Mark Raasveldt & Pedro Holanda

DuckDB
an Embeddable Analytical RDBMS

Outline

- **Internals at a Glance**
- Query processing pipeline
- Query execution
- Hands-On

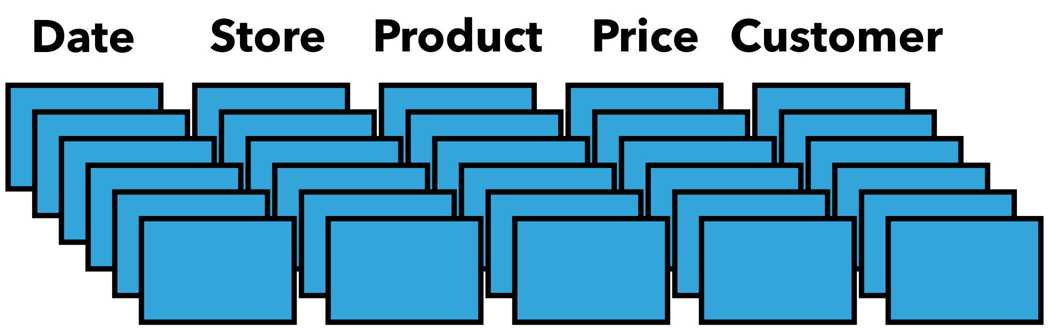
DuckDB

- Embedded analytical database
- Simple installation
  - `pip install duckdb`
- Fast and easy to use

