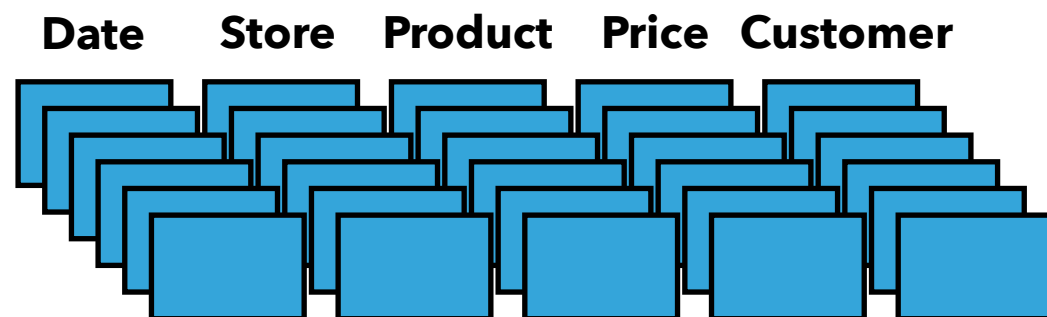**CWI**

Mark Raasveldt & Pedro Holanda

**DuckDB
an Embeddable Analytical RDBMS**
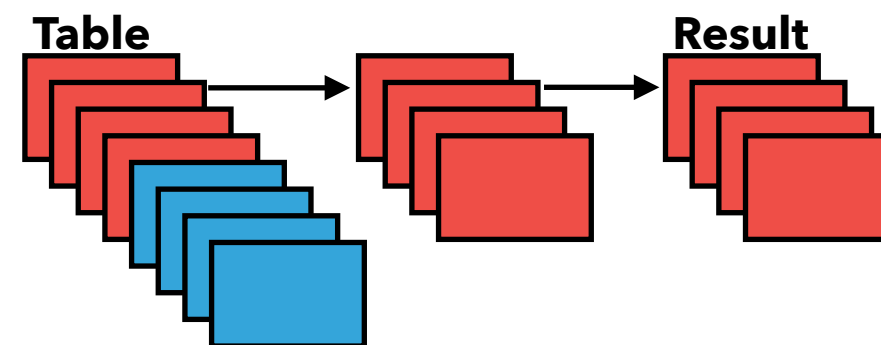
# **Outline**

▸ Internals at a Glance

▸ Query processing pipeline
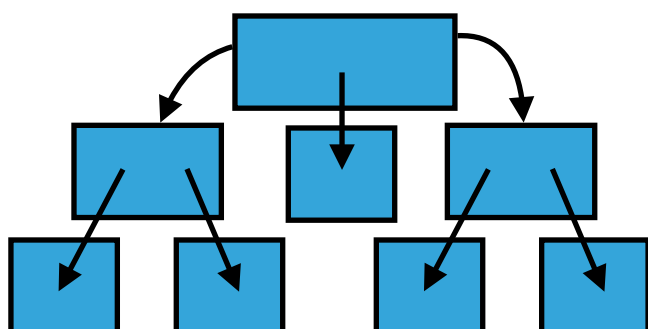
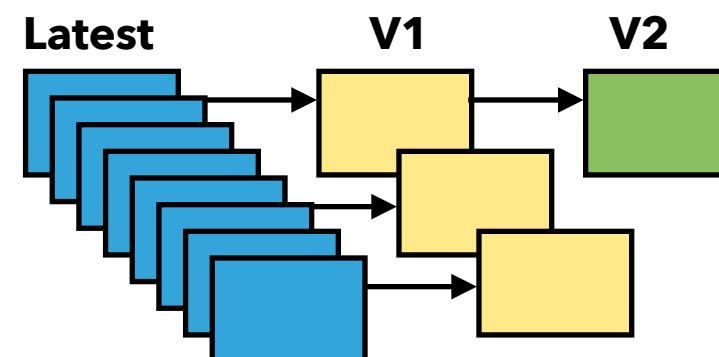▸ Query execution

▸ Hands-On

# Internals at a Glance

**CWI**

# Internals at a Glance

## Column-Store

Date  Store  Product  Price  Customer

## Vectorized Processing

Table → → Result

## ART Index

## Multi-Version Concurrency Control

Latest  V1  V2

## Single-File Storage

| HEADER | META | COL1 | COL2 | COL2 | COL2 |

4KB  256KB

**database.db**

C++

**Parser**

# CWI  Internals at a Glance

**Column-Store**

Date   Store   Product   Price   Customer

**Vectorized Processing**

Table          Result

**ART Index**

**Multi-Version Concurrency Control**

Latest          V1          V2

**Single-File Storage**

| HEADER | META | COL1 | COL2 | COL2 | COL2 |

4KB    256KB

database.db

C++

Parser
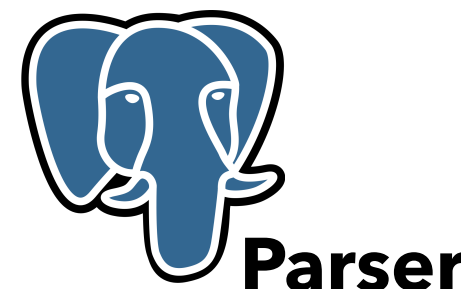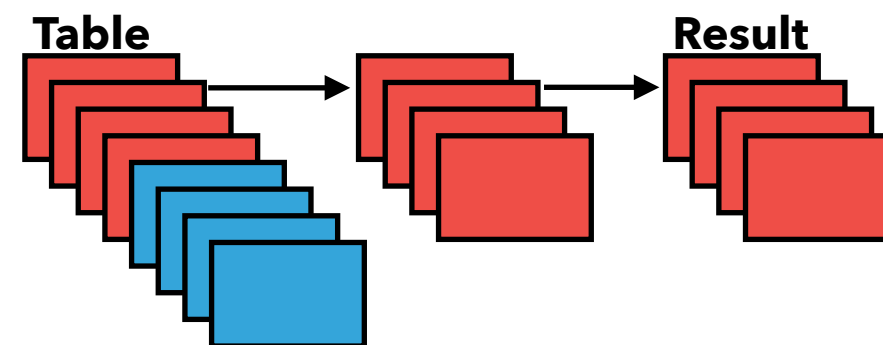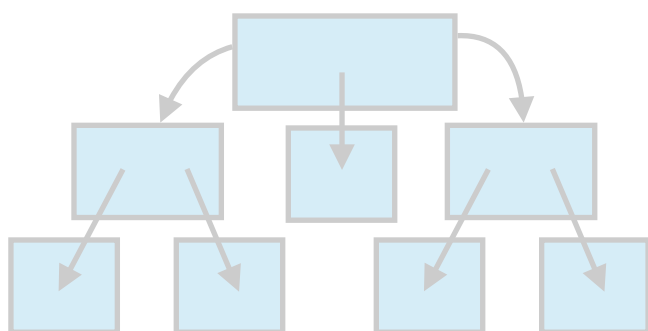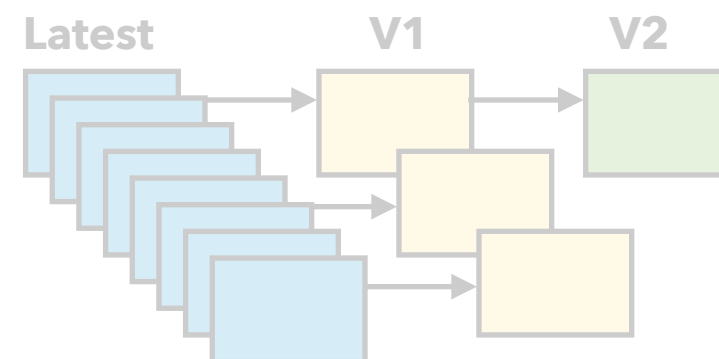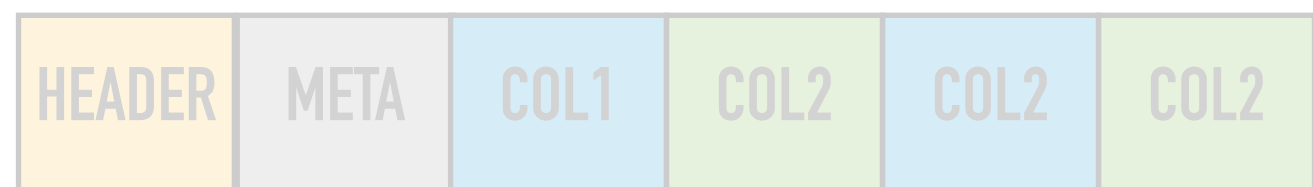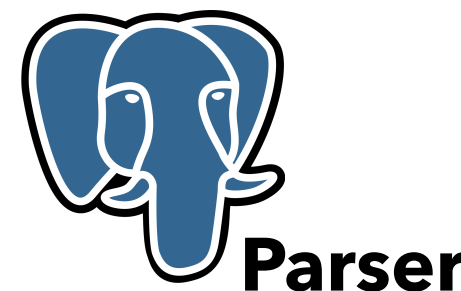
# Query Processing Pipeline

