CWI

Mark Raasveldt & Pedro Holanda

DuckDB
an Embeddable Analytical RDBMS

Outline

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

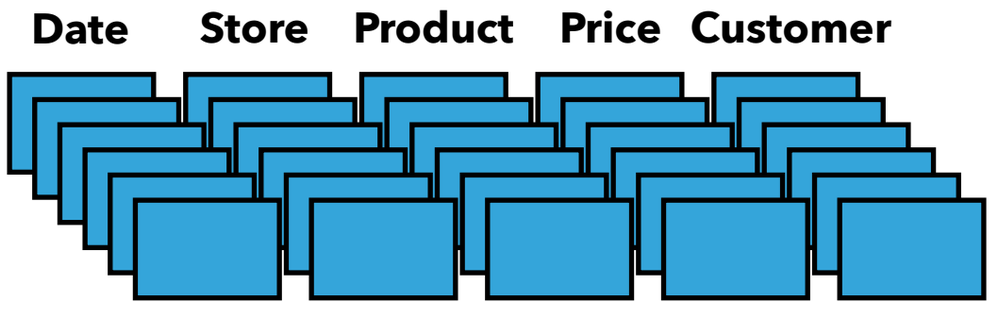
CWI

DuckDB

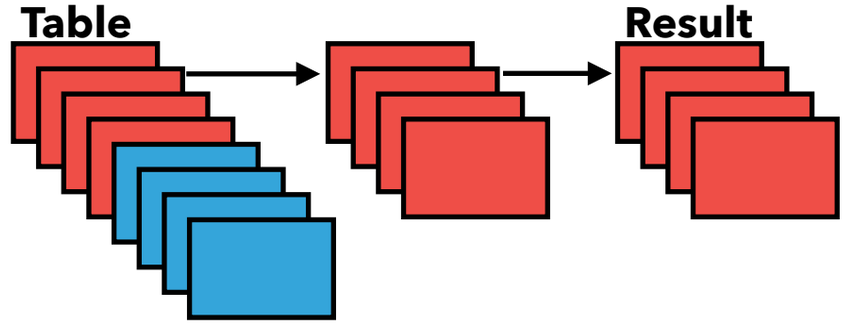
- Embedded analytical database
- Simple installation
  - `pip install duckdb`
- Fast and easy to use`
- https://www.duckdb.org

CWI Internals at a Glance

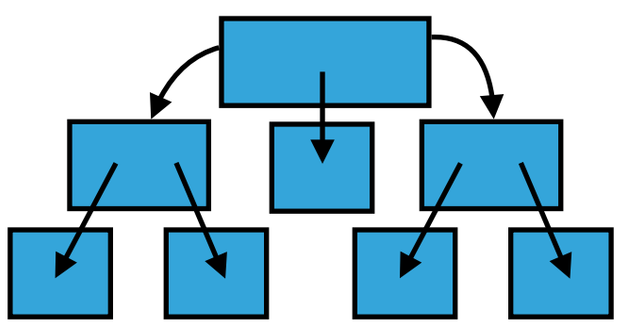
Column-Store



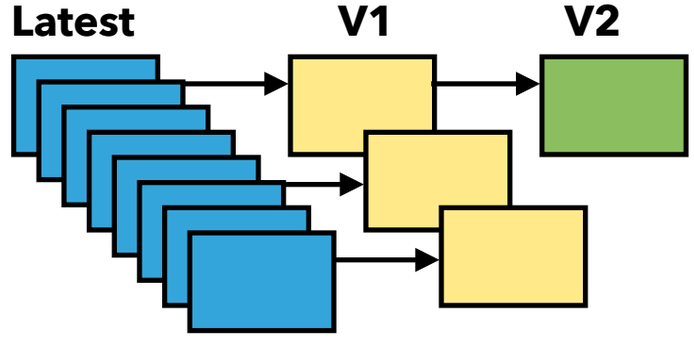
Vectorized Processing



ART Index

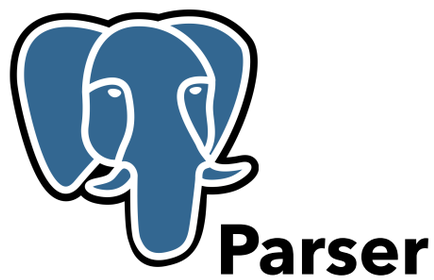


Multi-Version Concurrency Control



Single-File Storage

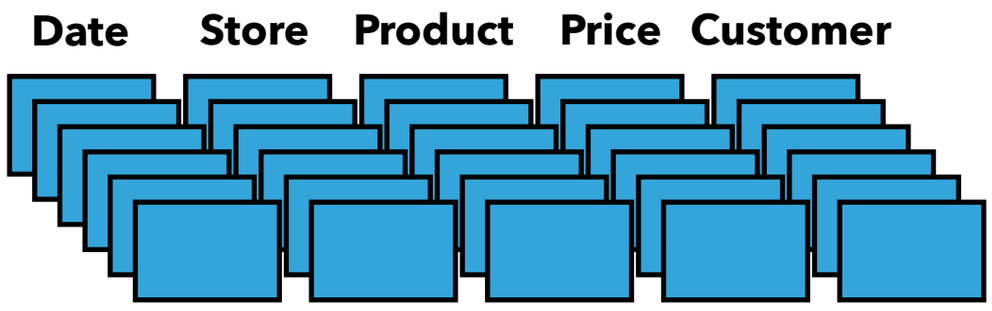
database.db



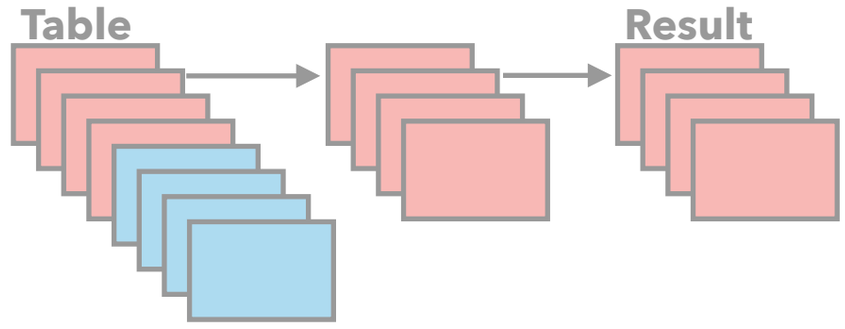
Parser

CWI Internals at a Glance

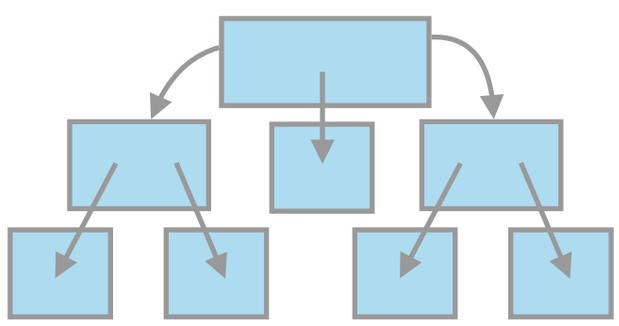
Column-Store



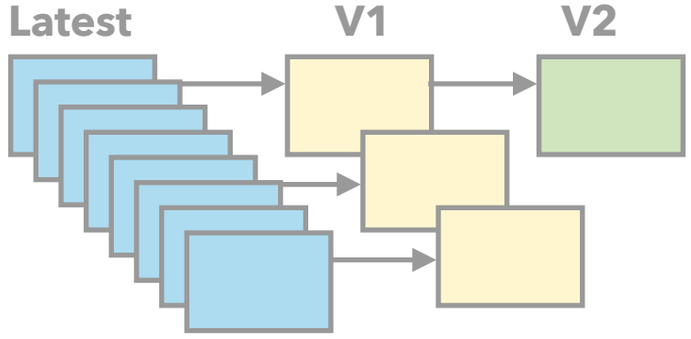
Vectorized Processing



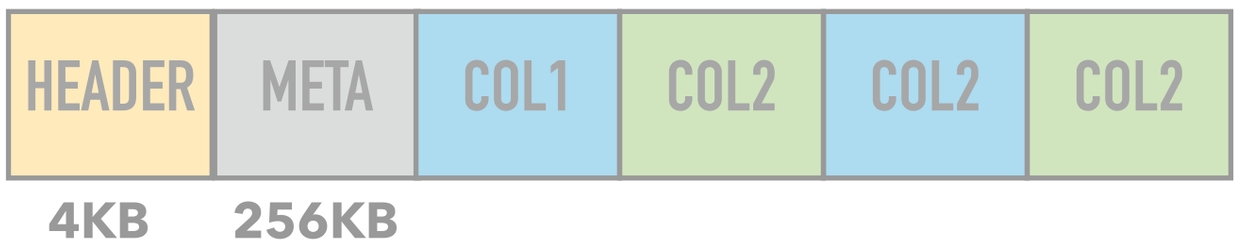
ART Index



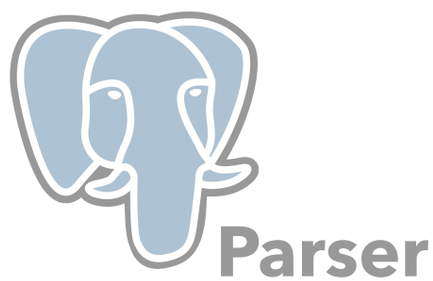
Multi-Version Concurrency Control



Single-File Storage



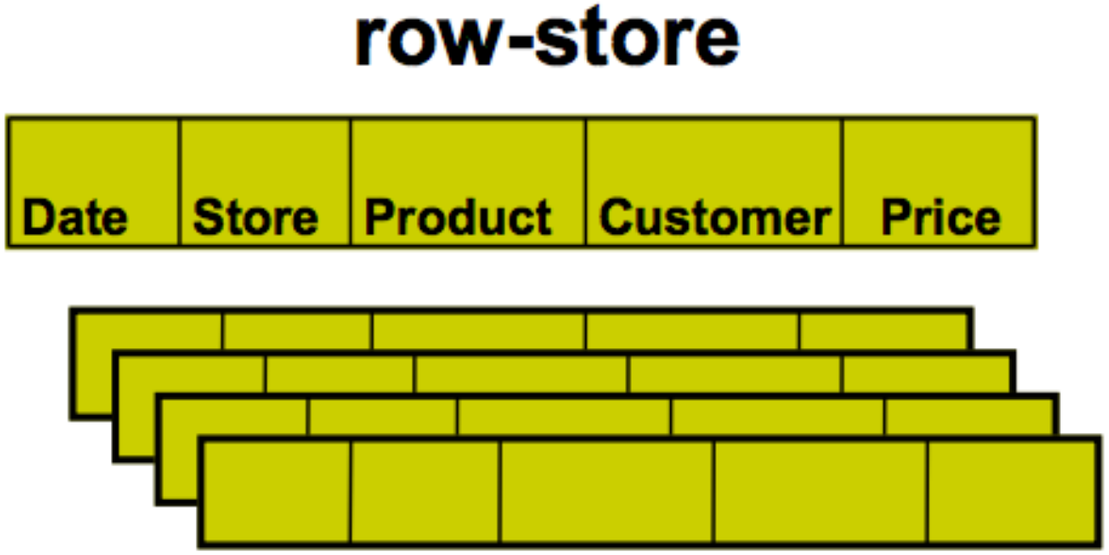
database.db

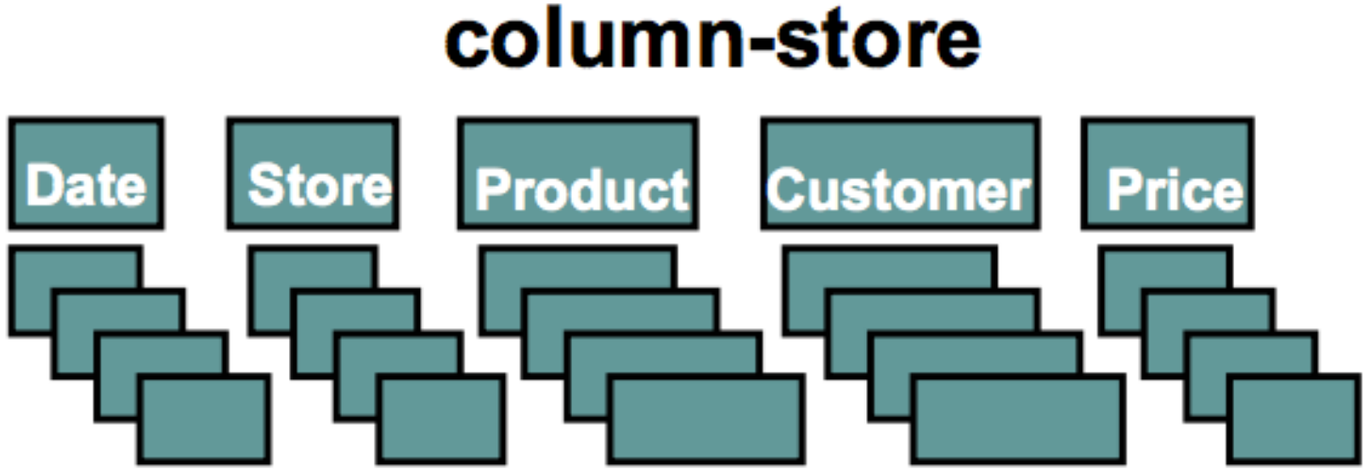


Parser

Why DuckDB?

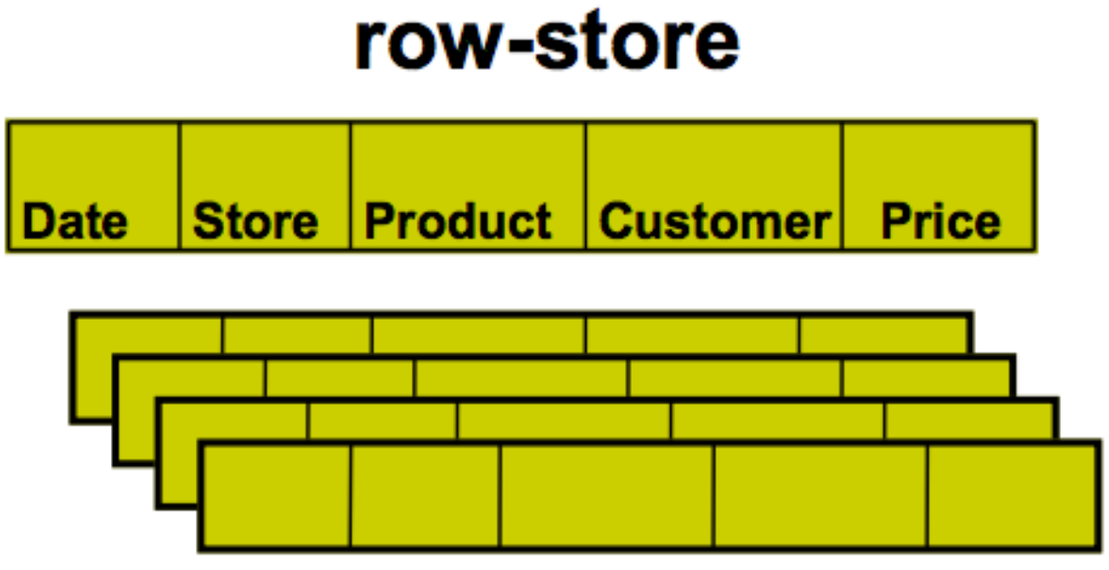
- **Storage Model**
- Traditional RDBMS use a row-storage model
- DuckDB uses a columnar storage model

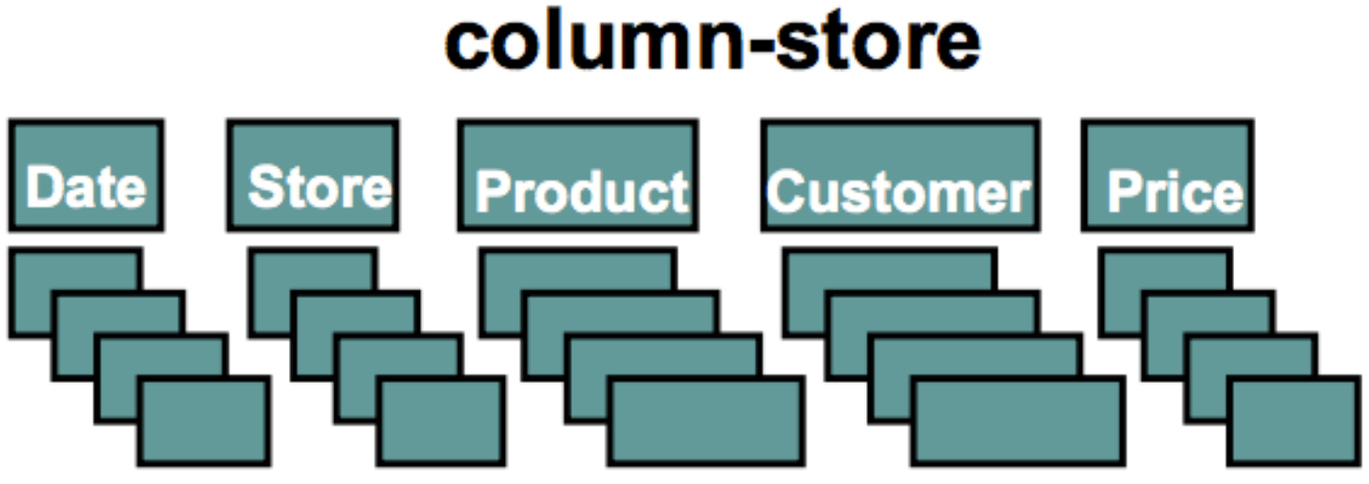




Why DuckDB?