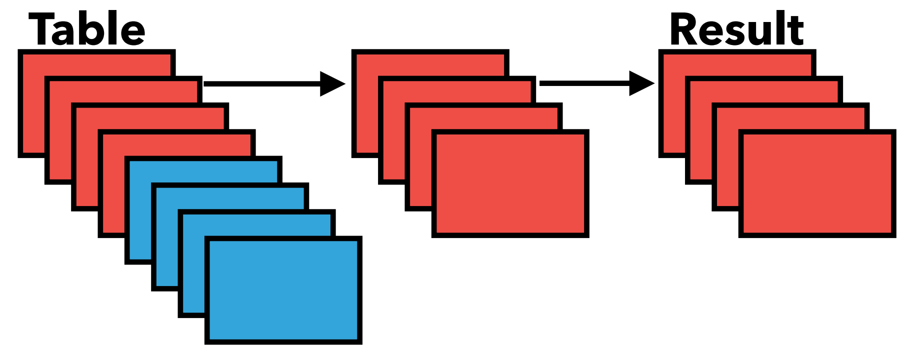
- https://www.duckdb.org

Internals at a Glance

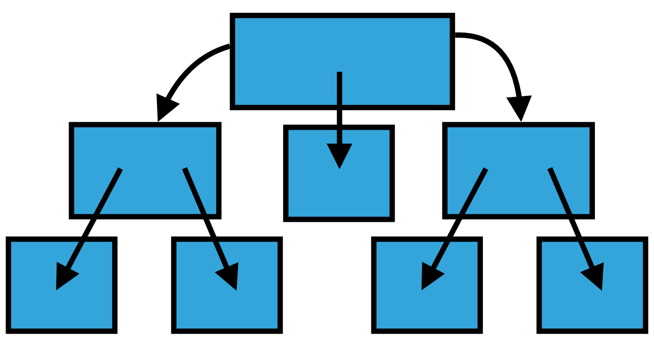
Column-Store



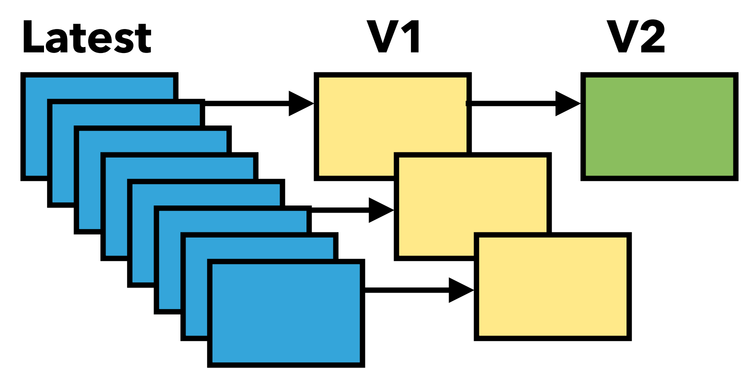
Vectorized Processing



ART Index



Multi-Version Concurrency Control



Single-File Storage

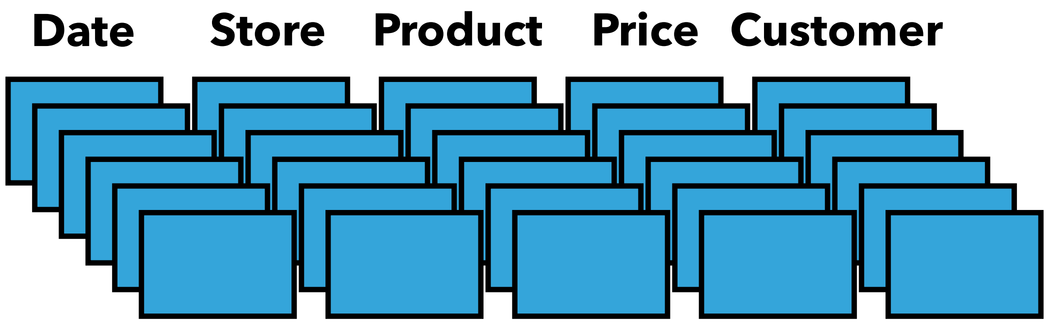
database.db

Parser

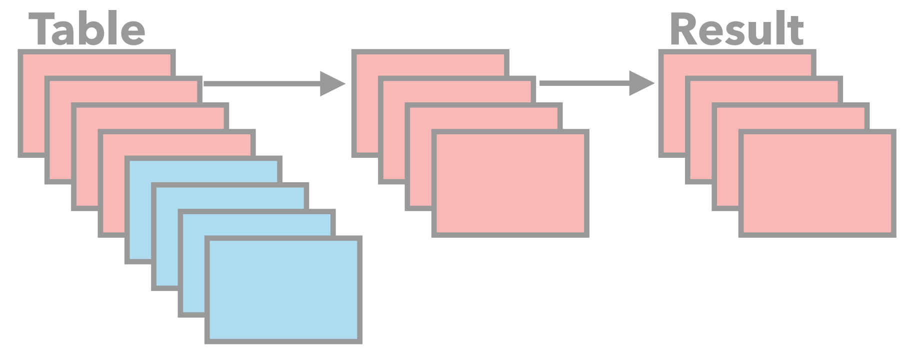
CWI

Internals at a Glance

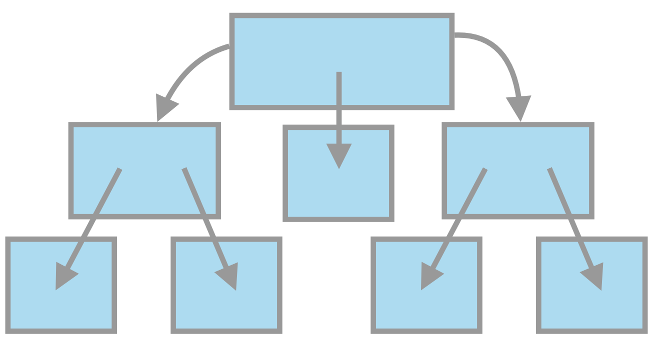
Column-Store



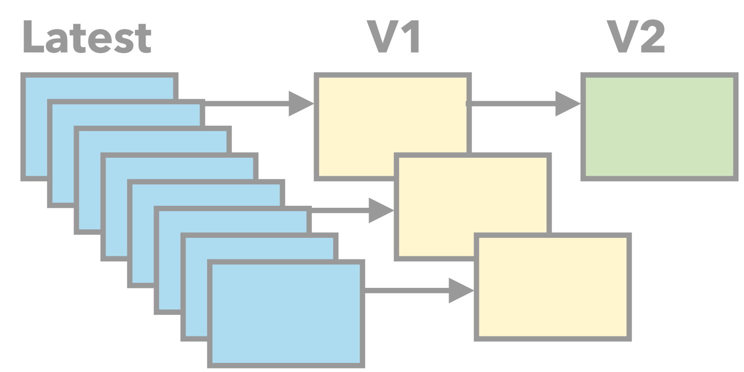
Vectorized Processing



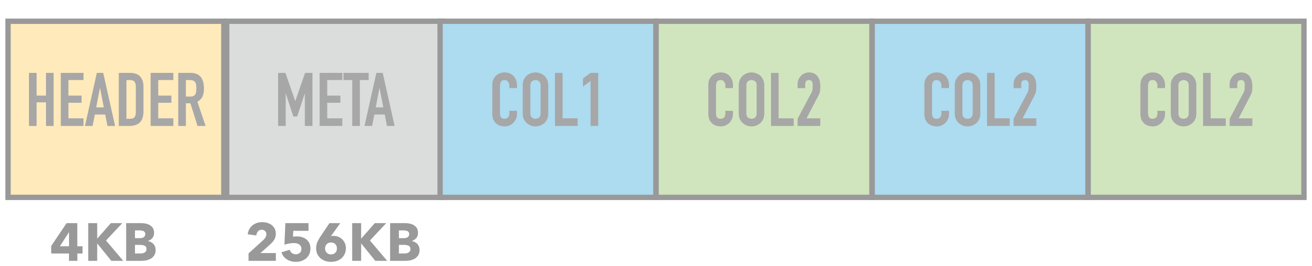
ART Index



Multi-Version Concurrency Control



Single-File Storage

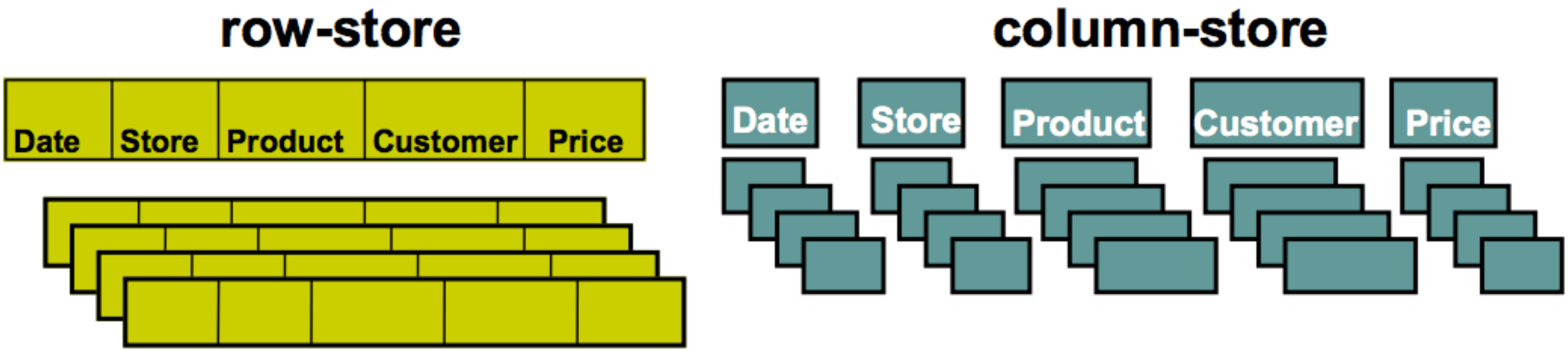


database.db

Parser

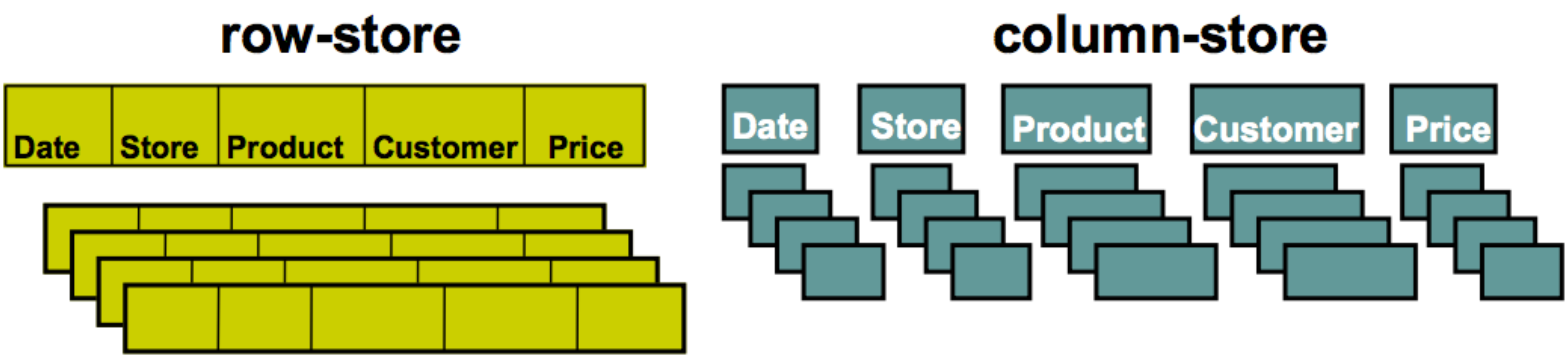
Why DuckDB?

- **Storage Model**
- Traditional RDBMS use a row-storage model
- DuckDB uses a columnar storage model



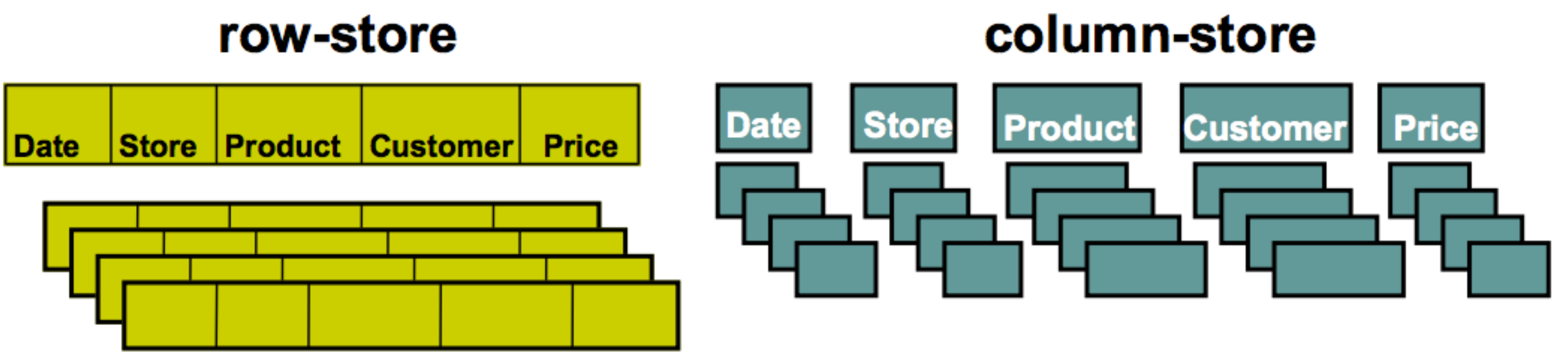
Why DuckDB?

- **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?



Why DuckDB?

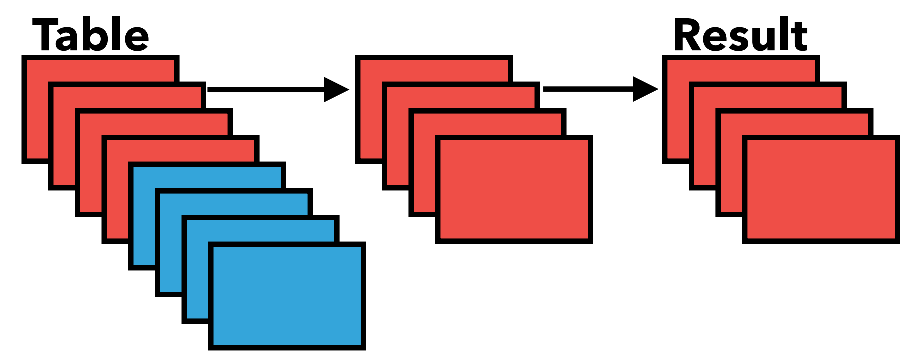
- **Column-Storage**:
  - We can fetch individual columns
  - Immense savings on disk IO/memory bw when only using few columns
- Queries that would take hours in a row-store can take seconds in a column-store