‣ **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='X'
      AND l_tax > 50;
```
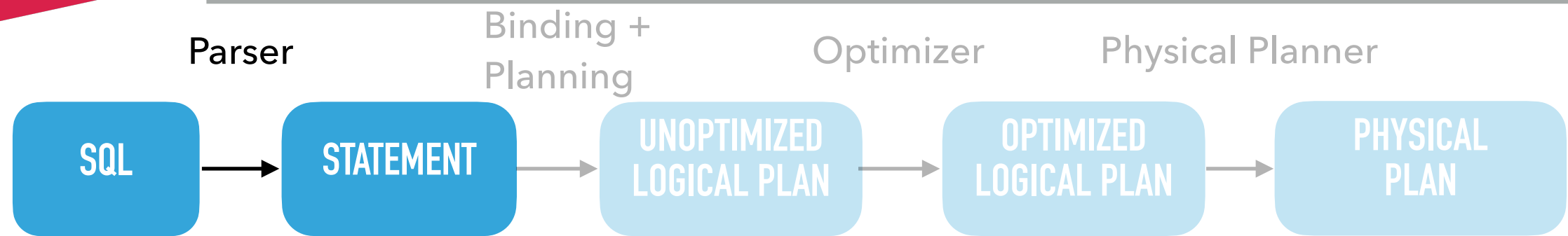
‣ **Aggregate:** COUNT(*)

‣ **Implicit join:** lineitem, orders **on** orderkey

‣ **Filters:** o_orderstatus='X' **and** l_tax>50

▸ DuckDB uses a typical pipeline for query processing

Parser        Binding +        Optimizer        Physical Planner
              Planning

| SQL | → | STATEMENT | → | UNOPTIMIZED LOGICAL PLAN | → | OPTIMIZED LOGICAL PLAN | → | PHYSICAL PLAN |

▸ 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;
```
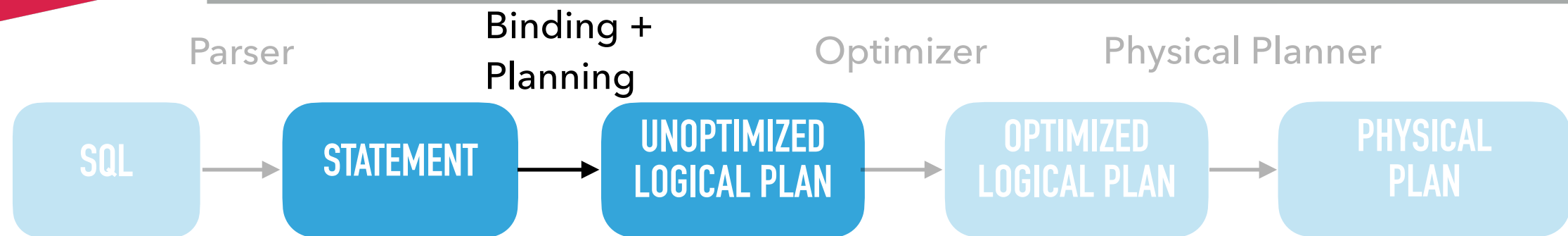
▸ The result of the **parsing** stage is the following:
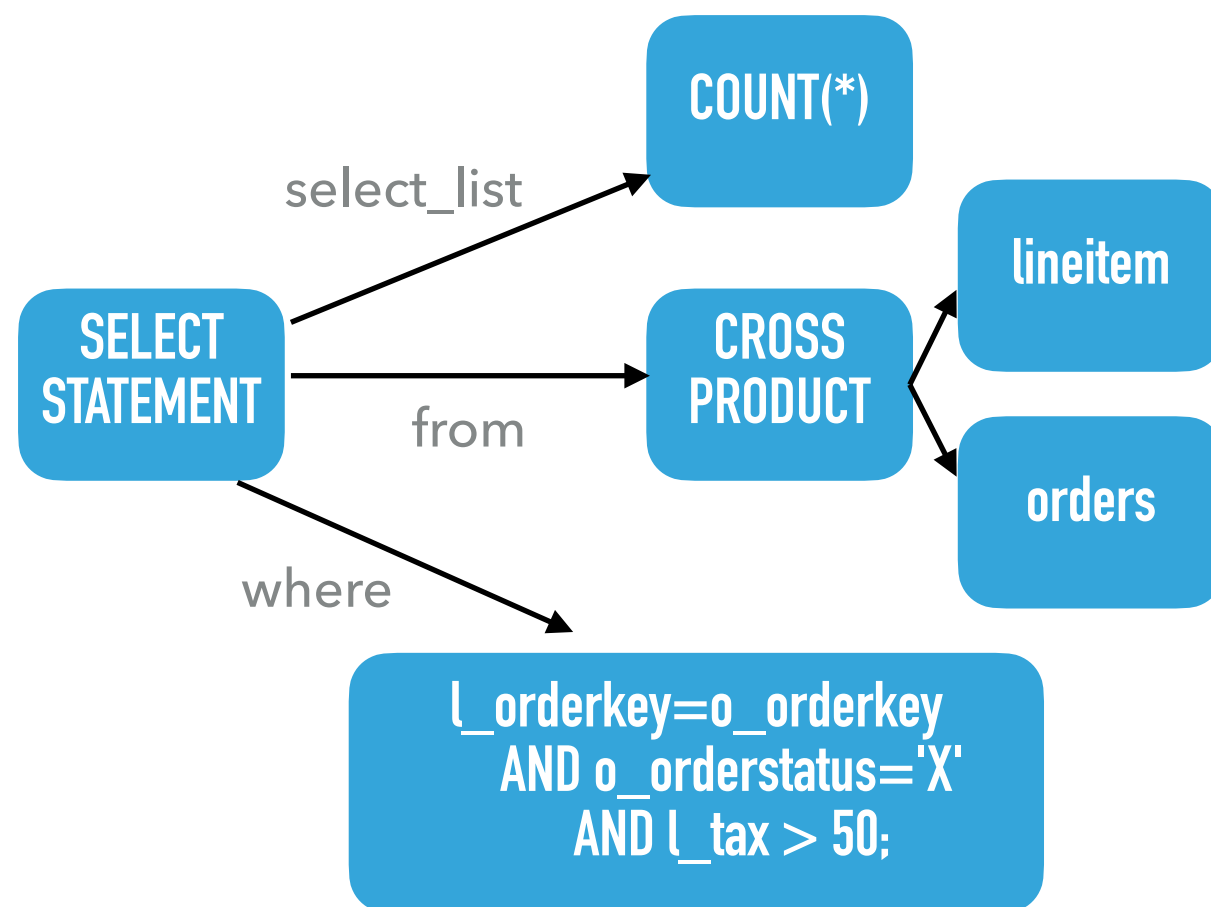


```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

▸ 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;
```

▸ <u>Note</u>: table/column names *are not resolved yet*

  ▸ e.g. if **lineitem** table does not exist, no error will be thrown yet

Parser | Binding + Planning | Optimizer | Physical Planner

| SQL | → | STATEMENT | → | UNOPTIMIZED LOGICAL PLAN | → | OPTIMIZED LOGICAL PLAN | → | PHYSICAL PLAN |

## ▸ **Binding phase**

- ▸ Catalog lookup of table/column names
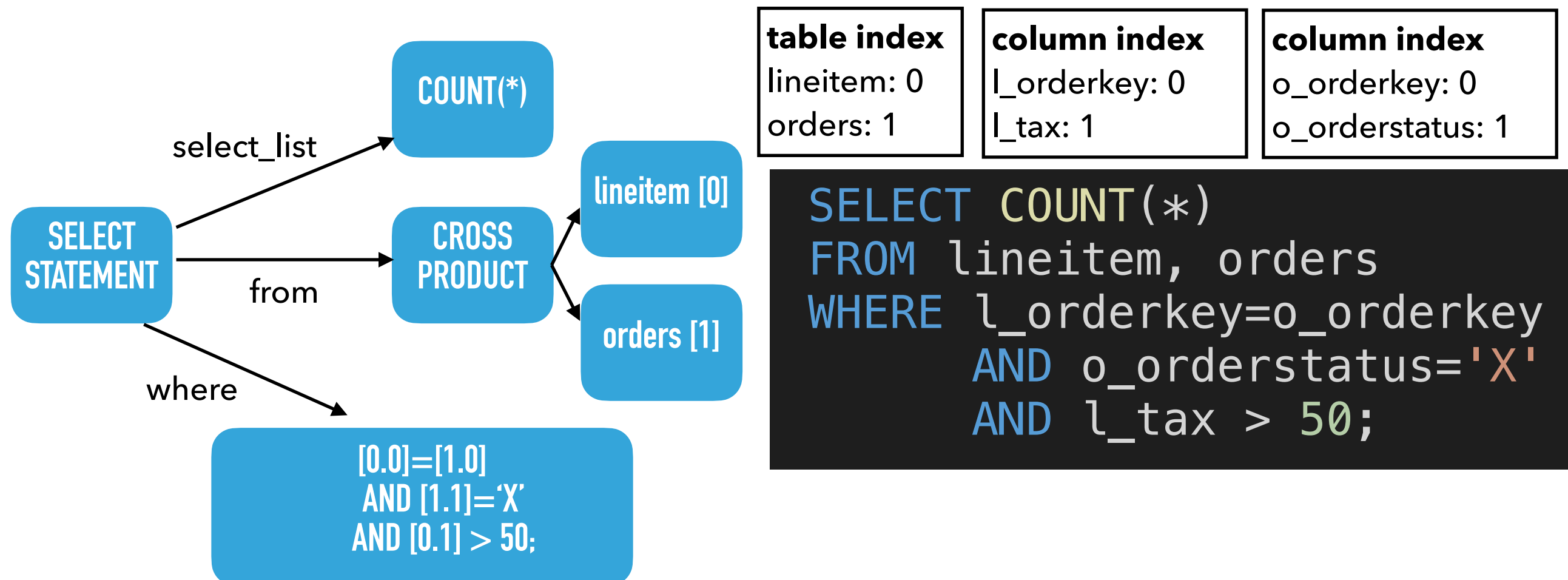
- ▸ Type resolution of expressions

```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```
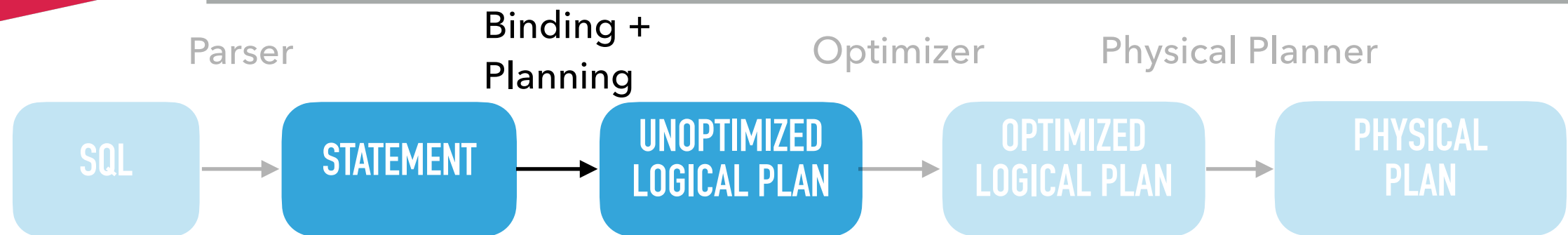
▸ Look up tables `lineitem` and `orders` tables

▸ 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;
```
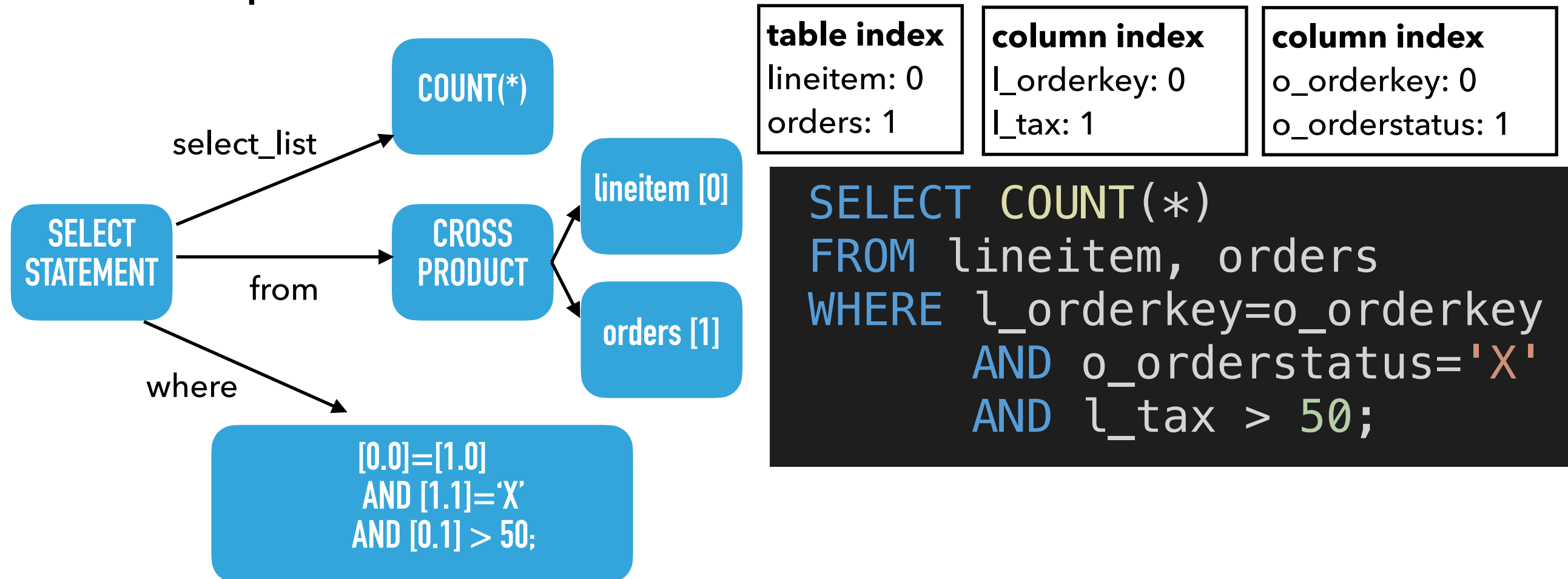
▸ Replace table names with **table indexes**

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

▸ **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 |

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

COUNT(*)

