Mark Raasveldt & Pedro Holanda

DuckDB
an Embeddable Analytical RDBMS

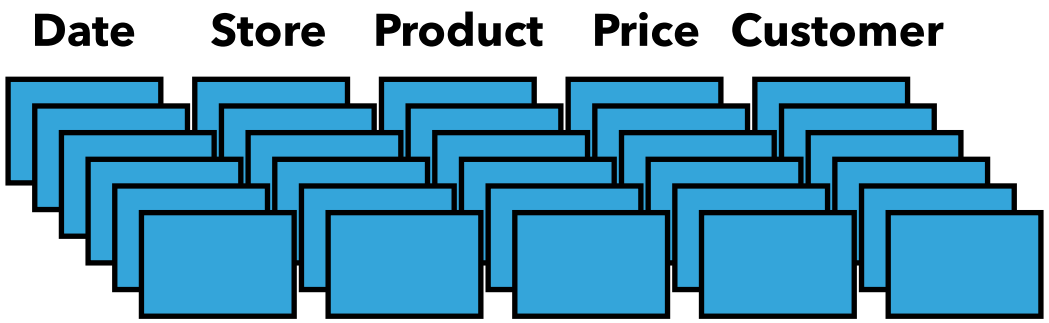
Outline

- Internals at a Glance
  - Column-Store
  - MVCC
  - ART Index
  - Storage
- Query processing pipeline
- Query execution
- **Hands-On**

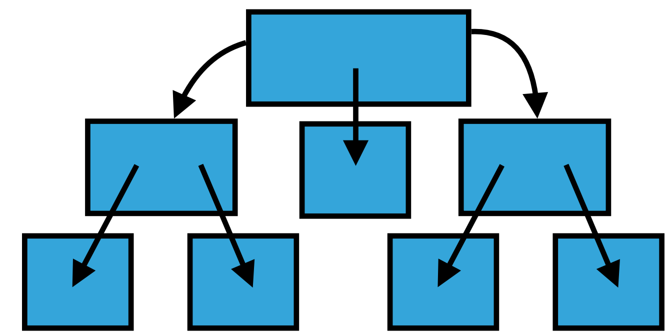
Internals at a Glance

Internals at a Glance

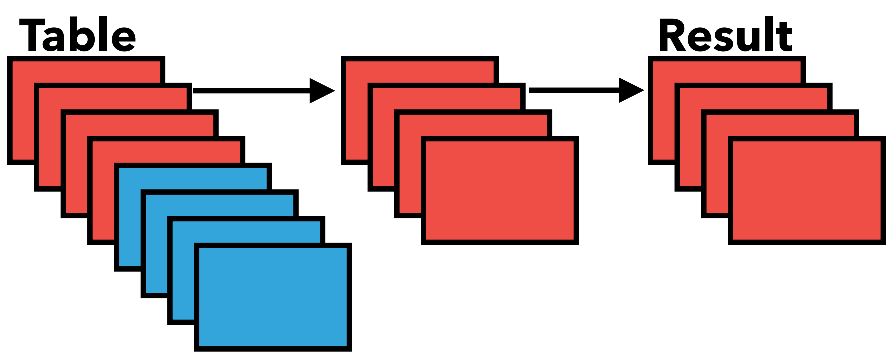
Column-Store



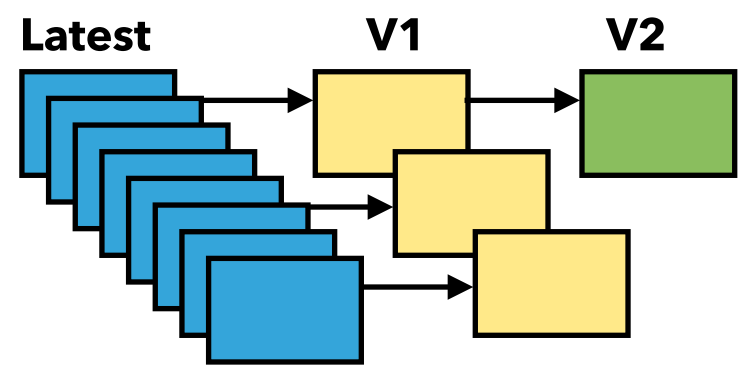
ART Index



Vectorized Processing



MVCC



Single-File Storage

database.db

Parser

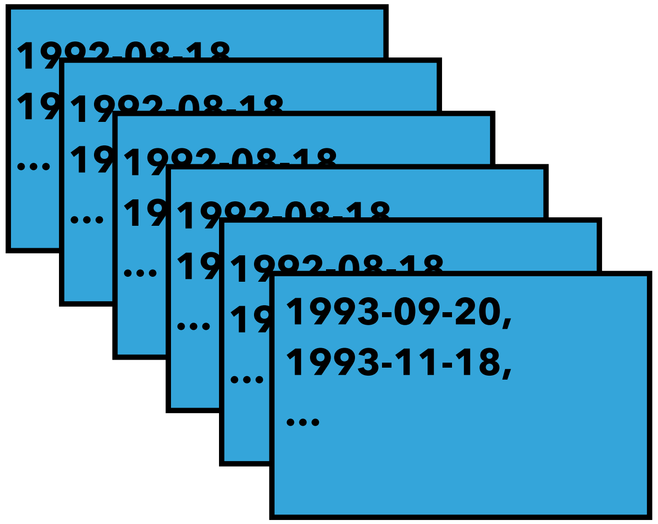
Column-Store

- DuckDB **vertically partitions** tables
  - This is also called a **column-store**
- **Advantages:**
  - Allows individual columns to be fetched/updated
  - Better compression ratio
  - Faster scans
- **Disadvantages:**
  - Less efficient fetch/update of individual rows

Column-Store

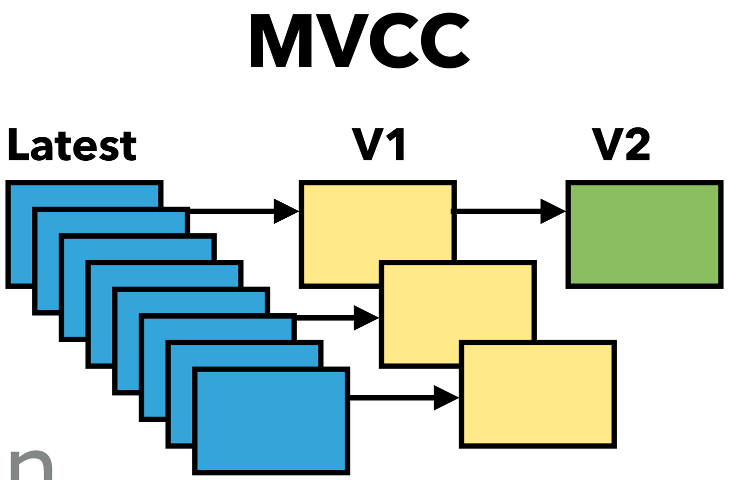
- Columns are again partitioned into **blocks**
- Blocks are **fixed-size** slices of a column
  - Currently *256KB*
- Individually scanned/decompressed
- Blocks have **min/max** indices (zonemaps)
  - Allow for skipping of blocks

Date



MVCC

- MVCC is used to provide **snapshot isolation**
- Data is updated **in place**
  - Avoids fragmentation
- Old versions are kept in version chain
  - Deleted once no longer required
- Version chain starts out empty
  - No overhead unless versions exist



MVCC

Two transactions: T1 T2

	Date
0	1992-01-01
1	1992-01-02
2	1992-01-03
3	1992-01-04
4	1992-01-05
5	1992-01-06
6	1992-01-07
7	1992-01-08
8	1992-01-09
9	1992-01-10
10	1992-01-11

MVCC

Two transactions: **T1** **T2**

Date

0	1992-01-01
1	1992-01-02
2	1992-01-03
3	1992-01-04
4	**1992-01-13**
5	1992-01-06
6	**1992-01-15**
7	1992-01-08
8	1992-01-09
9	1992-01-10
10	1992-01-11

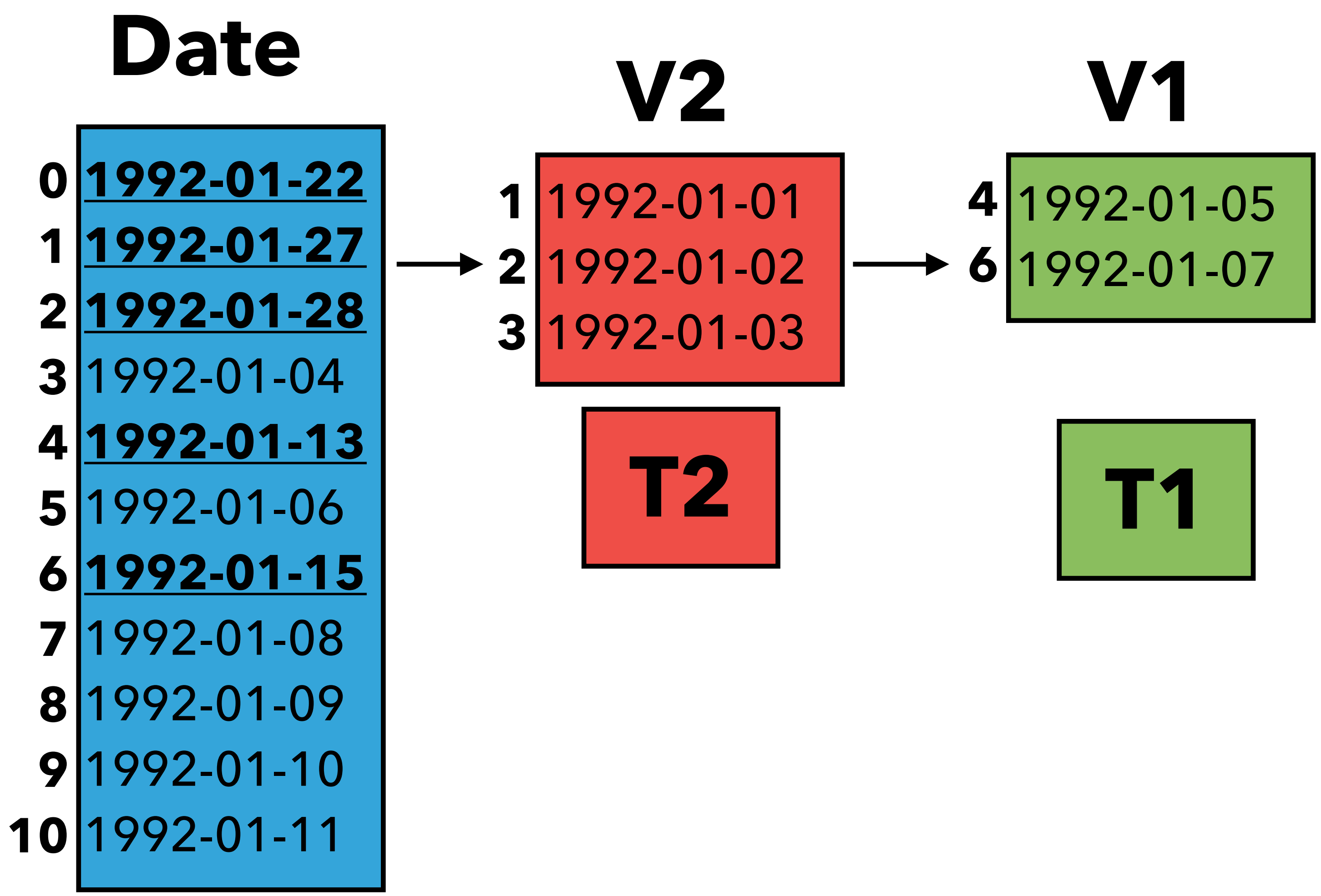
V1

4	1992-01-05
6	1992-01-07

T1

MVCC

Two transactions: T1 T2



MVCC

Two transactions: T1 T2

	Date	V2 (T2)	V1 (T1)	T1	T2
0	**1992-01-22**			1992-01-01	**1992-01-22**
1	**1992-01-27**	1992-01-01		1992-01-02	**1992-01-27**
2	**1992-01-28**	1992-01-02		1992-01-03	**1992-01-28**
3	1992-01-04	1992-01-03		1992-01-04	1992-01-04
4	**1992-01-13**		1992-01-05	**1992-01-13**	1992-01-05
5	1992-01-06			1992-01-06	1992-01-06
6	**1992-01-15**		1992-01-07	**1992-01-15**	1992-01-07
7	1992-01-08			1992-01-08	1992-01-08
8	1992-01-09			1992-01-09	1992-01-09
9	1992-01-10			1992-01-10	1992-01-10
10	1992-01-11			1992-01-11	1992-01-11

CWI

MVCC

After commit of **T1**

Date

0	**1992-01-22**
1	**1992-01-27**
2	**1992-01-28**
3	1992-01-04
4	**1992-01-13**
5	1992-01-06
6	**1992-01-15**
7	1992-01-08
8	1992-01-09
9	1992-01-10
10	1992-01-11

→

V2

1	1992-01-01
2	1992-01-02
3	1992-01-03

T2

→

V1

4	1992-01-05
6	1992-01-07

T1

T1

0	1992-01-01
1	1992-01-02
2	1992-01-03
3	1992-01-04
4	**1992-01-13**
5	1992-01-06
6	**1992-01-15**
7	1992-01-08
8	1992-01-09
9	1992-01-10
10	1992-01-11

T2

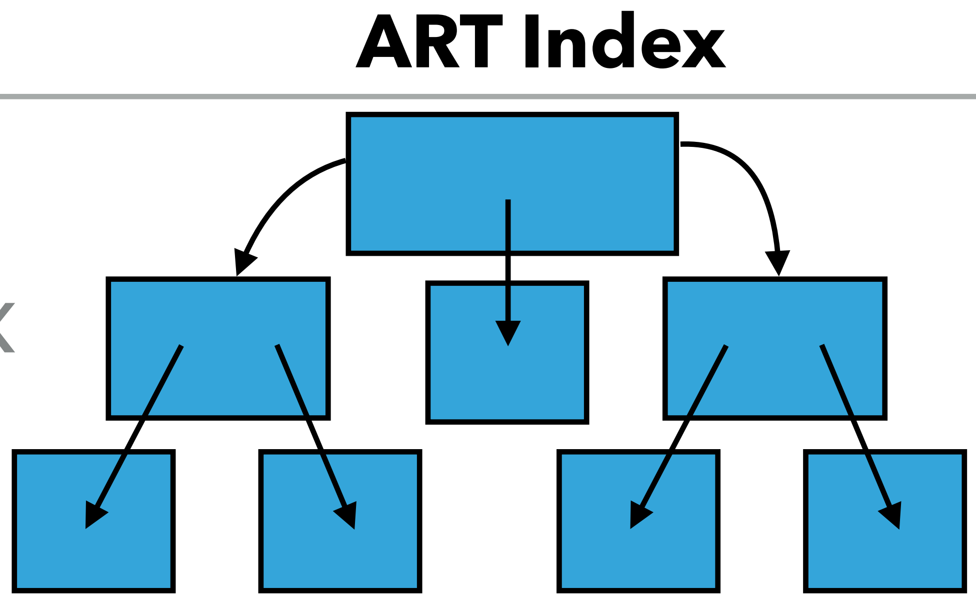
0	**1992-01-22**
1	**1992-01-27**
2	**1992-01-28**
3	1992-01-04
4	1992-01-05
5	1992-01-06
6	1992-01-07
7	1992-01-08
8	1992-01-09
9	1992-01-10
10	1992-01-11

