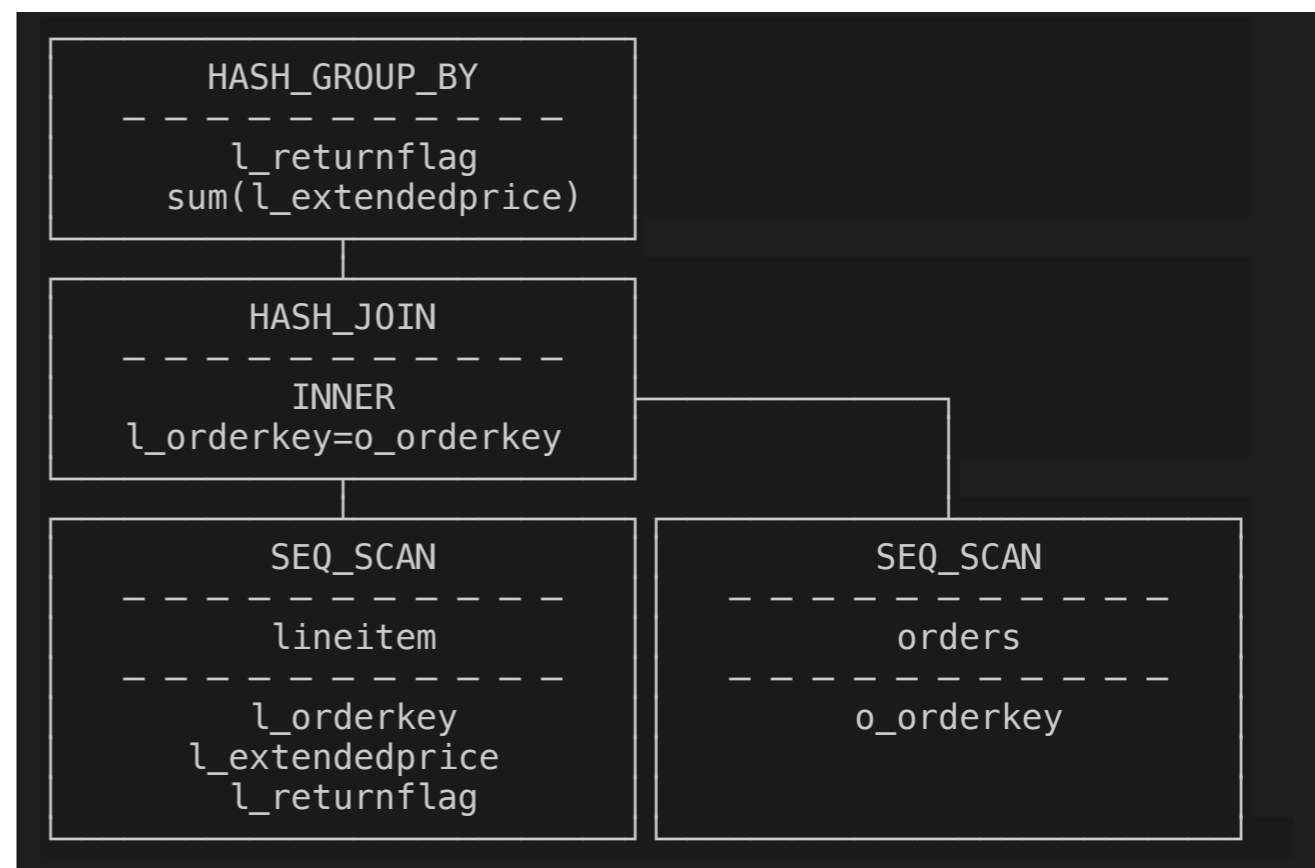


# Push-Based Execution in DuckDB

Mark Raasveldt

- DBMS transform SQL into query plans
- Query plans contain operators

```
SELECT SUM(l_extendedprice)
FROM lineitem
JOIN orders
ON (l_orderkey=o_orderkey)
GROUP BY l_returnflag;
```



- Operators need to be executed
- How?