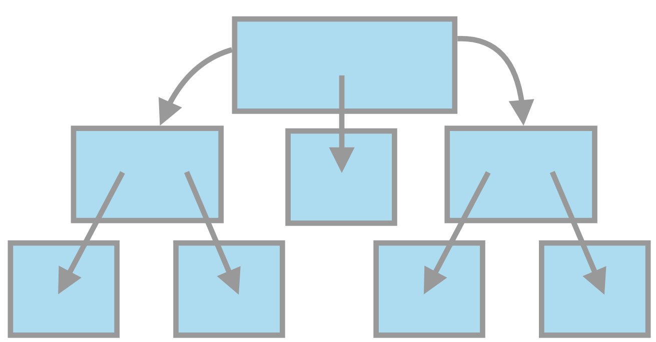
Internals at a Glance

Column-Store

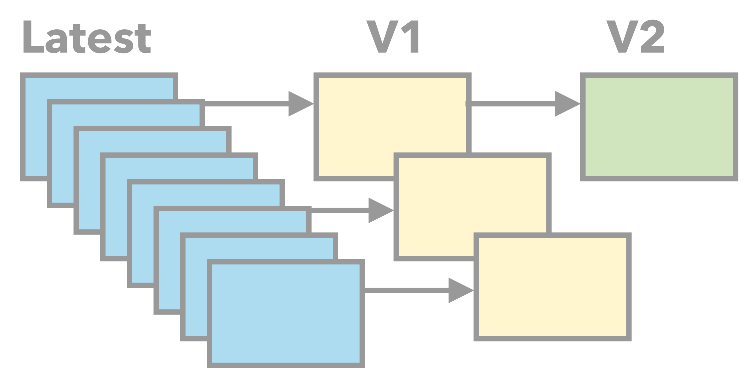
Vectorized Processing



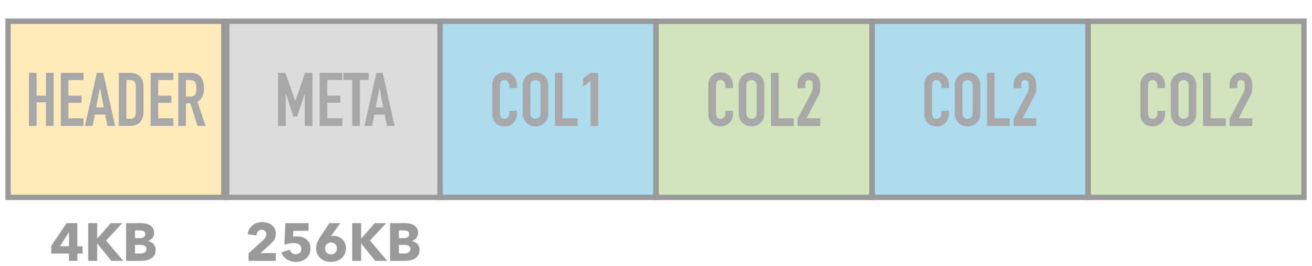
ART Index



Multi-Version Concurrency Control



Single-File Storage

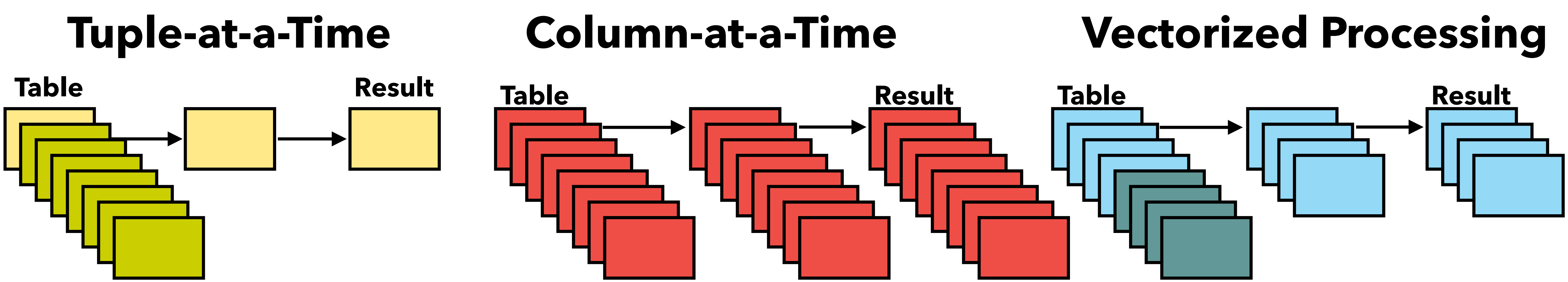


database.db

Parser

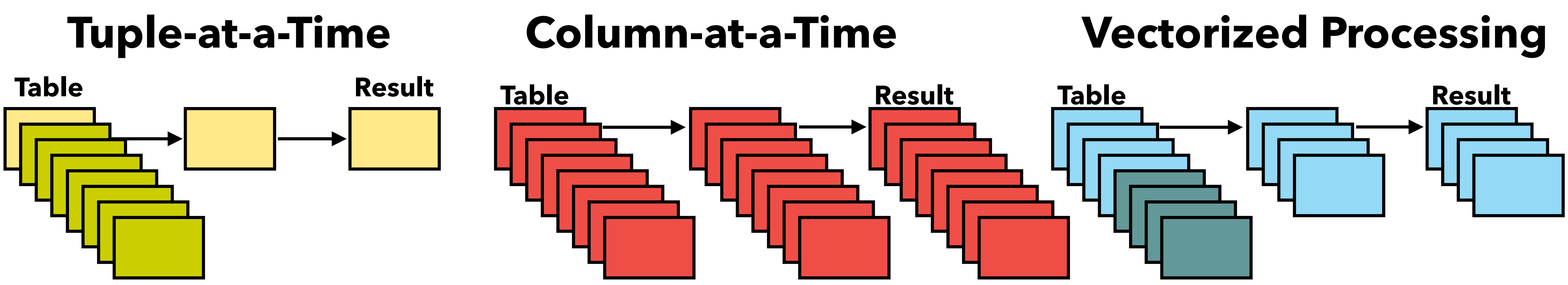
Why DuckDB?

- **Query Execution**
- **Traditional RDBMS** 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



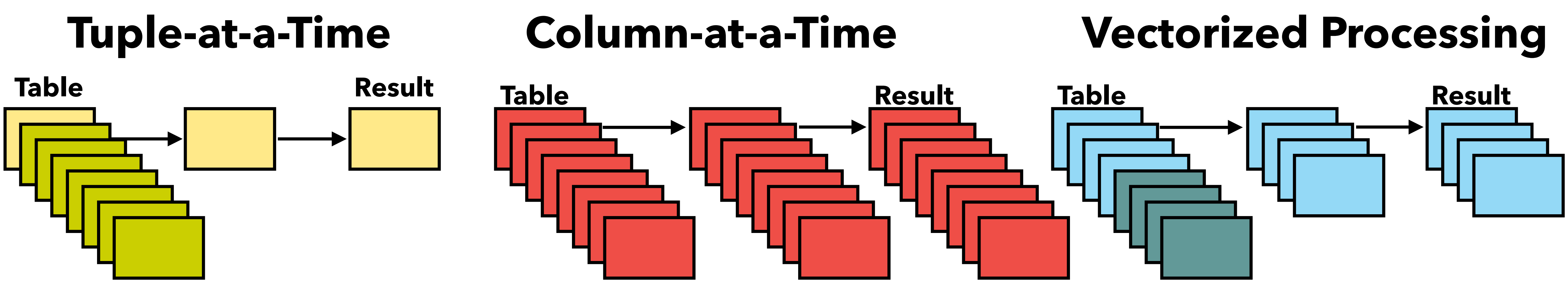
Why DuckDB?

- **Tuple-at-a-Time (Traditional RDBMS)**
  - 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!**



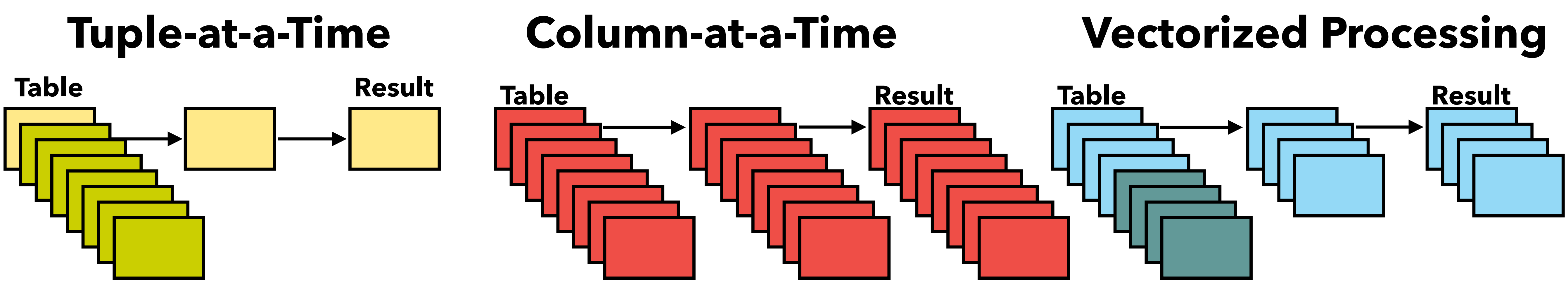
Why DuckDB?

- **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?