MVCC

After commit of T2

	Date	T1	T2
0	**1992-01-22**	**1992-01-22**	**1992-01-22**
1	**1992-01-27**	**1992-01-27**	**1992-01-27**
2	**1992-01-28**	**1992-01-28**	**1992-01-28**
3	1992-01-04	1992-01-04	1992-01-04
4	**1992-01-13**	**1992-01-13**	**1992-01-13**
5	1992-01-06	1992-01-06	1992-01-06
6	**1992-01-15**	**1992-01-15**	**1992-01-15**
7	1992-01-08	1992-01-08	1992-01-08
8	1992-01-09	1992-01-09	1992-01-09
9	1992-01-10	1992-01-10	1992-01-10
10	1992-01-11	1992-01-11	1992-01-11

ART Index



- DuckDB has support for ART index
  - **Explicitly**: `CREATE INDEX`
  - **Automatically:** `PRIMARY KEY/UNIQUE`
- Only unclustered indexes are supported
  - In leaf nodes, ART index contains **row ids**
  - To reconstruct tuples, base table must be accessed
- ART index only speeds up **selective queries (<1%)**

Storage

- Data is stored in a **single file**
  - **Purpose:** user convenience and simplicity
- File starts with a small header (4KB)
- Rest is divided into **equal-sized blocks**
  - Currently *256KB*
- Blocks are referred to by **block_id**

Single-File Storage

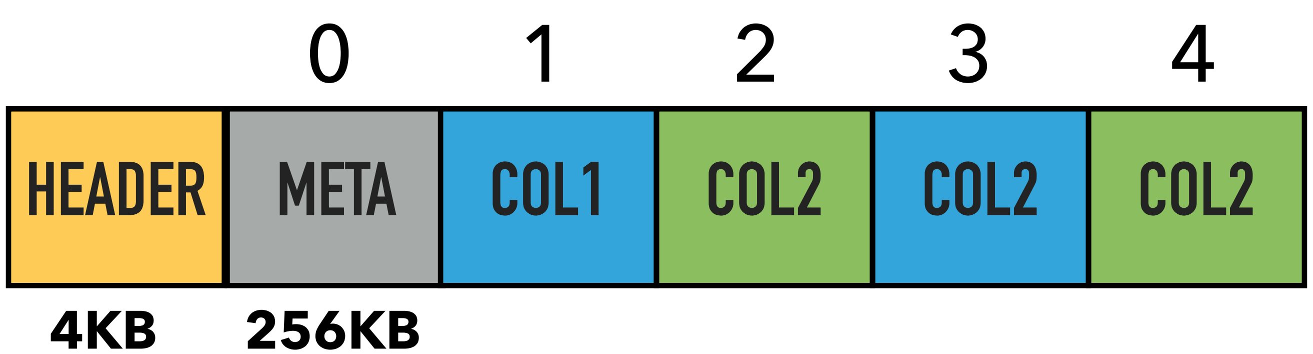
database.db

Storage

- Meta data contains **schema information**
  - Schema, table, column names
  - Column types
  - **Column Pointers:** Pointers to blocks where data is

Meta (0)

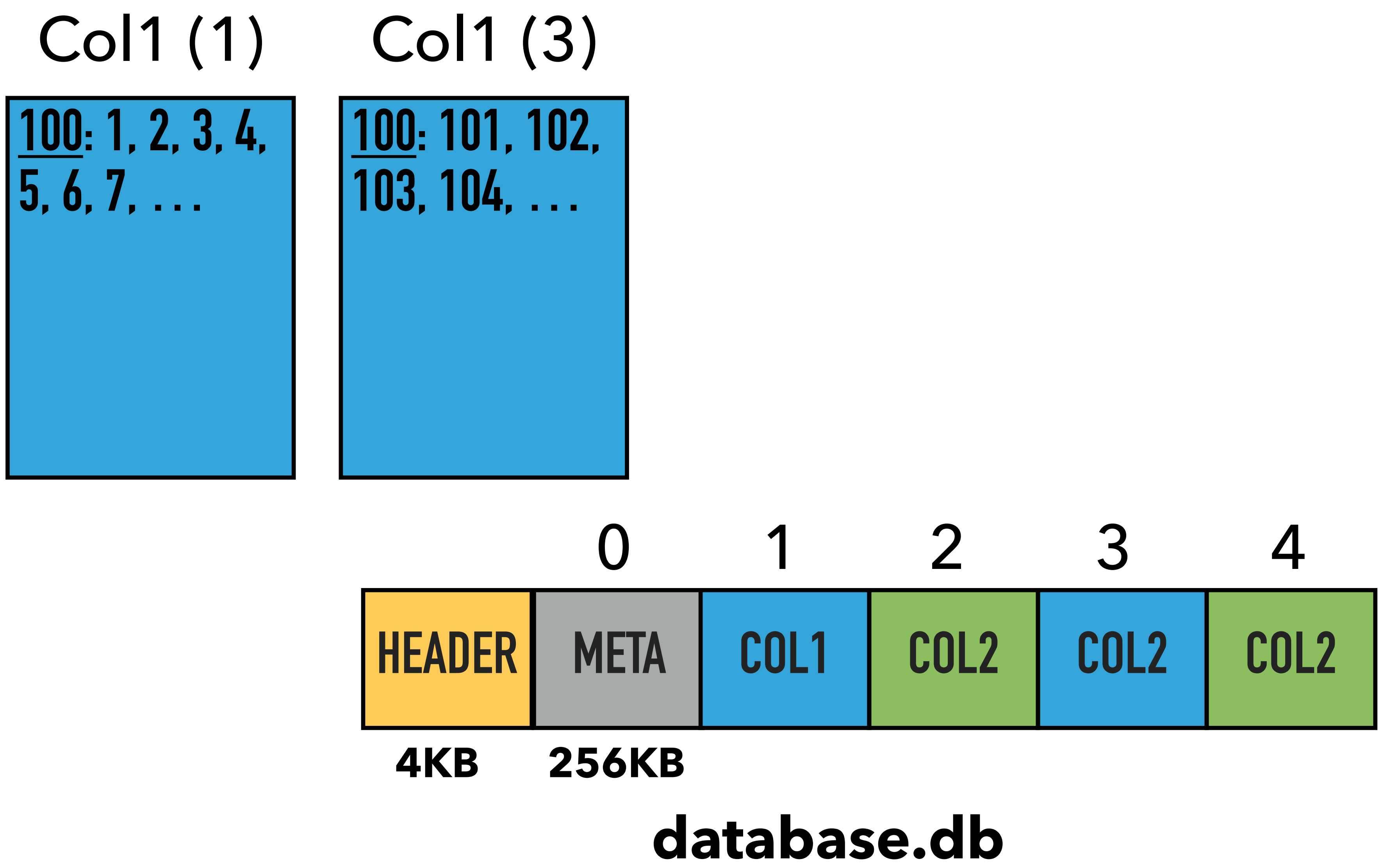
table
col1 INTEGER blocks: 1, 3
col2 VARCHAR blocks: 1, 4



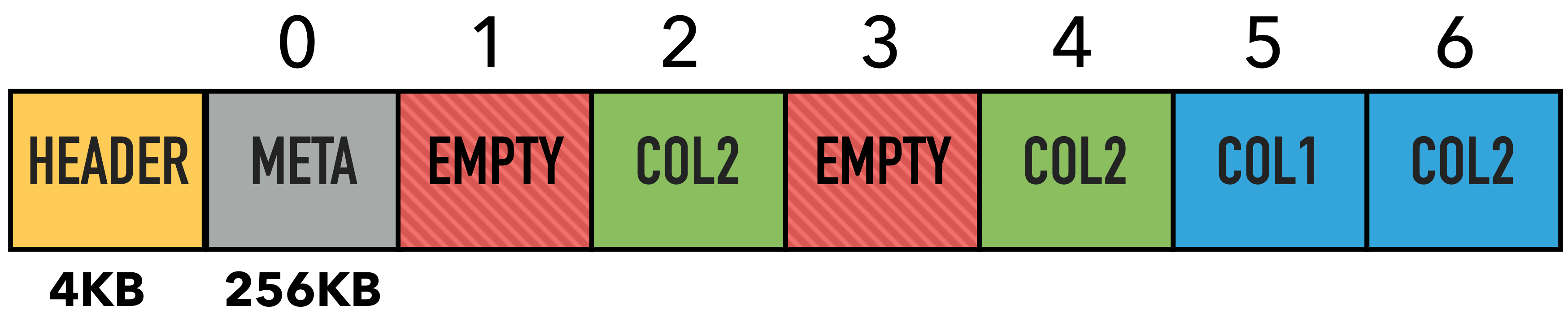
database.db

Storage

- Column data contains the data of that column
  - Tuple count + physical data
- Currently only uncompressed, compression WIP



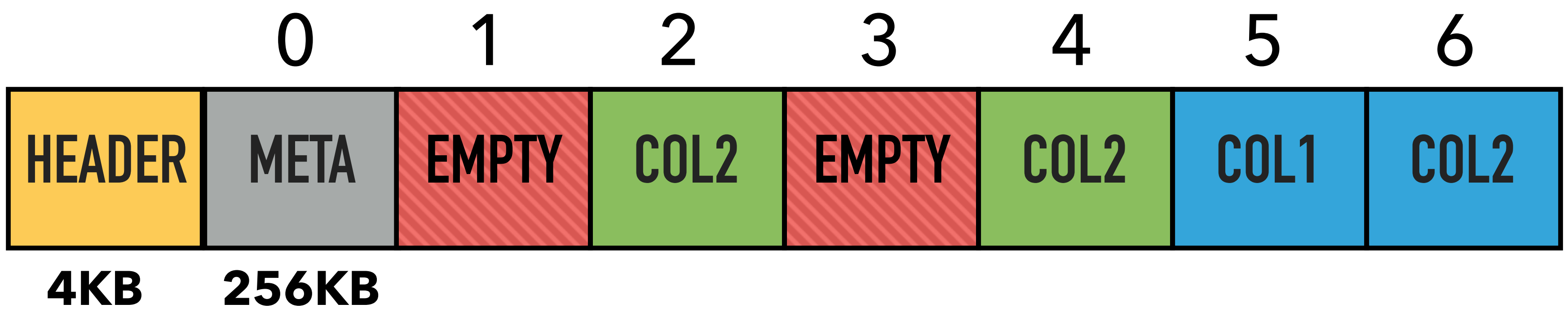
Storage



- Checkpointing rewrites **dirty blocks** first
- Header updated after writing successfully completes

- This ensures ACID properties
- **If writing fails during data write:**
  - Header is never updated
- Header is small enough to be updated atomically

Storage



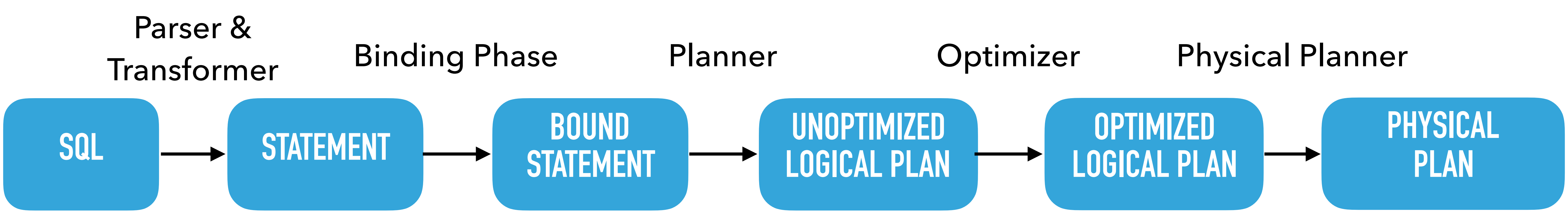
- Checkpointing leaves **empty blocks**
  - Old data is not deleted until new blocks are written
- Next checkpoint will overwrite the empty blocks

Contents

Query Processing Pipeline

DuckDB

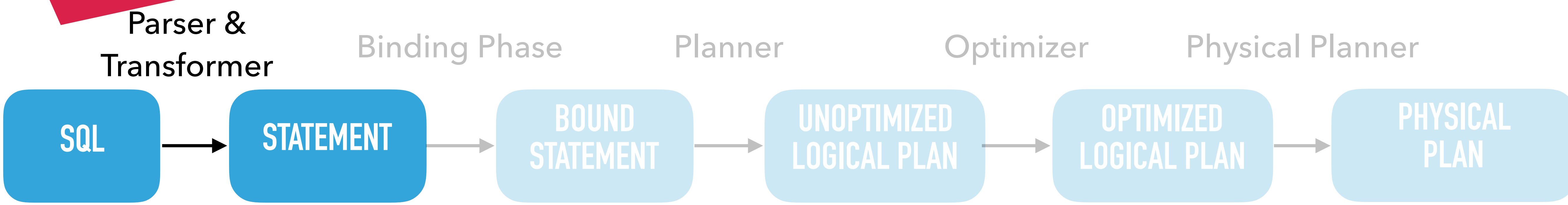
- DuckDB uses a typical pipeline for query processing

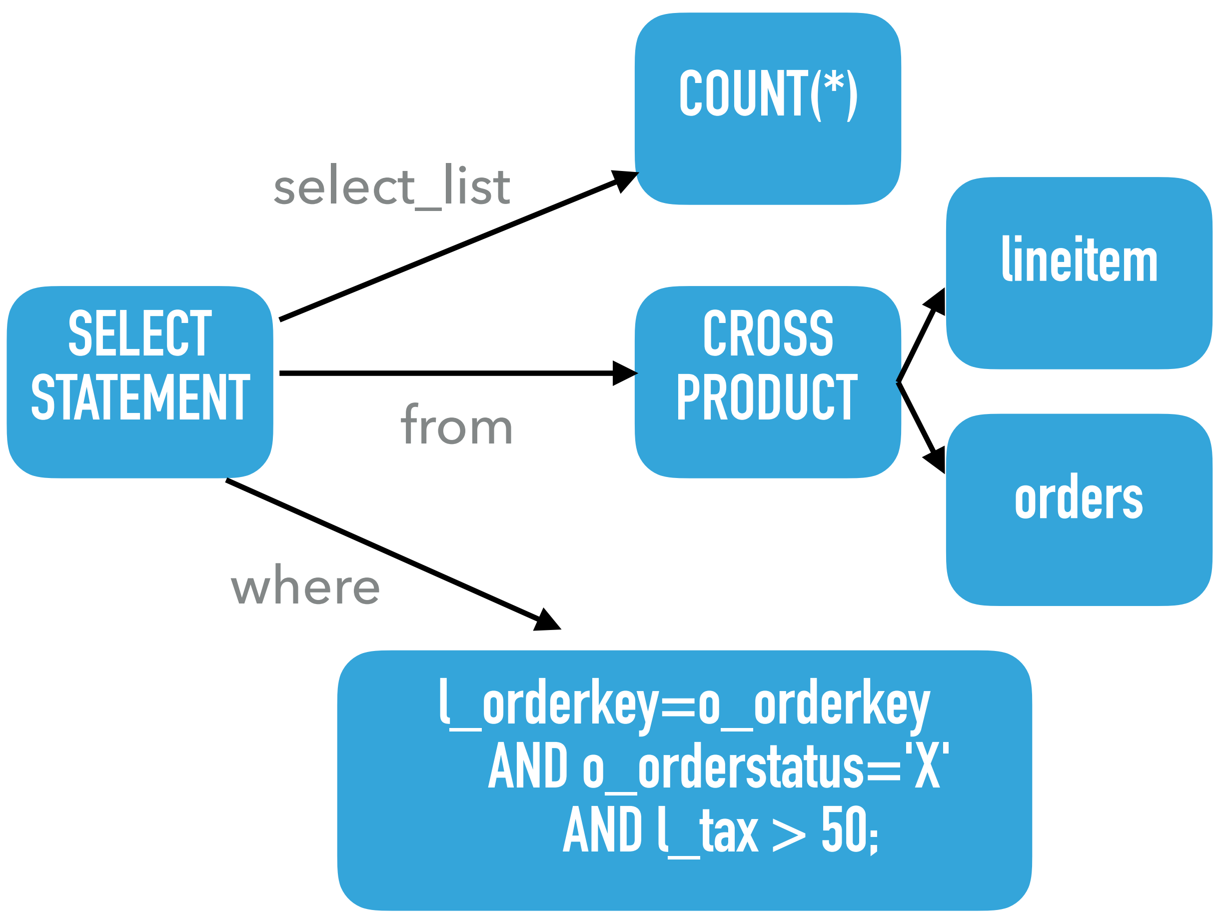


- **Life of a Query**

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

DuckDB

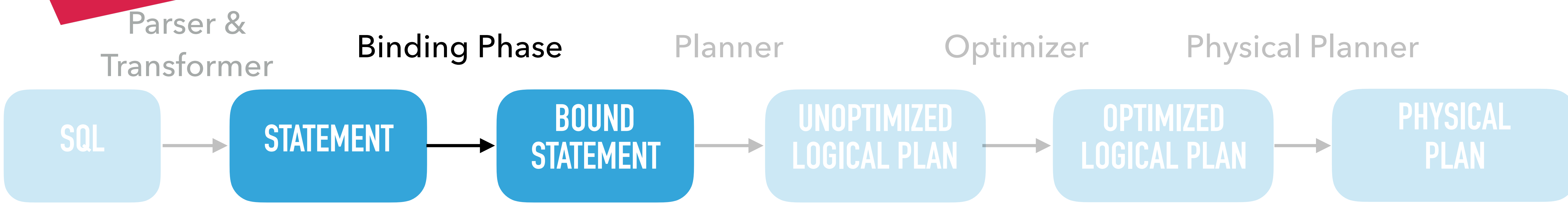




Tables and columns are strings
Nothing is resolved yet!