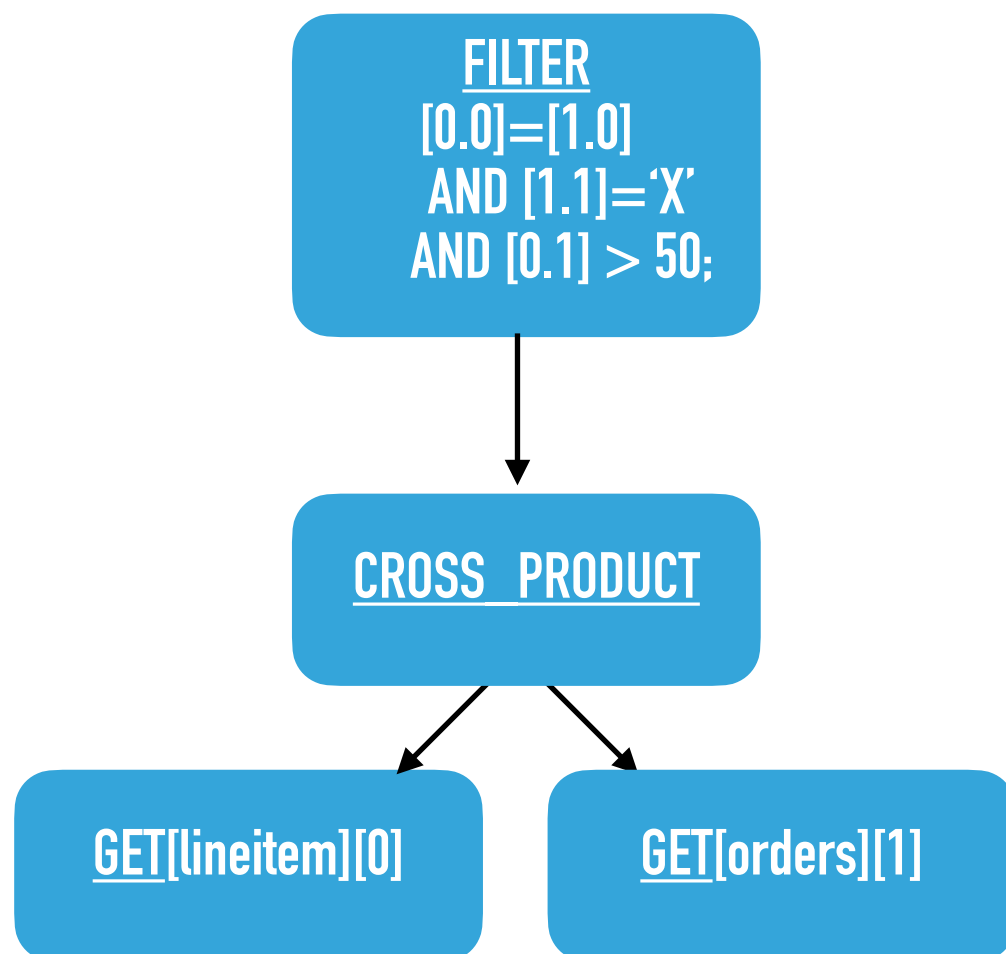SELECT STATEMENT

select_list

from

where

CROSS PRODUCT

lineitem [0]

orders [1]

[0.0]=[1.0]
AND [1.1]='X'
AND [0.1] > 50;

Parser | Binding + Planning | Optimizer | Physical Planner

| SQL | → | STATEMENT | → | UNOPTIMIZED LOGICAL PLAN | → | OPTIMIZED LOGICAL PLAN | → | PHYSICAL PLAN |

▸ **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='X'
      AND l_tax > 50;
```

- ▶ 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]

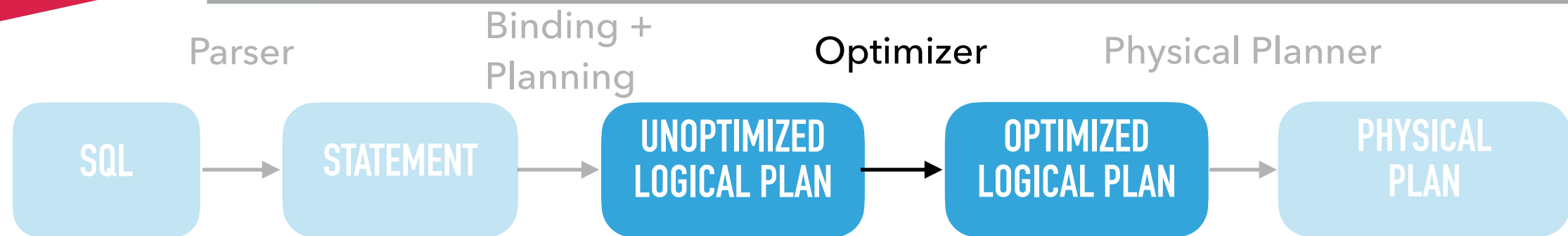```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

- The tables are combined with a **cross product**

  - There is no explicit join here

- The optimizer will later convert this into a join
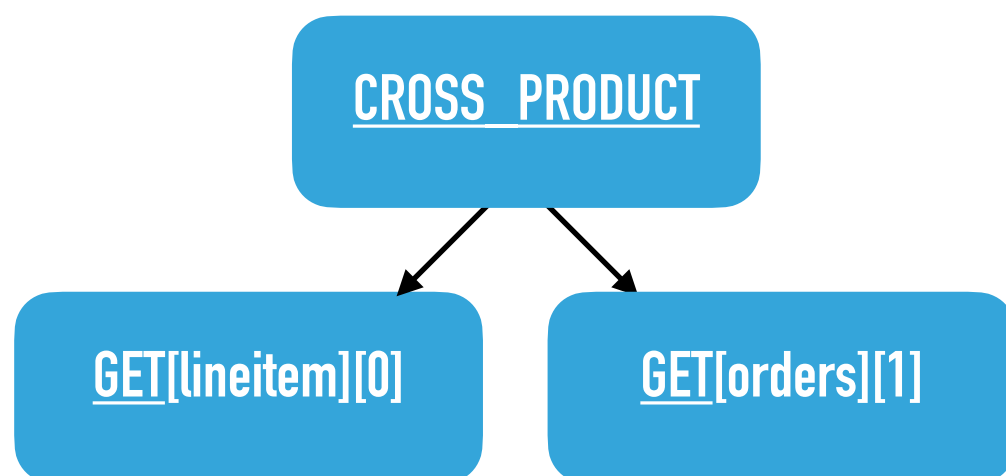
```
                    CROSS_PRODUCT

        GET[lineitem][0]        GET[orders][1]
```

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

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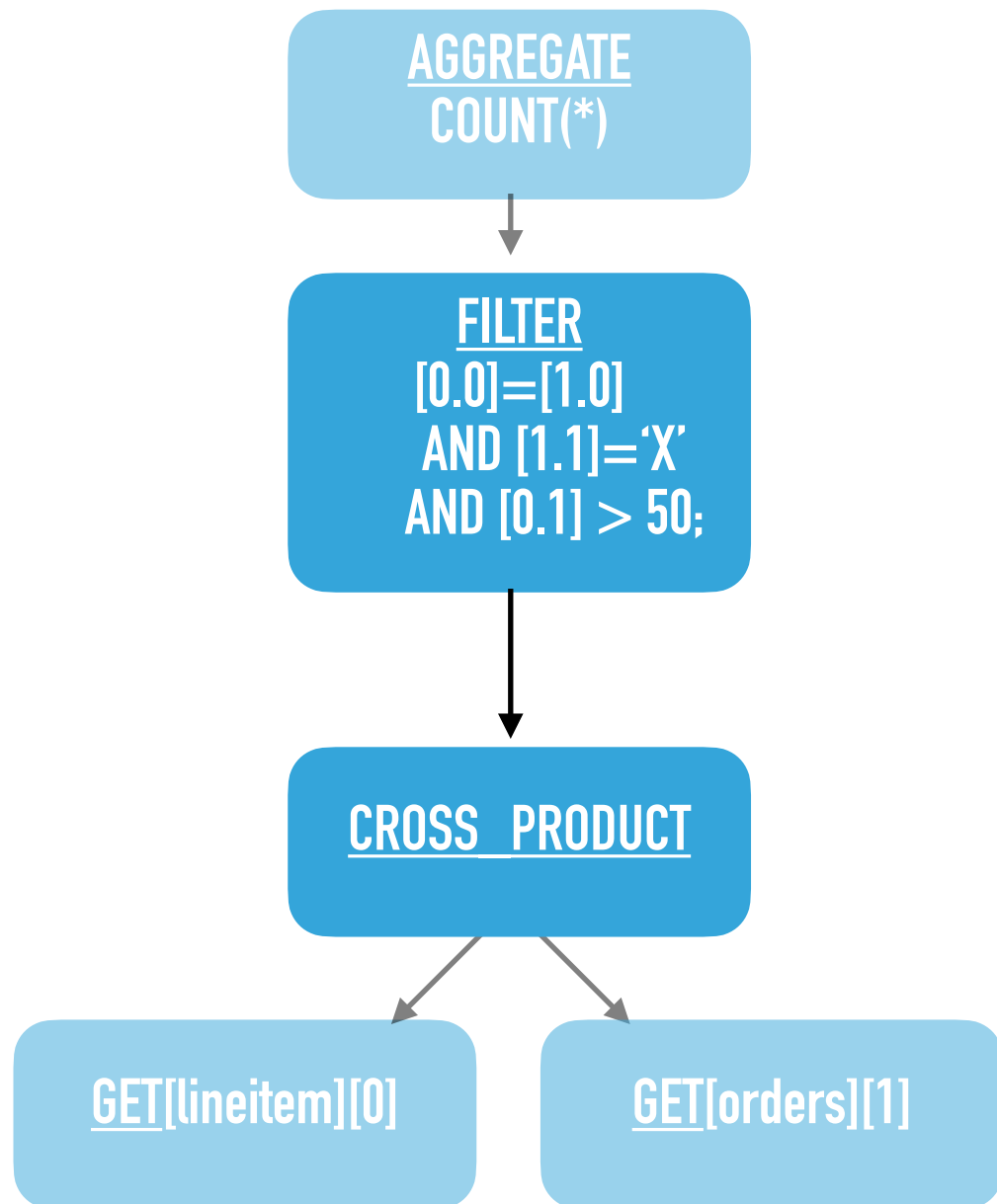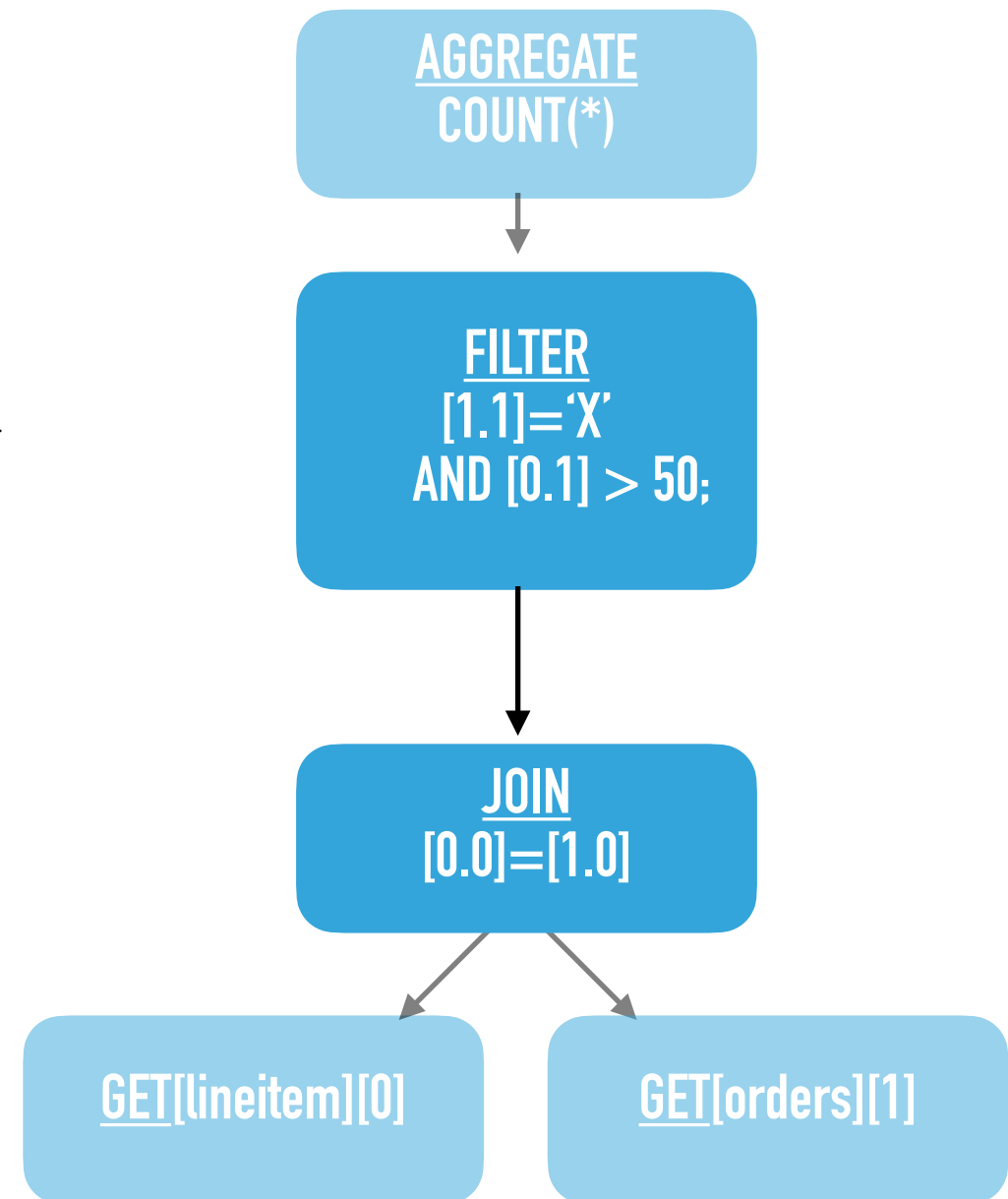
  ▸ Filter has single big expression