- **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?

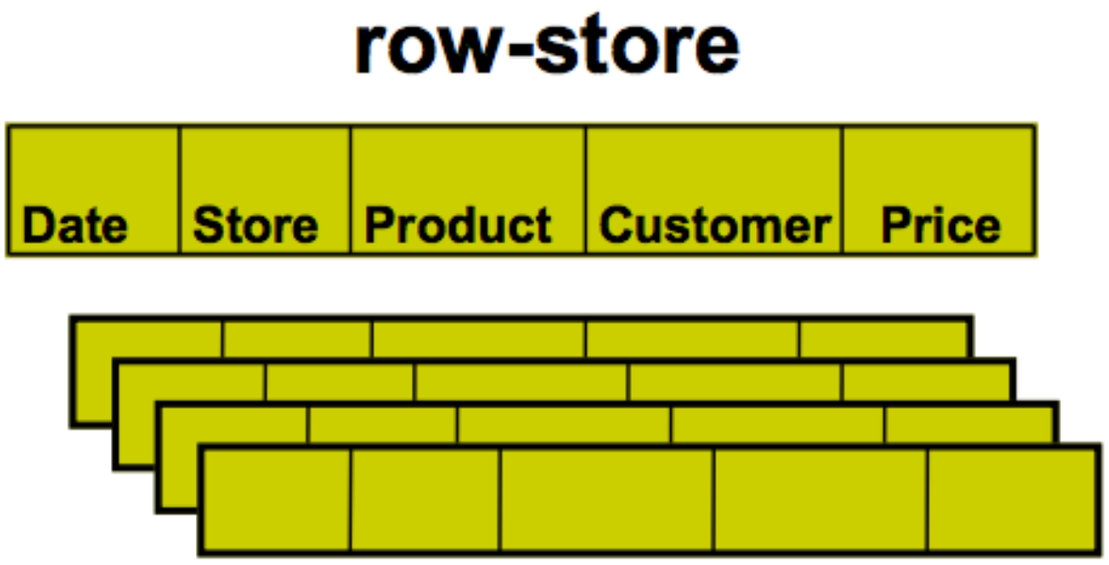


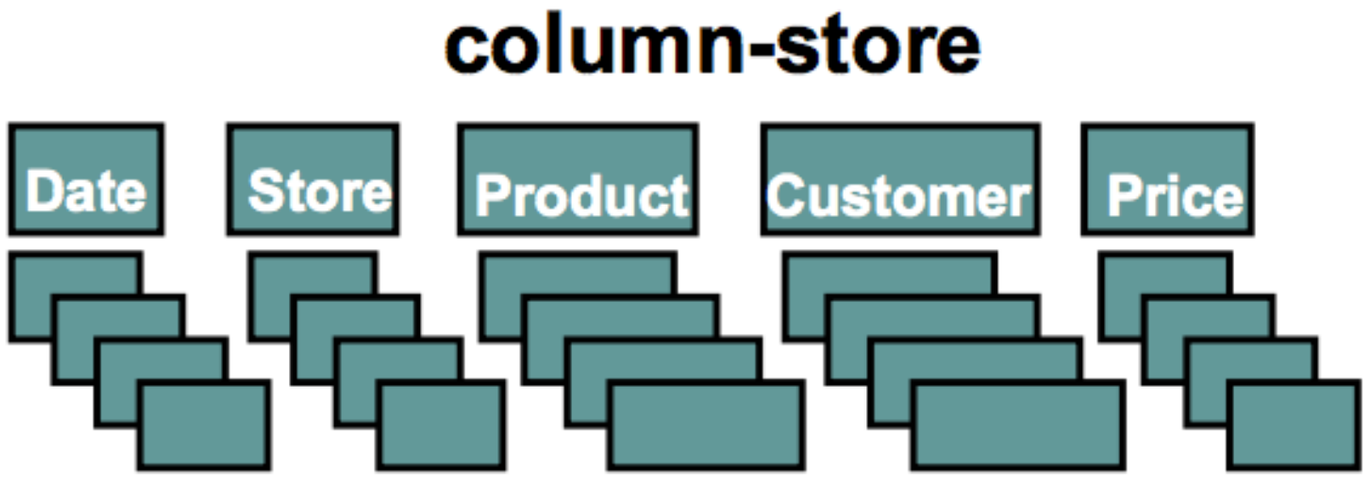


Why DuckDB?

- **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



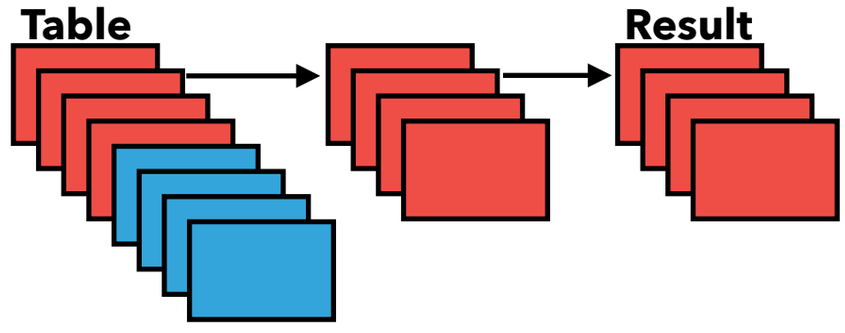


CWI

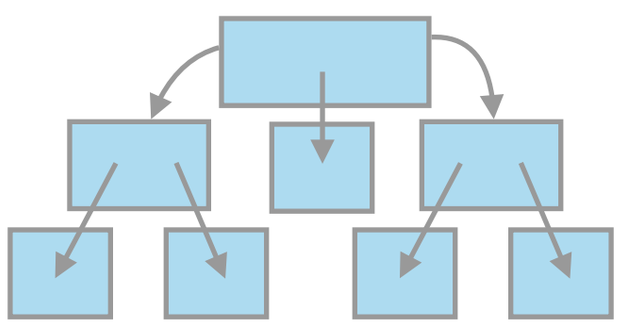
Internals at a Glance

Column-Store

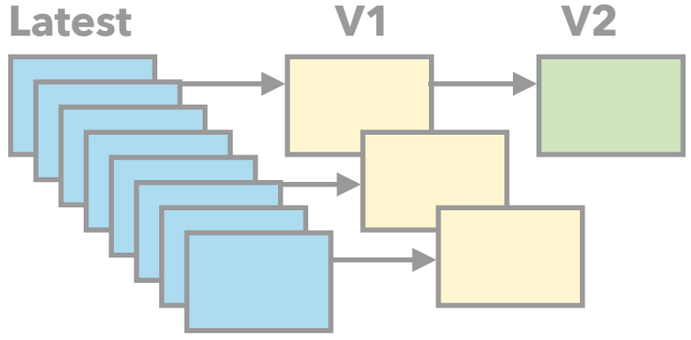
Vectorized Processing



ART Index

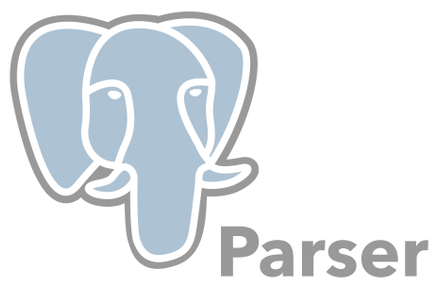


Multi-Version Concurrency Control



Single-File Storage

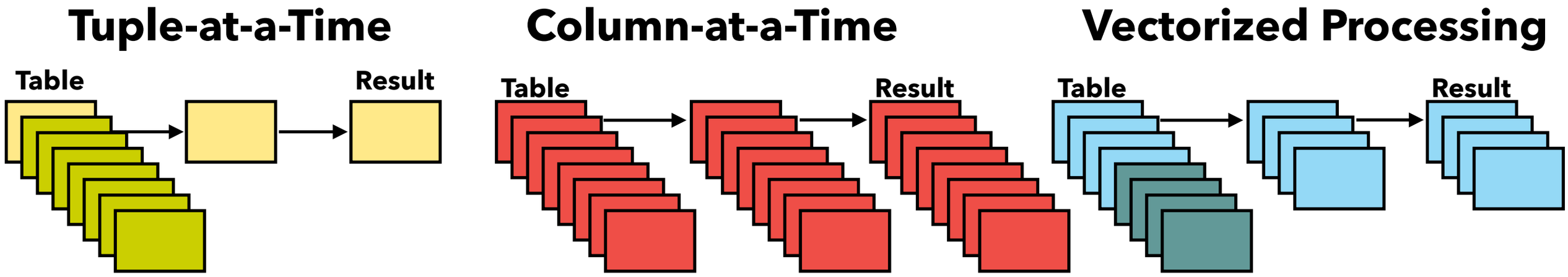
database.db



Parser

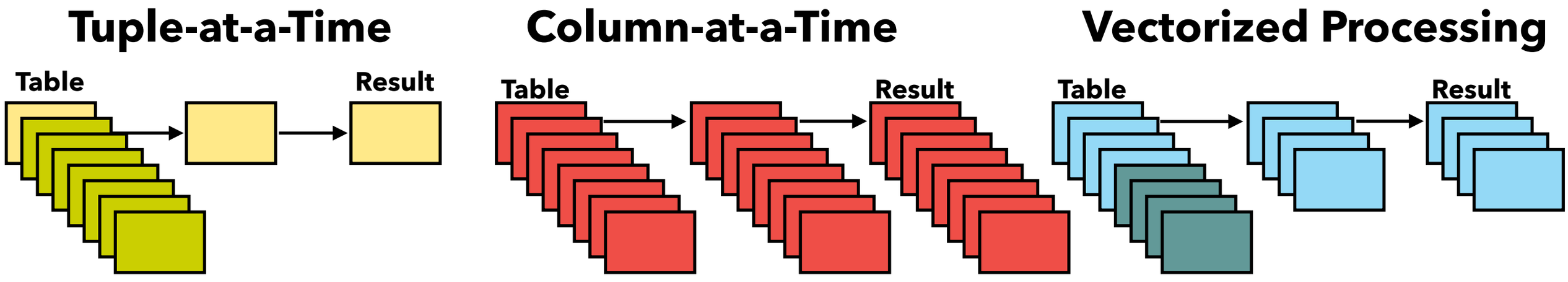
Why DuckDB?

- **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



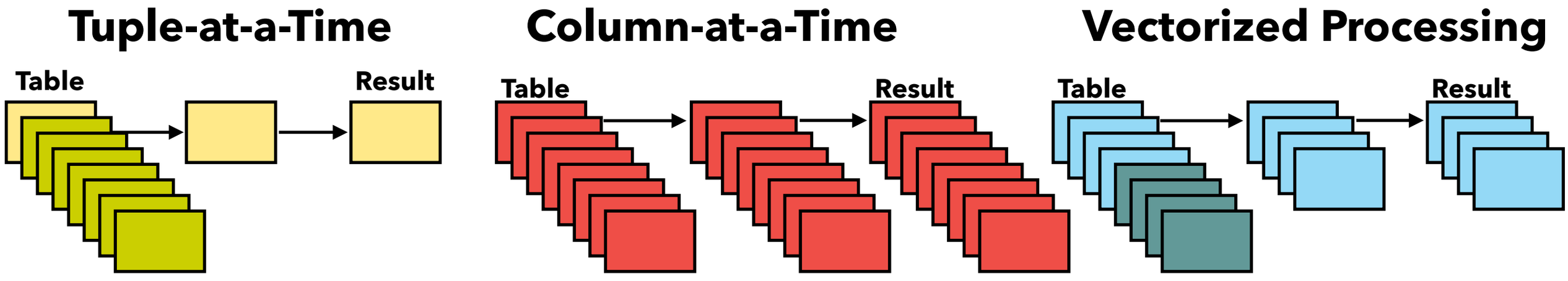
Why DuckDB?

- **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!**



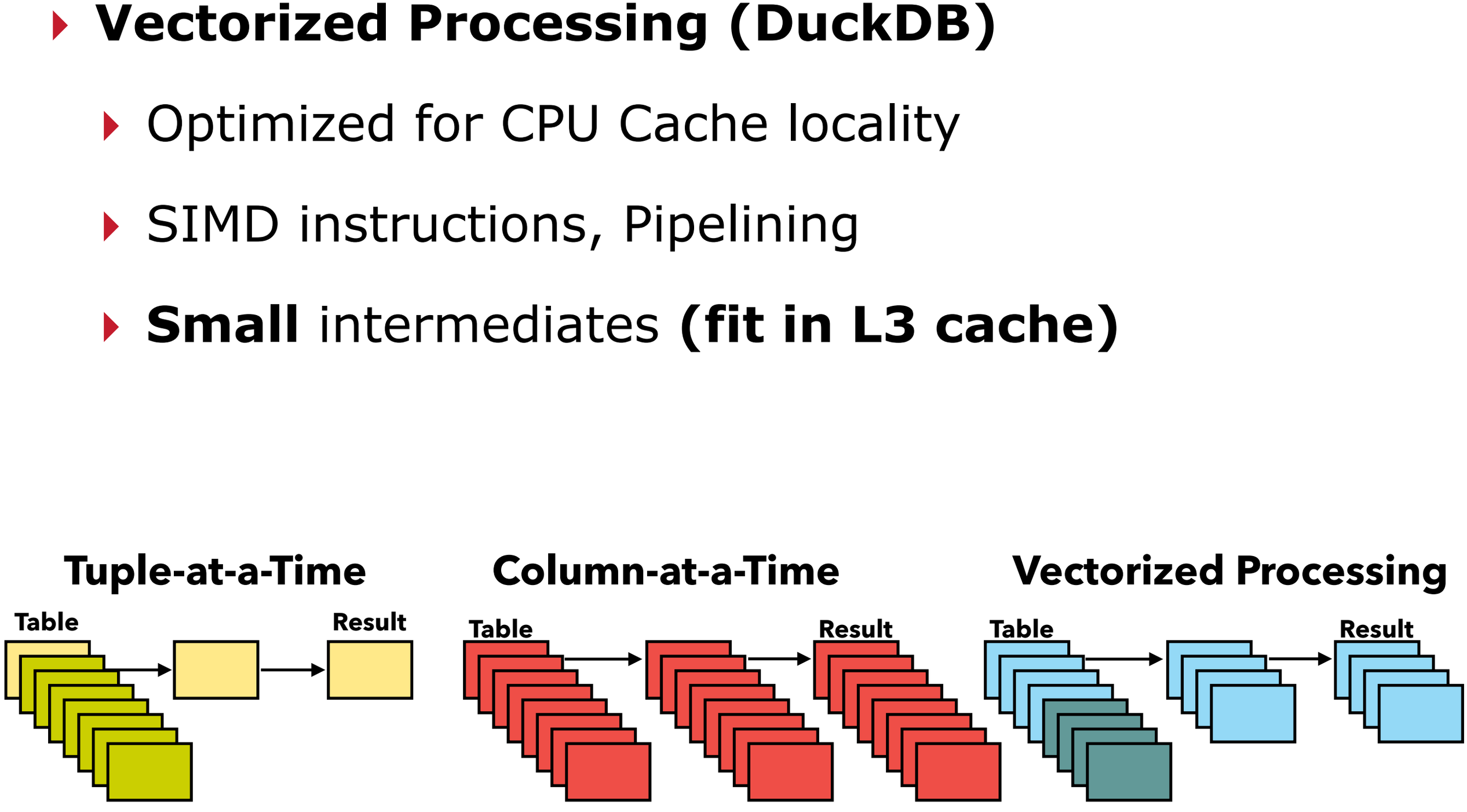
Why DuckDB?