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

DuckDB



Catalog lookup is done here

Table names → **table indexes**

Column names → **table index + column index**

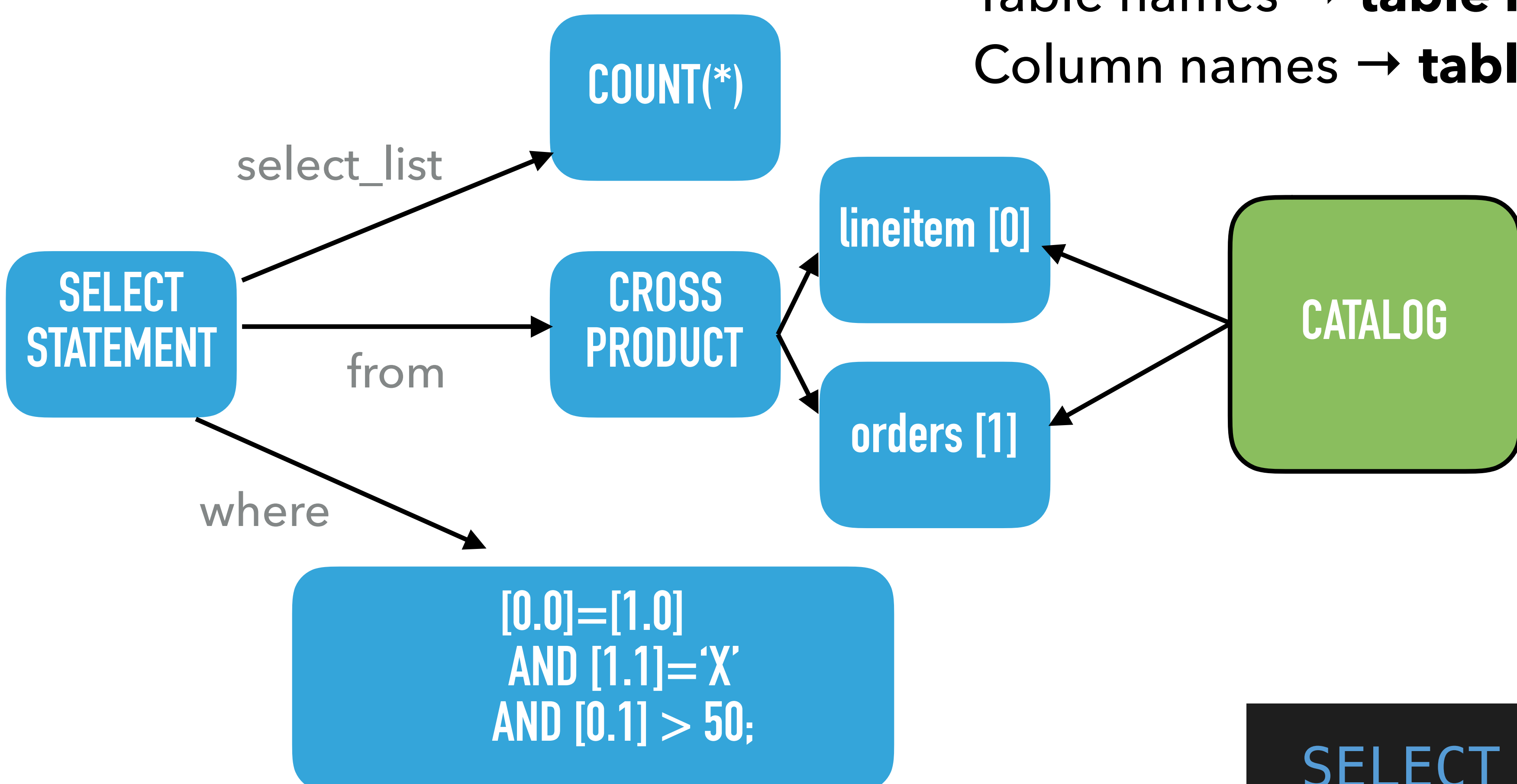


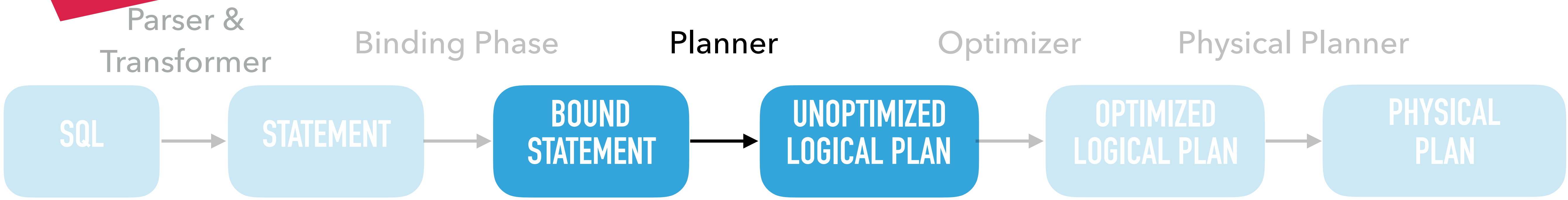
table index
lineitem: 0
orders: 1

column index
l_orderkey: 0
l_tax: 1

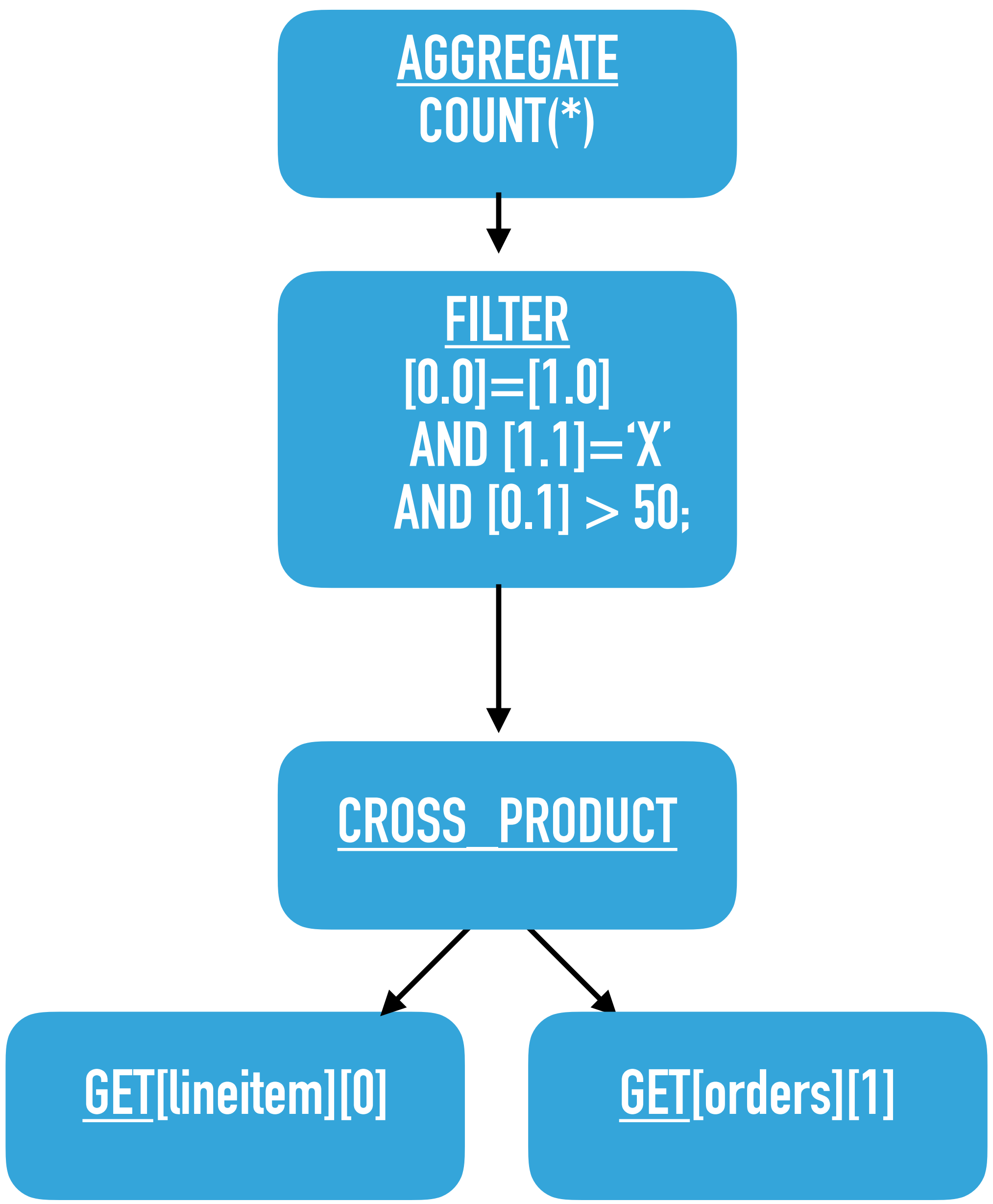
column index
o_orderkey: 0
o_orderstatus: 1

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

DuckDB



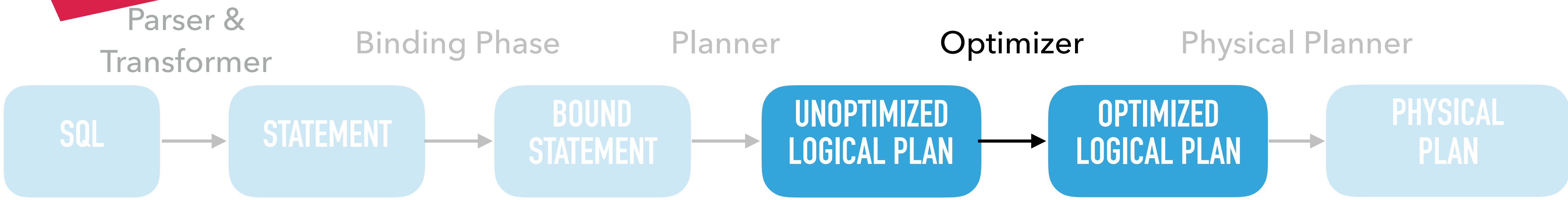
Logical Query Tree



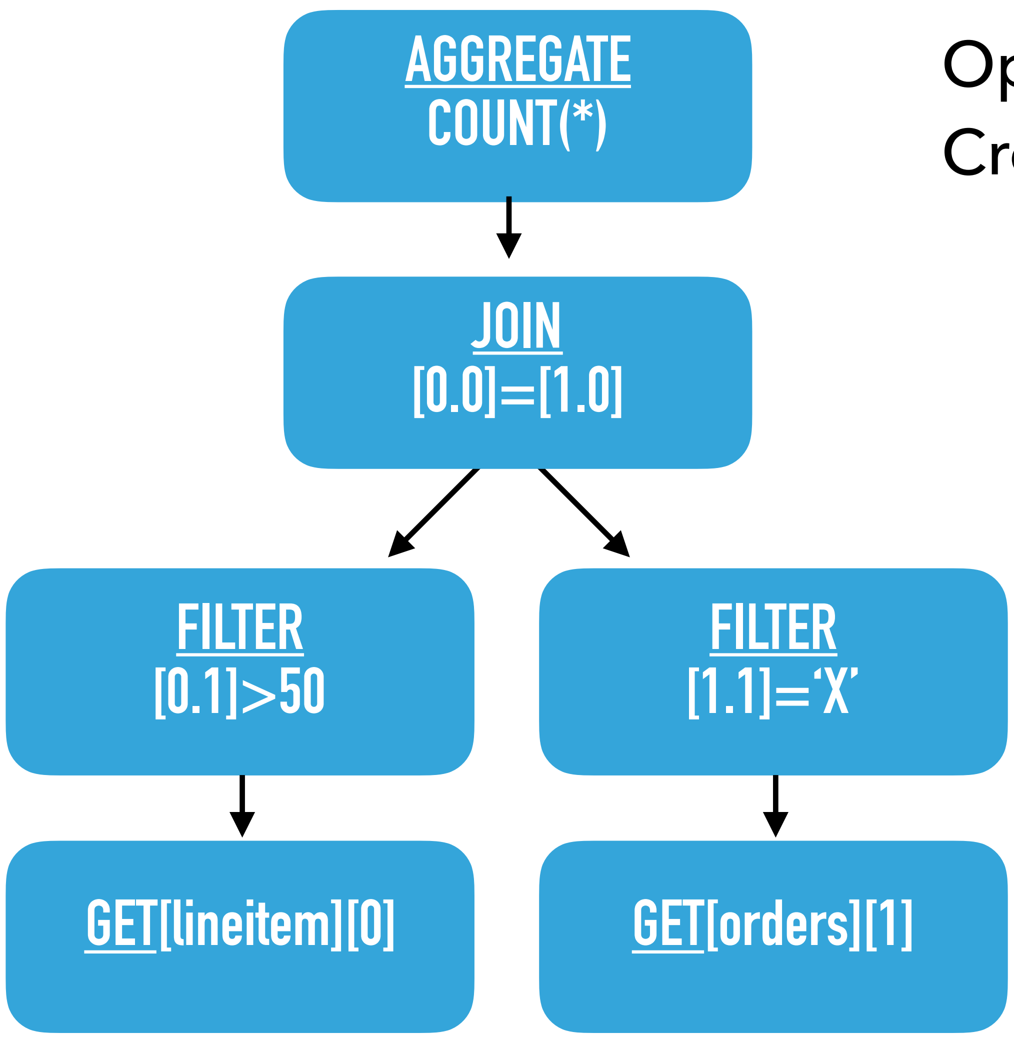
Internally, nodes are called **Logical**…
e.g. LogicalAggregate, LogicalFilter

