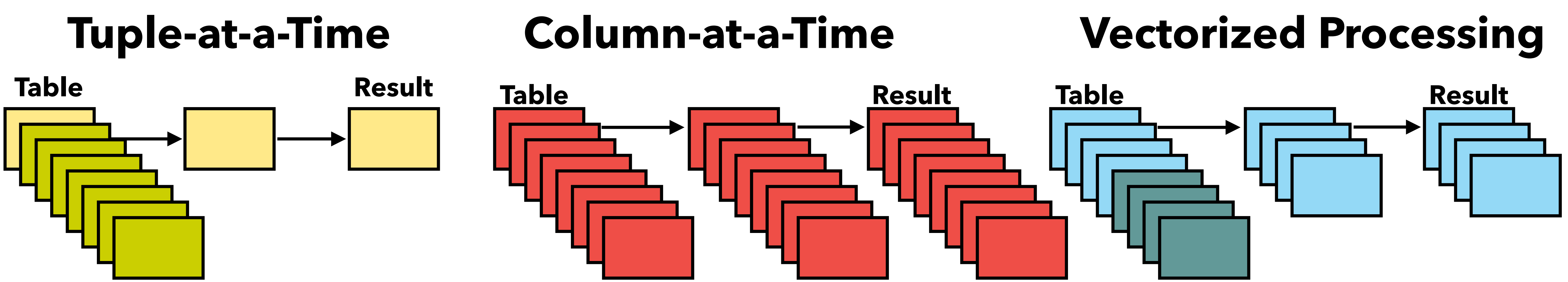
Why DuckDB?

- **Query Execution**
- **SQLite** use tuple-at-a-time processing
  - Process **one row** at a time
- **NumPy/R** use column-at-a-time processing
  - Process entire columns at once
- **DuckDB** uses **vectorized** processing
  - Process **batches** of columns at once



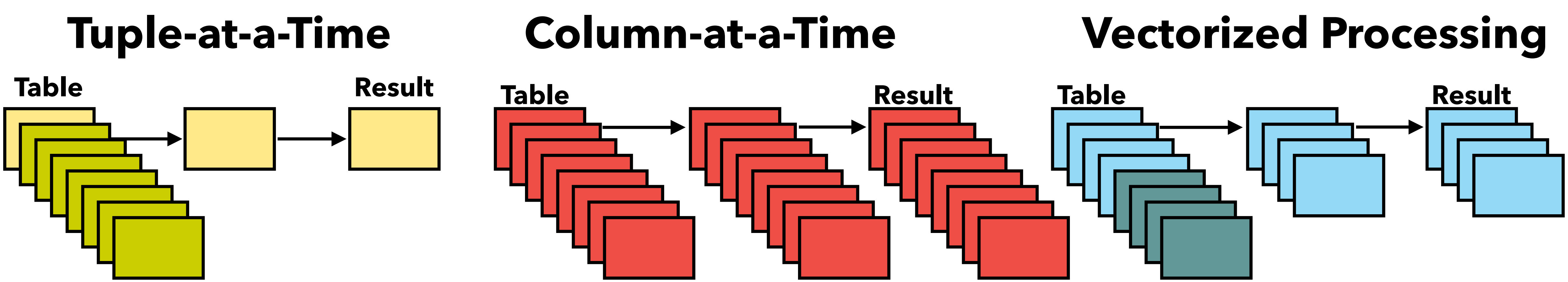
Why DuckDB?

- **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!**



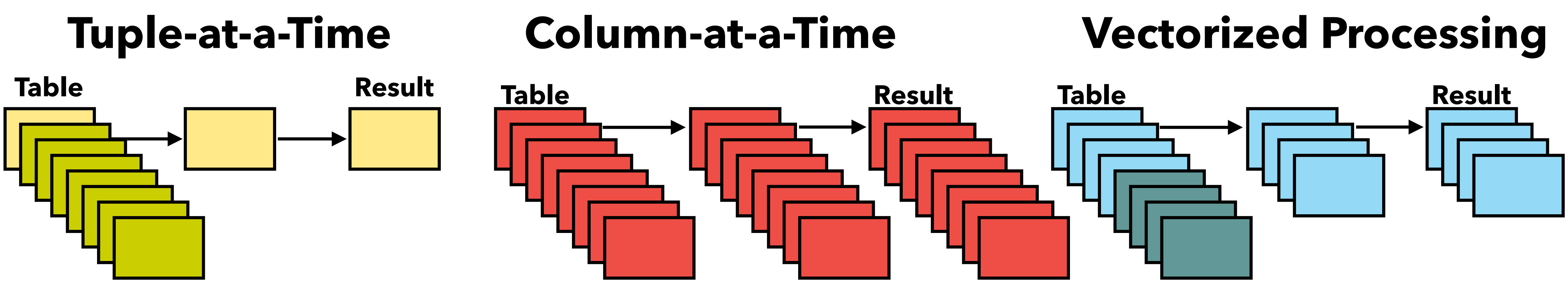
Why DuckDB?