- **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?

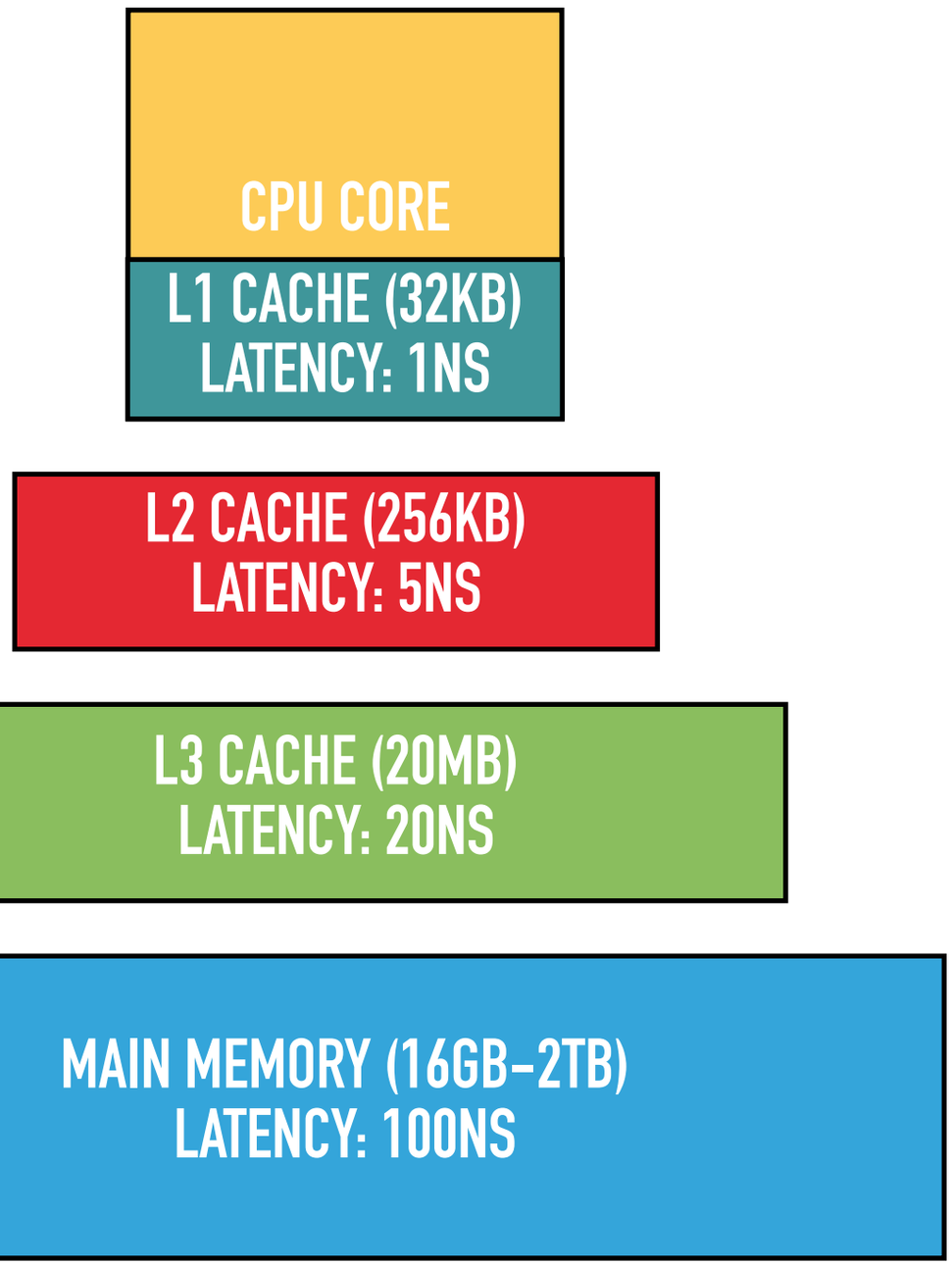
- **Vectorized Processing (DuckDB)**
  - Optimized for CPU Cache locality
  - SIMD instructions, Pipelining
  - **Small** intermediates **(fit in L3 cache)**



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

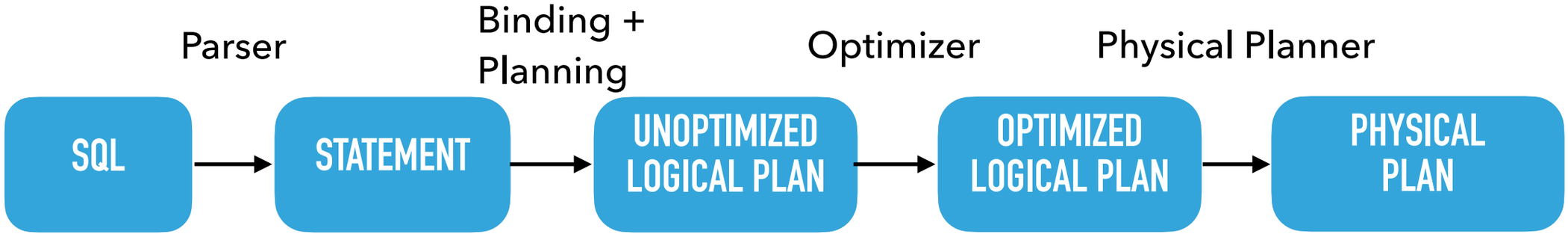
- **Life of a query**
- How does the system go from query to result?
- We will focus on the following query:

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

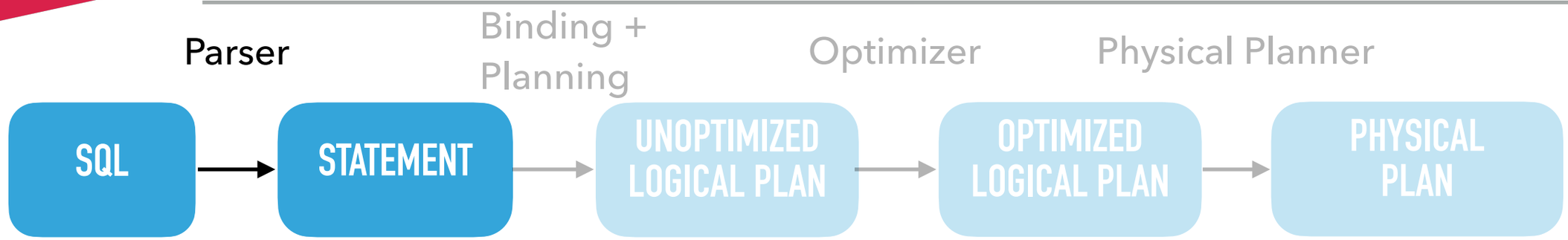
- **Aggregate:** `COUNT(*)`
- **Implicit join:** `lineitem, orders` **on** `orderkey`
- **Filters:** `o_orderstatus='X'` **and** `l_tax>50`

DuckDB

- DuckDB uses a typical pipeline for query processing



CWI DuckDB

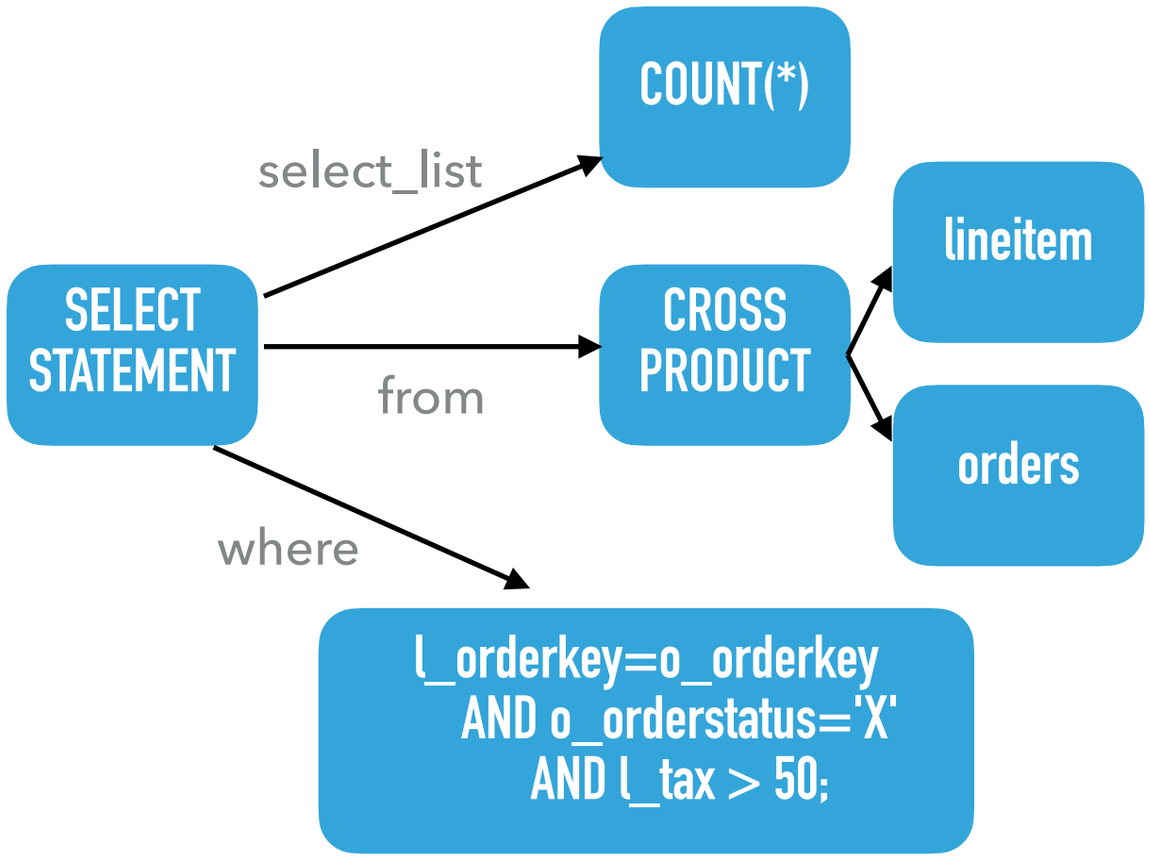


- Query is input into the system as a string
- The **lexer and parser** take the input string and convert it into a set of **statements**, **expressions** and **table references**
  - Note that this is not yet a query tree!
- We utilize the **Postgres parser** for this part

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

CWI DuckDB

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




```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

CWI DuckDB

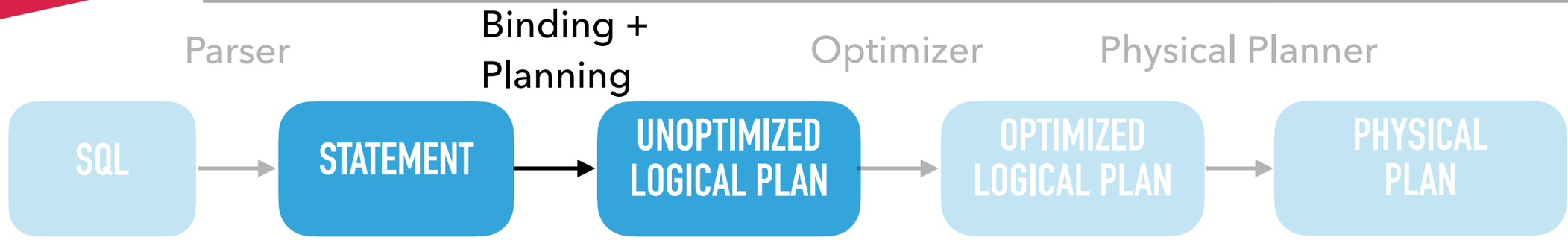
- In (pseudo) code, this is as follows:

```
class SelectStatement : Statement {
    vector<Expression*> select_list = { COUNT(*) }
    TableRef from_clause = CROSS_PRODUCT("lineitem", "orders");
    Expression *where_clause = "l_orderkey"="o_orderkey"
                               AND "o_orderstatus"="X"
                               AND "l_tax" > 50;
}
```

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

- Note: table/column names *are not resolved yet*
  - e.g. if **lineitem** table does not exist, no error will be thrown yet

CWI **DuckDB**



- **Binding phase**
  - Catalog lookup of table/column names
  - Type resolution of expressions

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

CWI DuckDB

- Look up tables **`lineitem`** and **`orders`** tables
- Look up **columns** within these tables

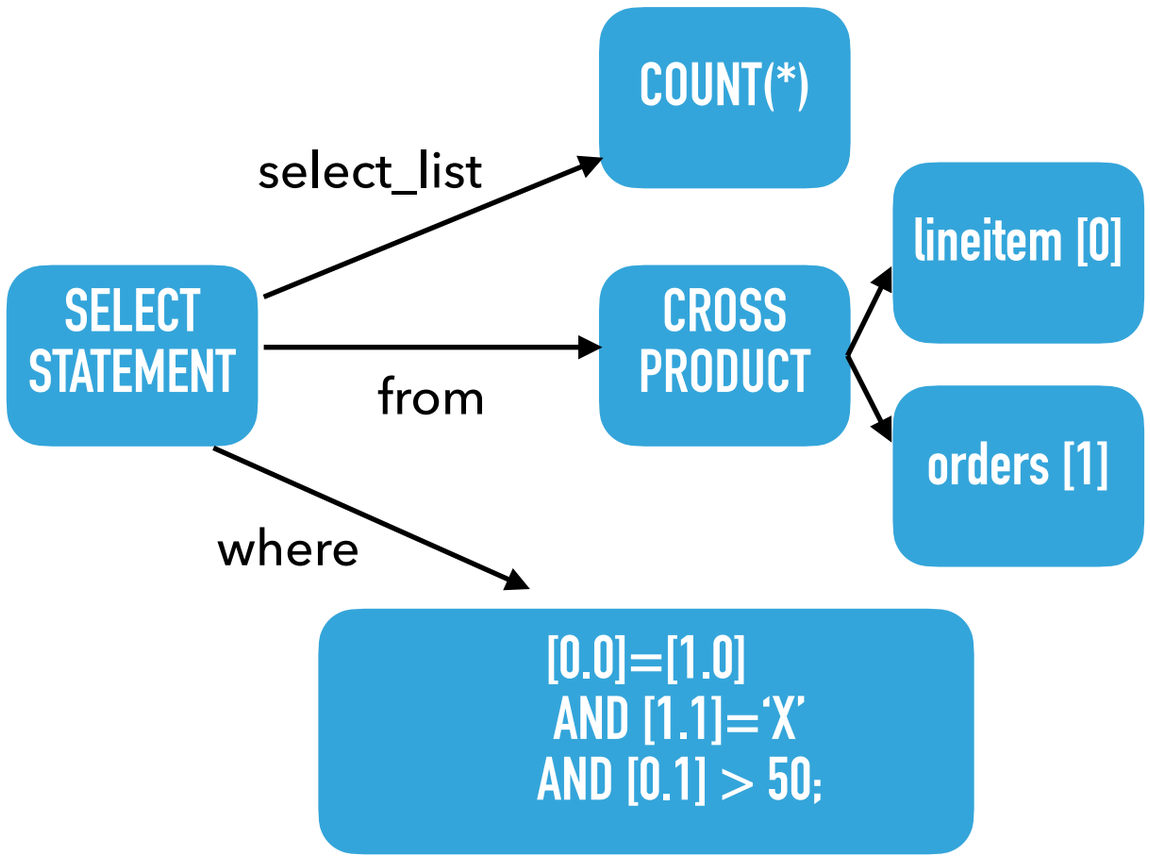


table index	column index	column index
lineitem: 0	l_orderkey: 0	o_orderkey: 0
orders: 1	l_tax: 1	o_orderstatus: 1

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

- Replace table names with **table indexes**
- Replace column names with **table+column** indexes

CWI DuckDB

- **Type resolution:** look up the types from the tables
  - `l_orderkey : INTEGER`
  - `o_orderkey : INTEGER`
- `l_orderkey = o_orderkey : BOOLEAN`

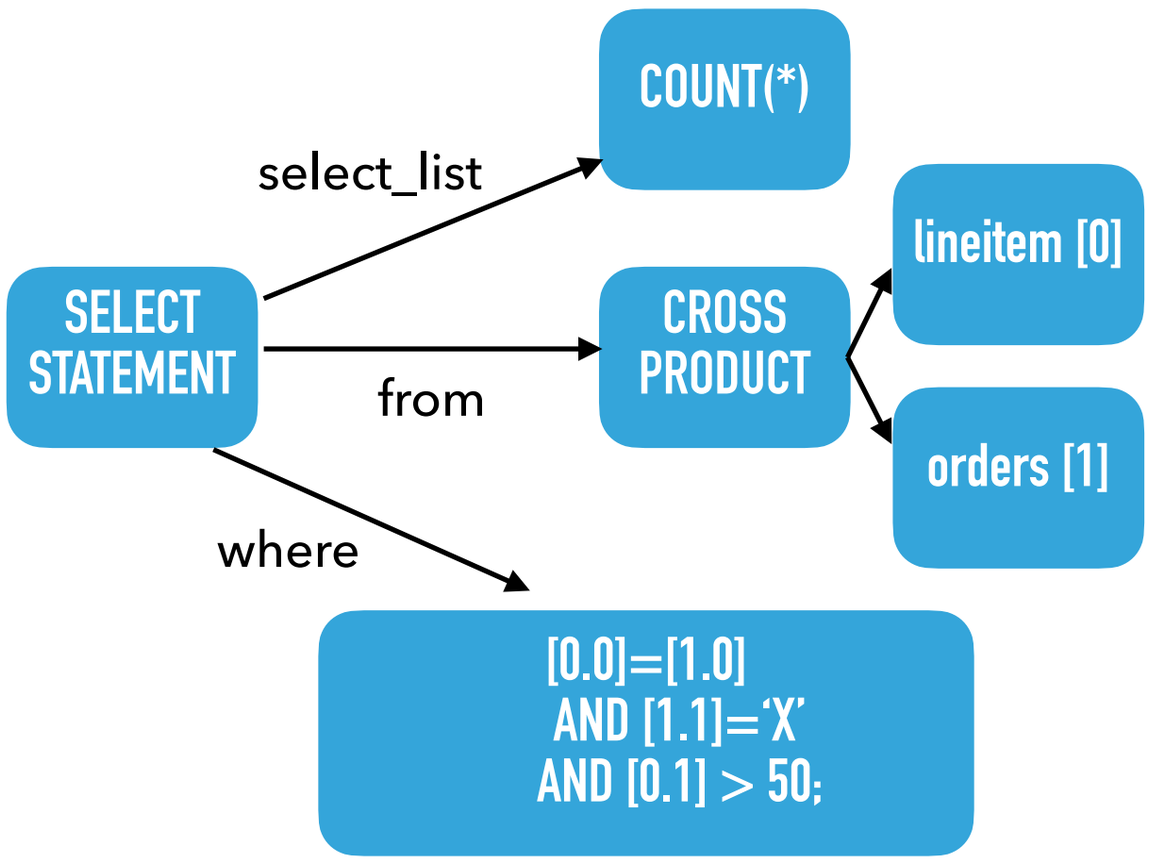
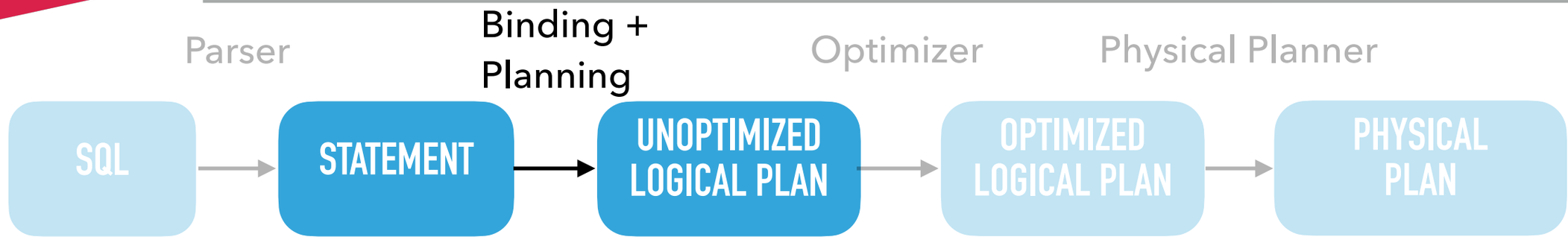


table index	column index	column index
lineitem: 0	l_orderkey: 0	o_orderkey: 0
orders: 1	l_tax: 1	o_orderstatus: 1

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

CWI 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`"

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

DuckDB

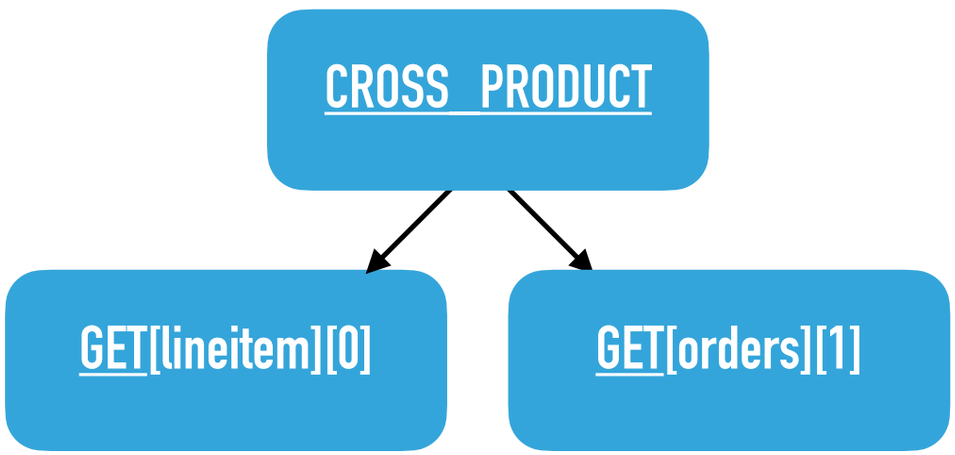
- Query tree starts with **tables**
- We have two tables: **lineitem** and **orders**
- These will result in two **LogicalGet** operations

GET[lineitem][0]

GET[orders][1]

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

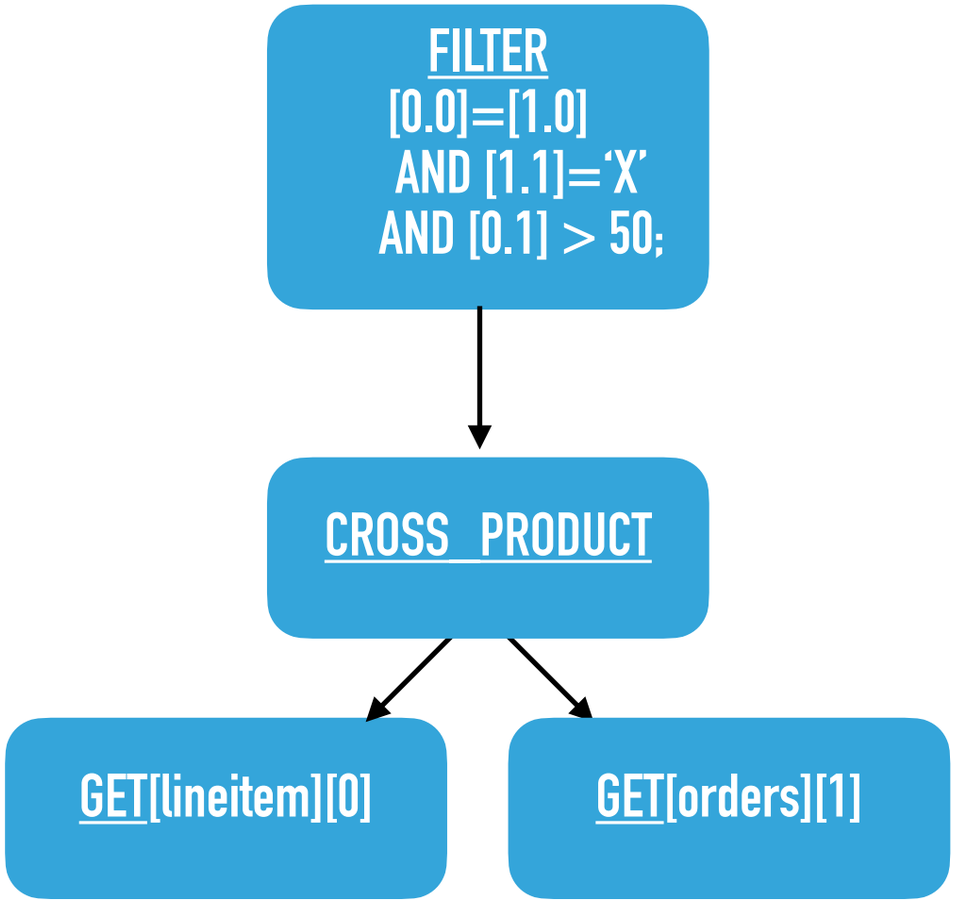
- The tables are combined with a **cross product**
  - There is no explicit join here
- The optimizer will later convert this into a join




```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

DuckDB

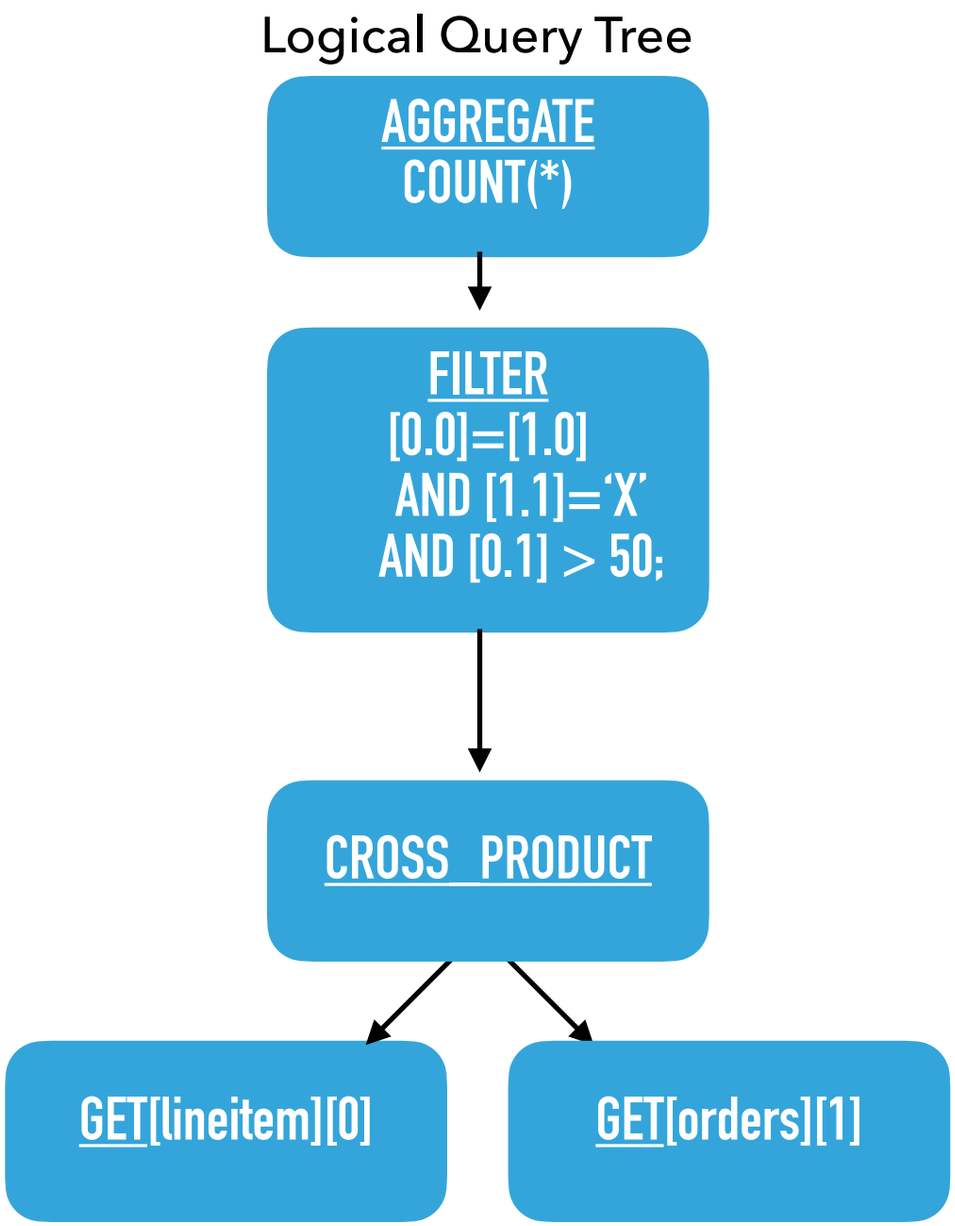
- After **Filter** is added
  - Filter has single big expression




```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

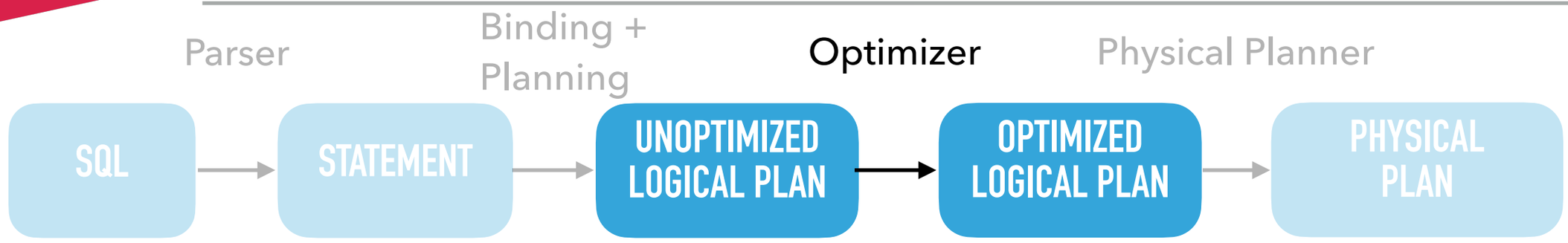
DuckDB

- Finally add aggregate computation