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

DuckDB



Logical Query Tree

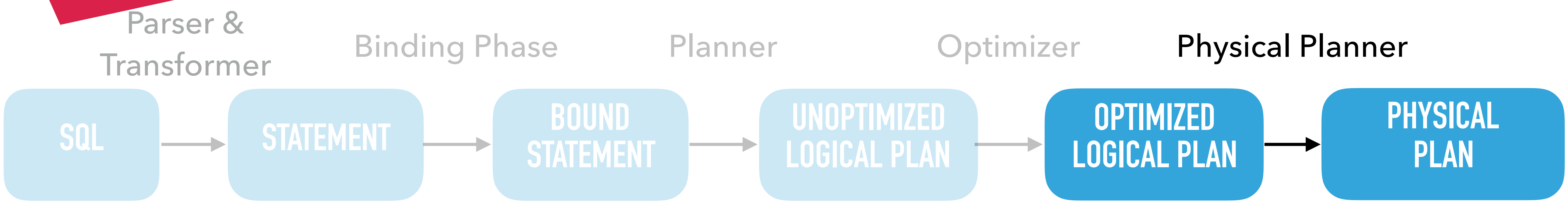


Optimizer **transforms** logical query tree

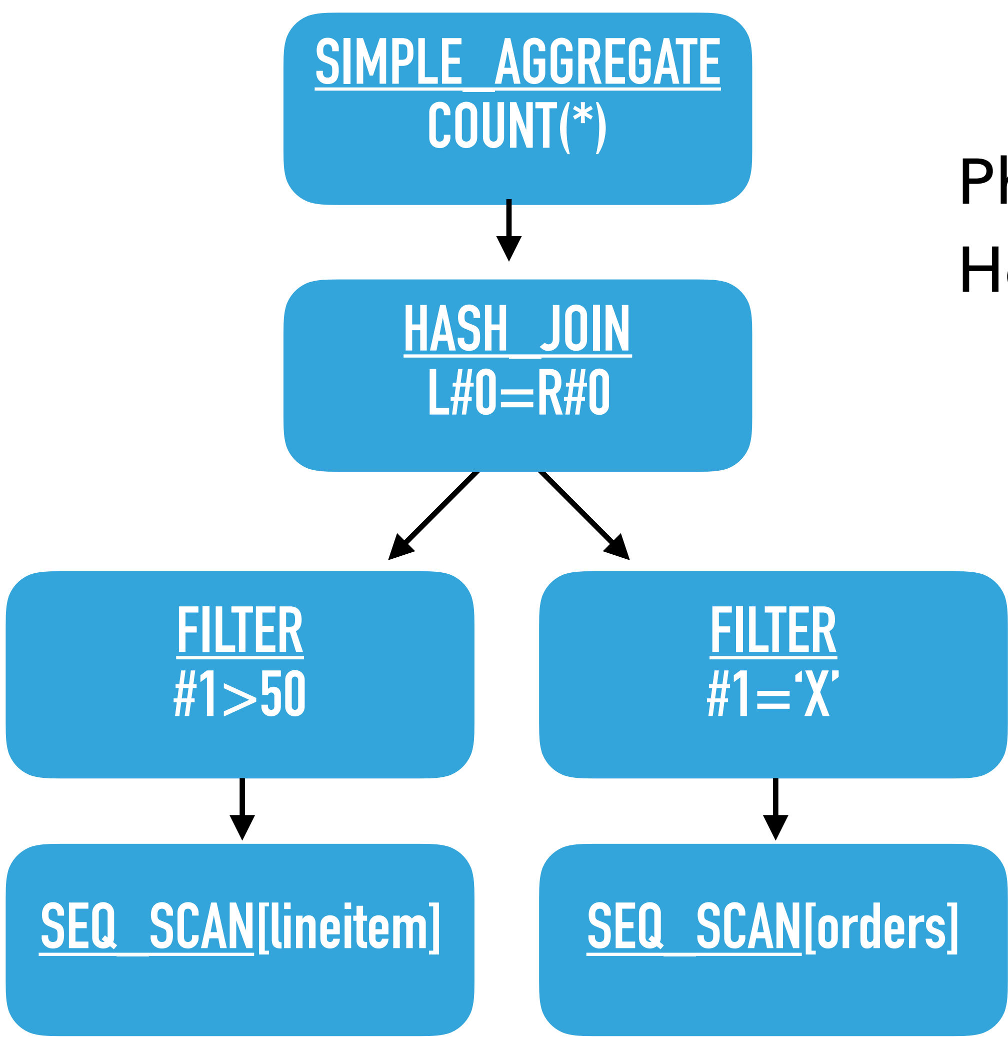
Created plan is equivalent but (hopefully) faster

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

DuckDB



Physical Query Tree



Physical planner creates **physical query tree**

Here implementations of operators are chosen

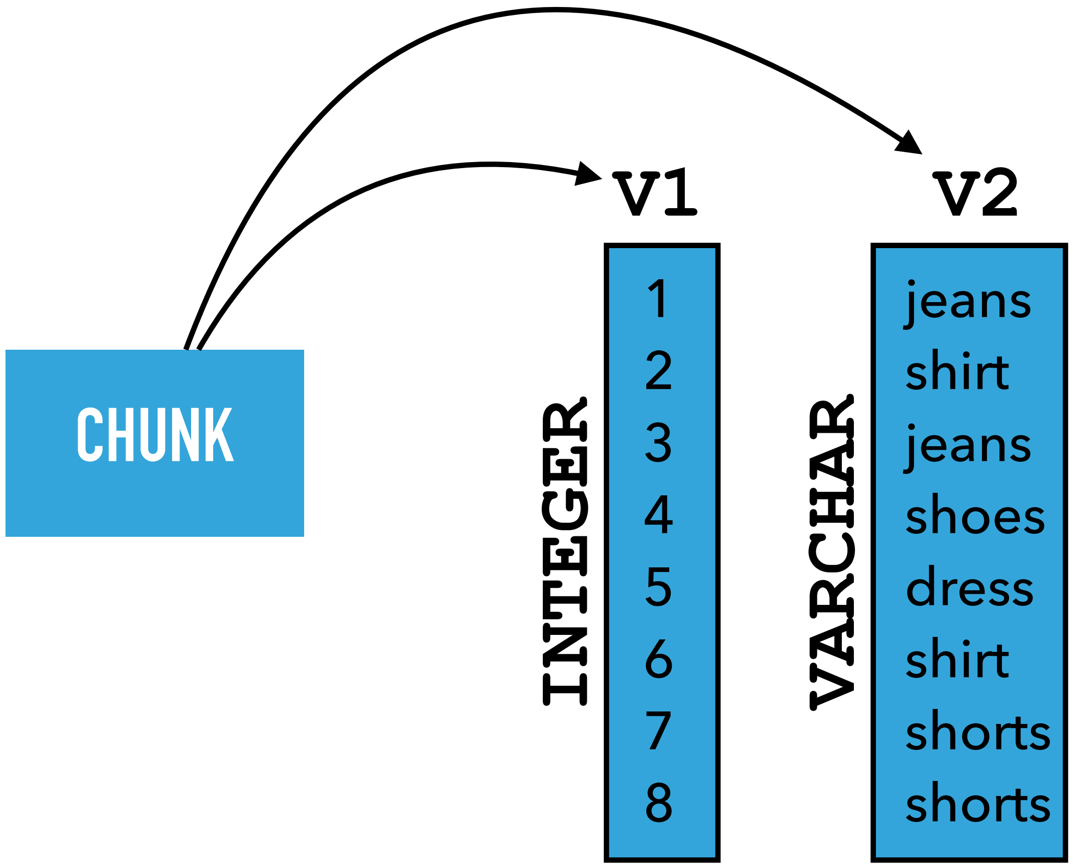
```
SELECT COUNT(*)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey
      AND o_orderstatus='X'
      AND l_tax > 50;
```

Contents

Query Execution

Query Execution

- Basic units: `Vector` and `DataChunk`
- `Vector` is a **column-slice**
  - Set of **up to** *1024* values of a single type
- `DataChunk` is a **table-slice** (set of vectors)



Query Execution

```
class Vector {
public:
    TypeId type;
    index_t count;
    data_ptr_t data;
    sel_t *sel_vector;
    nullmask_t nullmask;
```

```
class DataChunk {
public:
    index_t column_count;
    unique_ptr<Vector[]> data;
```

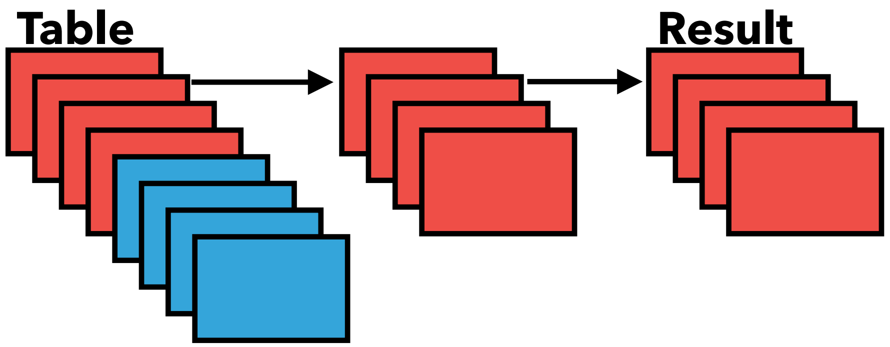
- **`nullmask`**: bitmap indicating which values are NULL
- **`sel_vector`**: optional selection vector indicating **which** values to use in the vector

Query Execution

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

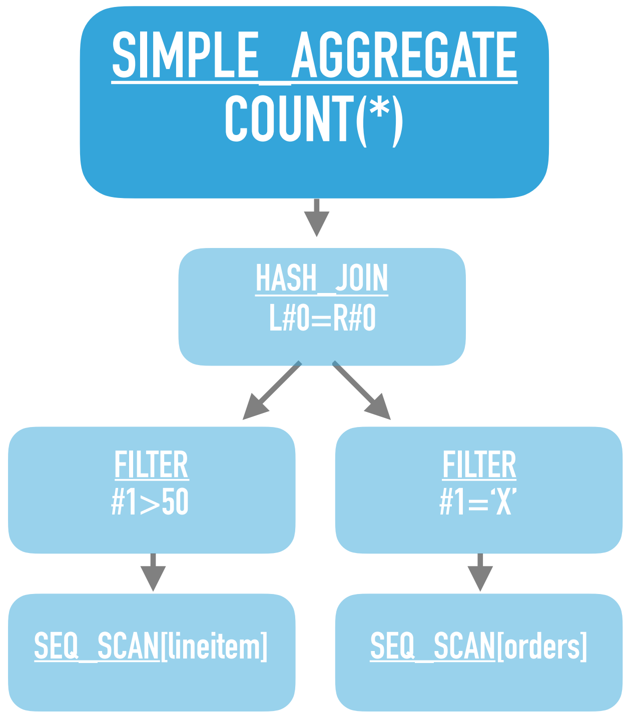
- Query starts by calling **`GetChunk`** on the root node
- Root node recursively calls **`GetChunk`** on children
- Scans fetch data from the base tables

Vectorized Processing



DuckDB

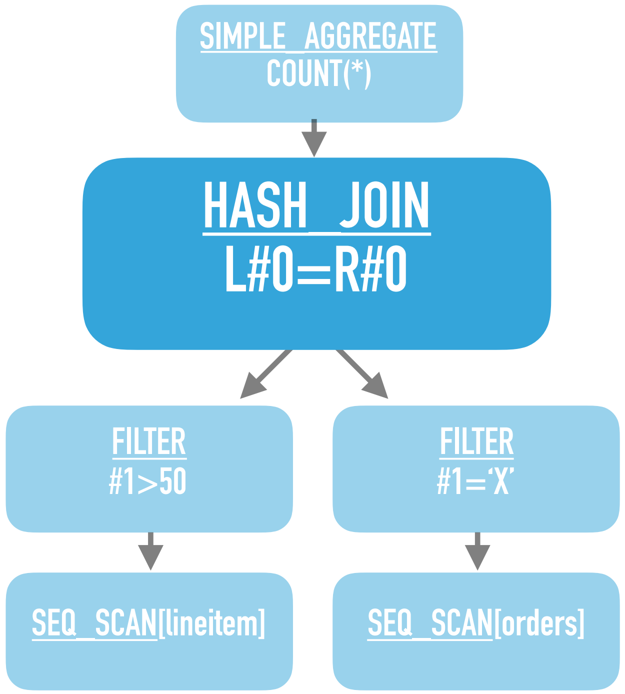
- Start with root node: **SimpleAggregate**
  - Aggregate without groups
- Immediately calls **`GetChunk`** on child