```
FILTER
[0.0]=[1.0]
AND [1.1]='X'
AND [0.1] > 50;
```

```
CROSS_PRODUCT
```

```
GET[lineitem][0]
```

```
GET[orders][1]
```

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

▶ Finally add aggregate computation

Logical Query Tree
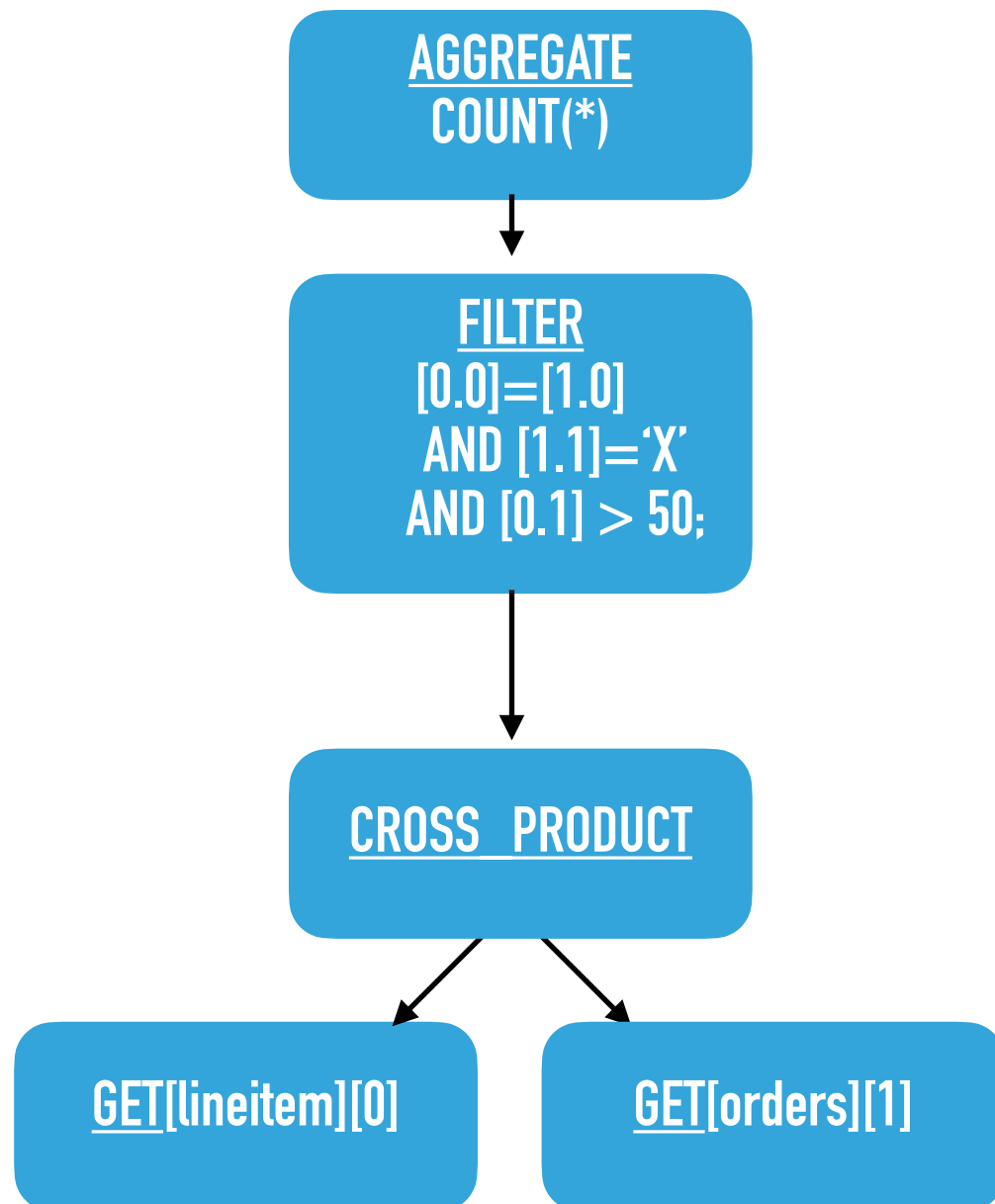


```sql
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

Parser — Binding + Planning — Optimizer — Physical Planner

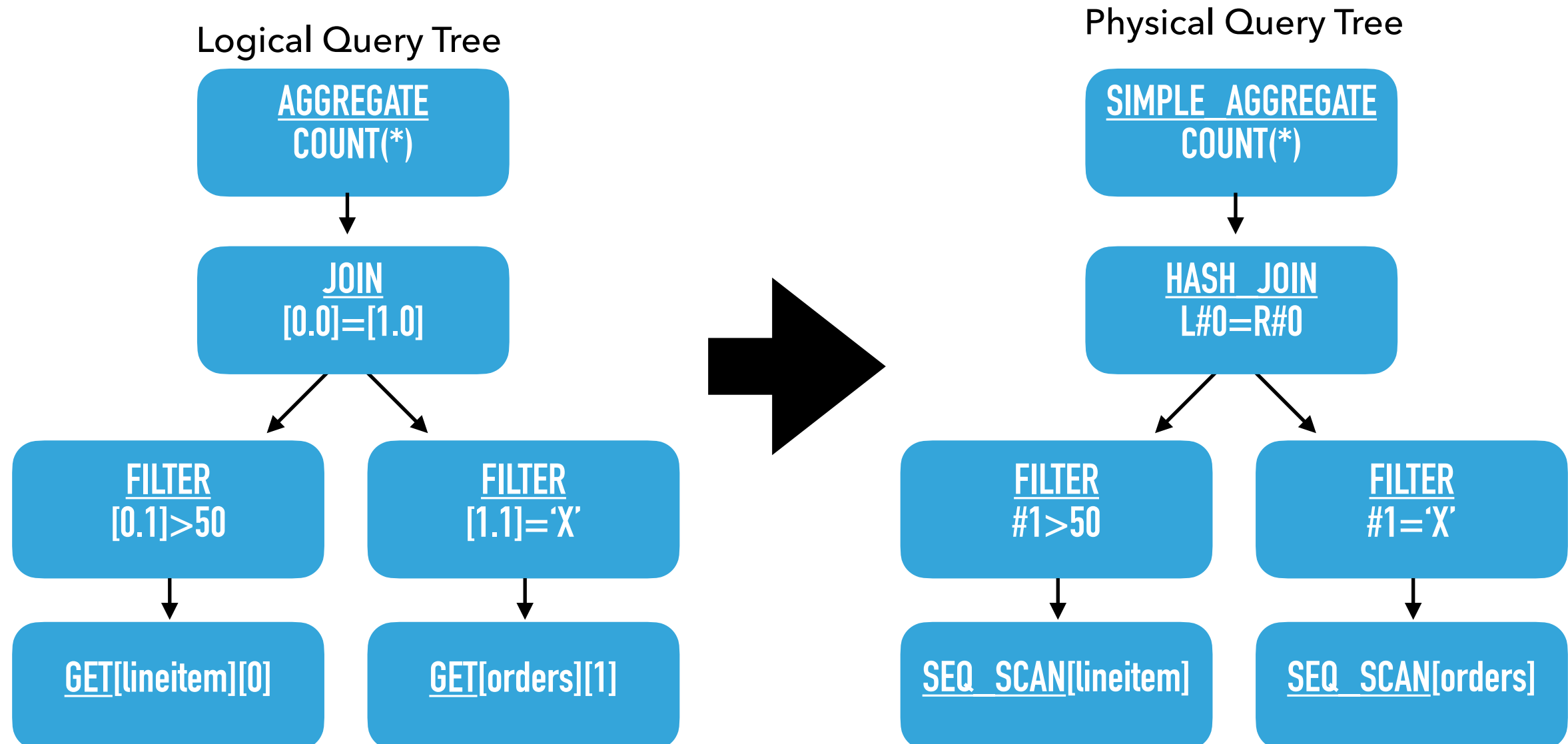SQL → STATEMENT → UNOPTIMIZED LOGICAL PLAN → OPTIMIZED LOGICAL PLAN → PHYSICAL PLAN