- **Column-at-a-Time (NumPy/R)**
  - Better CPU utilization, allows for SIMD
  - Materialize **large intermediates** in memory!
- Intermediates can be gigabytes each…
- **Problematic** when data sizes are large



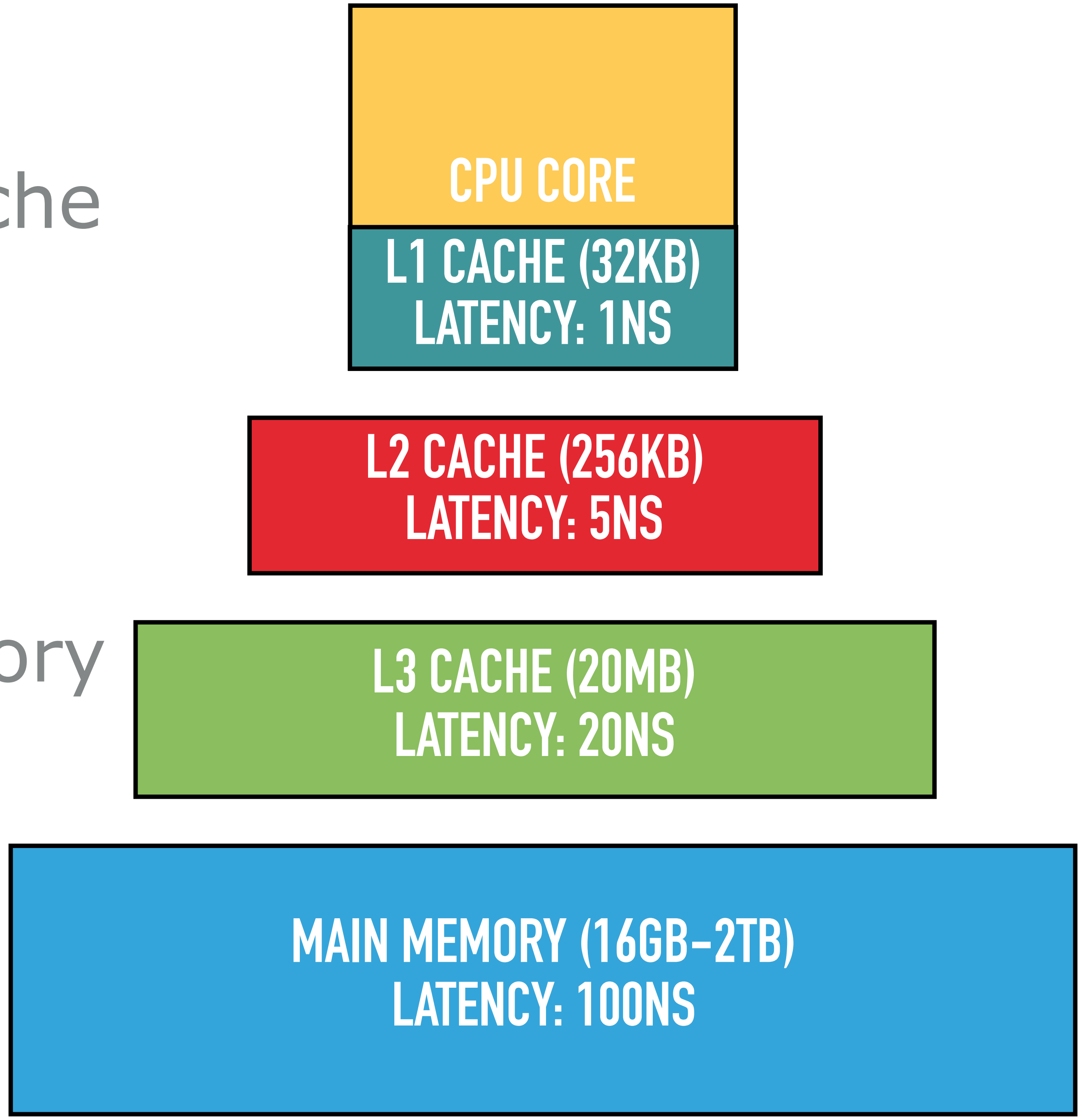
Why DuckDB?

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



Why DuckDB?

- **Vectorized Processing**
- Intermediates fit in L3 cache
- **Column-at-a-Time**
- Intermediates go to memory



DuckDB

- DuckDB is free and open-source
- Currently in pre-release (v0.1)
    - Storage is on beta version.
    - Might have bugs
    - Execution engine is solid and well tested!
- We have a website: www.duckdb.org
- Source Code: https://github.com/cwida/duckdb
- Feel free to try it
- ***And send us a bug report if anything breaks!***

Hands-on

Hands-on

- Goal: See in practice the differences of:
  - Pandas;
  - SQLite;
  - DuckDB.
- Tasks (Benchmark):
  - Loading Data;
  - Queries (e.g., aggregations, filters and joins);
  - Transactions.

Hands-on

- Download the .ipynb file from:
  - https://github.com/pdet/duckdb-tutorial/blob/master/DuckDB_Exercise1.ipynb

- Upload it to:
- https://colab.research.google.com/ as a Python 3 Notebook.