```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

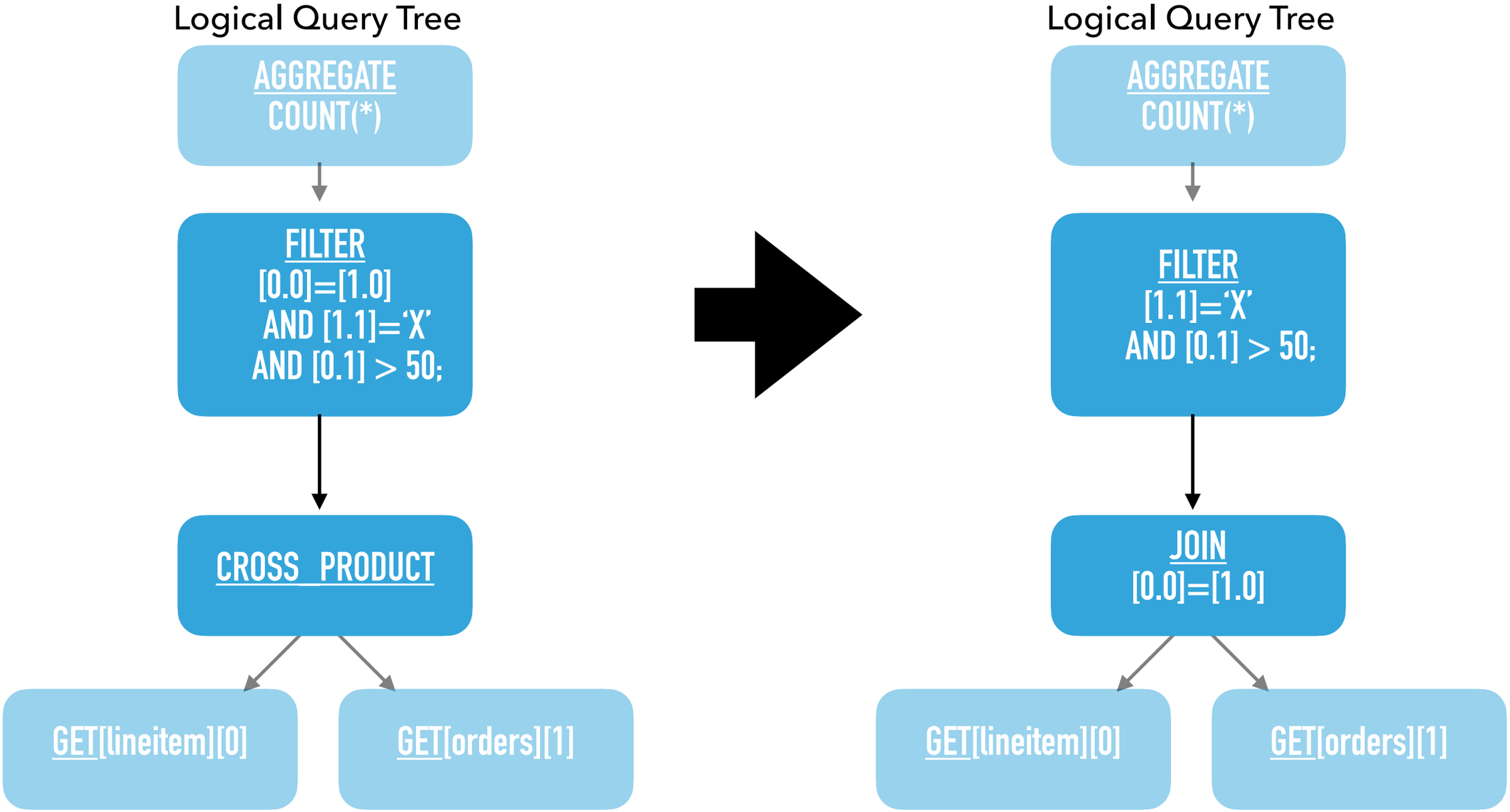
CWI DuckDB



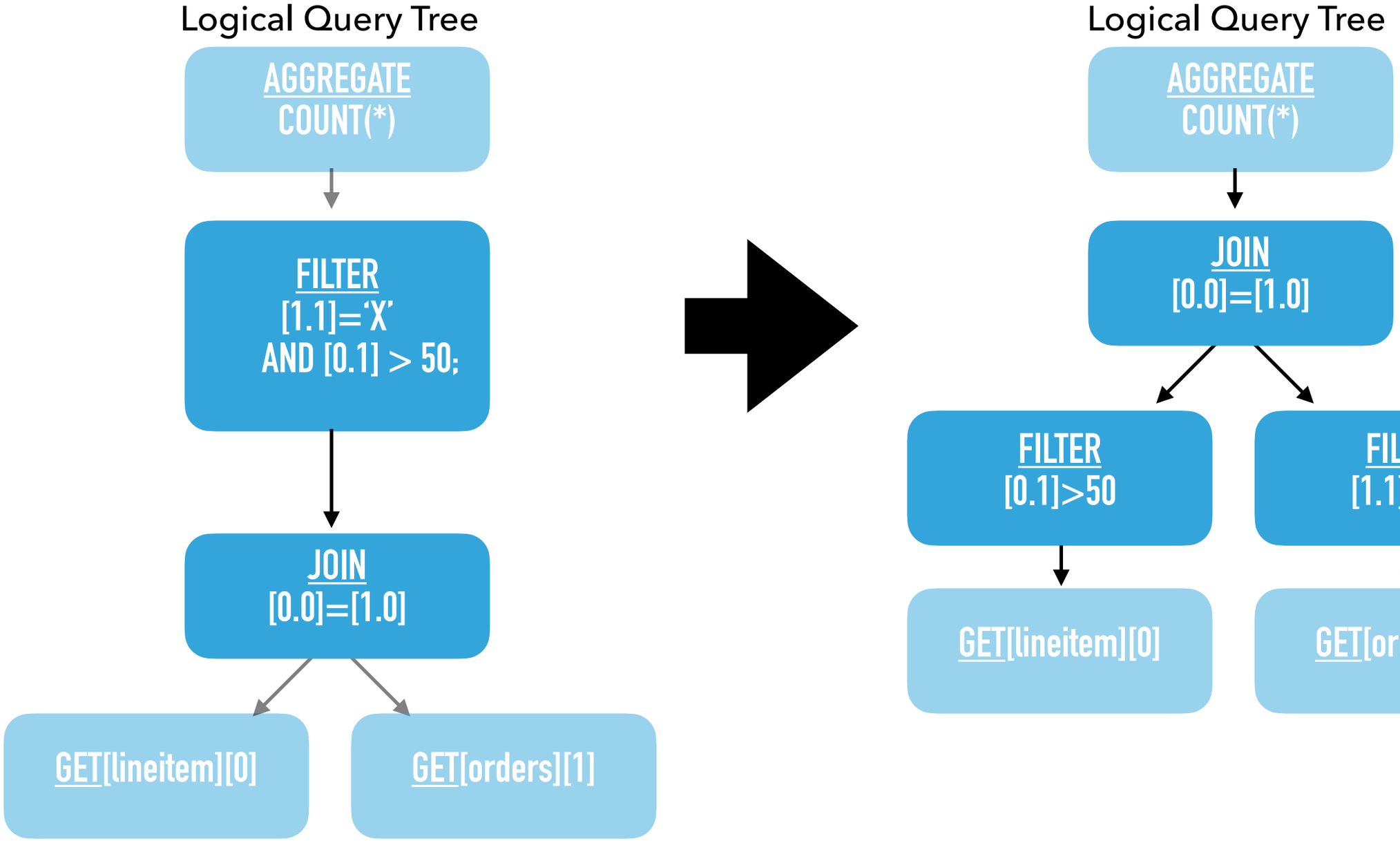
- **Optimizer:** transforms the logical query tree
- Created plan is logically equivalent
- But (hopefully) faster

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

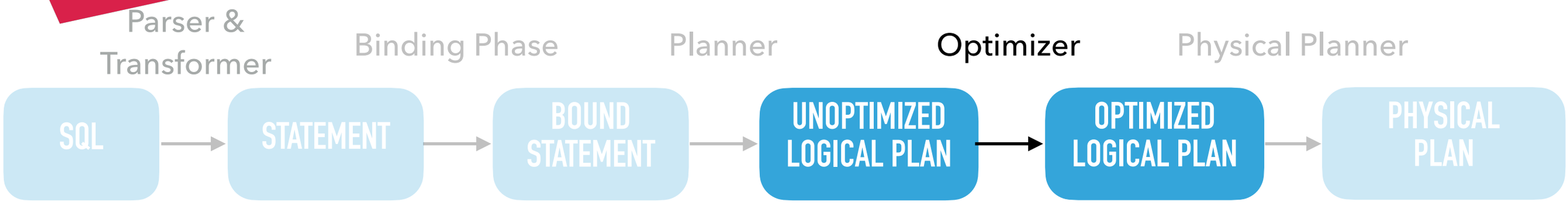
- Pushdown filter into cross product: creates a join



- Pushdown filters below the join



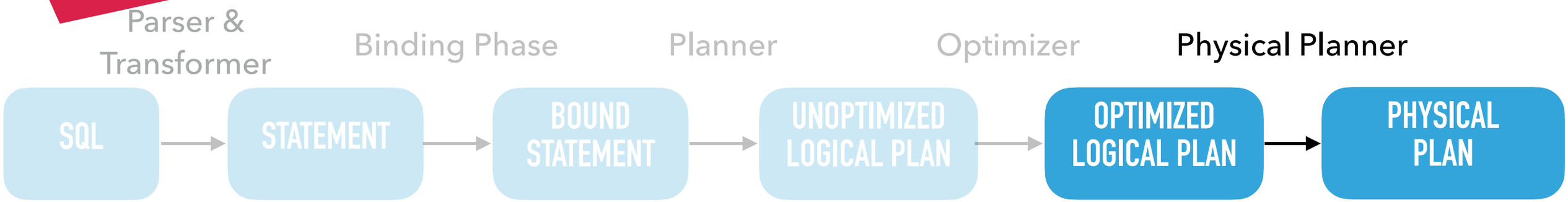
DuckDB



- Many possible optimizations possible here
  - Join ordering, constant folding, CSE, subquery flattening, common subtree elimination, projection pushdown, etc…
- For this query only filter pushdown is necessary

```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

CWI DuckDB

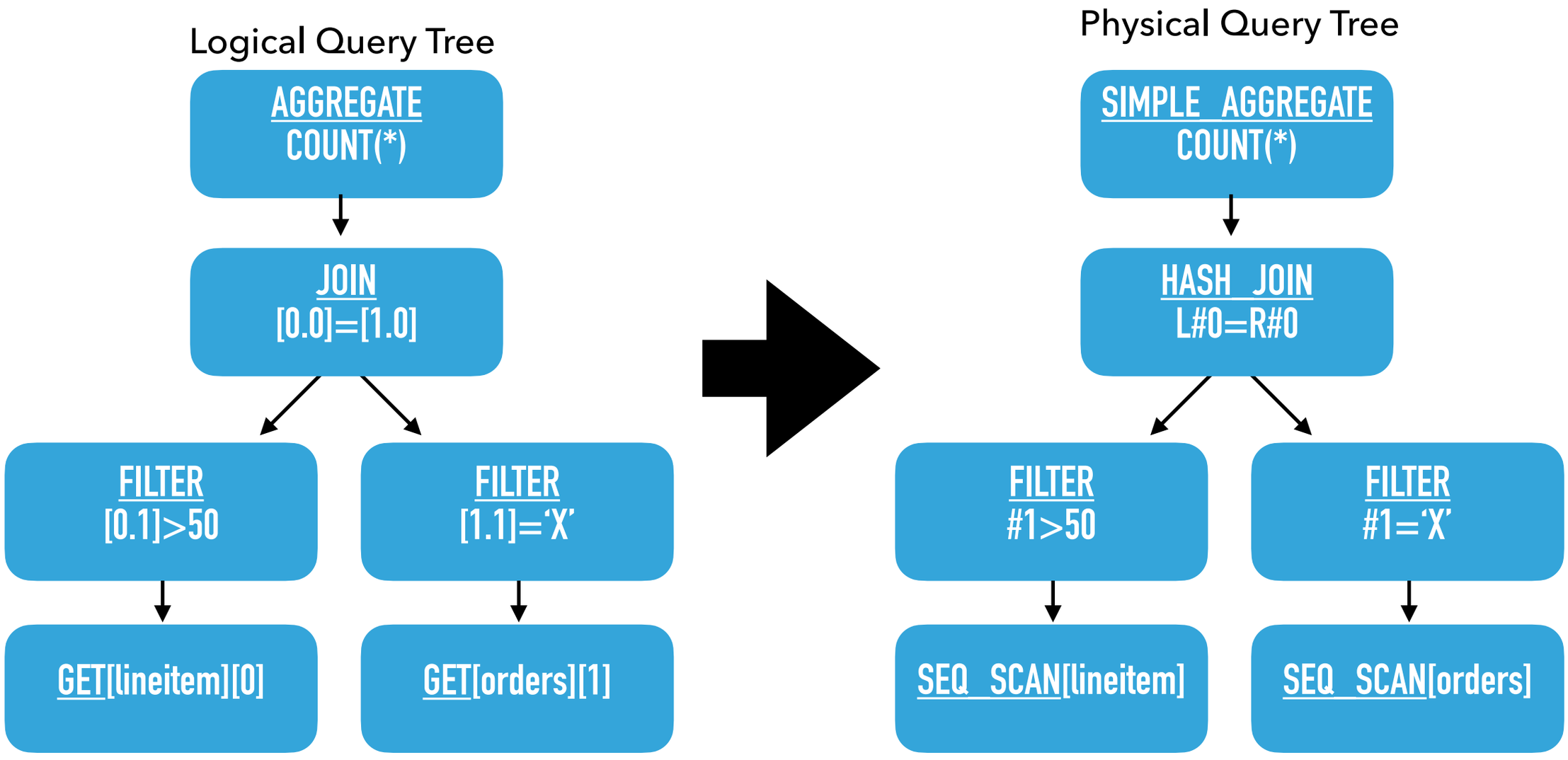


- **Physical planner:** converts logical plan into physical (executable) plan
- Makes decision on implementations of operators
  - e.g. use a HashJoin, MergeJoin or NestedLoopJoin

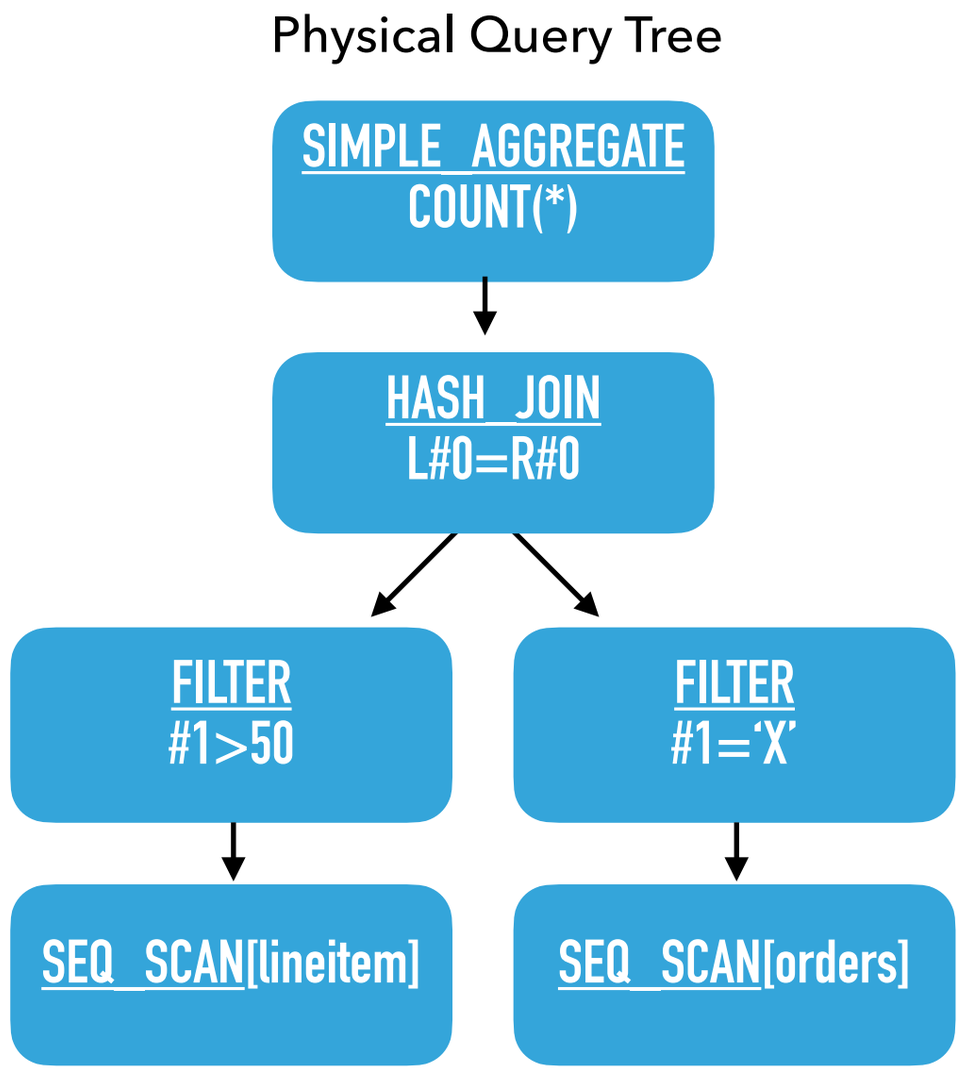
```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