CWI

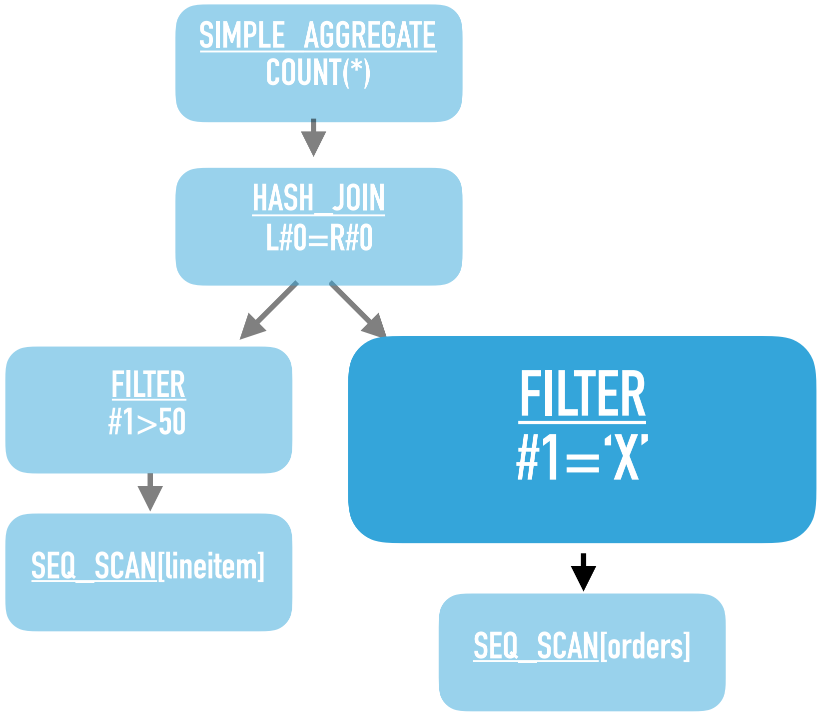
DuckDB

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

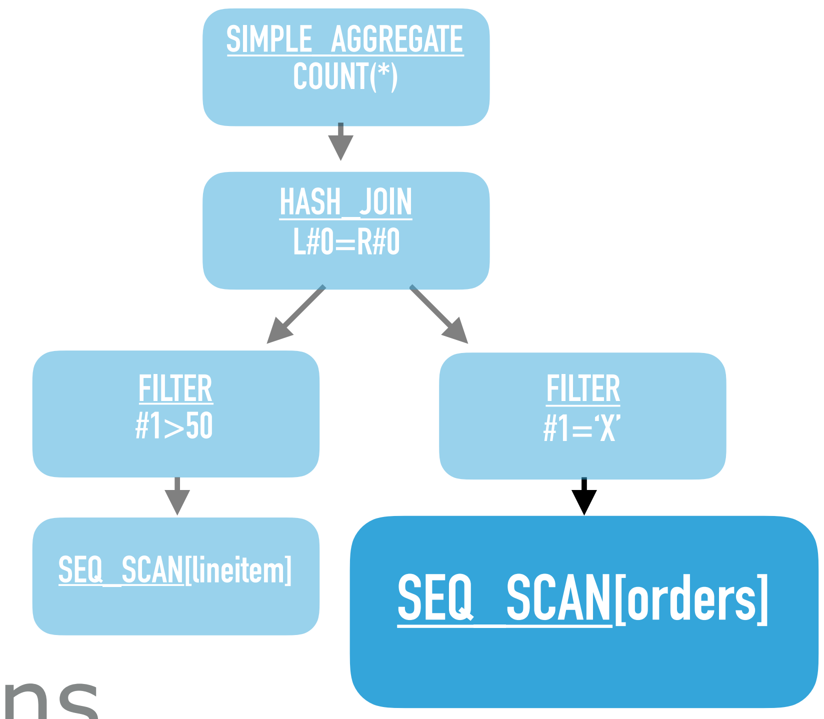


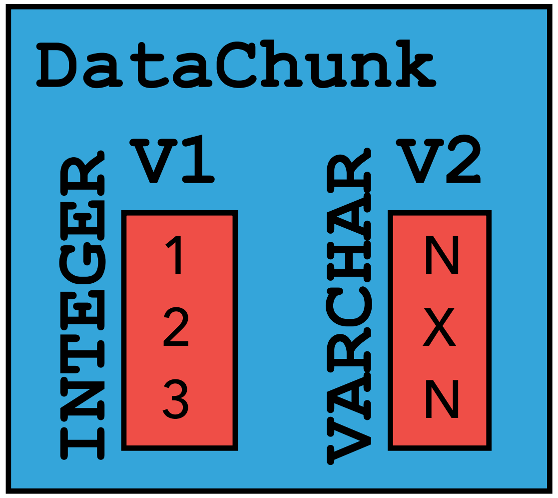
DuckDB

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



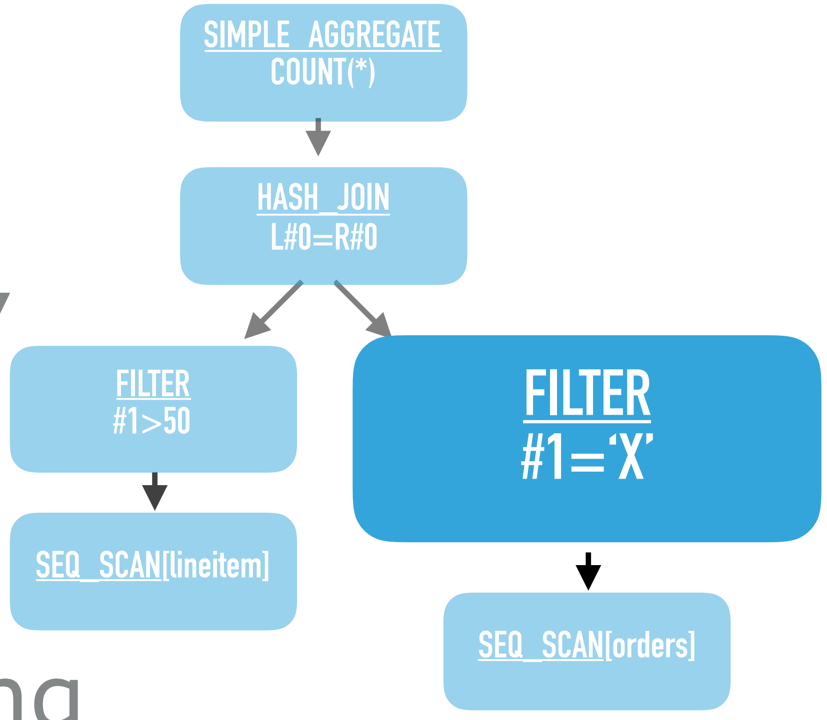
- **Sequential Scan**
- Finally we can start executing
- Scan the base table
- Return a DataChunk with two columns
  - `o_orderkey` and `o_orderstatus`

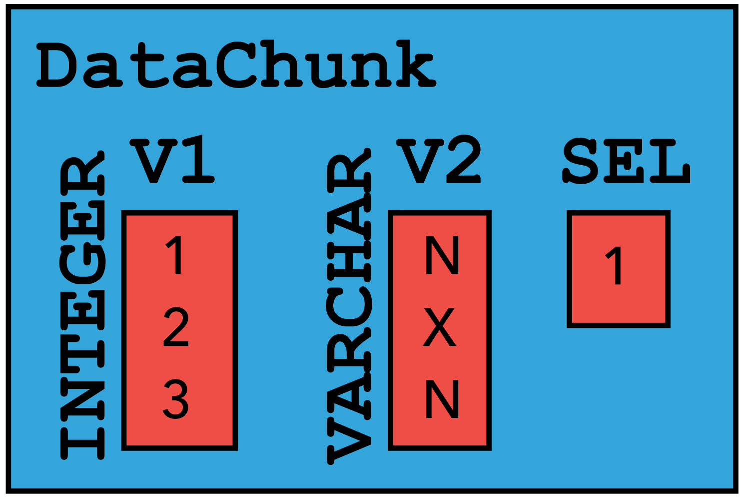




DuckDB

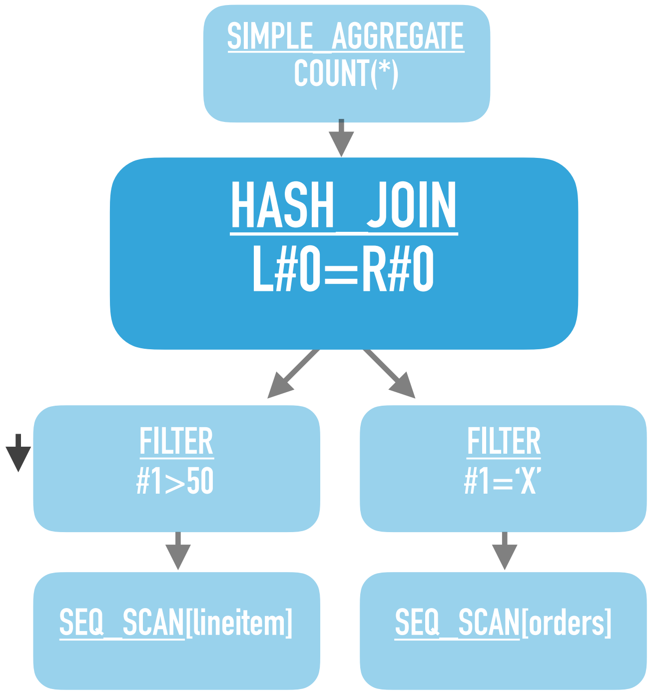
- **Filter**
- Now we can perform the filter #1='X'
  - Only the second tuple passes
- **Selection vector** pointing to surviving tuple is created
- Note that no data is copied or changed

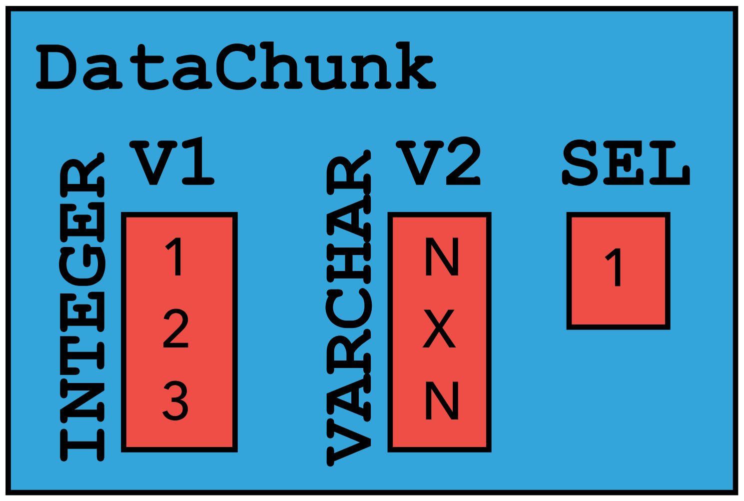


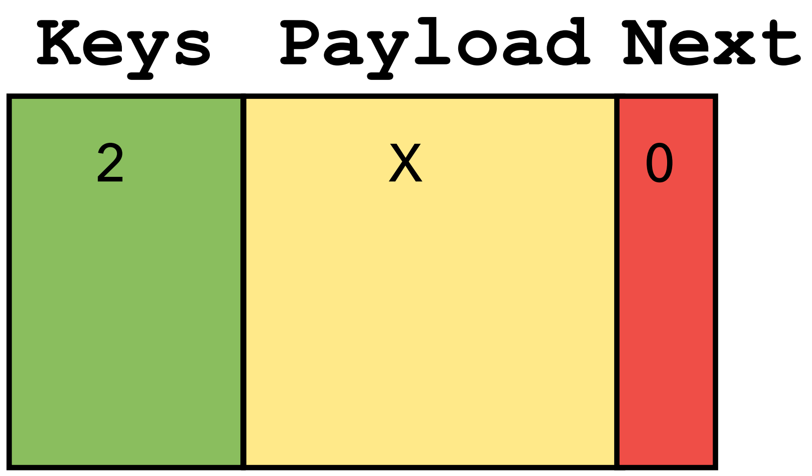


DuckDB

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



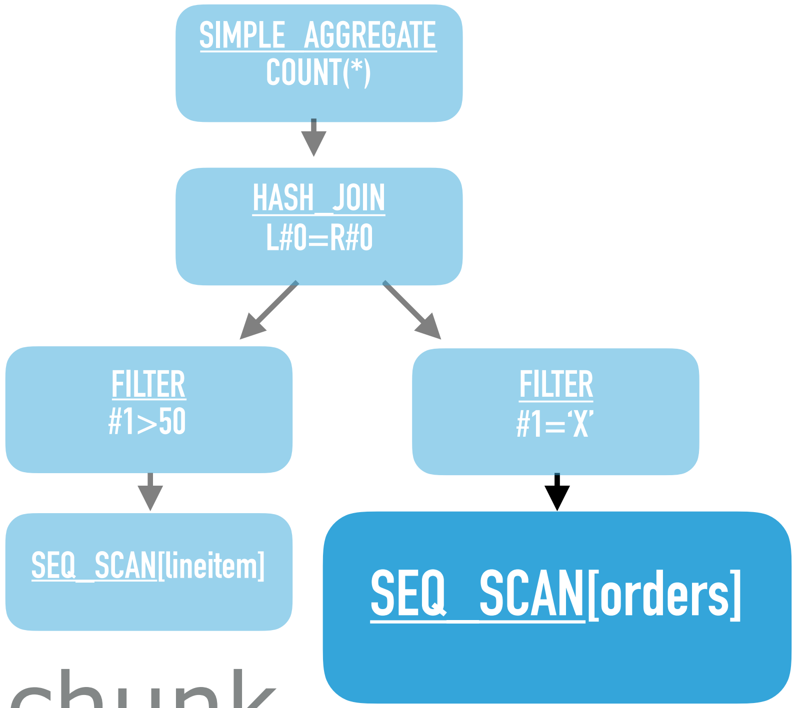


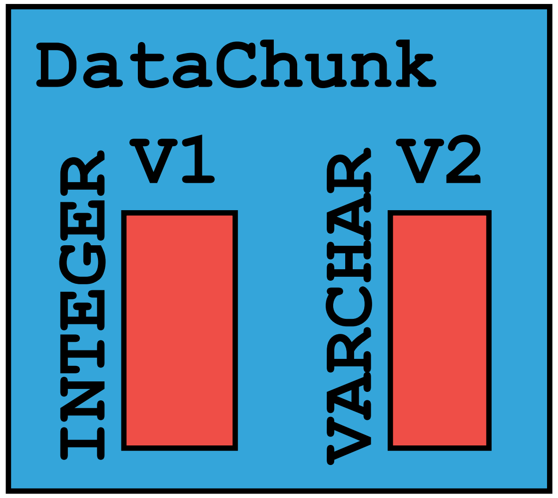


CWI

DuckDB

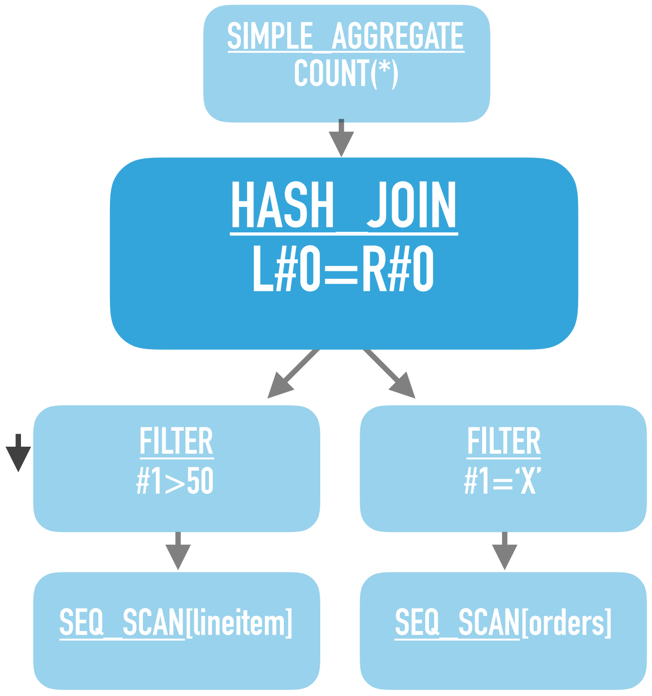
- **Sequential Scan**
- The filter again calls **`GetChunk`**
- Scan base table again:
  - The scan is finished, return empty chunk