▸ **Optimizer:** transforms the logical query tree

▸ Created plan is logically equivalent

   ▸ But (hopefully) faster

```sql
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

Logical Query Tree

**AGGREGATE**
COUNT(*)

**FILTER**
[0.0]=[1.0]
AND [1.1]='X'
AND [0.1] > 50;

**CROSS_PRODUCT**

GET[lineitem][0]

GET[orders][1]

Logical Query Tree

**AGGREGATE**
COUNT(*)

**FILTER**
[1.1]='X'
AND [0.1] > 50;

**JOIN**
[0.0]=[1.0]

GET[lineitem][0]

GET[orders][1]

# CWI DuckDB
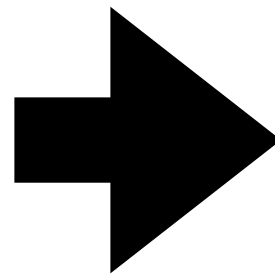
▸ Pushdown filters below the join
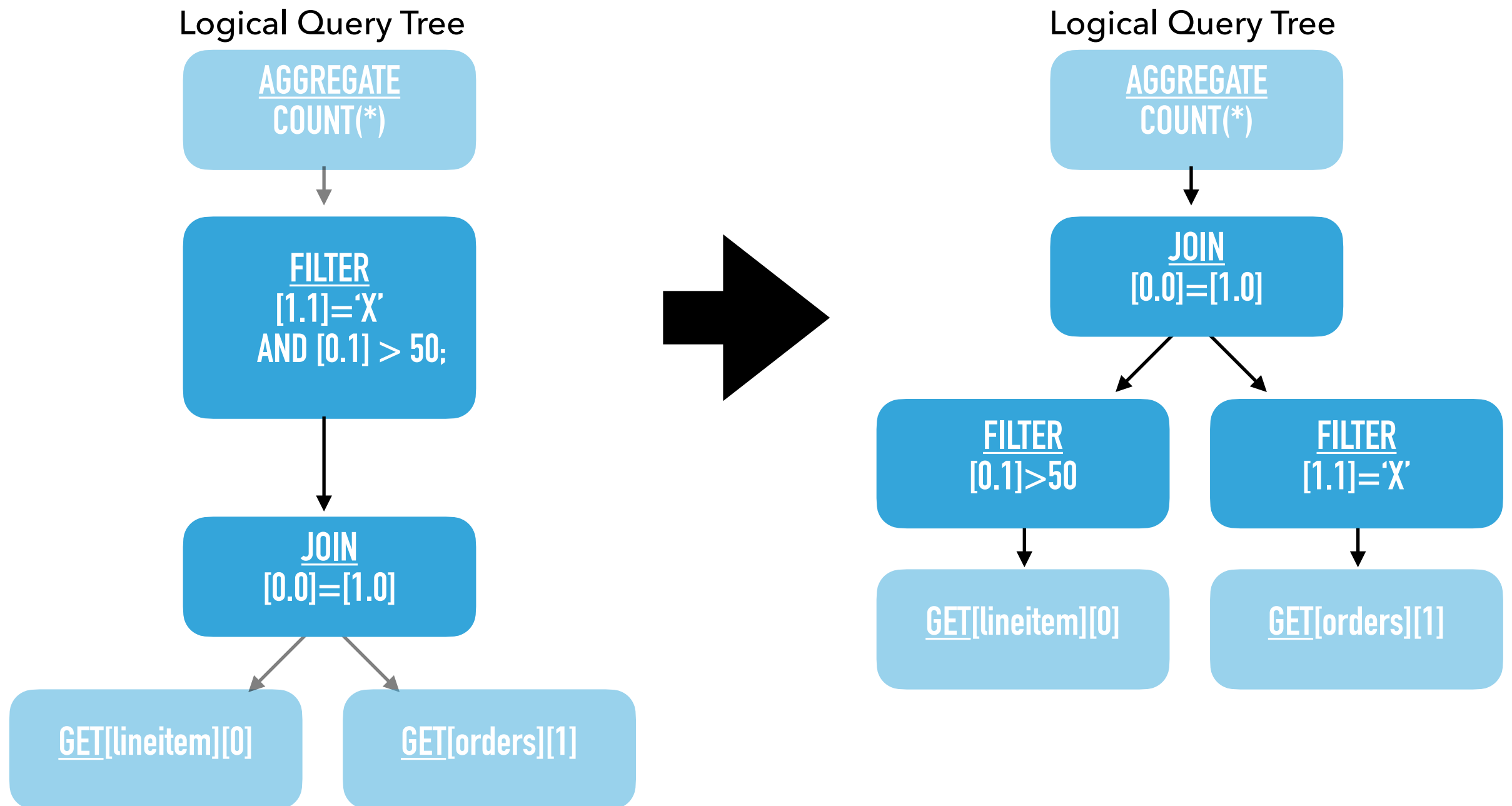
# DuckDB

| SQL | → | STATEMENT | → | BOUND STATEMENT | → | UNOPTIMIZED LOGICAL PLAN | → | OPTIMIZED LOGICAL PLAN | → | PHYSICAL PLAN |

▸ 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

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

SQL → STATEMENT → BOUND STATEMENT → UNOPTIMIZED LOGICAL PLAN → OPTIMIZED LOGICAL PLAN → PHYSICAL PLAN

▸ **Physical planner:** converts logical plan into physical (executable) plan

▸ Makes decision on implementations of operators

  ▸ e.g. use a HashJoin, MergeJoin or NestedLoopJoin

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

# DuckDB

- **SimpleAggregate:** no groups, no hash required

- **Hash Join:** Most effective for this equality join

- **Sequential Scan:** No index that helps us speed up

Logical Query Tree

```
AGGREGATE
COUNT(*)
   │
   ▼
  JOIN
[0.0]=[1.0]
```

| FILTER [0.1]>50 | FILTER [1.1]='X' |
| --- | --- |
| GET[lineitem][0] | GET[orders][1] |

Physical Query Tree