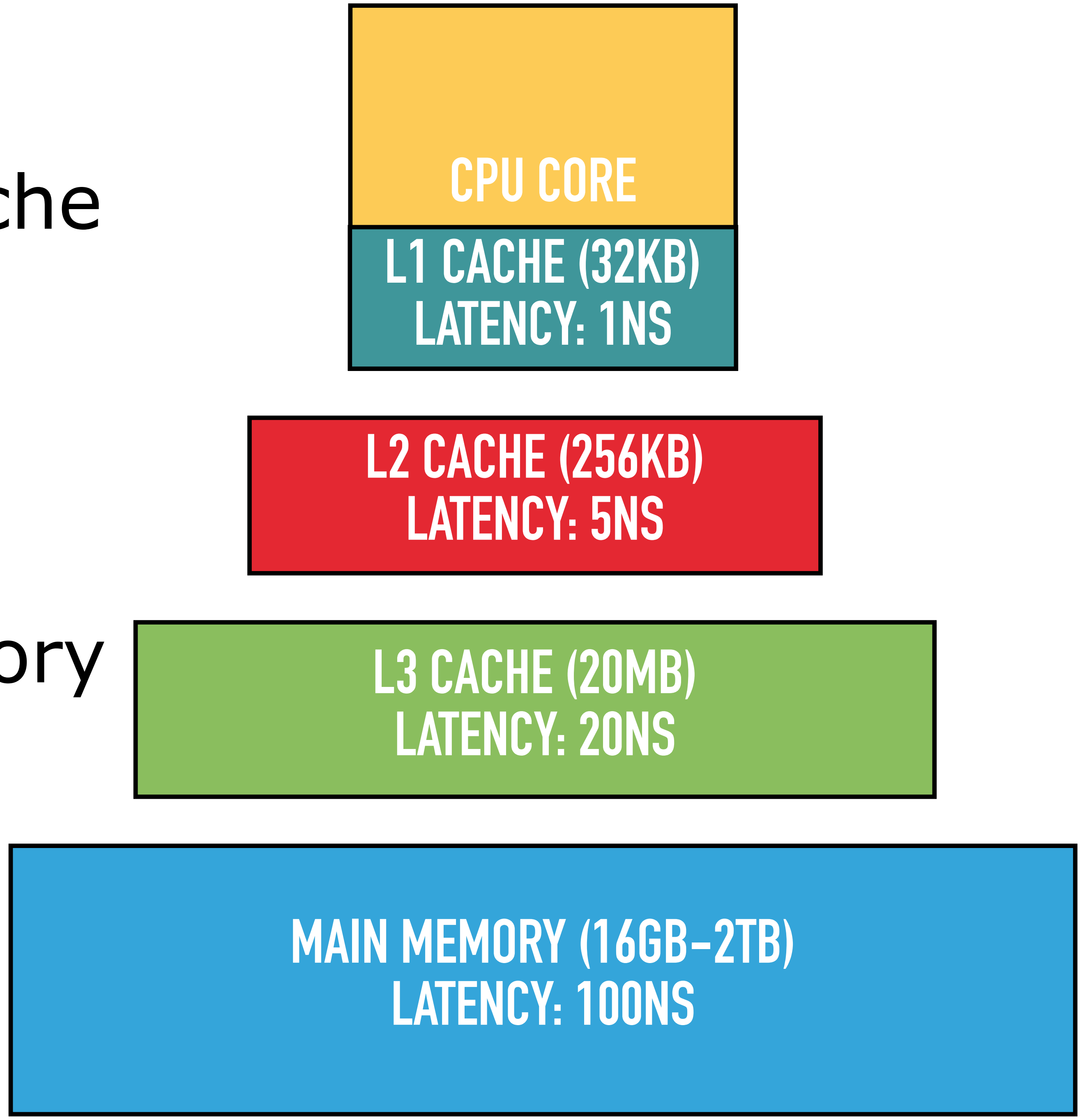
- **Vectorized Processing (DuckDB)**
  - Optimized for CPU Cache locality
  - SIMD instructions, Pipelining
  - **Small** intermediates **(fit in L3 cache)**



Why DuckDB?

- **Vectorized Processing**
- Intermediates fit in L3 cache

- **Column-at-a-Time**
- Intermediates go to memory



Outline

- Internals at a Glance
- **Query processing pipeline**
- Query execution
- Hands-On

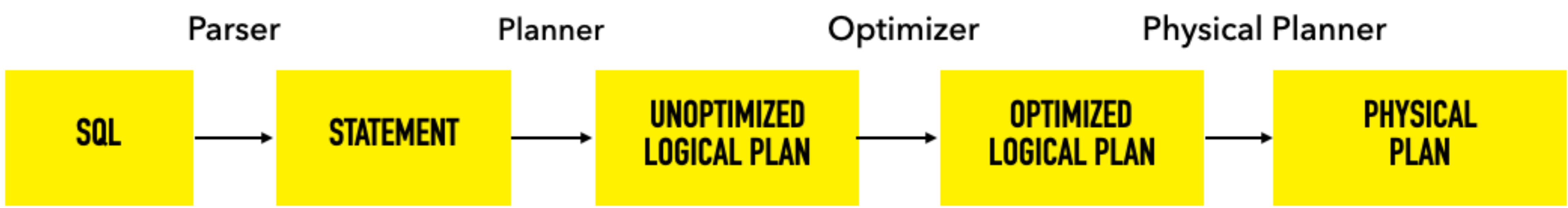
- **Life of a query**
- How does the system go from query to result?
- We will focus on the following query:

```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

- **Aggregate:** `COUNT(*)`
- **Implicit join:** `lineitem, orders` **on** `orderkey`
- **Filters:** `o_orderstatus='F'` **and** `l_tax>0.04`

DuckDB

DuckDB uses a typical pipeline for query processing

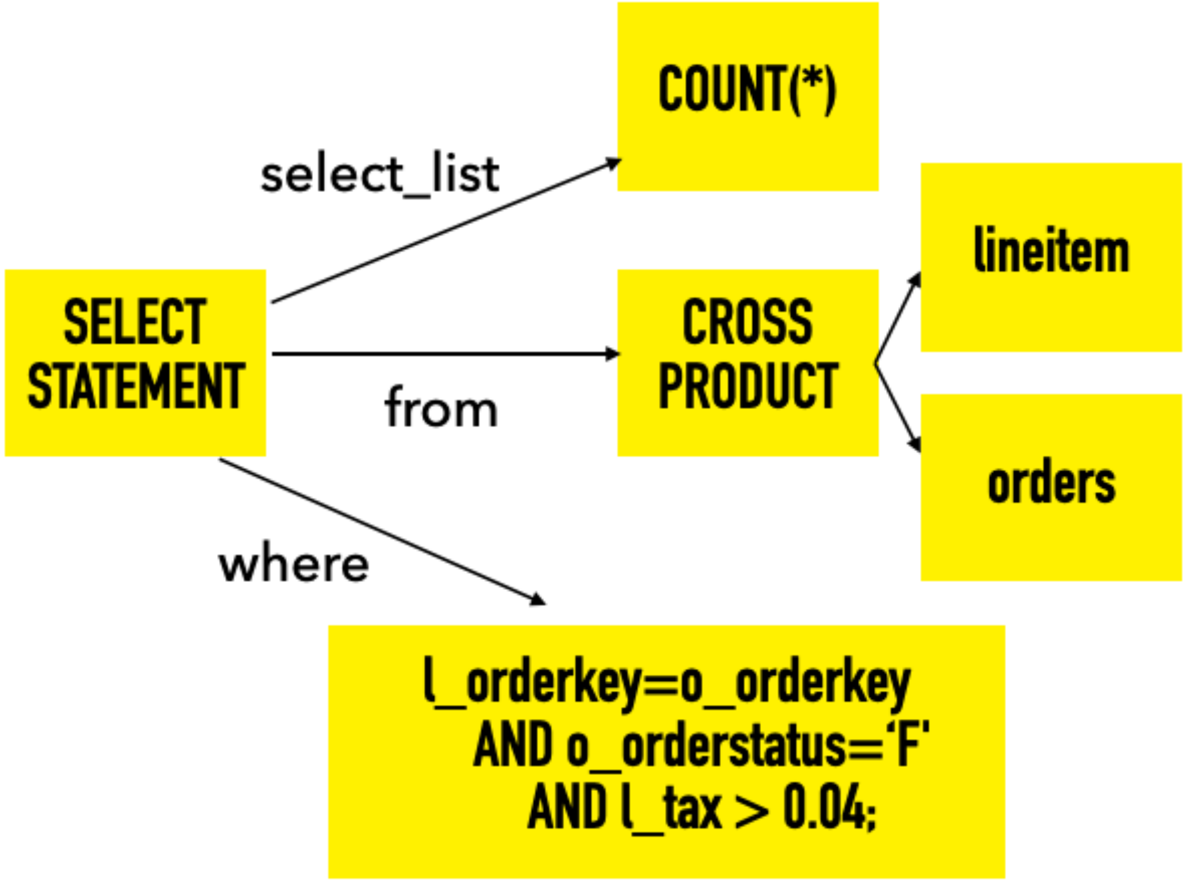


DuckDB

- Query is input into the system as a string
- The **lexer and parser** take the input string and convert it into a set **statements, parsed expressions** and **table references**
  - Note that this is not yet a query tree!
- We utilize the **Postgres parser + Transformer** for this part

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

- The result of the **parsing** stage is the following:




```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

DuckDB

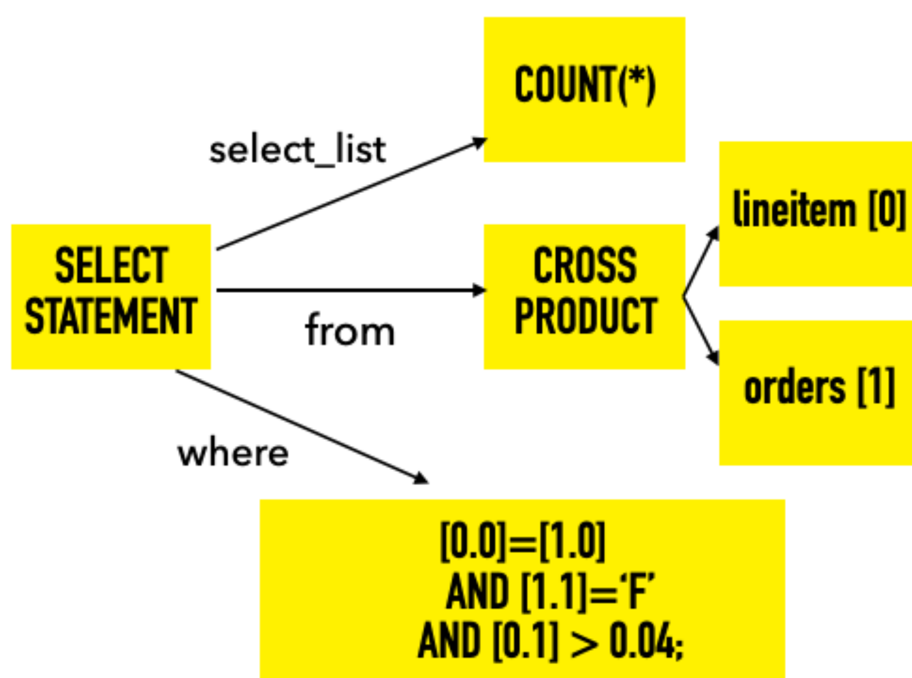
- Look up tables (**`lineitem`** and **`orders`**)
- Look up **columns** within these tables
- Assign **table** and **column** indexes