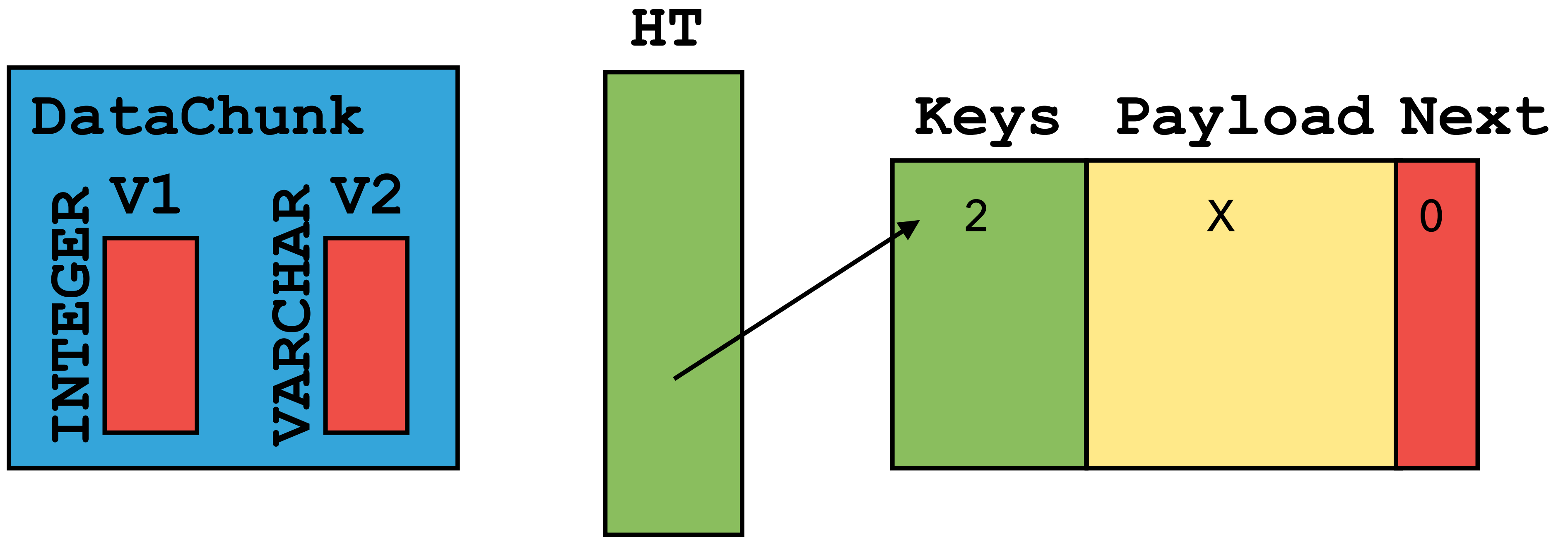




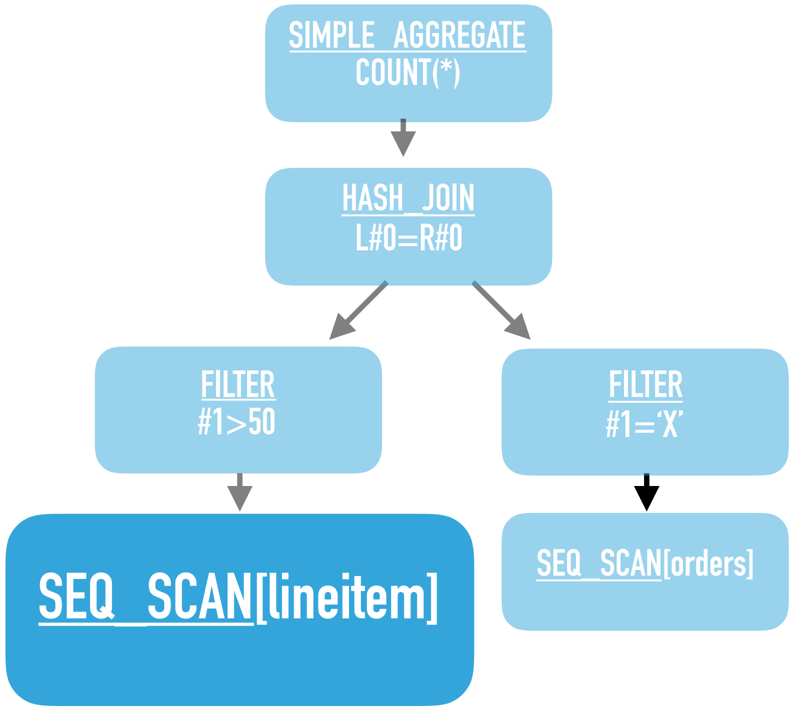
DuckDB

- **Hash Join**
- HT receives second input chunk
  - But it is empty!
- The RHS is exhausted
- Finish building HT and call `GetChunk` on LHS

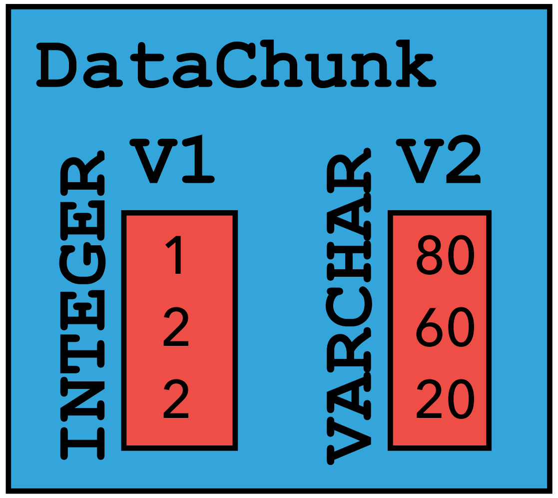




- **Sequential Scan**
- We arrive at scan on lineitem

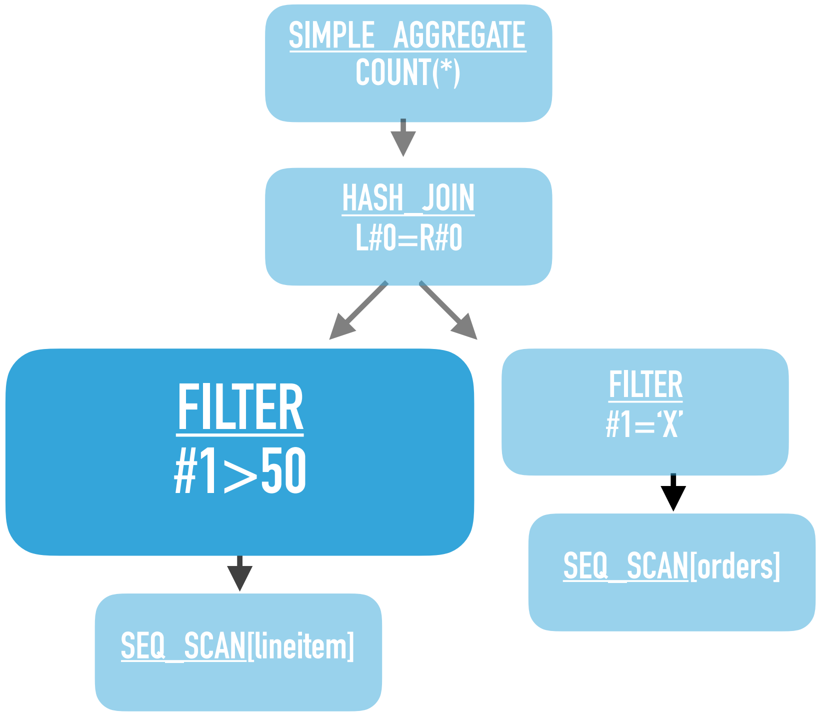


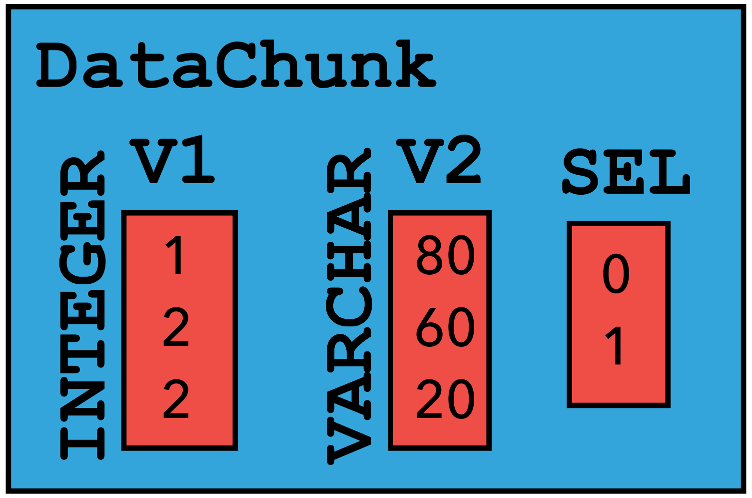
- DataChunk with two columns
  - `l_orderkey` and `l_tax`



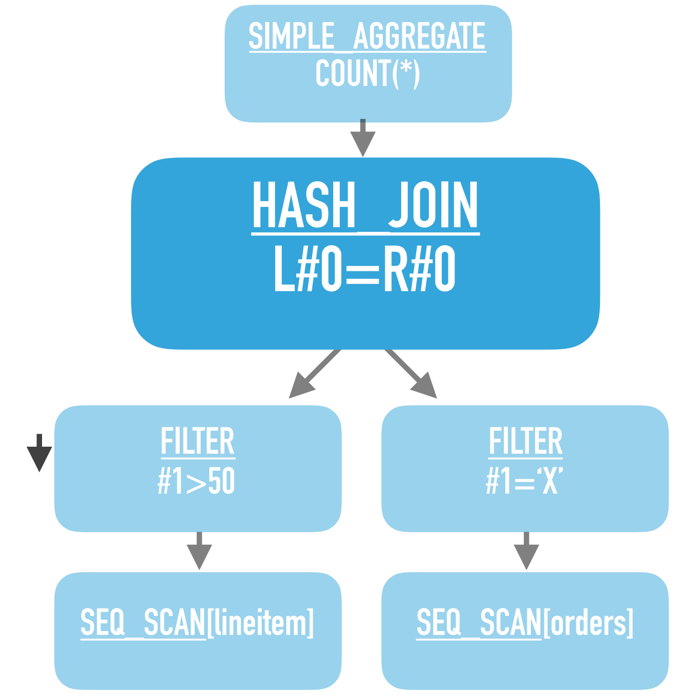
DuckDB

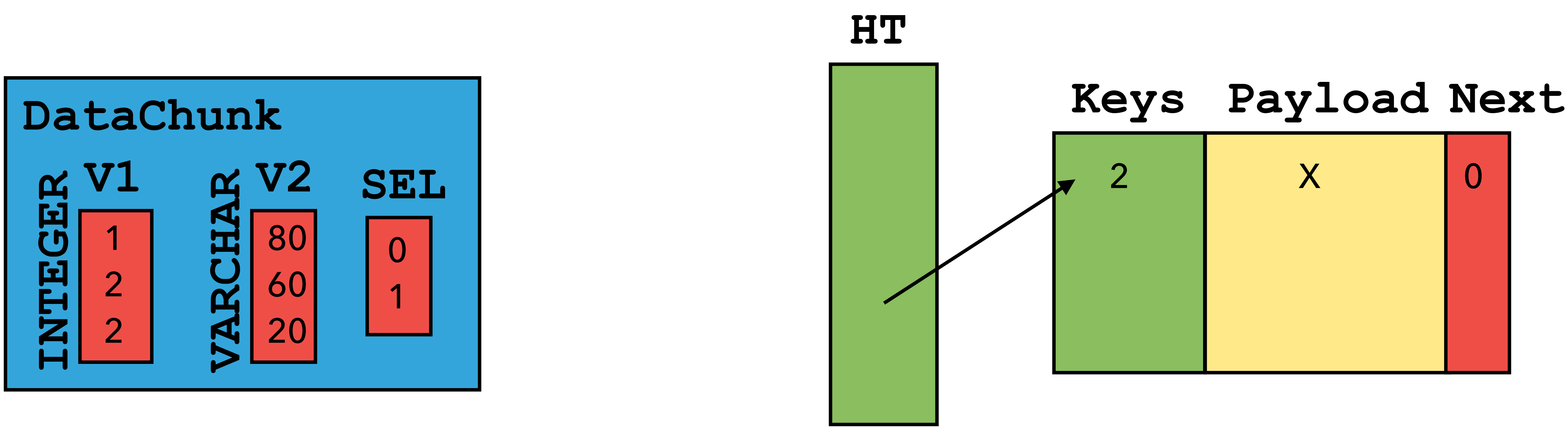
- **Filter**
- Performs the filter **#1>50**
- Again, add a selection vector





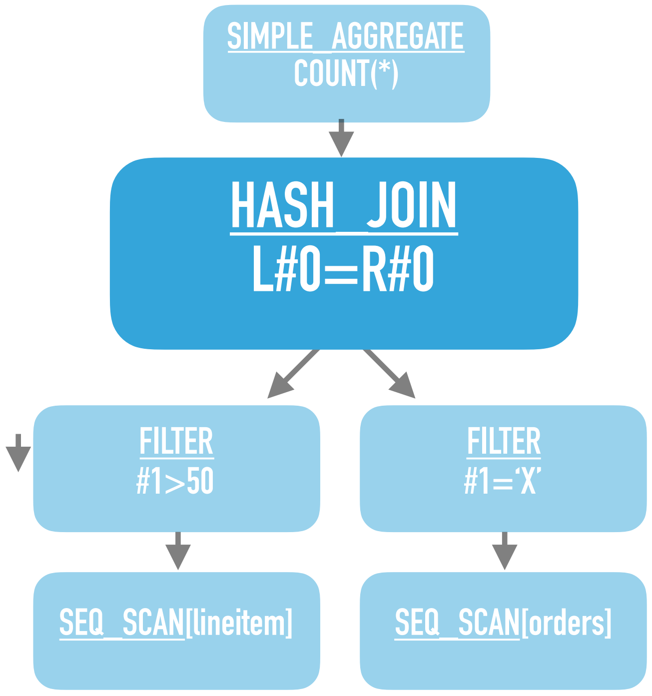
- **Hash Join**
- Now it is time to probe the HT
- We compute the hash for each tuple
- Then lookup in the HT

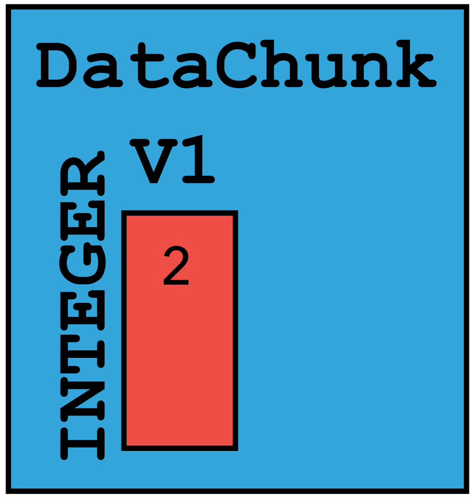




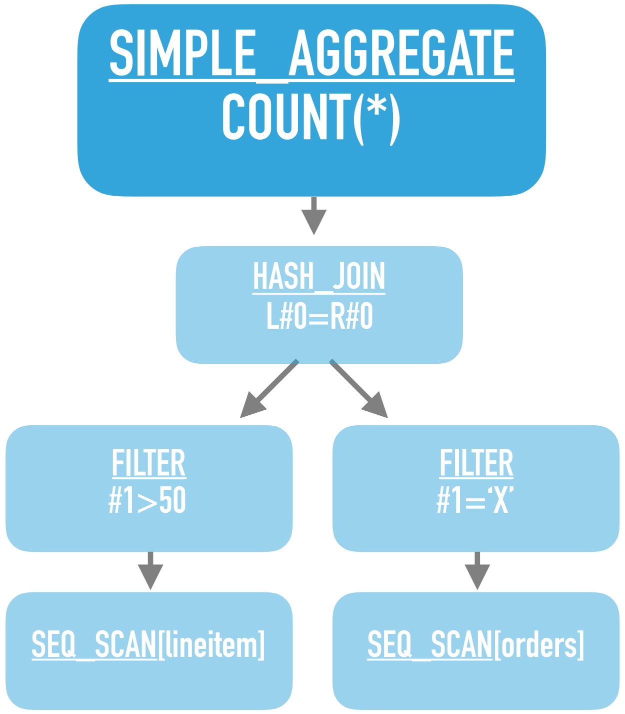
DuckDB

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



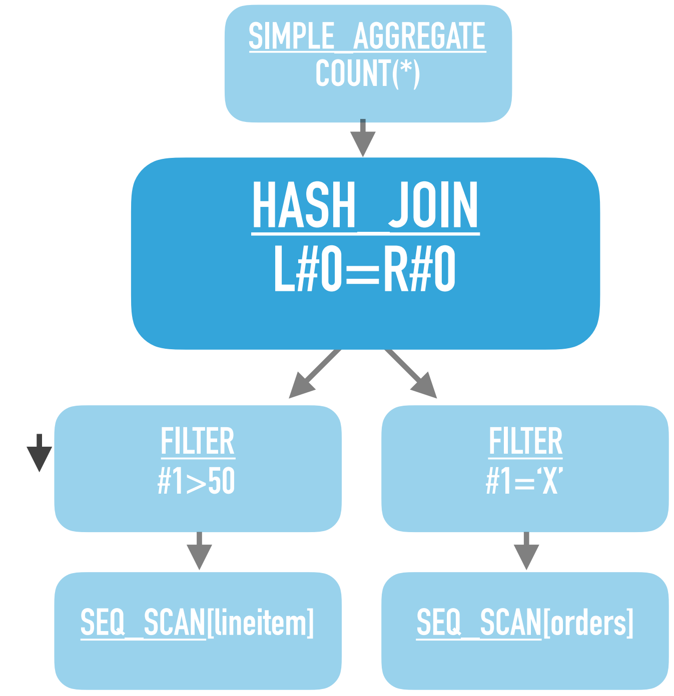


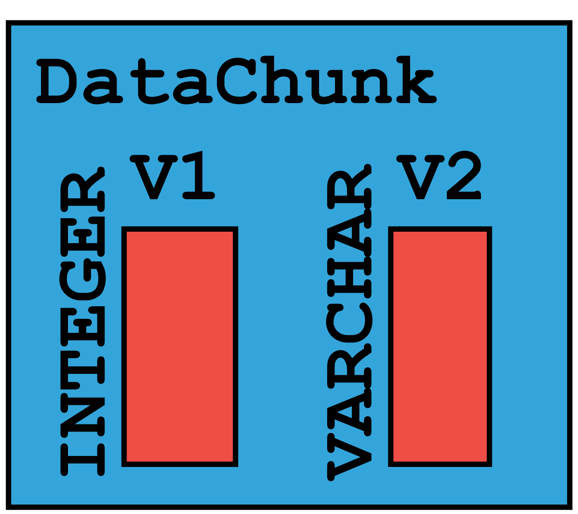
- The aggregate takes our input chunk
- Updates the aggregate
- Then fetches from the child again