```
SIMPLE_AGGREGATE
COUNT(*)
   │
   ▼
HASH_JOIN
L#0=R#0
```

| FILTER #1>50 | FILTER #1='X' |
| --- | --- |
| SEQ_SCAN[lineitem] | SEQ_SCAN[orders] |

▸ Now we have the final query tree

   ▸ This is what we **execute** to run the query

Physical Query Tree
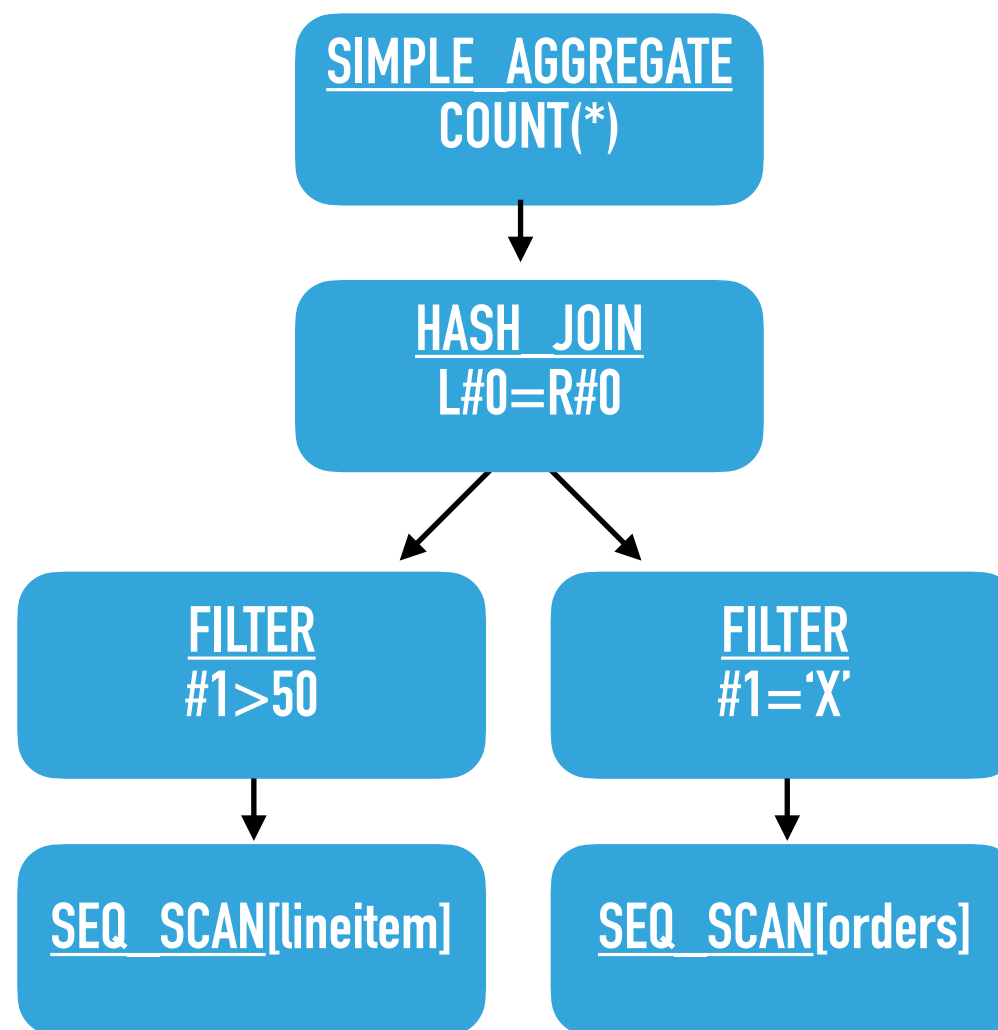
# Query Execution

# 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

**Vectorized Processing**

Table                    Result
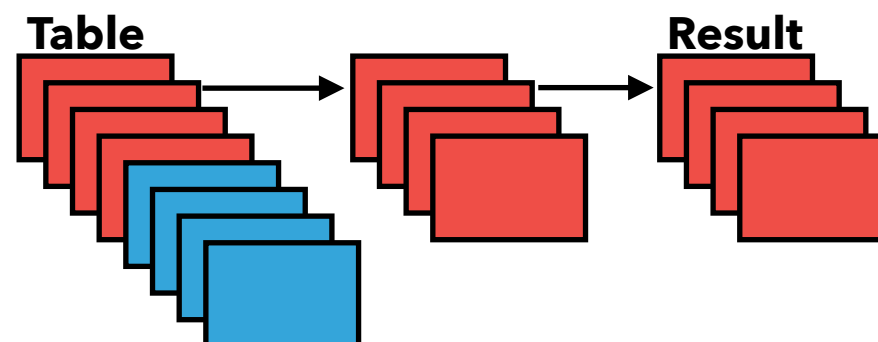
- ▸ Start with root node: **SimpleAggregate**

  - ▸ Aggregate without groups

- ▸ Immediately calls `GetChunk` on child
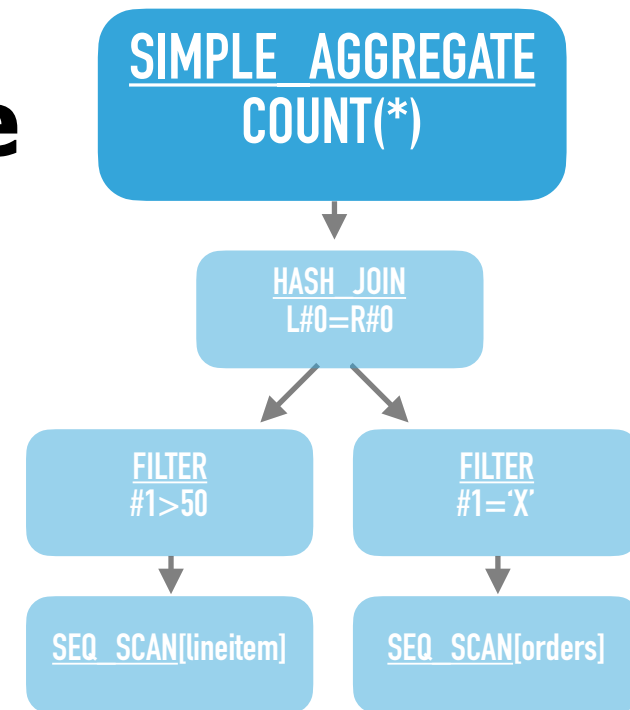
▸ **Hash Join**

▸ Start by **building HT**

▸ Call `GetChunk` on right node



SIMPLE_AGGREGATE
COUNT(*)

HASH_JOIN
L#0=R#0

FILTER
#1>50

FILTER
#1='X'

SEQ_SCAN[lineitem]

SEQ_SCAN[orders]
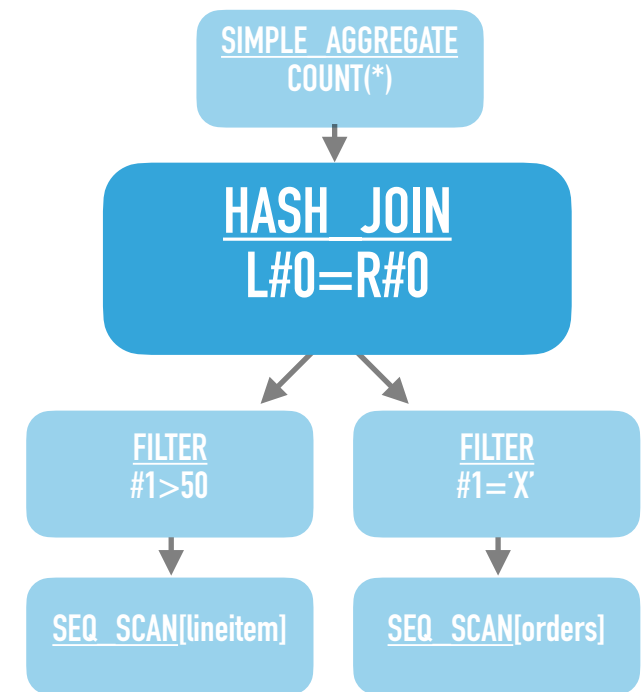
▸ **Filter**

▸ Again, pull a chunk from child
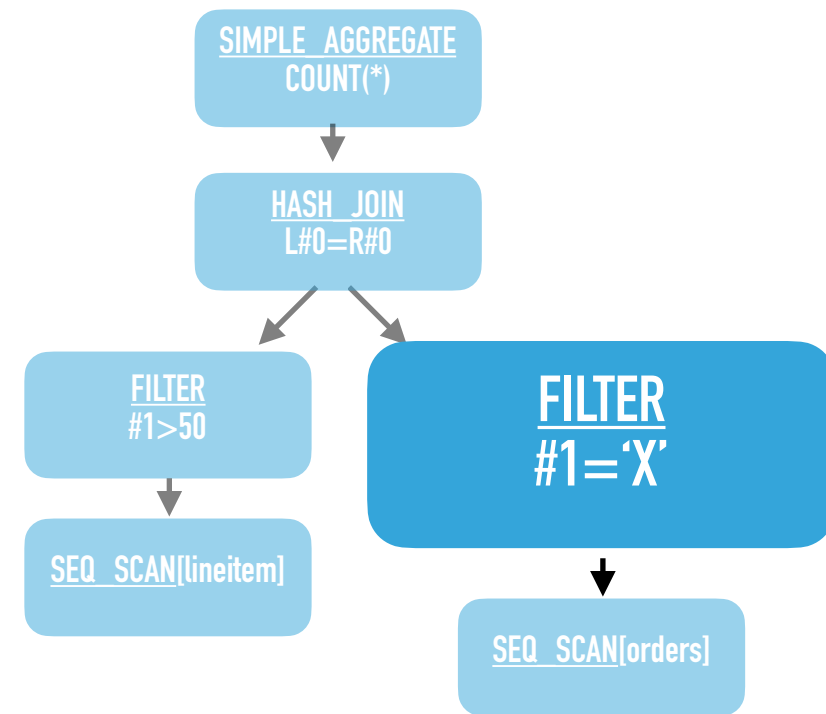
▸ **Sequential Scan**

▸ Finally we can start executing

▸ Scan the base table

▸ Return a DataChunk with two columns

　　▸ `o_orderkey` and `o_orderstatus`

SIMPLE_AGGREGATE
COUNT(*)

HASH_JOIN
L#0=R#0

FILTER
#1>50

FILTER
#1='X'

SEQ_SCAN[lineitem]

SEQ_SCAN[orders]

DataChunk

V1

V2

INTEGER

1
2
3
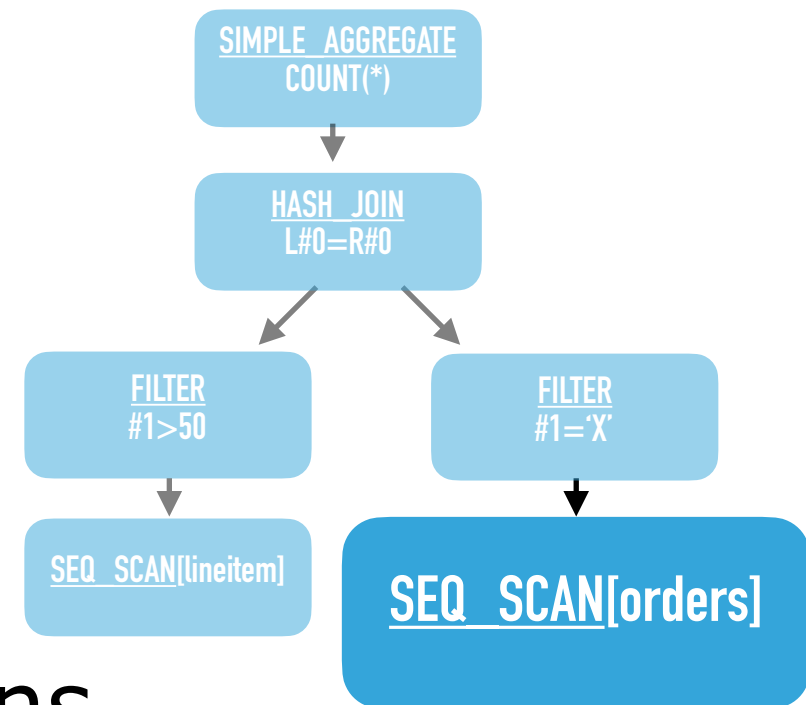
VARCHAR

N
X
N

▸ **Filter**

▸ Now we can perform the filter #1='X'

　　▸ Only the second tuple passes