table index
lineitem: 0
orders: 1

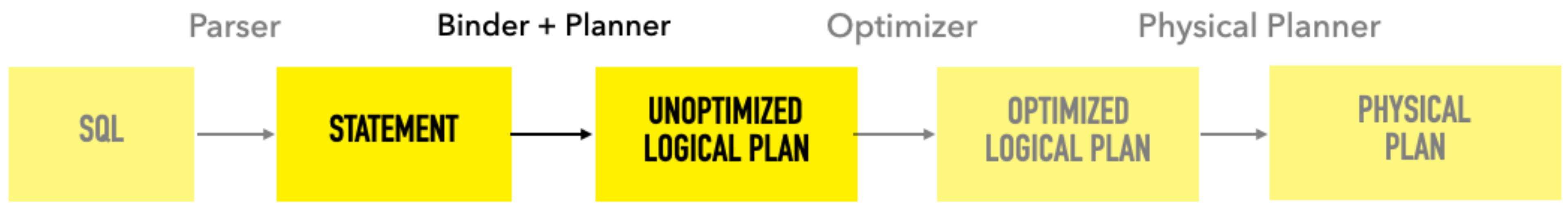
column index
l_orderkey: 0
l_tax: 1

column index
o_orderkey: 0
o_orderstatus: 1

```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```



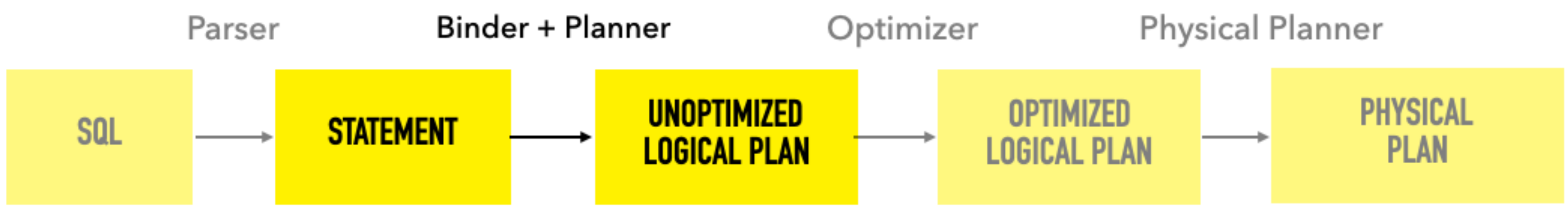
DuckDB




```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

- **Type resolution** happens in this stage
  - `l_orderkey : INTEGER`
  - `o_orderkey : INTEGER`
- `l_orderkey = o_orderkey : BOOLEAN`
- Types are propagated through every expression/operator
- Types are used for function resolution
  - Potential implicit casts are added

DuckDB



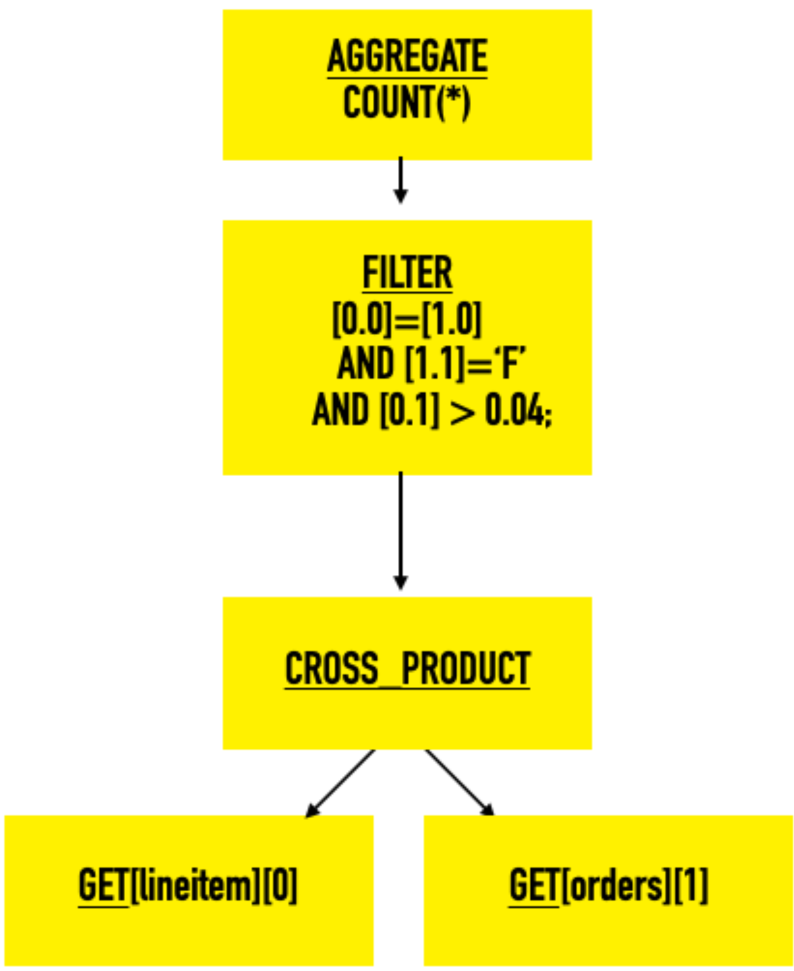
- **Planner:** Create **logical** query tree
- The logical query tree contains **logical operations**
  - Describes what to do, not how to do it
  - e.g. `Join`, not `HashJoin` or `MergeJoin`

```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

DuckDB

- Query tree starts with **FROM** clause
- Followed by **WHERE**
- Followed by **SELECT**

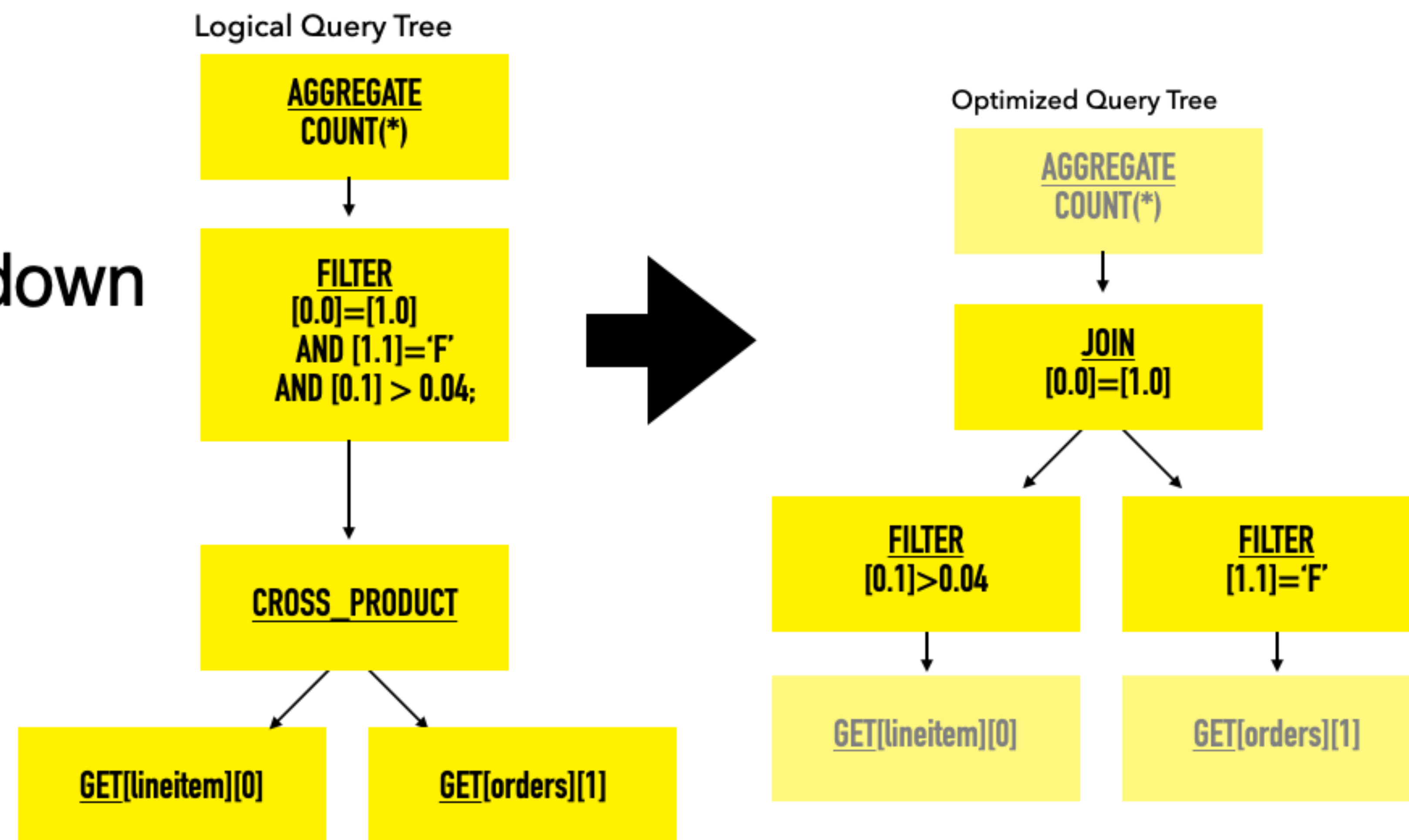
Logical Query Tree