CWI DuckDB

- **SimpleAggregate:** no groups, no hash required
- **Hash Join:** Most effective for this equality join
- **Sequential Scan:** No index that helps us speed up



- Now we have the final query tree
  - This is what we **execute** to run the query



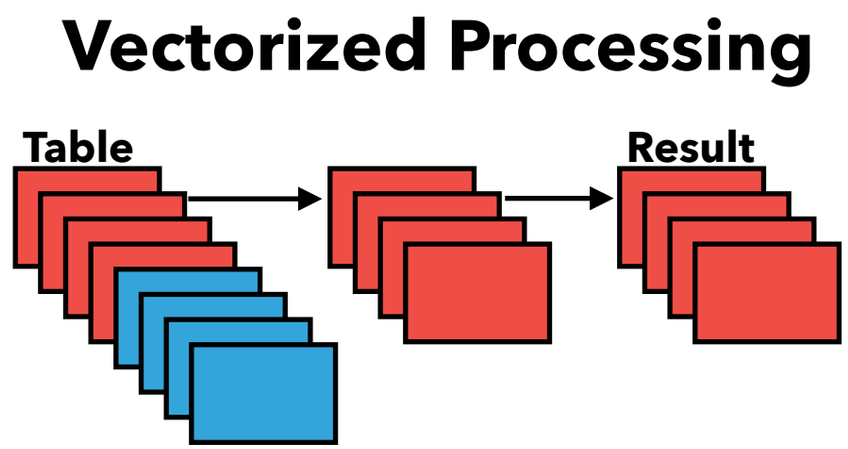
Outline

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

Query Execution

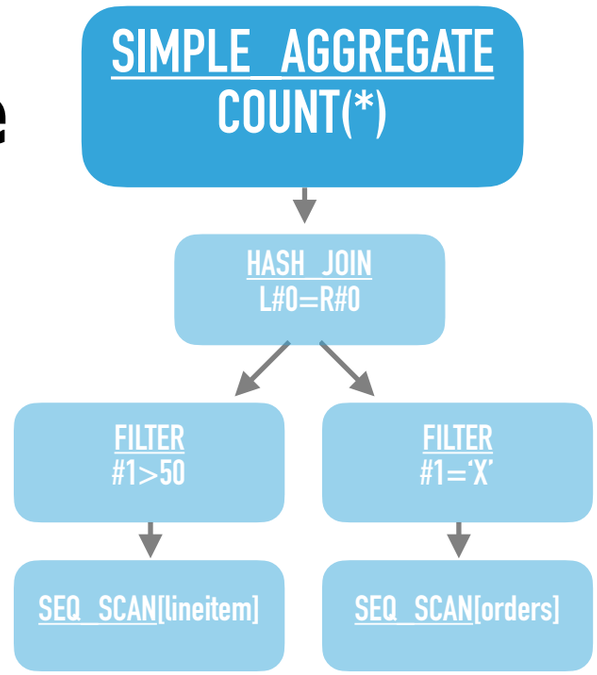
- DuckDB uses a vectorized pull-based model
  - "vector volcano”

- Query starts by calling **`GetChunk`** on the root node
- Root node recursively calls **`GetChunk`** on children
- Scans fetch data from the base tables



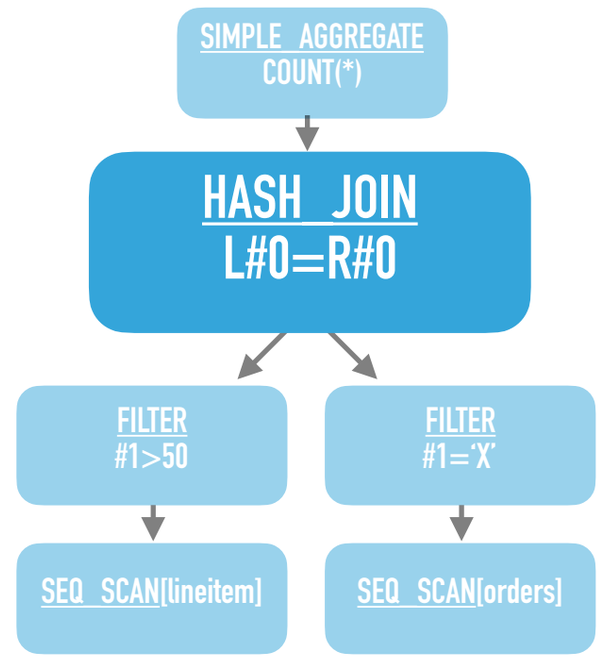
CWI DuckDB

- Start with root node: **SimpleAggregate**
  - Aggregate without groups
- Immediately calls **`GetChunk`** on child

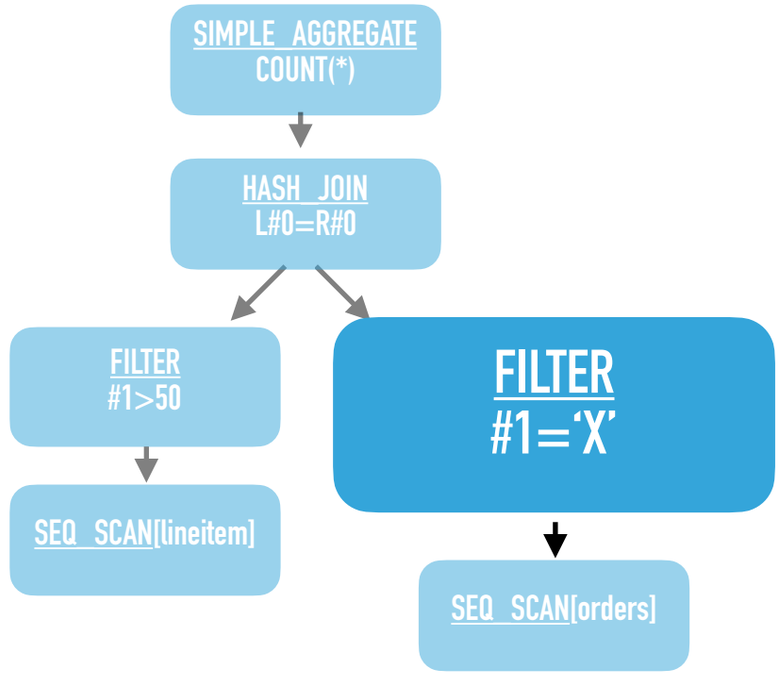


CWI **DuckDB**

- **Hash Join**
- Start by **building HT**
- Call **`GetChunk`** on right node

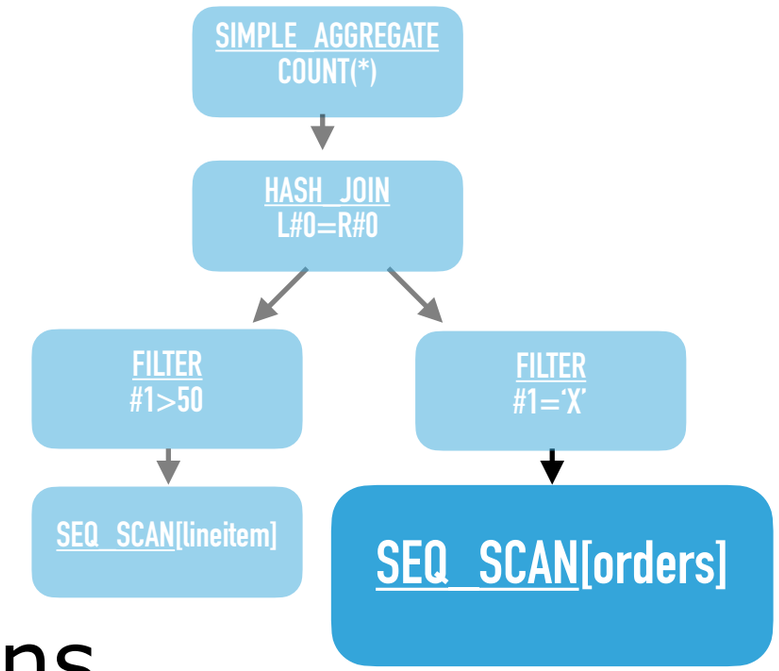


- **Filter**
- Again, pull a chunk from child

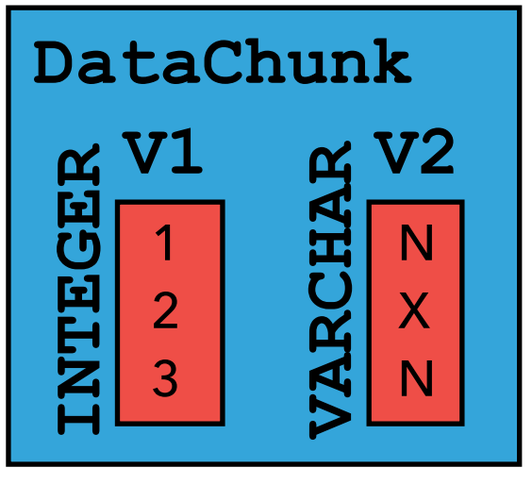


CWI DuckDB

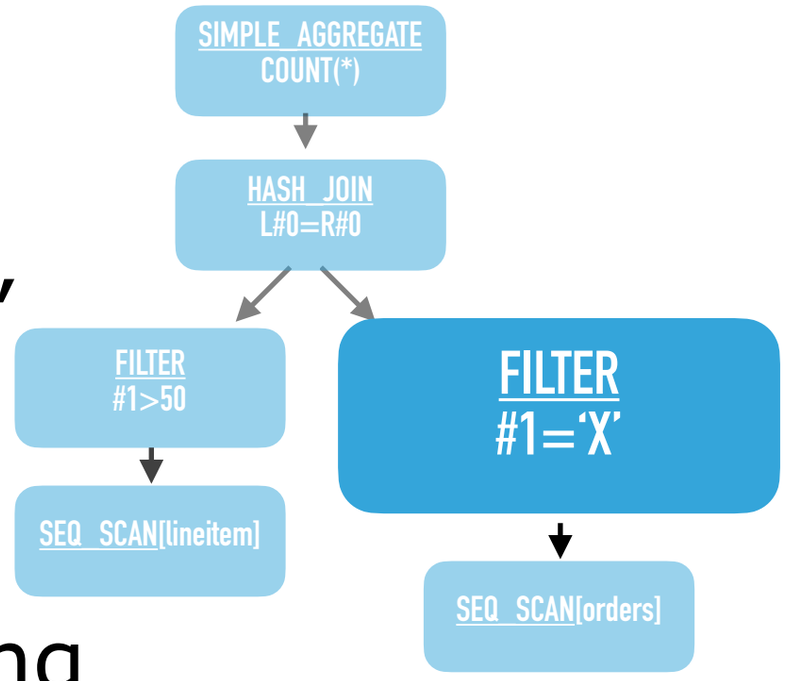
- **Sequential Scan**
- Finally we can start executing
- Scan the base table
- Return a DataChunk with two columns



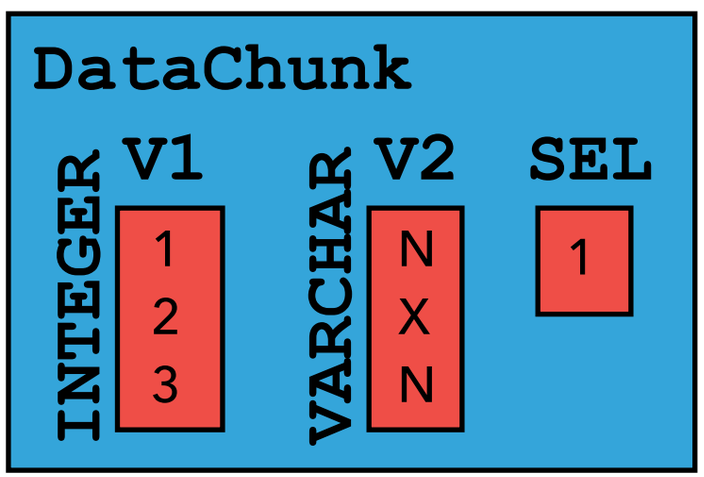
  - `o_orderkey` and `o_orderstatus`



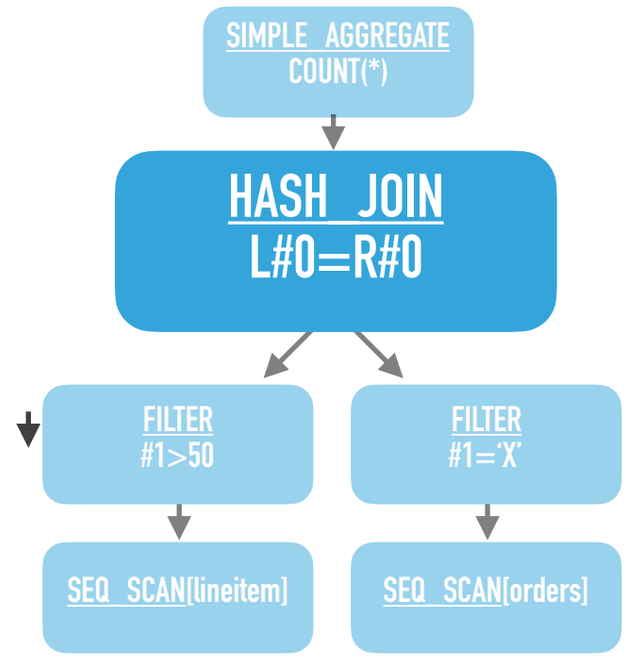
CWI DuckDB

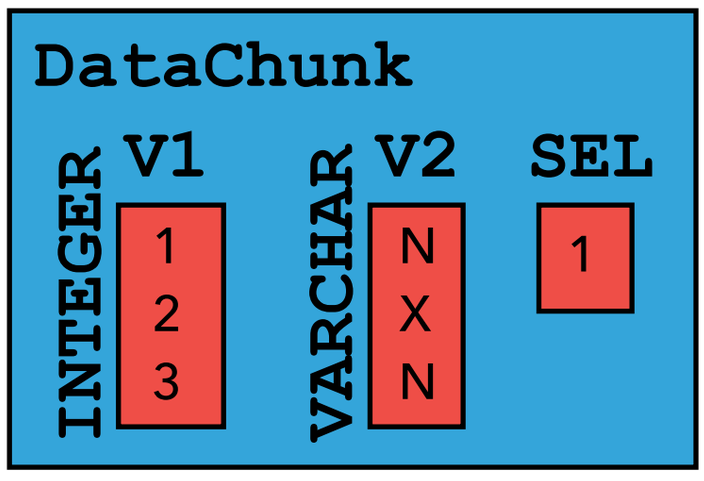


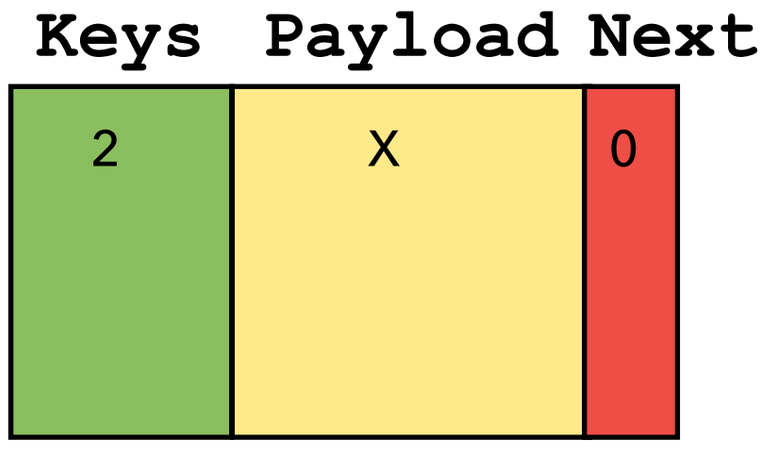
- **Filter**
- Now we can perform the filter #1='X'
  - Only the second tuple passes
- **Selection vector** pointing to surviving tuple is created
- Note that no data is copied or changed