▸ **Selection vector** pointing to surviving tuple is created

▸ Note that <u>no data is copied or changed</u>
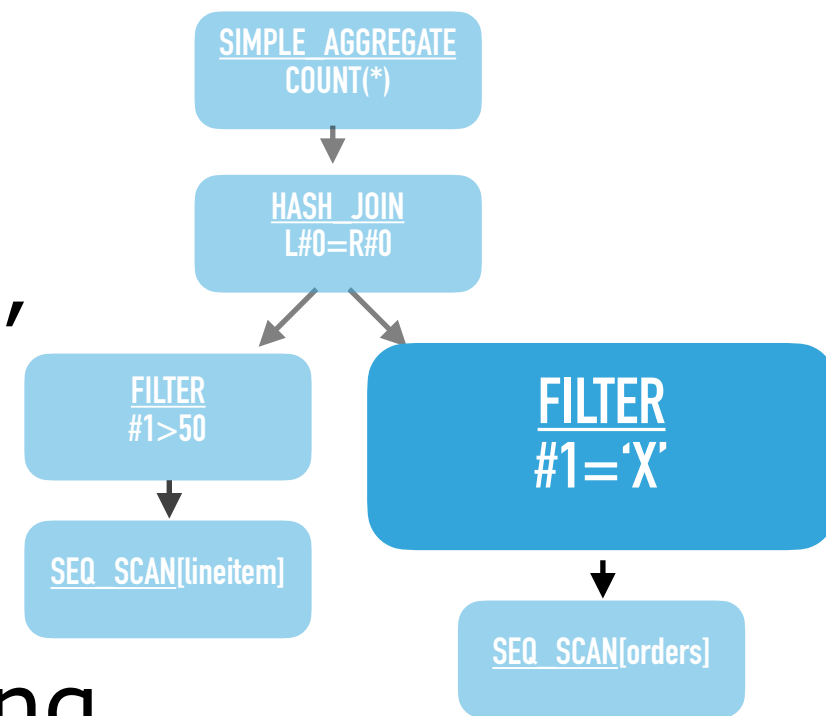
▸ **Hash Join**

▸ Now we have our first input chunk

▸ We input it into the HT

▸ Now we fetch another chunk from RHS

SIMPLE_AGGREGATE
COUNT(*)

HASH_JOIN
L#0=R#0

FILTER
#1>50

FILTER
#1='X'

SEQ_SCAN[lineitem]

SEQ_SCAN[orders]

**DataChunk**

V1 — INTEGER: 1 2 3

V2 — VARCHAR: N X N

SEL: 1
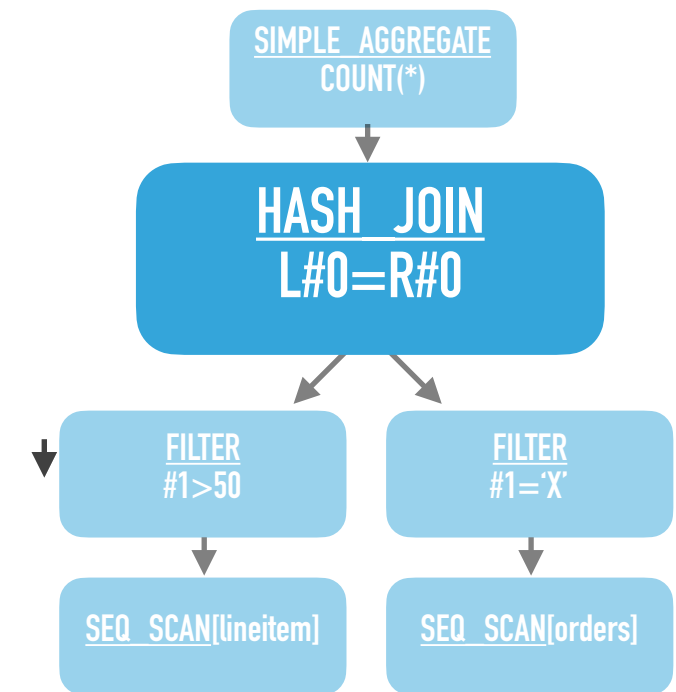
**Keys Payload Next**

| 2 | X | 0 |

▸ **Sequential Scan**

▸ The filter again calls `GetChunk`

▸ Scan base table again:

  ▸ The scan is finished, return empty chunk
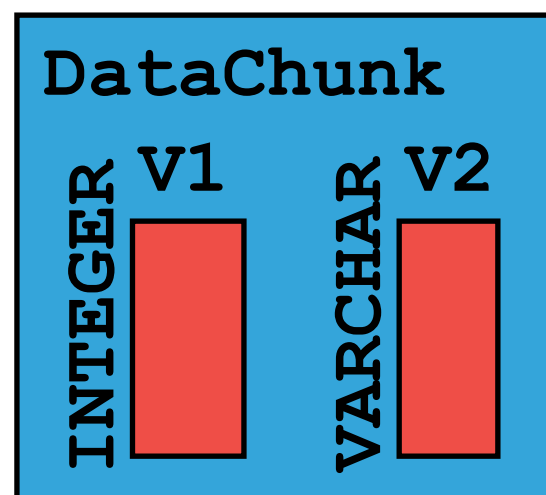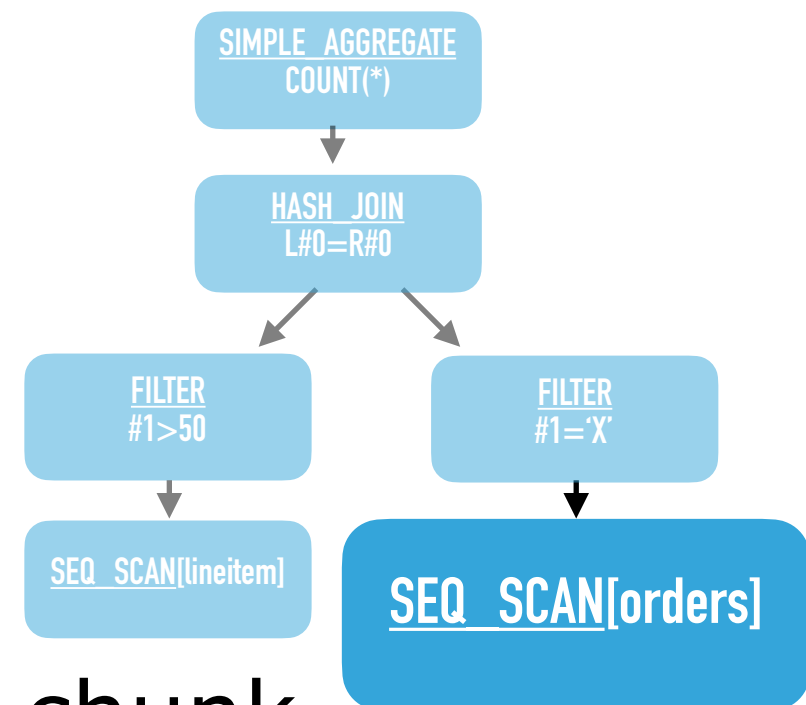
## **Hash Join**

- HT receives second input chunk

  - But it is empty!

- The RHS is exhausted

- Finish building HT and call `GetChunk` on LHS
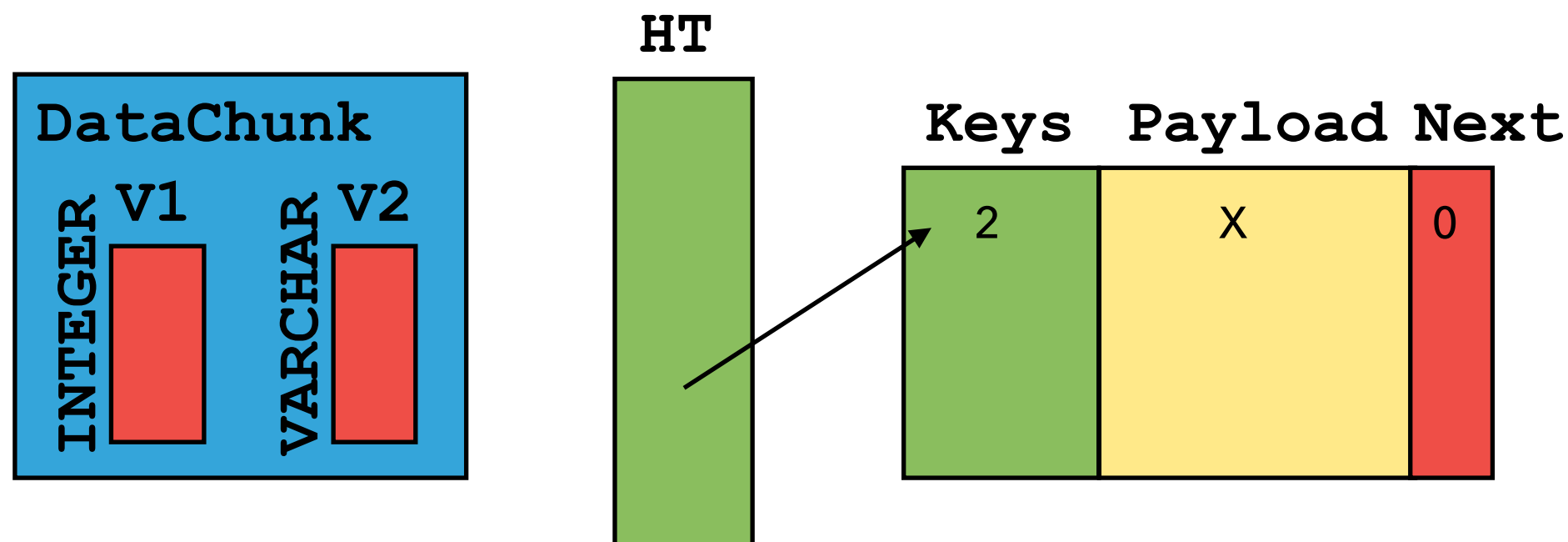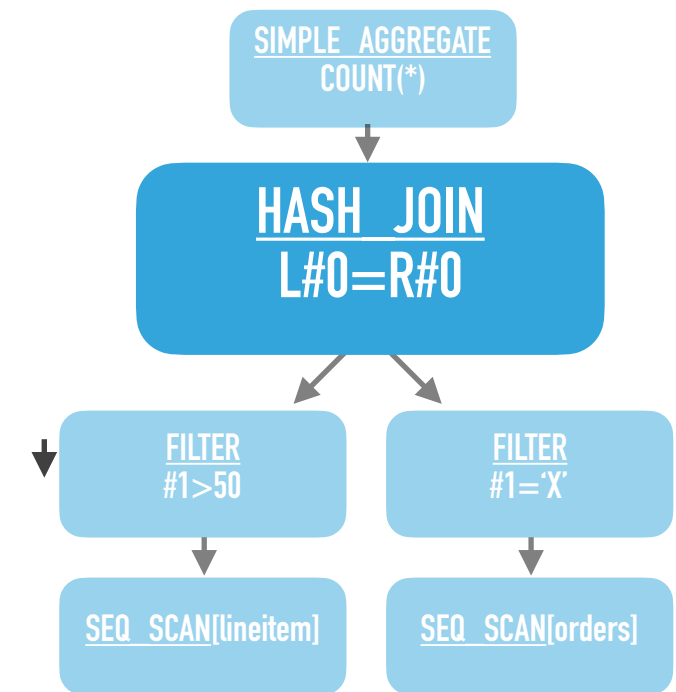
▸ **Sequential Scan**

▸ 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

SIMPLE_AGGREGATE
COUNT(*)

HASH_JOIN
L#0=R#0

FILTER
#1>50

FILTER
#1='X'

SEQ_SCAN[lineitem]

SEQ_SCAN[orders]

DataChunk

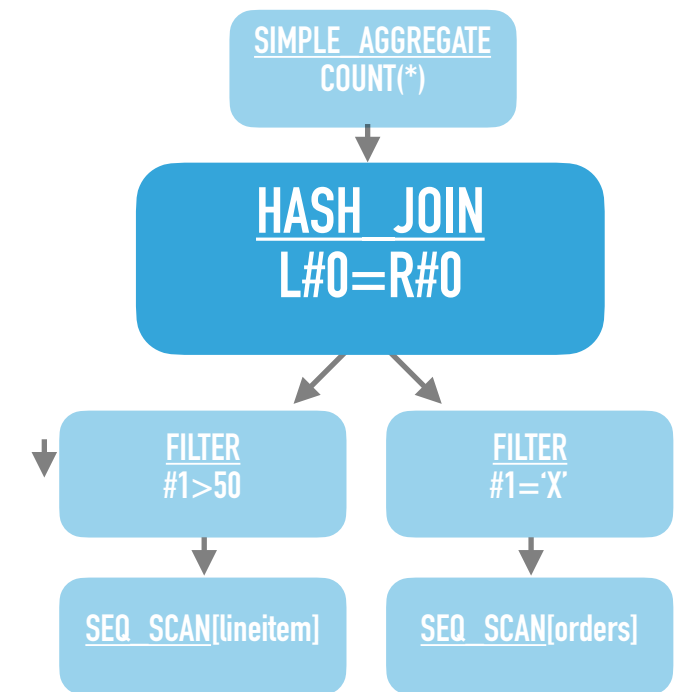INTEGER V1
1
2
2

VARCHAR V2
80
60
20

SEL
0
1
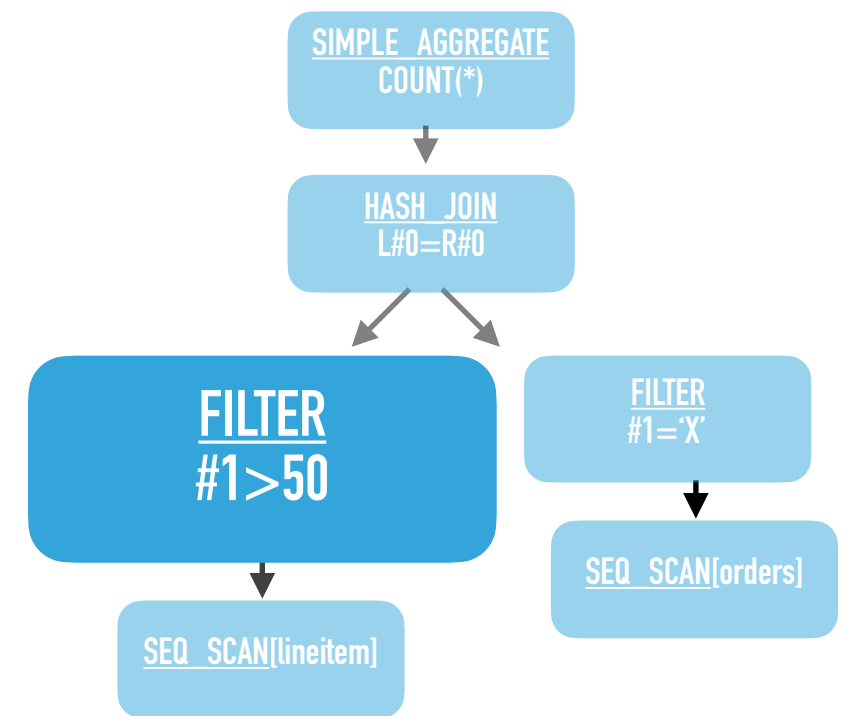
### **Hash Join**

- Now it is time to probe the HT