```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

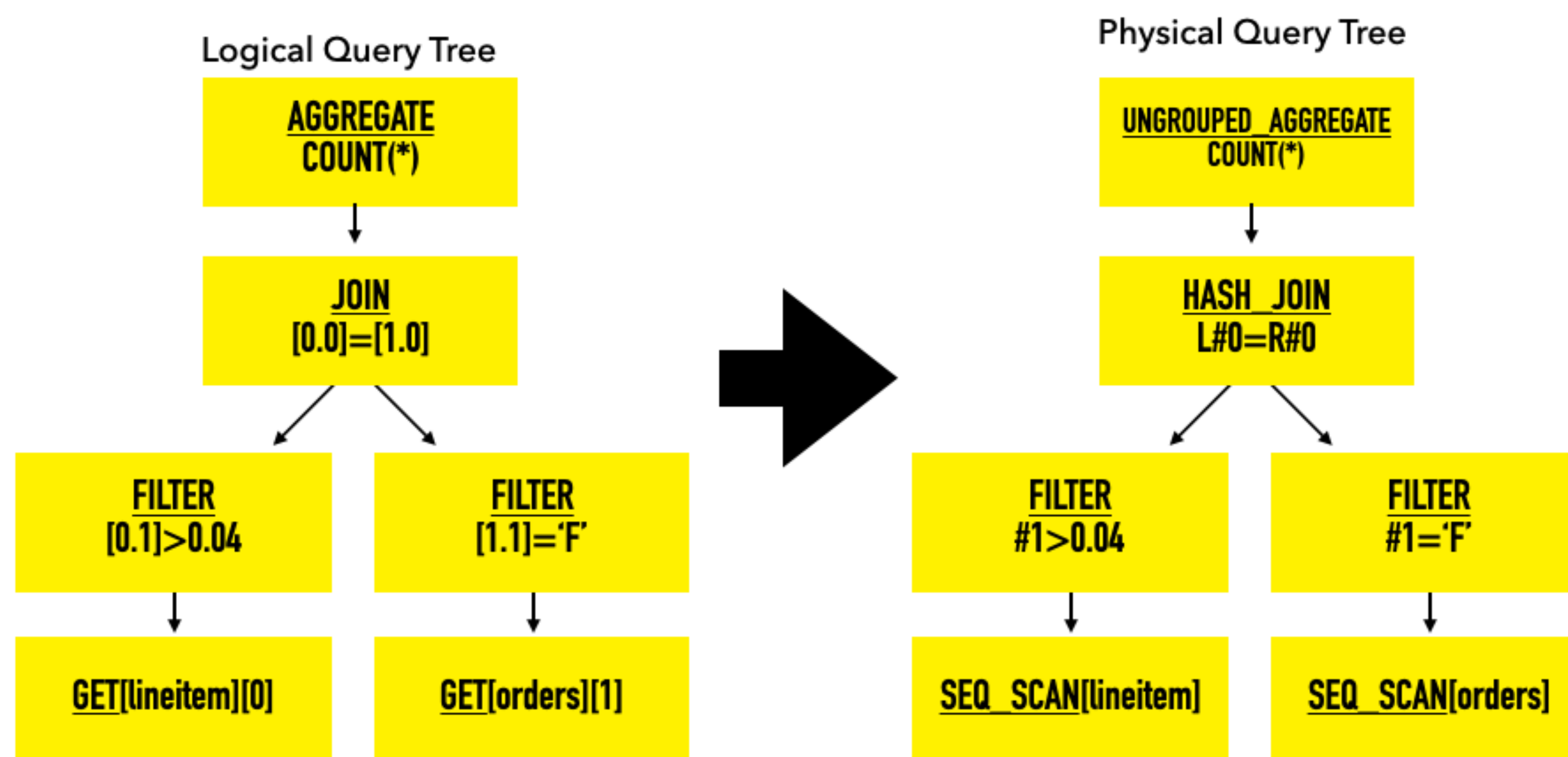
DuckDB

- **Optimizer**: rewrite **logical query tree**
- Mix of **rule-based** and **cost-based** optimizers
  - Join ordering/filter pushdown
  - Expression rewriter
  - Statistics propagation
  - …



DuckDB

- **Physical planner:** convert logical into physical plan
- Make decisions on implementations of operators
- Convert **column bindings** into **references**



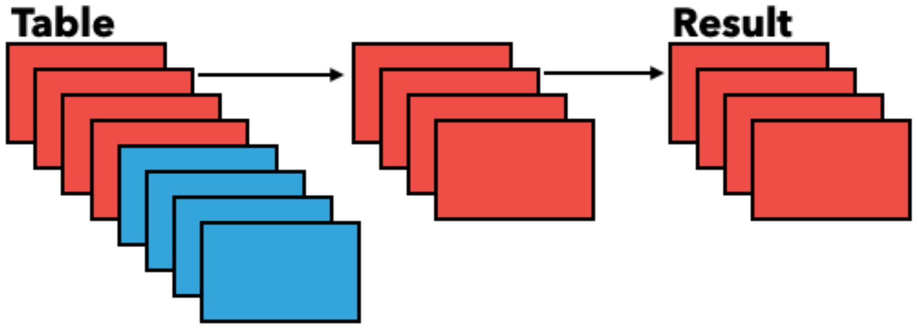
Outline

- Internals at a Glance
- Query processing pipeline
- **Query execution**
- Hands-On