- Two paradigms: Pull-based and push-based
- **Pull-based**
  - Pull data from other operators when required
- **Push-based**
  - Push data into operator when data is available

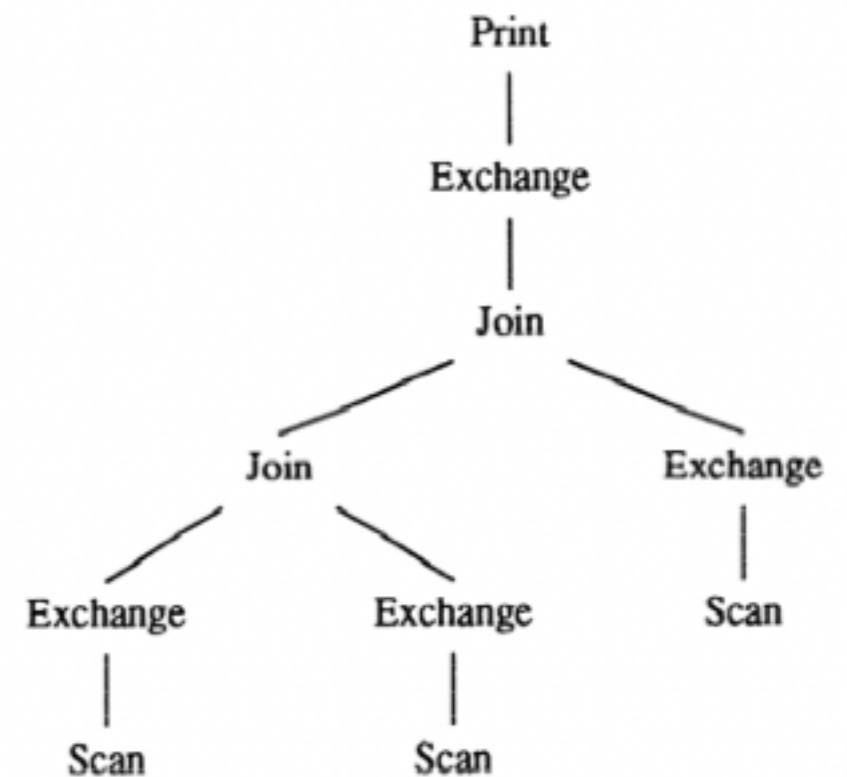
- DuckDB initially used a pull-based execution model
  - "Vector Volcano"
- Every operator implements **GetChunk**
- Query starts by calling **GetChunk** on the root
- Nodes recursively call **GetChunk** on children

## ● Simplified Projection Example

```
void Projection::GetChunk(DataChunk &result) {  
    // get the next chunk from the child  
    child->GetChunk(child_chunk);  
    if (child_chunk.size() == 0) {  
        return;  
    }  
  
    // execute expressions  
    executor.Execute(child_chunk, result);  
}
```

- **In this model:**
- Single-threaded execution is straightforward
- Multi-threaded not so much...
- How do you make operators parallelism-aware?

- <sup>D</sup> Exchange operator
- <sup>D</sup> Optimiser splits query plan into partitions
- <sup>D</sup> Partitions can be executed independently
- <sup>D</sup> **Problems:**
  - <sup>D</sup> Load imbalance issues
  - <sup>D</sup> Plan explosion
  - <sup>D</sup> Added materialization costs



- D Morsel-Driven Parallelism
- D Individual operators are parallelism-aware
- D Query is divided into pipelines
- D Pipelines are executed in parallel

**Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age**

Viktor Leis<sup>1</sup>, Peter Boncz<sup>2</sup>, Alfons Kemper<sup>1</sup>, Thomas Neumann<sup>1</sup>

<sup>1</sup>Technische Universität München, <sup>2</sup>CWI  
 {leis,kemper,neumann}@in.tum.de, pboncz@cwi.nl

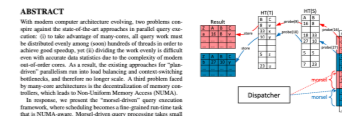


Figure 1: Idea of morsel-driven parallelism. © P. Leis, P. Boncz, T. Neumann

Boncz's forthcoming presentation series in Berlin, DE, which can be found on his slides, will be available. We use the term 'morsel' for our data chunks, with this as a nod to the past.