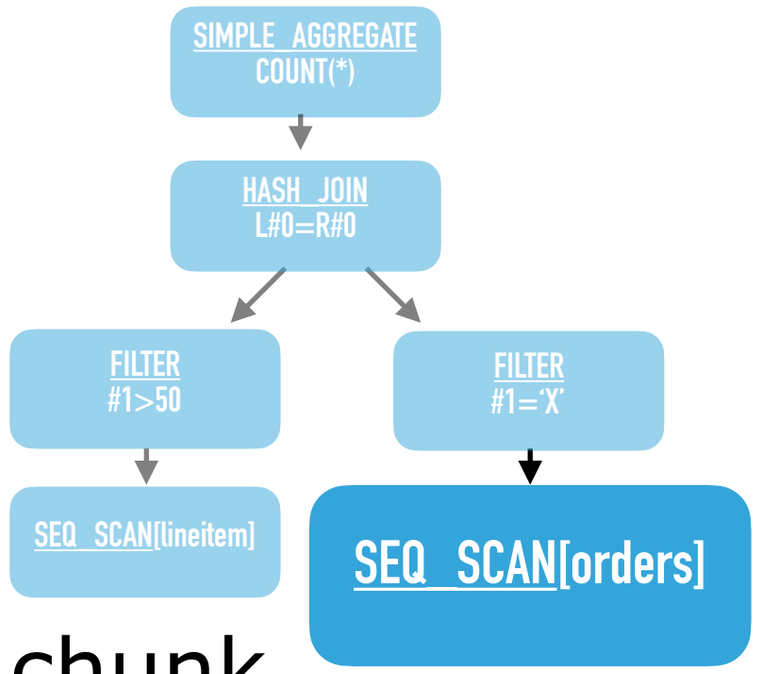
- **Hash Join**
- Now we have our first input chunk
- We input it into the HT
- Now we fetch another chunk from RHS

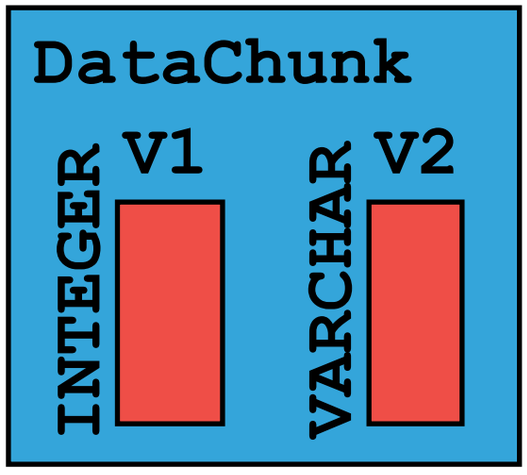




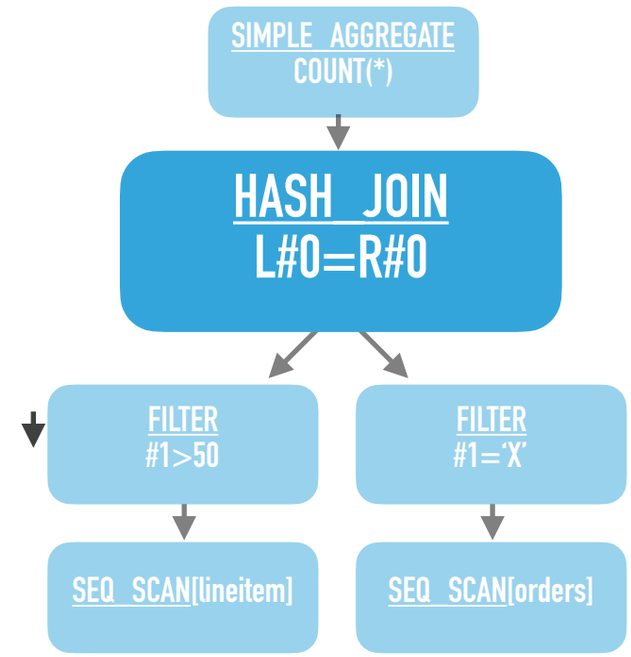


- **Sequential Scan**
- The filter again calls **`GetChunk`**
- Scan base table again:
  - The scan is finished, return empty chunk

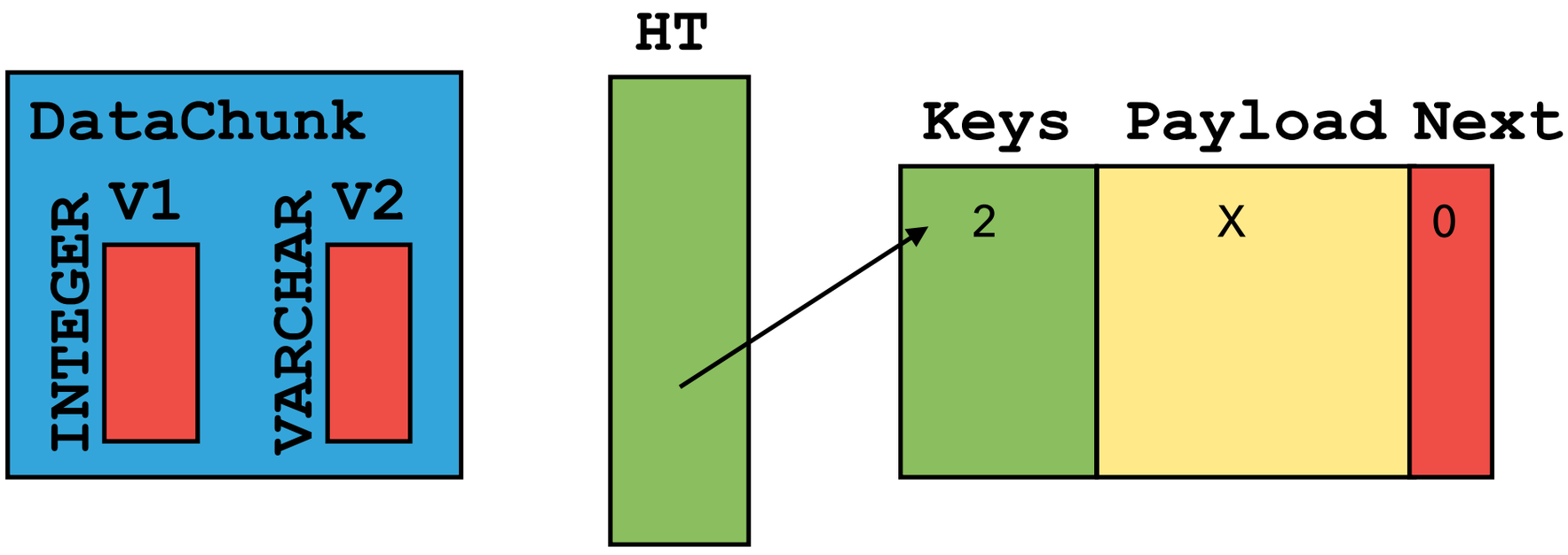




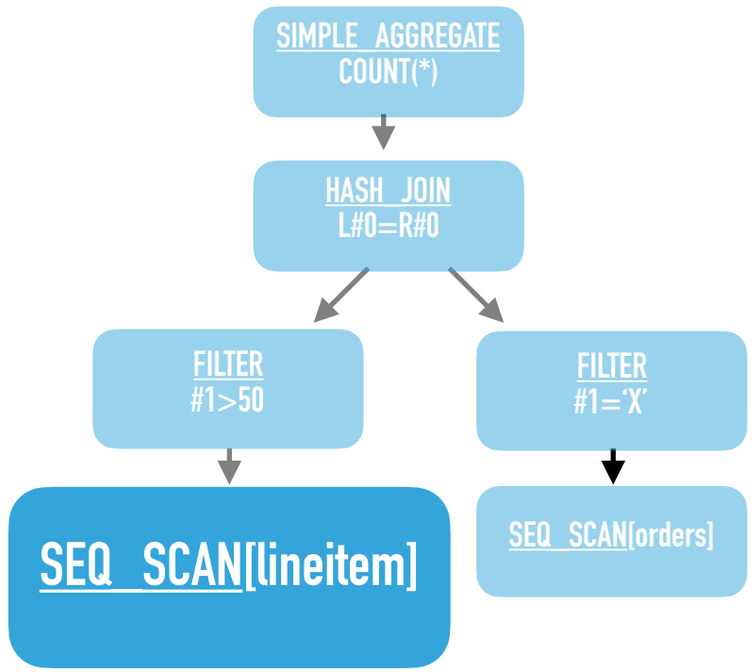
- **Hash Join**
- HT receives second input chunk
  - But it is empty!
- The RHS is exhausted



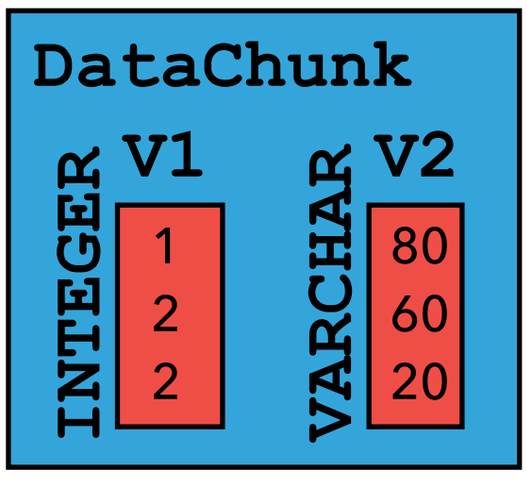
- Finish building HT and call **`GetChunk`** on LHS



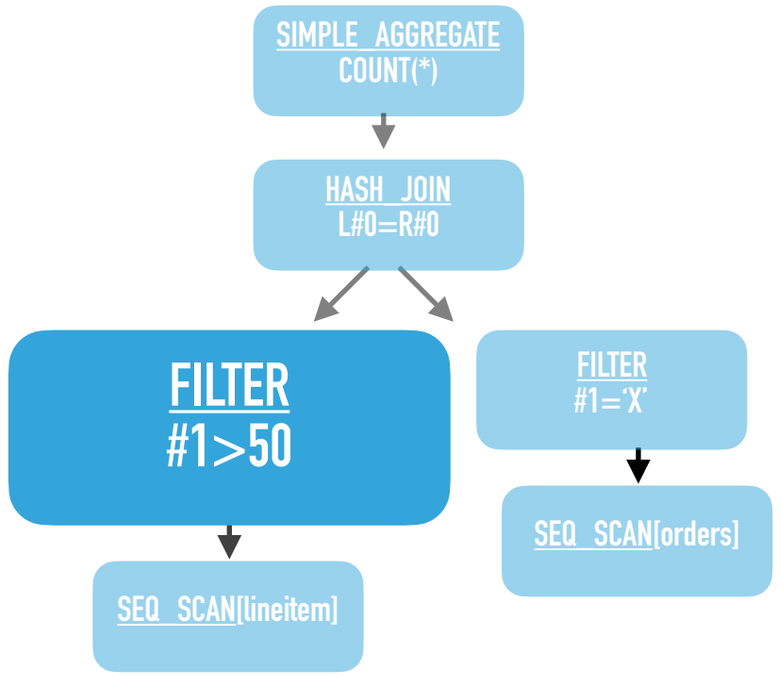
- **Sequential Scan**
- We arrive at scan on lineitem

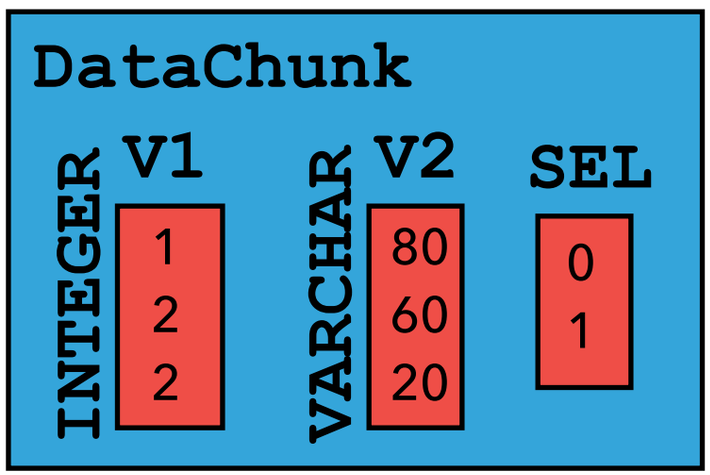


- DataChunk with two columns
- `l_orderkey` and `l_tax`

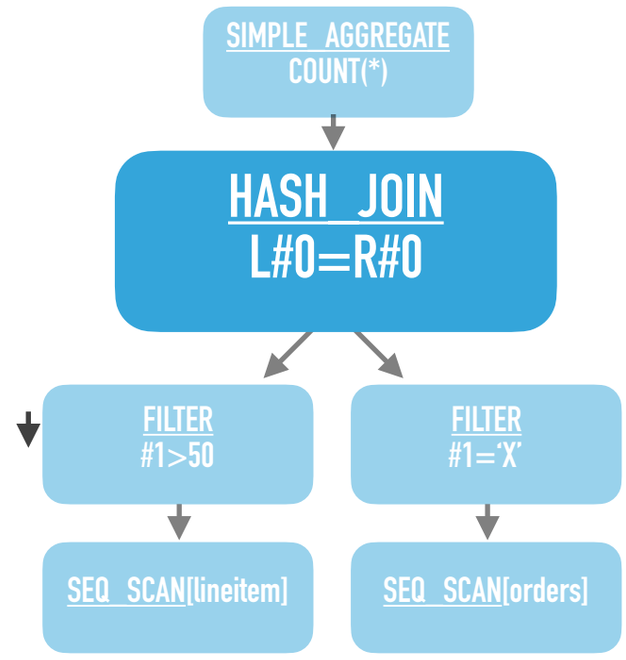


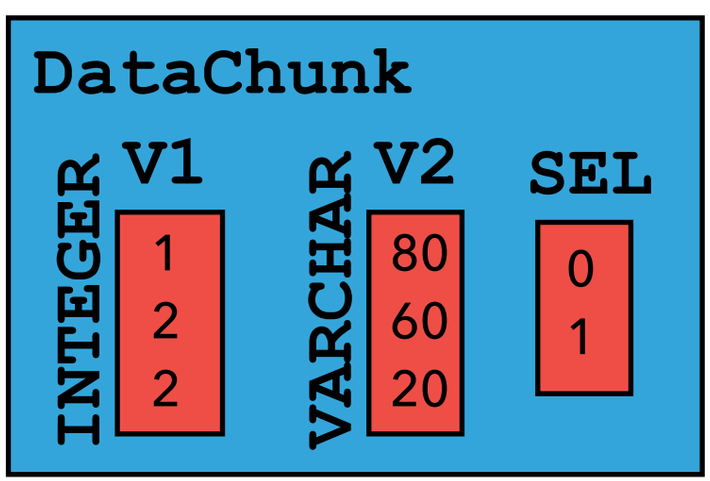
- **Filter**
- Performs the filter **#1>50**
- Again, add a selection vector