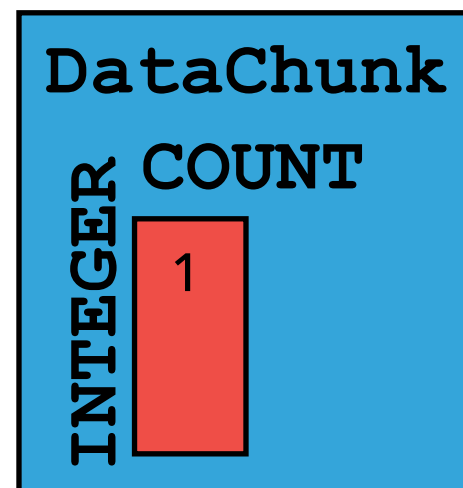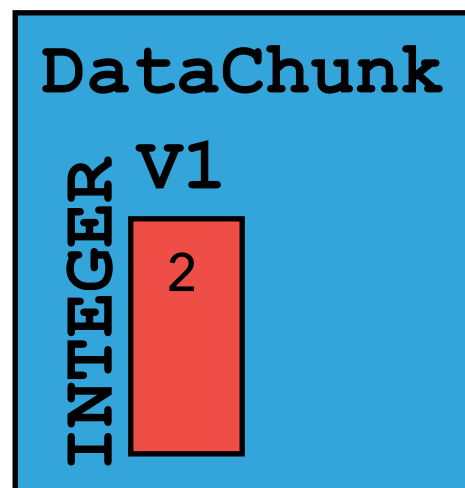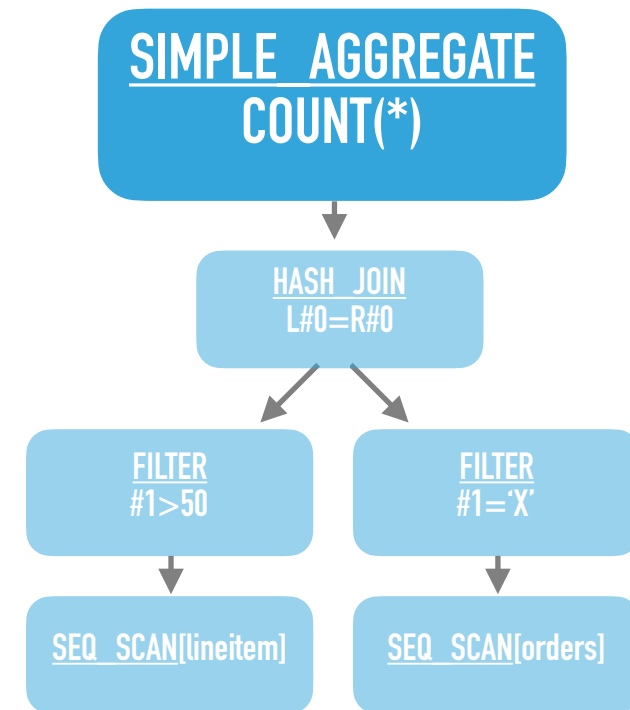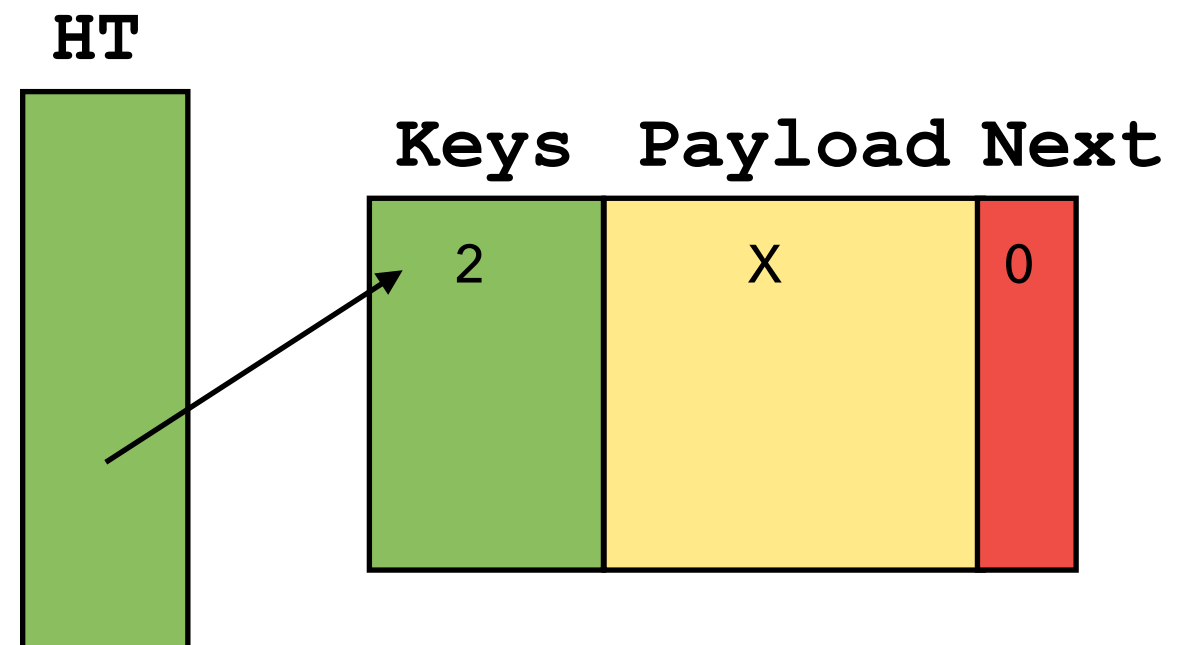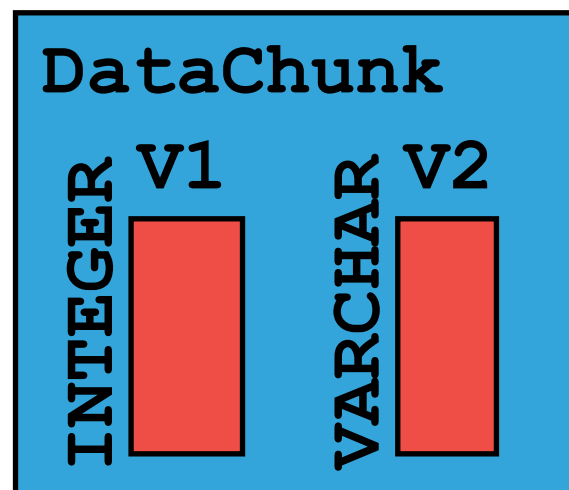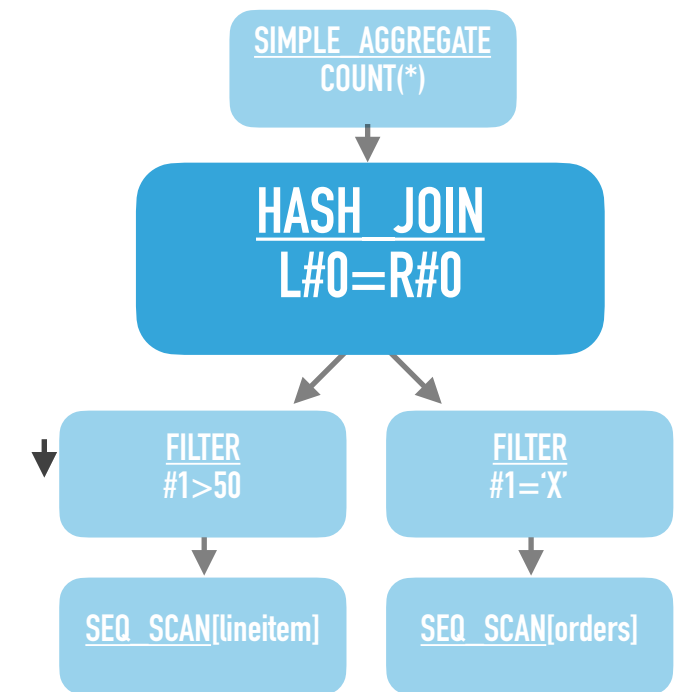- We compute the hash for each tuple

- Then lookup in the HT
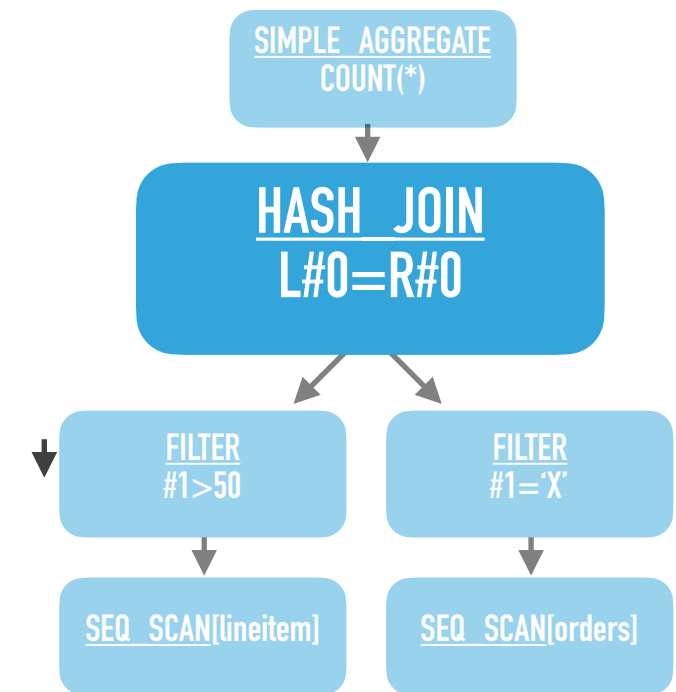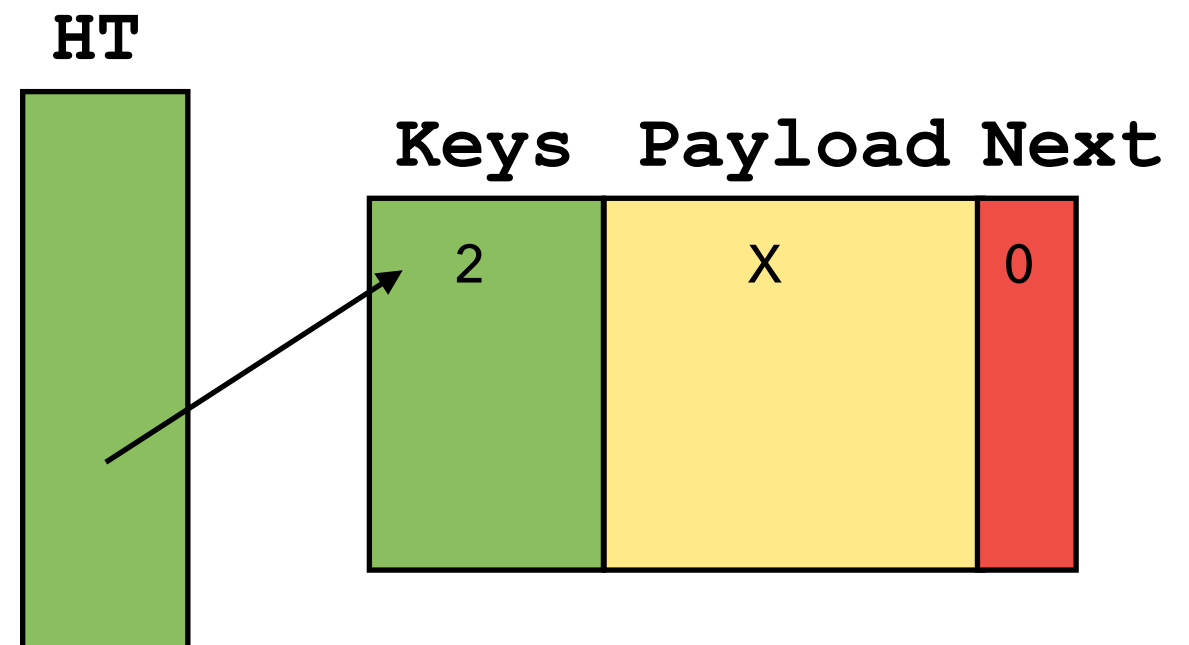
## **Hash Join**

▶ We get one hit on our join!

▶ The hash join now produces the result

▶ We return this to the aggregate
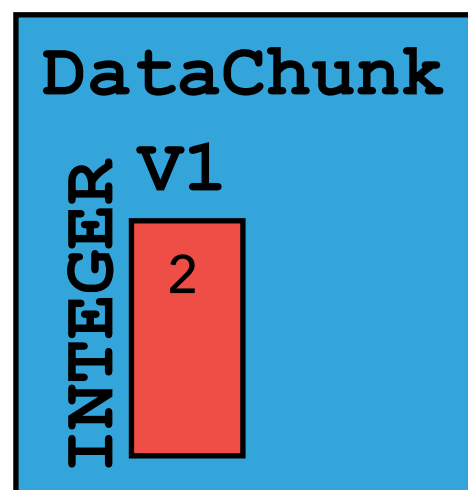
▸ The aggregate takes our input chunk
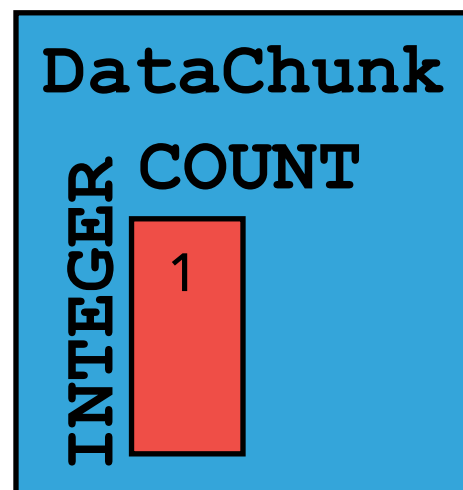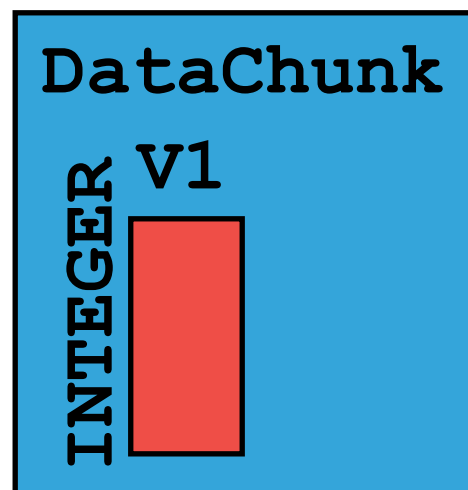
▸ Updates the aggregate

▸ Then fetches from the child again

SIMPLE_AGGREGATE
COUNT(*)

HASH_JOIN
L#0=R#0

FILTER
#1>50

FILTER
#1='X'

SEQ_SCAN[lineitem]

SEQ_SCAN[orders]

**DataChunk**
INTEGER V1
2

**DataChunk**
INTEGER COUNT
1

- 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

**SIMPLE_AGGREGATE**
COUNT(*)

HASH_JOIN
L#0=R#0

FILTER
#1>50

FILTER
#1='X'

SEQ_SCAN[lineitem]

SEQ_SCAN[orders]

**DataChunk**
INTEGER V1

**DataChunk**
INTEGER COUNT
1
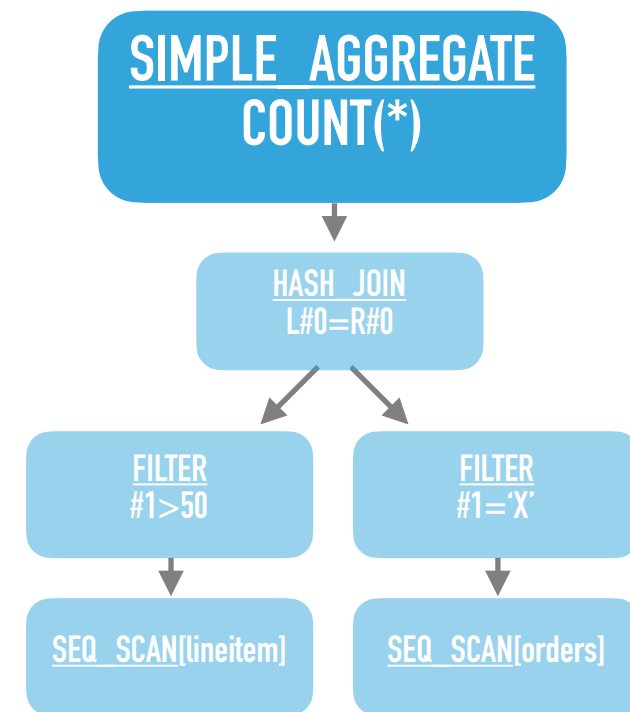
# Hands On

▸ **Assignment**: Hand-optimize a query in DuckDB

▸ We will disable the query optimizer, and allow you to

▸ Implement one of those

　　▸ For those that are successful, submit a PR!

▸ **Slides are online**

▸ [https://github.com/pdet/duckdb-tutorial](https://github.com/pdet/duckdb-tutorial)

▸ Feel free to ask any questions!