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O-bound, and the large parallel compute resources of many cores can be fully exploited. Unfortunately, the trend to more memory consumption and the chip-and-board de-commodification of memory systems, which was needed to scale throughput in high-memory nodes, has now led to a new trend: NUMA. In contrast to the traditional multi-processor architecture, NUMA architectures use data from various processors to which they do not have direct access. Therefore, more query parallelization needs to take RAM and cache hierarchies into account. In particular, the NUMA access of the RAM has to be considered carefully to ensure the database work (mostly on NUMA local data).

One of the main reasons to design a form of parallelism independent of database systems is the fact that parallel processing led to the development of the Volcano [12] model, which is now being largely abandoned. Parallelism is now being replaced by so-called "morsel" operators that cover tight strata between multiple morsels, each receiving identical portions of the query plan. Such implementations of the Volcano model can be called parallelism, the operator actually determines a query variable into how many morsels should run, instantiates one query operator plan for each morsel, and connects these with outgoing operators.

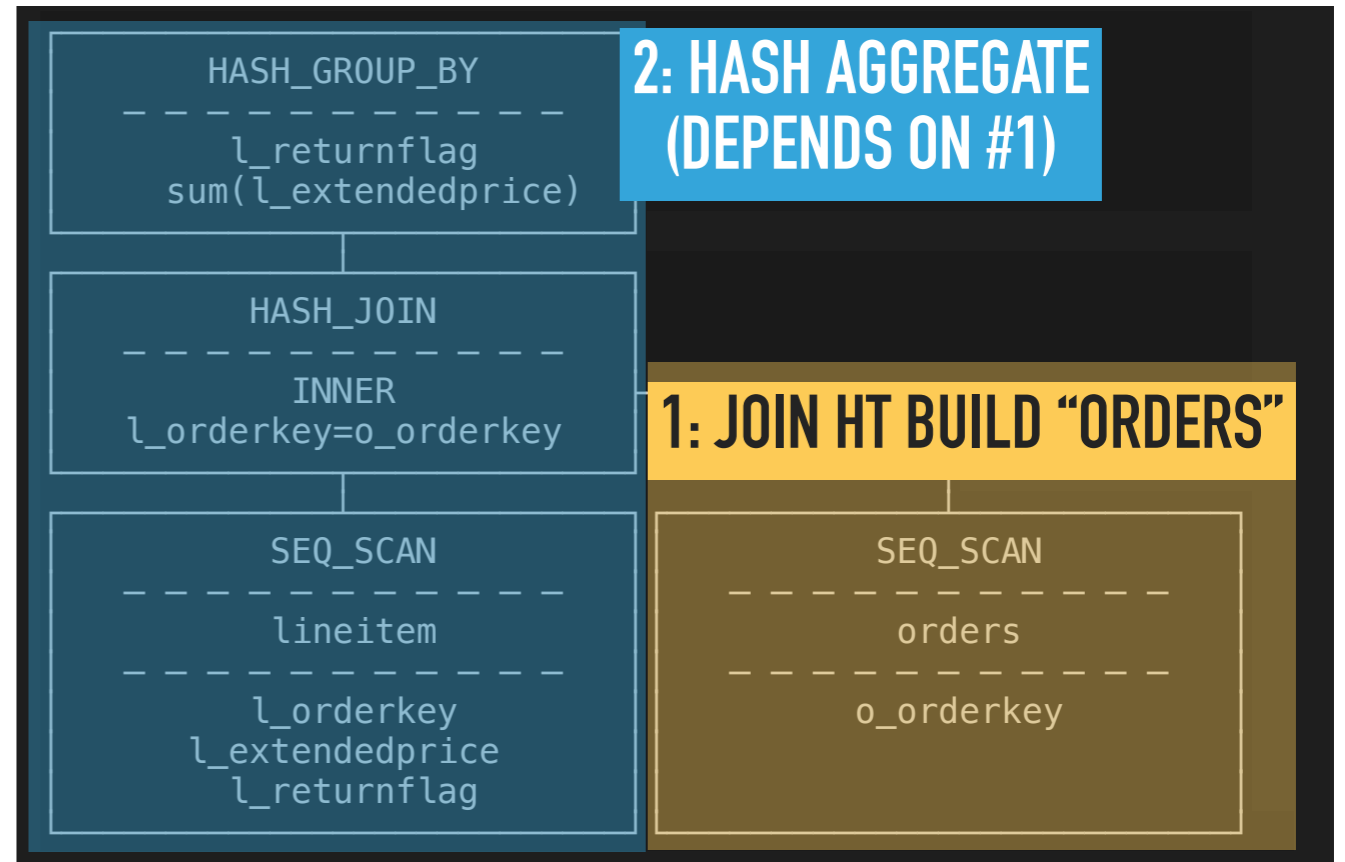
In this paper we present the morsel-driven query execution framework, which was designed for our main-memory database system in [11]. Our approach is outlined in Figure 1 for the three-way join query  $R \bowtie S \bowtie T$ . Parallelism is achieved