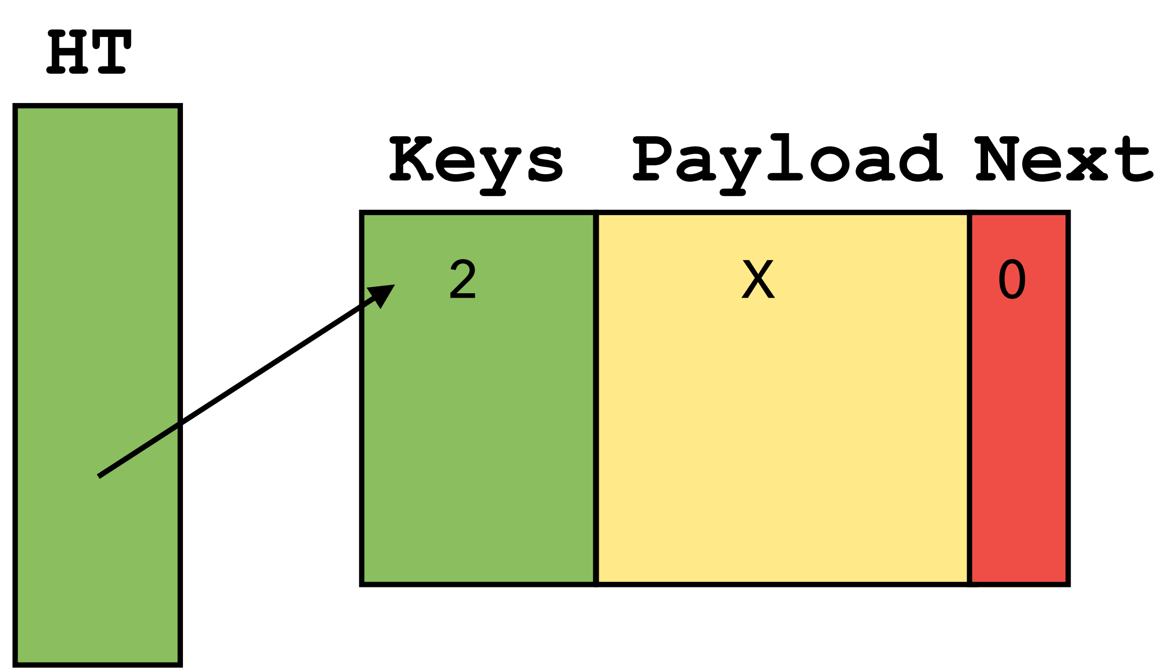


DuckDB

- 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!

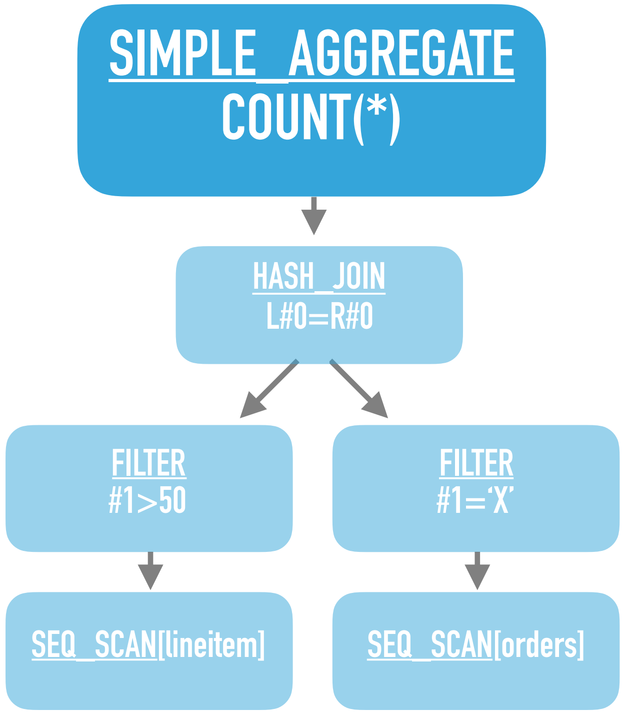


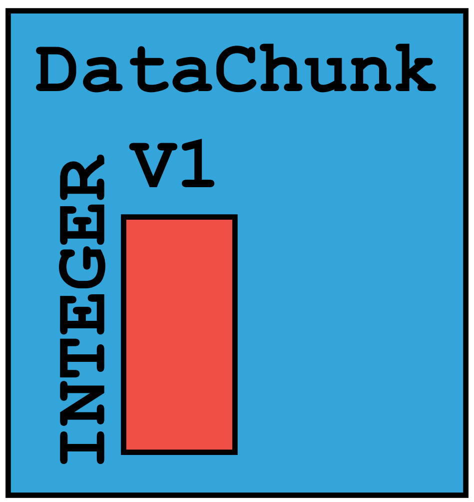


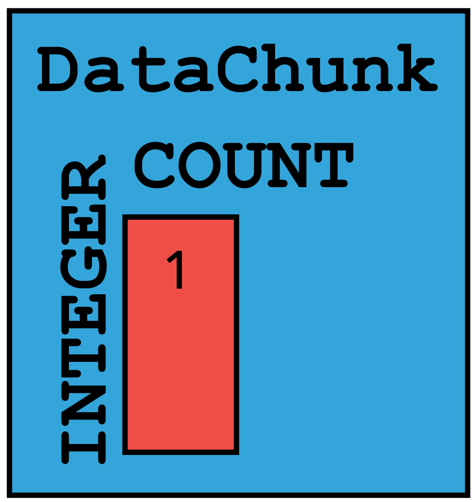


DuckDB

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



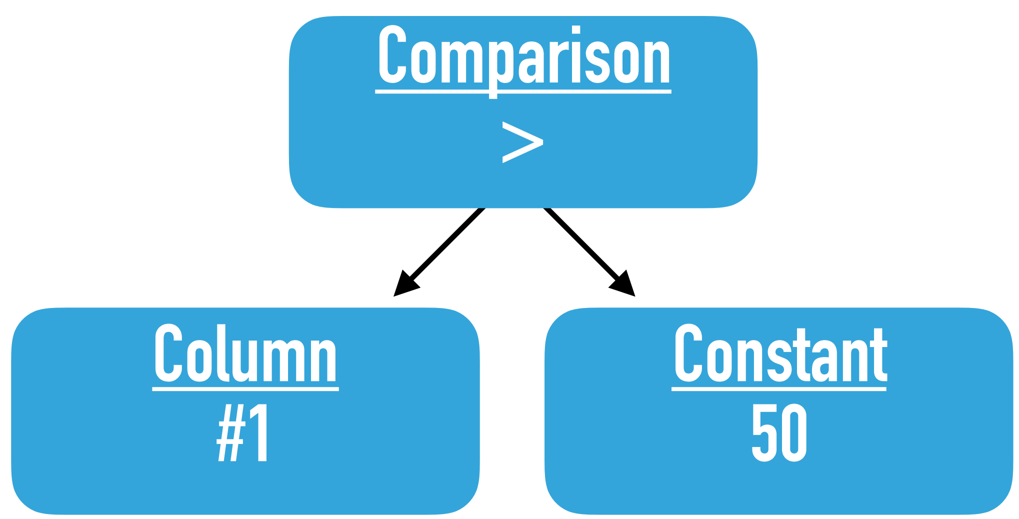




Expression Execution

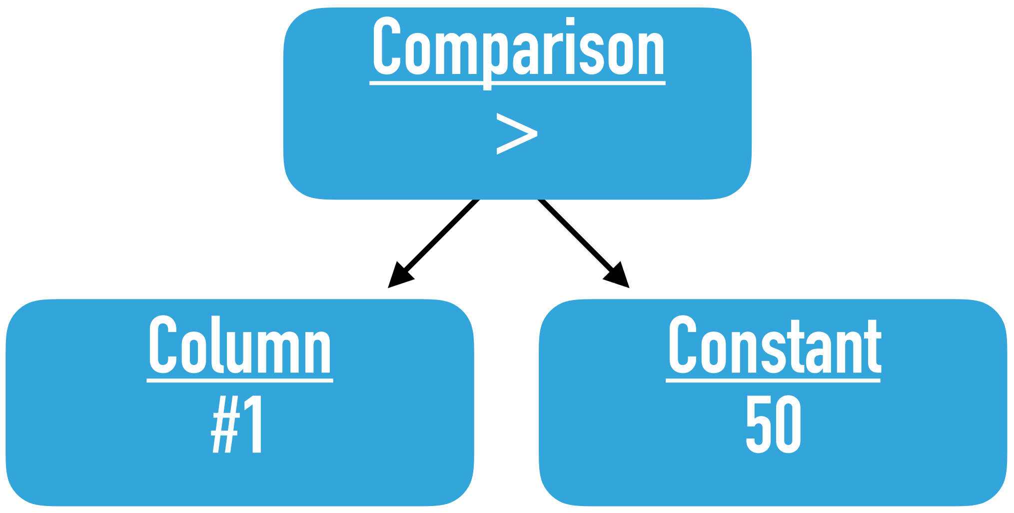
Expression Execution

- **Expressions** exist within the query tree nodes
  - Filter has a set of **filter predicates**
  - Projection has **projection list**
- Represented as **expression tree**

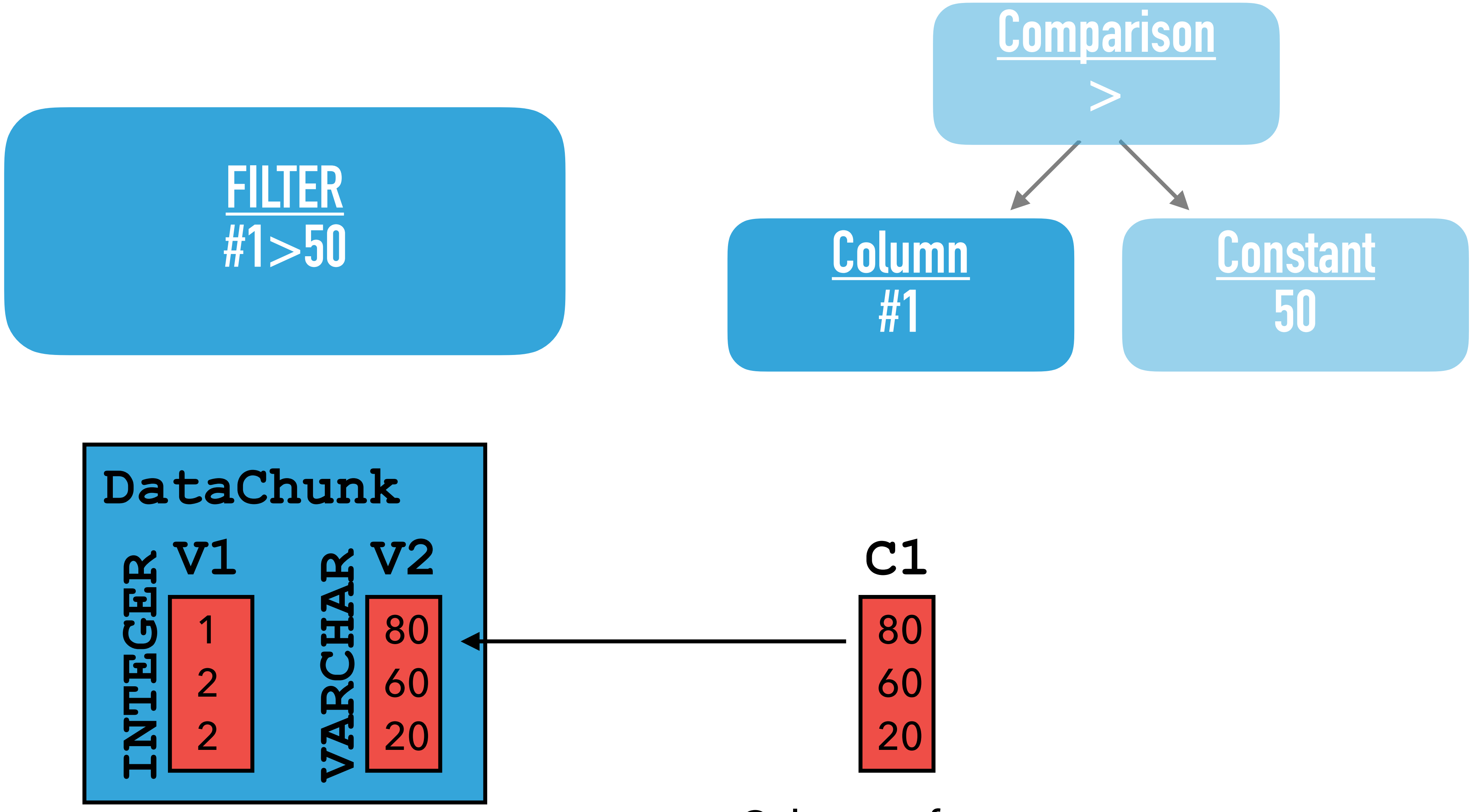


Expression Execution

- **ExpressionExecutor** runs the expressions
- This occurs as part of the execution of the node
- Expressions are executed in vectorized fashion