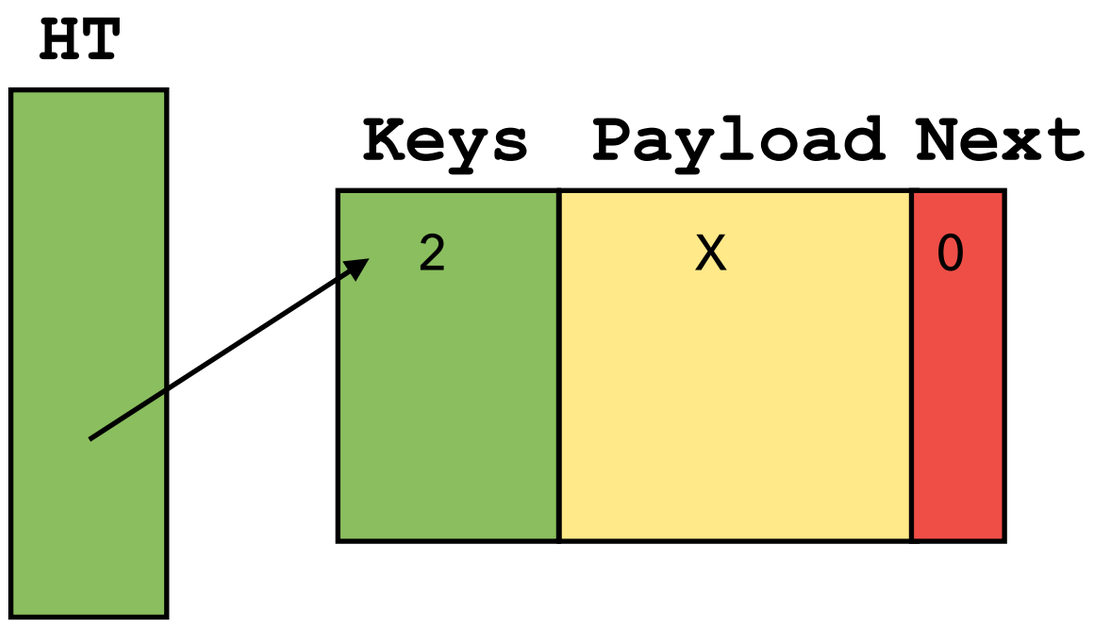




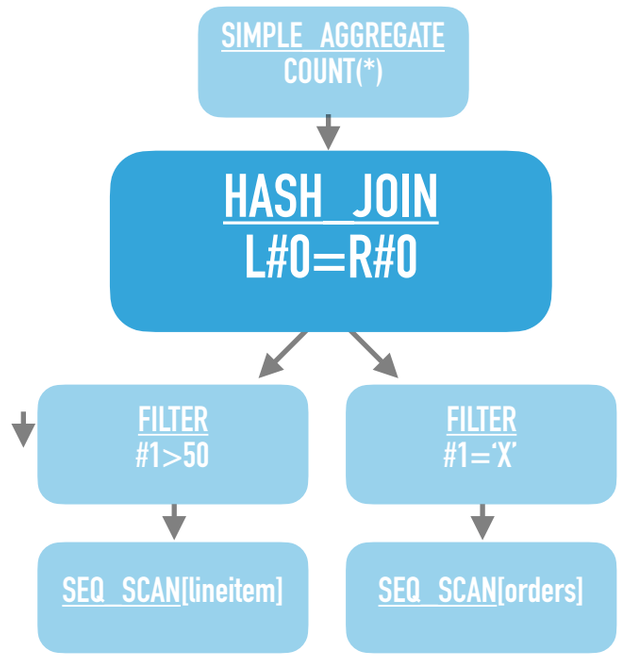
- **Hash Join**
- Now it is time to probe the HT
- We compute the hash for each tuple
- Then lookup in the HT

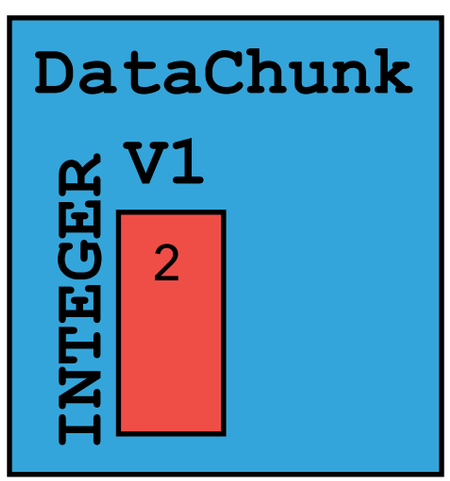




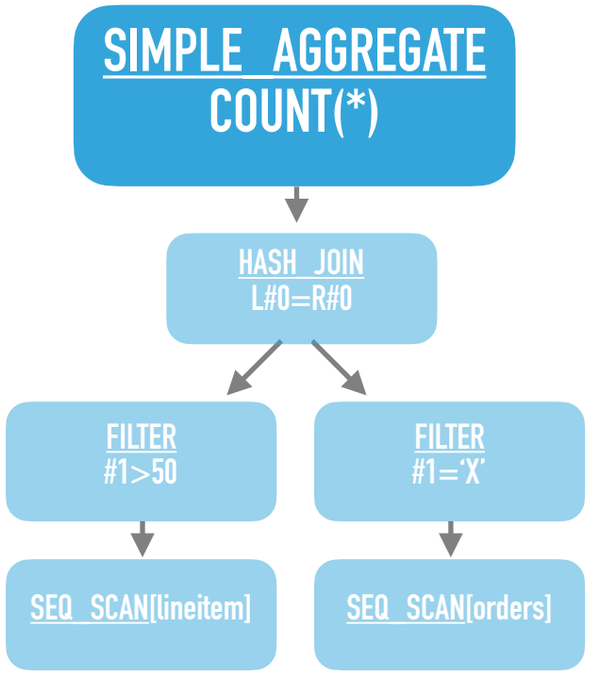


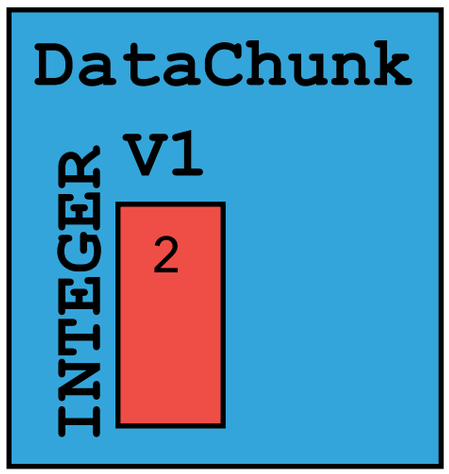
- **Hash Join**
- We get one hit on our join!
- The hash join now produces the result
- We return this to the aggregate

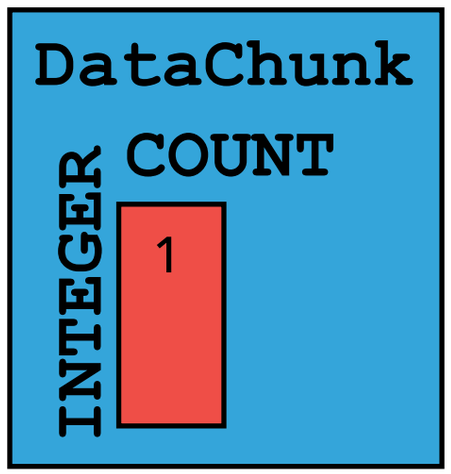




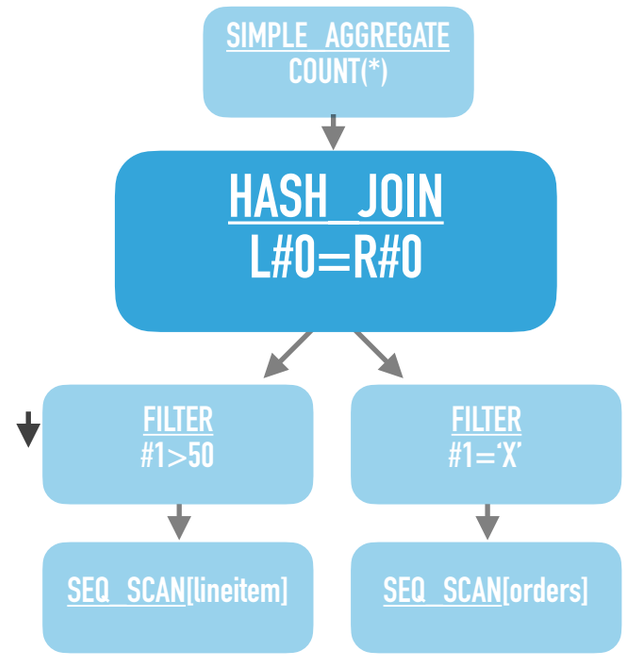
- The aggregate takes our input chunk
- Updates the aggregate
- Then fetches from the child again

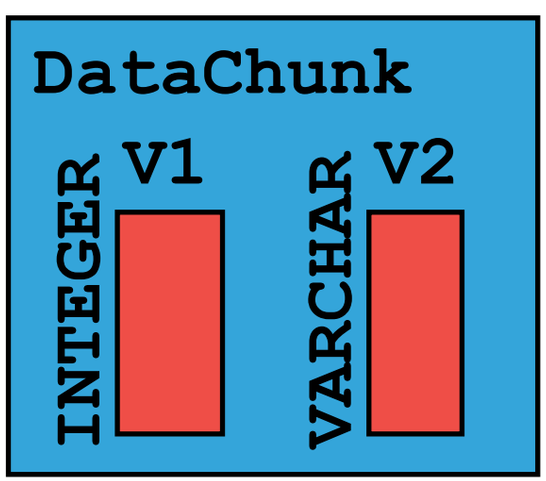


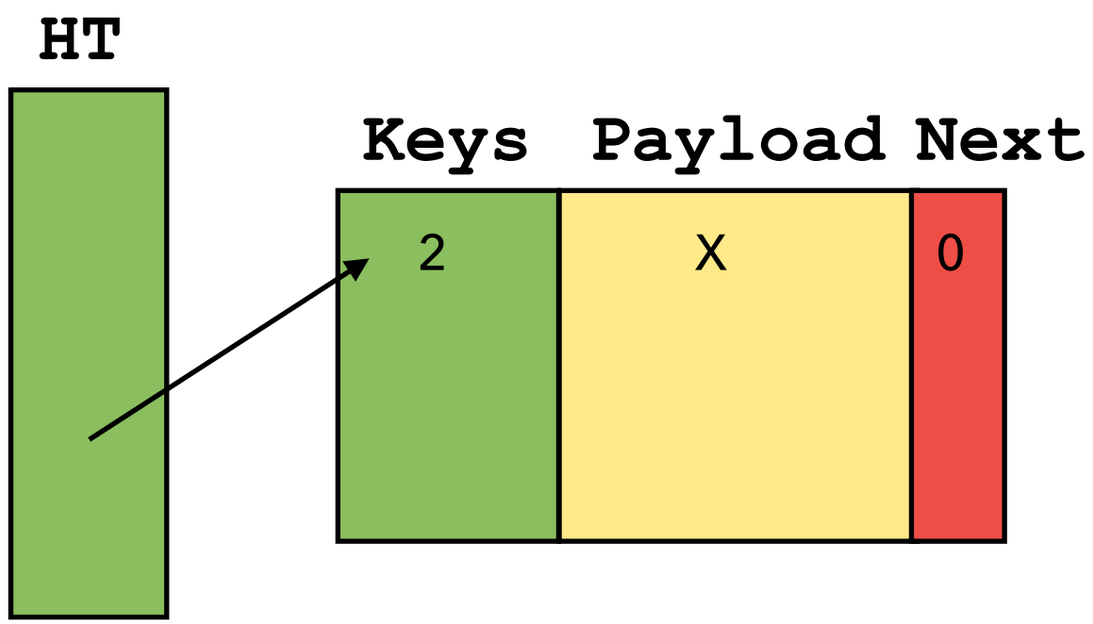




- We go back to the hash join
- Fetch from probe side again
- This time, input chunk is empty
- Now the hash join is entirely finished!

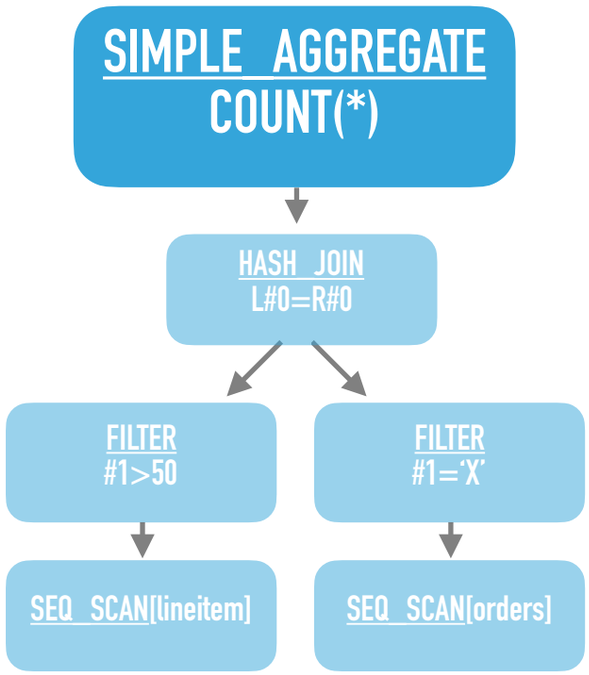


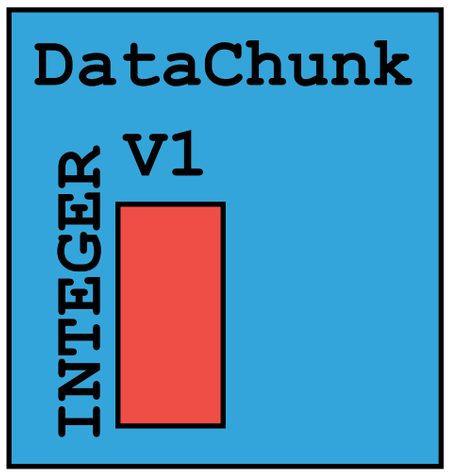


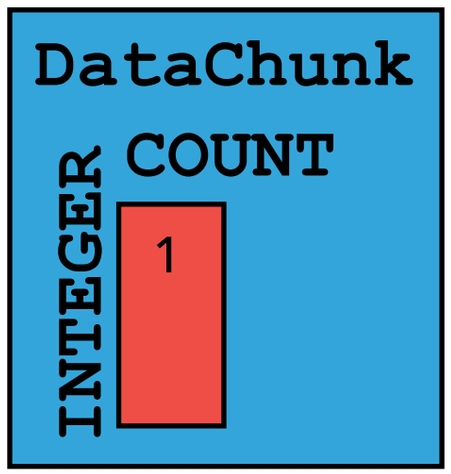


CWI DuckDB

- Aggregate gets an empty chunk
- Returns the final result of our query







CWI

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!