[2014] Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age



```

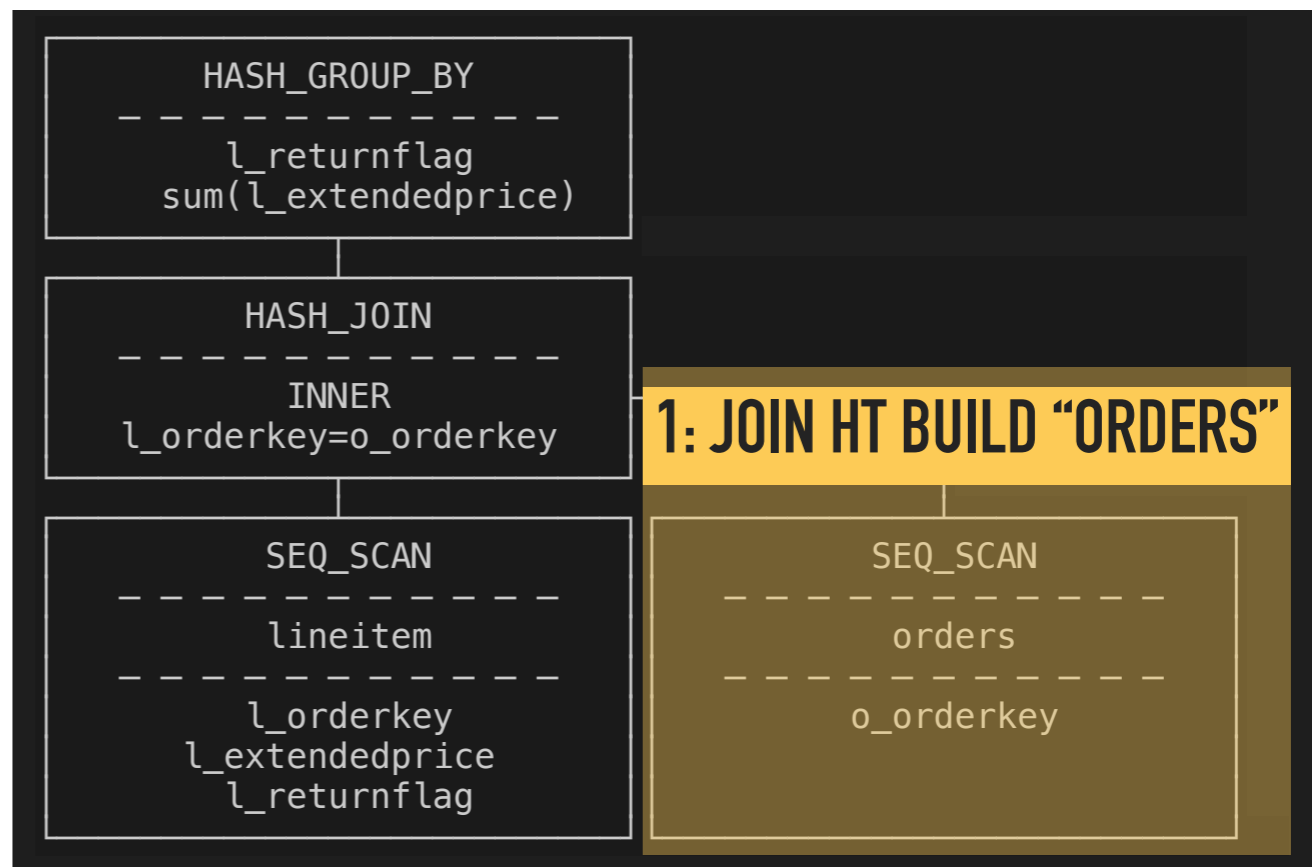
SELECT SUM(l_extendedprice)
FROM lineitem
JOIN orders
ON (l_orderkey=o_orderkey)
GROUP BY l_returnflag;
    
```