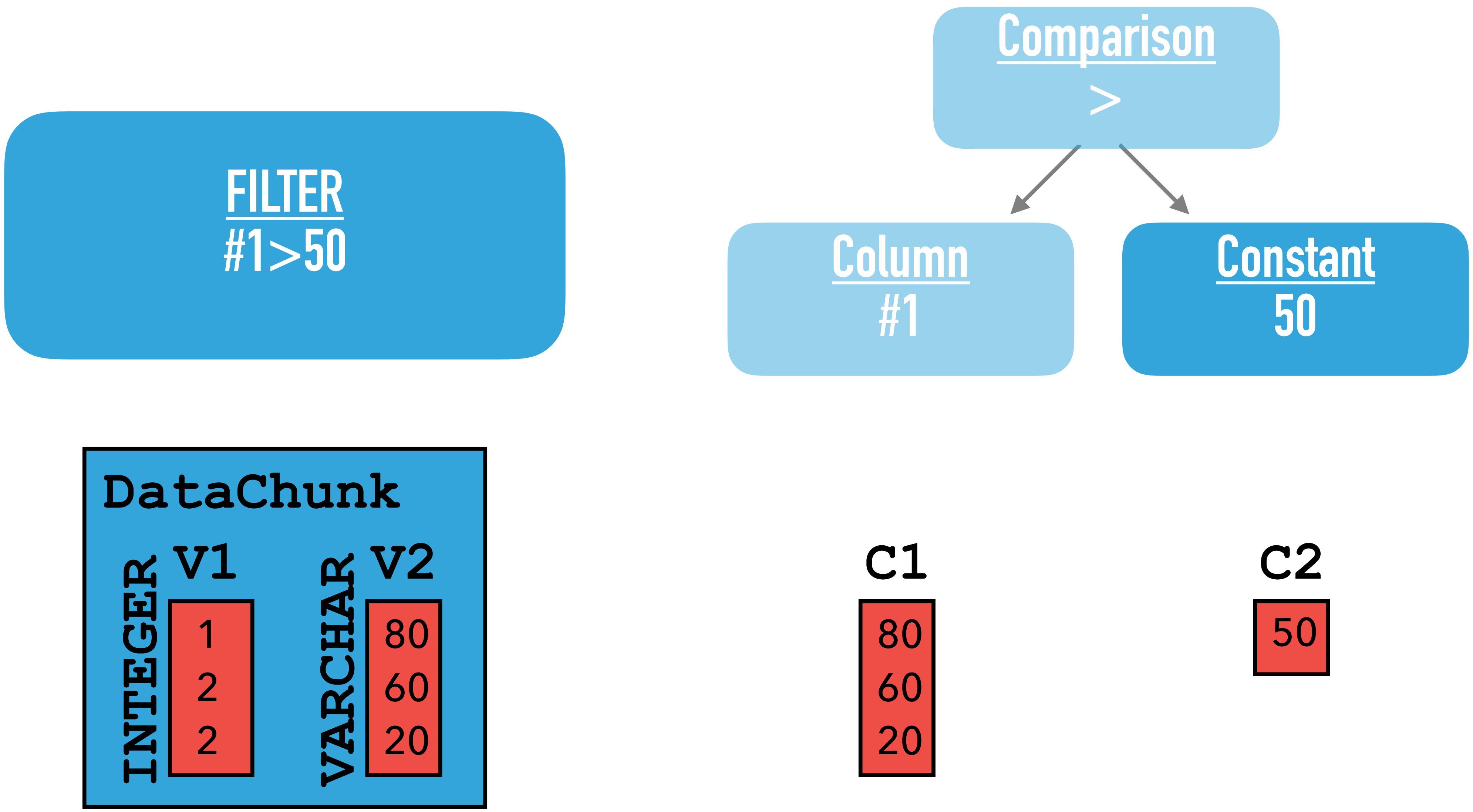


Expression Execution



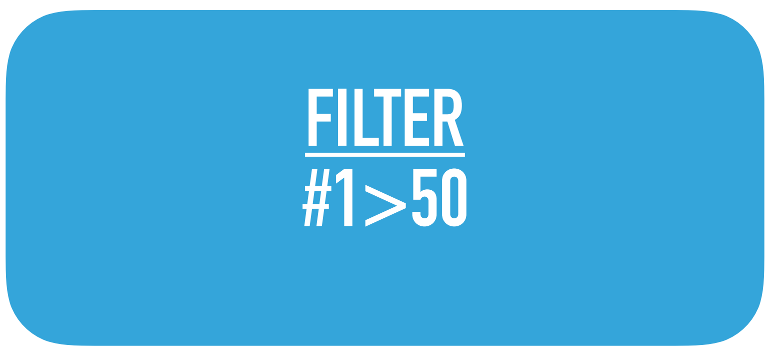
Column reference #1 fetches second column from input

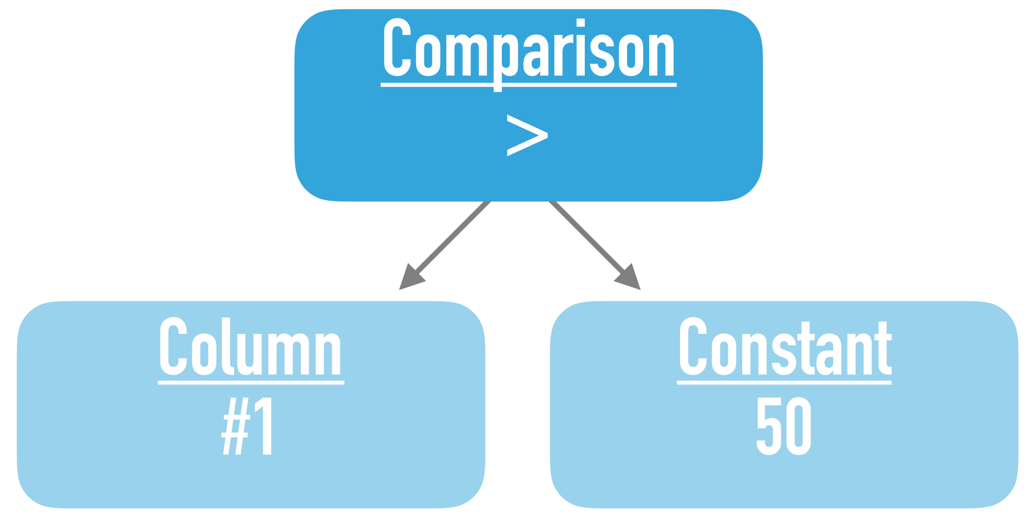
Expression Execution

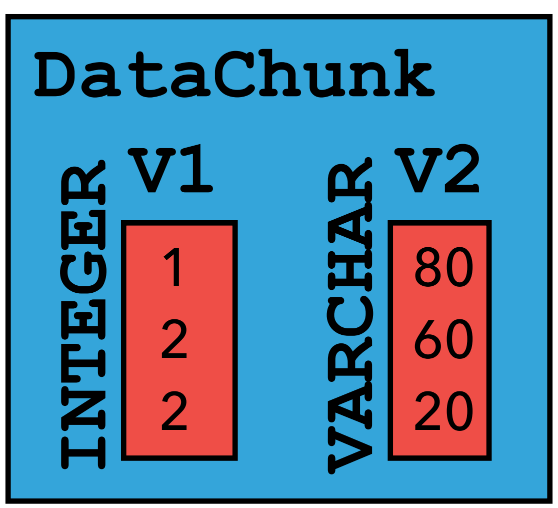


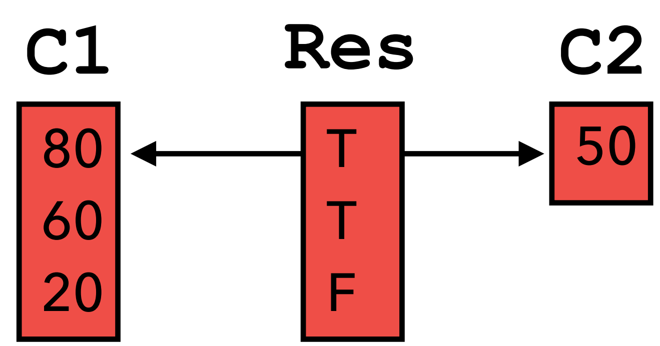
Constant is a single value

Expression Execution









Comparison
runs and returns
matching tuples

Hands On

Hands On

- **Assignment**: Implement a function in DuckDB
- Open issues for functions from other systems:
- https://github.com/cwida/duckdb/issues/193
- Implement one of those
  - For those that are successful, submit a PR!

Set Up & Testing

CWI

Set up

- **Set up:**
- 1. Download the source code
  - `git clone https://github.com/cwida/duckdb`
- 2. Compile the source code
  - First download CMake if you don't have it
  - **Linux/OSX:** `make debug`
  - **Windows:** Use CMake to generate a Visual Studio project, then build it from Visual Studio

CWI

Set up

DuckDB

- Tests are in the `test` directory
  - We use the Catch framework for tests
- Tests look like this:

```
TEST_CASE("Test scalar queries", "[scalarquery]") {
	unique_ptr<QueryResult> result;
	DuckDB db(nullptr);
	Connection con(db);
	con.EnableQueryVerification();

	result = con.Query("SELECT 42");
	REQUIRE(CHECK_COLUMN(result, 0, {42}));

	result = con.Query("SELECT 42 + 1");
	REQUIRE(CHECK_COLUMN(result, 0, {43}));

	result = con.Query("SELECT 2 * (42 + 1), 35 - 2");
	REQUIRE(CHECK_COLUMN(result, 0, {86}));
	REQUIRE(CHECK_COLUMN(result, 1, {33}));
```

Create in-memory database

Run queries & verify result

CWI

Set up

DuckDB

- Tests can be run as follows:
- **Linux/OSX:**
- `build/debug/test/unittest “Test scalar queries”`
- **Windows**
- Run `unittest` project
- Command line parameter: `“Test scalar queries”`

```
TEST_CASE("Test scalar queries", "[scalarquery]") {
	unique_ptr<QueryResult> result;
	DuckDB db(nullptr);
	Connection con(db);
	con.EnableQueryVerification();

	result = con.Query("SELECT 42");
```

Function Definition

Function Definition

- Each function has different **overloads**

Addition

+

- e.g. addition operator:
  - +(SMALLINT,SMALLINT)
  - +(INTEGER,INTEGER)
  - +(BIGINT,BIGINT)
  - …
- **Binder** chooses which version to use

Function Definition

- Set of permitted **implicit casts**

Addition

+

- TINYINT → SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE
- SMALLINT → INTEGER, BIGINT, FLOAT, DOUBLE
- INTEGER → BIGINT, FLOAT, DOUBLE
- BIGINT → FLOAT, DOUBLE
- FLOAT → DOUBLE

Function Definition

Addition

+

- Binder prefers to cast as little as possible
- e.g. TINYINT + INTEGER has multiple eligible options
- INTEGER + INTEGER will be chosen
  - Requires only one implicit cast
- Other options require two casts:
  - BIGINT + BIGINT, FLOAT+FLOAT, DOUBLE + DOUBLE

Function Definition

- The same binding rules apply to **functions**
  - **`substring(string, start, length)`**
  - Three parameters: **`VARCHAR`**, **`INTEGER`**, **`INTEGER`**
- Binder will automatically insert CAST if required
  - e.g. TINYINT → INTEGER
- In the code for **substring** we only need to implement the case with parameters `VARCHAR, INTEGER, INTEGER`

Expression Execution

- Code: how to add a function definition

```
set.AddFunction(ScalarFunction(
    "substring",          // name of function
    { SQLType::VARCHAR,   // argument list
      SQLType::INTEGER,
      SQLType::INTEGER },
    SQLType::VARCHAR,     // return type
    substring_function)); // pointer to function implementation
```

- Function code is implemented in `substring_function`

Creating a Simple Function

Creating a Simple Function

- Create a simple function:
  - `add_one(INTEGER) -> INTEGER`
- This function adds one to its integer input
- Returns the result

Creating a Simple Function

- **Step one: Create tests**
- Navigate to `test/sql/function`
- Create a new file: `test_add_one.cpp`
- Add it to `CMakeLists.txt` in that folder

CWI

Creating a Simple Function

- **Step one: Create tests**

```
TEST_CASE("Te Loading... function", "[function]") {
    unique_ptr<QueryResult> result;
    DuckDB db(nullptr);
    Connection con(db);
    con.EnableQueryVerification();

    REQUIRE_NO_FAIL(con.Query("CREATE TABLE integers(i INTEGER)"
    REQUIRE_NO_FAIL(con.Query("INSERT INTO integers VALUES "
                              "(1), (2), (3), (NULL)"));

    result = con.Query("SELECT add_one(1)");
    REQUIRE(CHECK_COLUMN(result, 0, {2}));
    result = con.Query("SELECT add_one(NULL)");
    REQUIRE(CHECK_COLUMN(result, 0, {Value()}));
    result = con.Query("SELECT add_one(i) FROM integers"
                       " ORDER BY 1");
    REQUIRE(CHECK_COLUMN(result, 0, {Value(), 2, 3, 4}));
    result = con.Query("SELECT add_one(i) FROM integers"
                       " WHERE i>1 ORDER BY 1");
    REQUIRE(CHECK_COLUMN(result, 0, {3, 4}));
}
```

Scalar tests

Table + selection vector tests

Creating a Simple Function

- Step one: Create tests

```
> build/debug/test/unittest "Test add one function"
Query failed with message: Catalog: Function with name add_one does not exist!

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
unittest is a Catch v2.4.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
Test add one function
-------------------------------------------------------------------------------
/Users/myth/Programs/duckdb/test/sql/function/test_add_one.cpp:7
...............................................................................

/Users/myth/Programs/duckdb/test/sql/function/test_add_one.cpp:18: FAILED:
  REQUIRE( CHECK_COLUMN(result, 0, {2}) )
with expansion:
  false

===============================================================================
test cases: 1 | 1 failed
assertions: 3 | 2 passed | 1 failed
```

Creating a Simple Function

- **Step two: Create the function**
- Navigate to `src/function/scalar`
- All function implementations are here
- In `math` directory, create new file: `add_one.cpp`
  - And add it to the CMakeLists.txt

Creating a Simple Function

- **Step two: Create the function**
- Add code to register function:

```
void AddOne::RegisterFunction(BuiltinFunctions &set) {
    set.AddFunction(ScalarFunction("add_one",
        { SQLType::INTEGER },
        SQLType::INTEGER,
        add_one_function));
}
```

Creating a Simple Function

- **Step two: Create the function**

- Now add actual function code:

```
static void add_one_function(
                ExpressionExecutor &exec,
                Vector inputs[]  index_t input_count,
                BoundFunctionExpression &expr, Vector &result) {
    result.Initialize(TypeId::INTEGER);
    VectorOperations::UnaryExec<int32_t, int32_t>(
        inputs[0], result, [&](int32_t input) {
        return input + 1;
    });
}
```

Initialize result

Loop over input & compute result

Creating a Simple Function

- **Step two: Create the function**
- Finally add some more bookkeeping code:
- `include/function/scalar/math_functions.hpp`

```
struct AddOne {
    static void RegisterFunction(BuiltinFunctions &set);
};
```

- `function/scalar/math_functions.cpp`

```
void BuiltinFunctions::RegisterMathFunctions() {
    Register<AddOne>();
```

Creating a Simple Function

- **Step two: Create the function**

- Now run the tests

```
> build/debug/test/unittest "Test add one function"
===============================================================================
All tests passed (6 assertions in 1 test case)
```

- Everything passes!

Creating a Simple Function

- Time to implement your own function
- **Advice**: Start with the `add_one` function
- Once that works, move on to different functions

Creating a Simple Function

- **Suggestions:**
- `RTRIM(VARCHAR) -> VARCHAR` [MySQL]
  - Remove spaces on right side of string
- `REVERSE(VARCHAR) -> VARCHAR` [MySQL]
  - Remove spaces on right side of string
- `REPEAT(VARCHAR, INTEGER) -> VARCHAR` [MySQL]
  - Repeat the specified string a number of times
- `INSTR(VARCHAR, VARCHAR) -> BOOL` [SQLite]
  - Returns true if second string is part of first string

Creating a Simple Function

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

- Feel free to ask any questions!