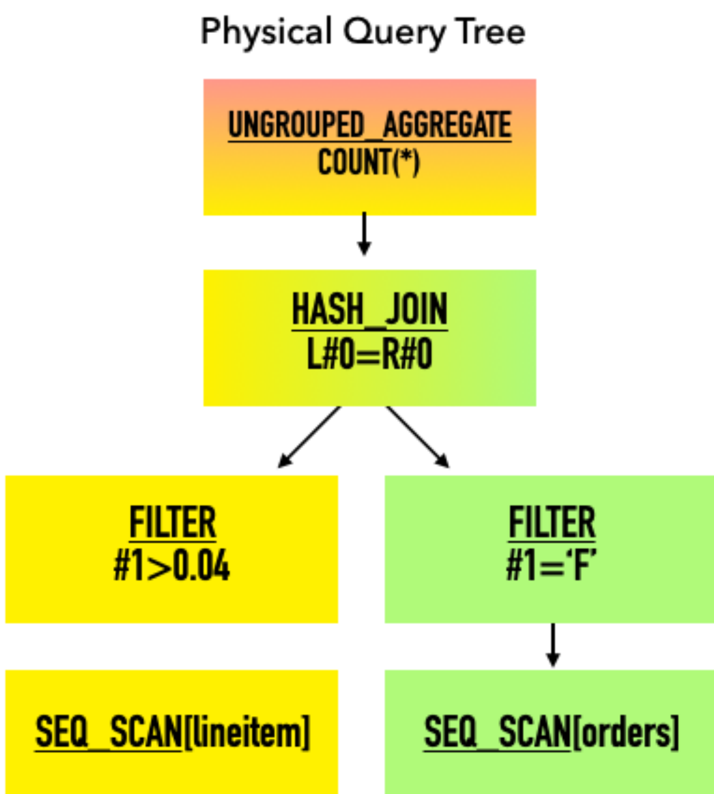
Query Execution

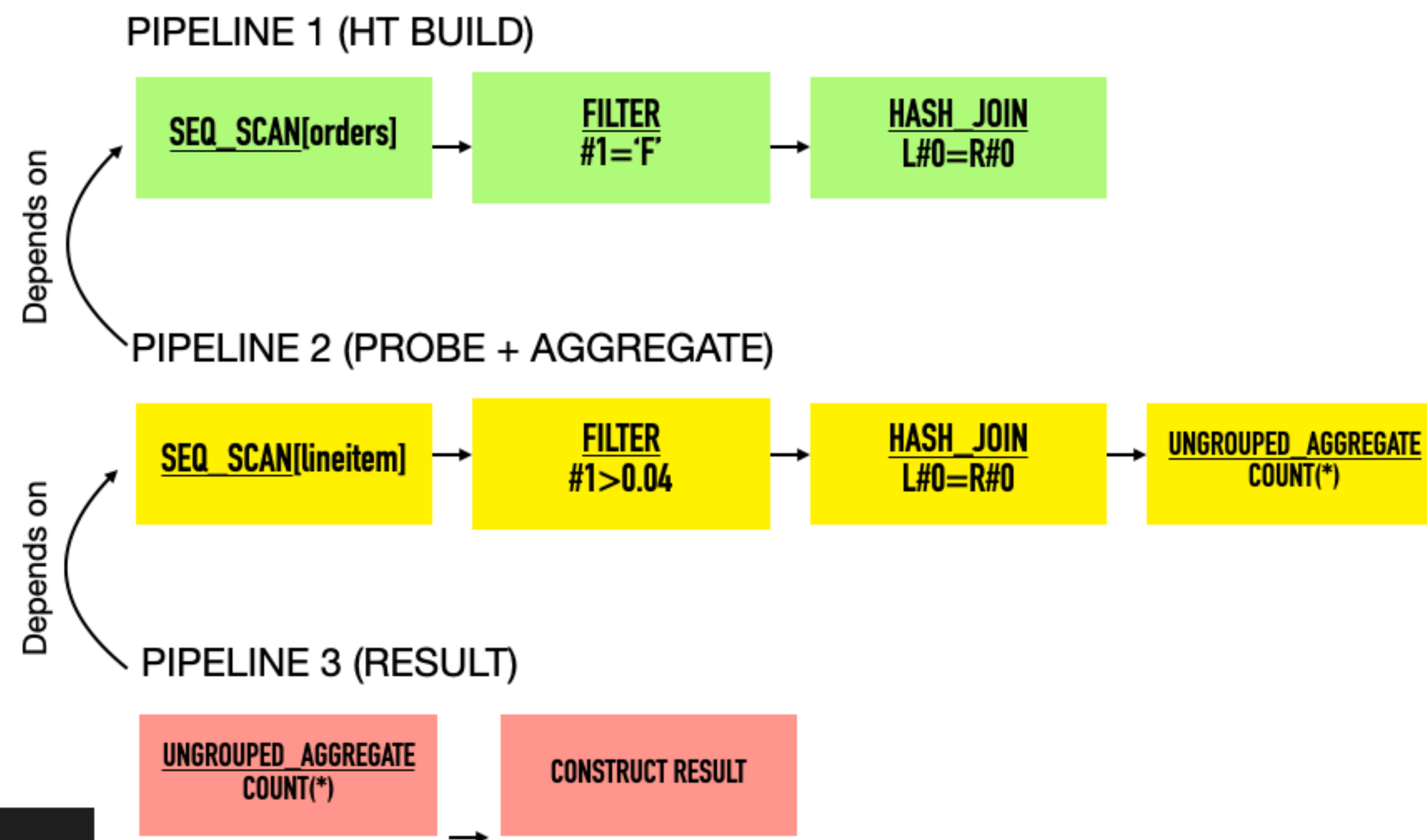
- DuckDB uses a **vectorized push-based model**
- Query tree is divided into **pipelines**
  - Pipelines have a **source, operators** and **sink**
- **Data flows from source to sink one chunk at a time**

Vectorized Processing



DuckDB



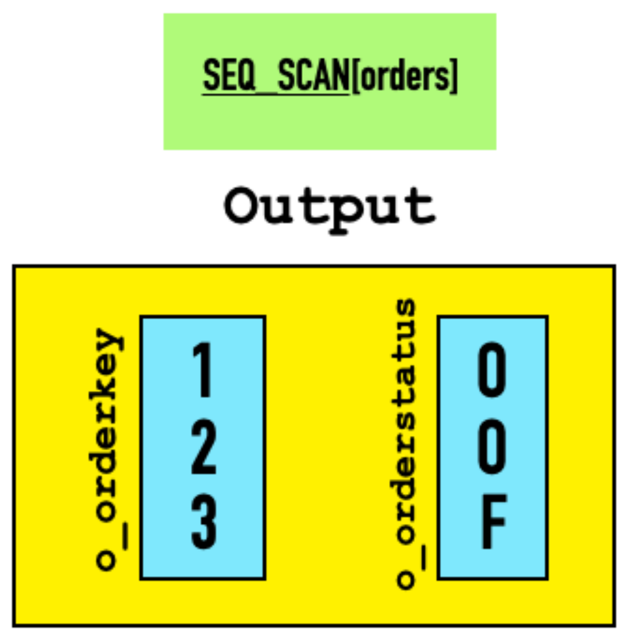



```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```

DuckDB

PIPELINE 1 (HT BUILD)

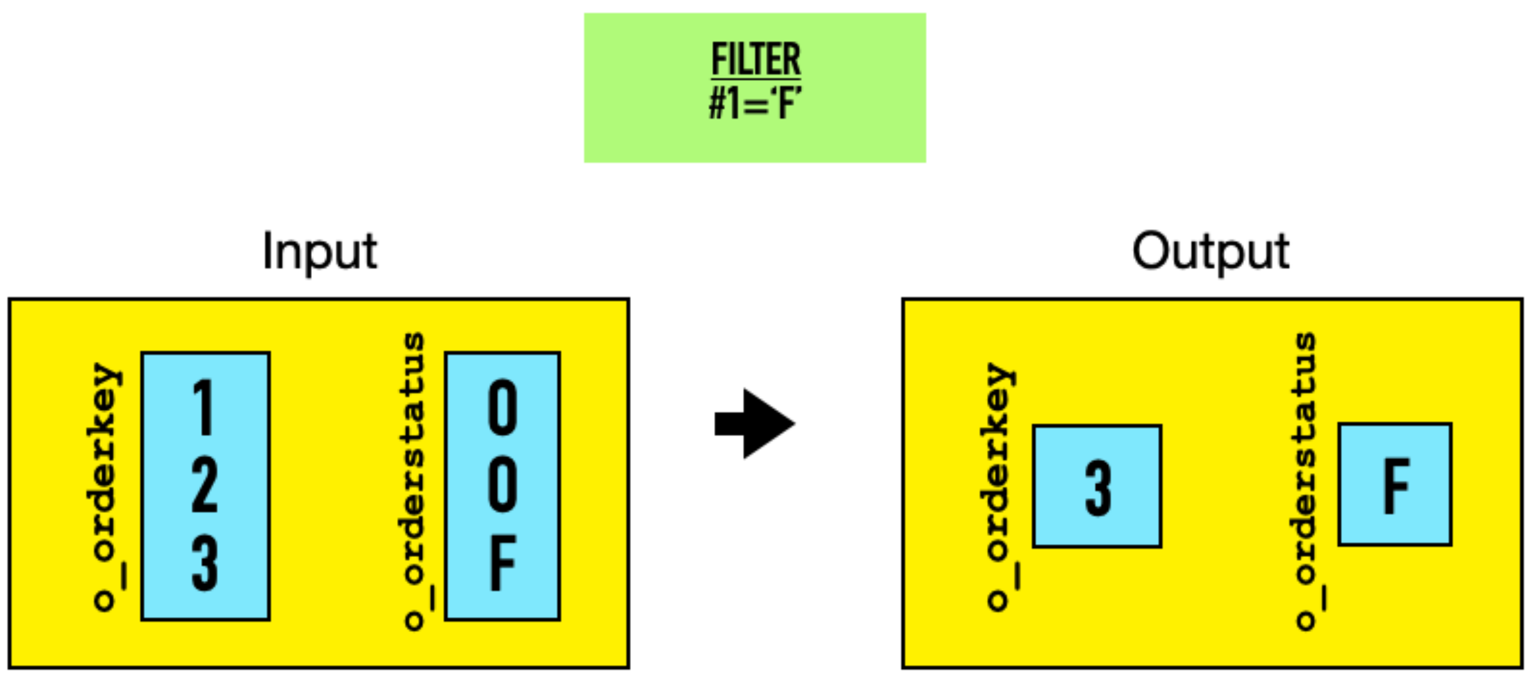

```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```



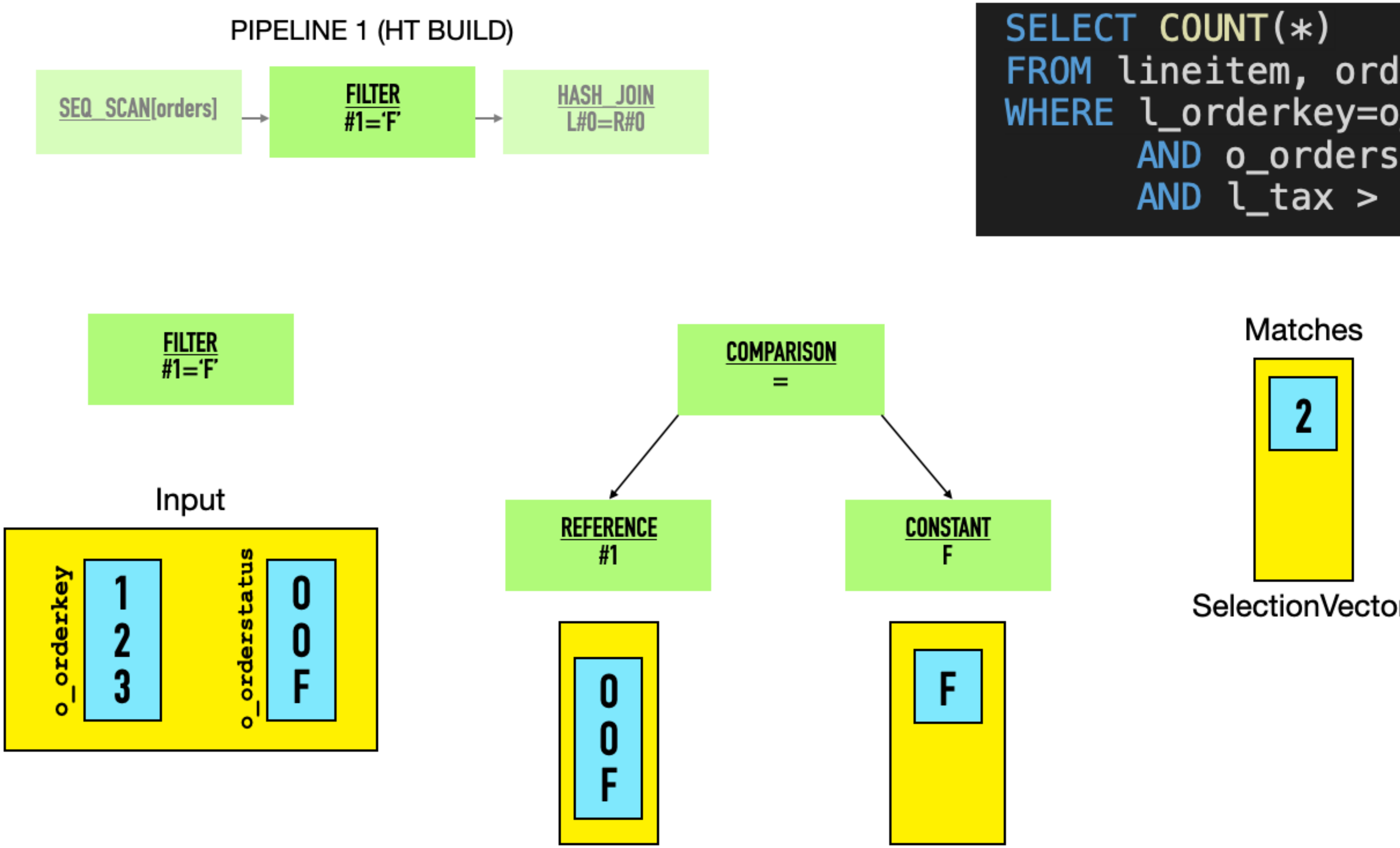
DuckDB

PIPELINE 1 (HT BUILD)

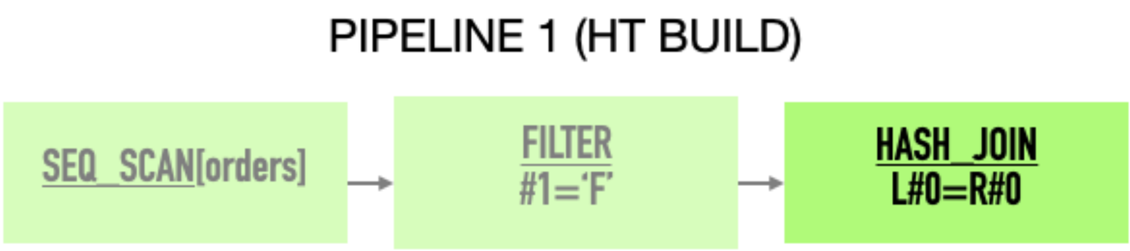

```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```



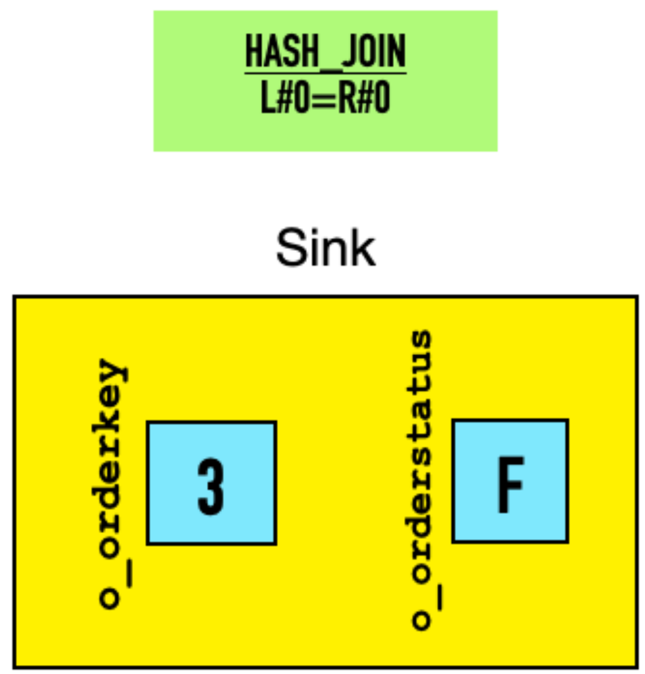
```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```



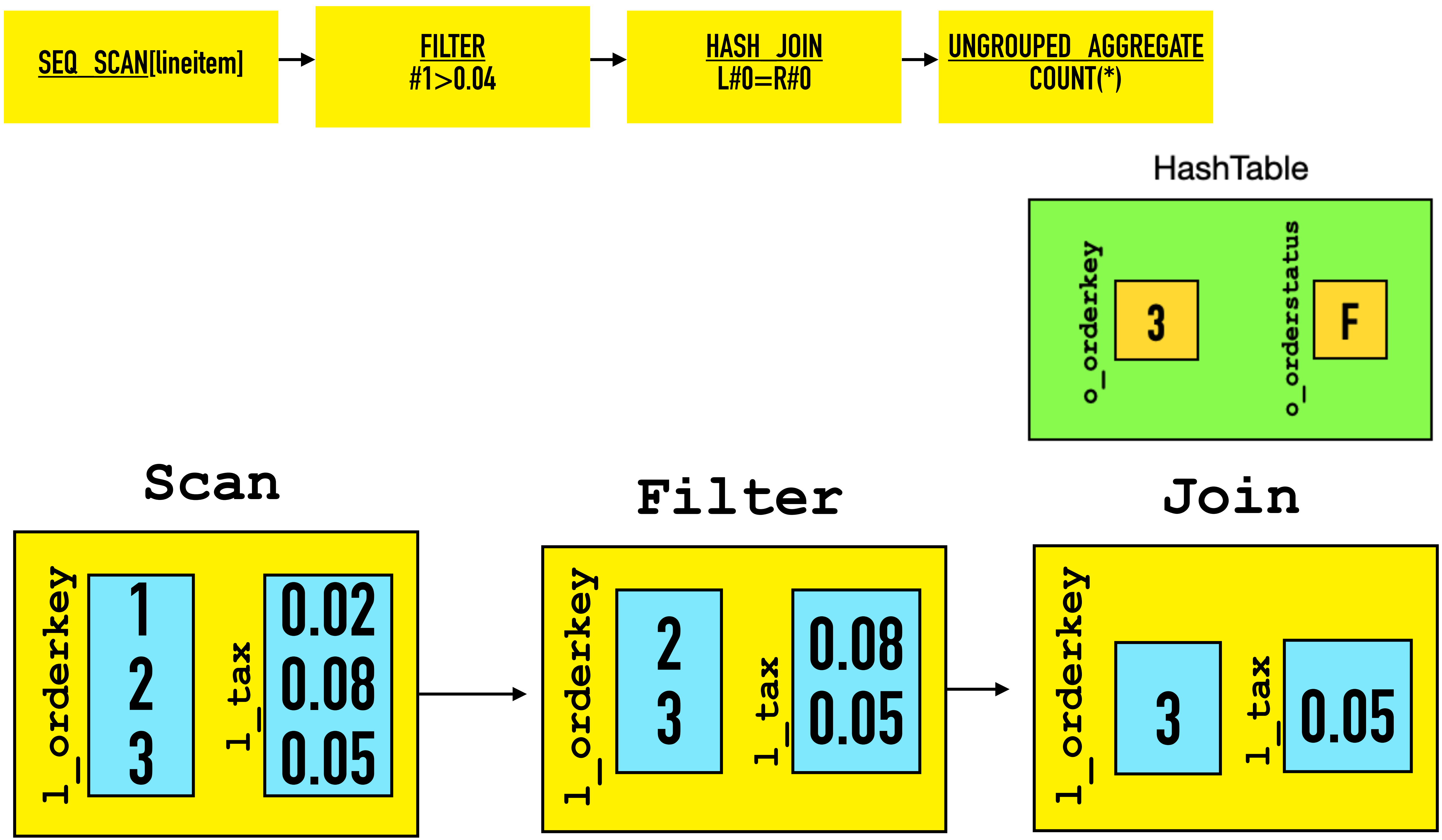
DuckDB




```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='F'
      AND l_tax > 0.04;
```



DuckDB



SEQ_SCAN[lineitem]
FILTER
#1>0.04
HASH_JOIN
L#0=R#0
UNGROUPED_AGGREGATE
COUNT(*)
HashTable
o_orderkey
3
o_orderstatus
F
Scan
l_orderkey
1
2
3
l_tax
0.02
0.08
0.05
Filter
l_orderkey
2
3
l_tax
0.08
0.05
Join
l_orderkey
3
l_tax
0.05

Outline

- Internals at a Glance
- Query processing pipeline
- Query execution
- **Hands-On**

Creating a Simple Function

- **Slides are online**
- https://github.com/pdet/duckdb-tutorial

- Feel free to ask any questions!