Contention happens **at endpoints**

**Source:** Scan of orders

**Sink:** HT build of join



Use **parallelism-aware** operators at endpoints

Other operators (HT probe, projection, filter, etc...) don't need to be aware

- Sink Interface

- Sinks can define **global** and **local** states

- **Sink** is called until all data is exhausted

- **Combine** is called (once per thread)

- **Finalize** is called (once)

```

void Sink(
    ExecutionContext &context,
    GlobalSinkState &gstate,
    LocalSinkState &lstate,
    DataChunk &input);
void Combine(
    ExecutionContext &context,
    GlobalSinkState &gstate,
    LocalSinkState &lstate);
void Finalize(
    ClientContext &context,
    GlobalSinkState &gstate);
    
```

## ● Simplified Hash Join Example

```
void HashJoin::Sink(DataChunk &input) {  
    // build the hash table  
    BuildHashTable(input);  
}  
  
void HashJoin::GetChunk(DataChunk &result) {  
    // probe the hash table  
    left_child->GetChunk(child_chunk);  
    ProbeHashTable(child_chunk, result);  
}
```

- Pipelines are run by pulling from child of sink
- After child is exhausted, call **Combine/Finalize**
- Mix of push/pull: sink is push, rest is pull...

```

void RunPipeline() {
    // fetch data from child of sink
    while(sink->child->GetChunk(child_chunk)) {
        sink->Sink(child_chunk, ...);
    }
    // finished: combine
    sink->Combine(...);
    if (all_threads_finished) {
        // all threads are finished: finalize sink
        sink->Finalize(...)
        ScheduleNextPipeline();
    }
}

```

- How do we partition **Sources**?
- Not as straightforward...
- Sources are located at the bottom of the pipeline

- Set up a **tasks** in thread context
- Tasks define how the scan is partitioned
- Read those tasks in the **GetChunk**

```
void TableScan::GetChunk() {  
    // check if there is a task scheduled for this operator  
    table.ScanTask(thread_context.tasks.find(this));  
}
```

- This mostly works
- Problems:
  - Data flow duplicated in every operator
  - No clean interface for source parallelism
  - How to parallelize **UNION** nodes?
  - How to parallelize **FULL/RIGHT** outer joins?
  - Scan Sharing?
  - Async I/O?



# Push-Based Execution

- What is push-based execution?
- Our previous model was **pull-based**:
  - **GetChunk** called when an operator requires data
- **Push-based** is the other way around
  - Push data into operators
- **Sink interface** is already push-based!

- **Push-Based** moves data flow **out of operators**
  - Data flow is handled in central location
  - Simplifies implementation of operators
    - But reduces flexibility!

- Define **Operator** and **Source** interface
- **Operator** processes data
  - Projection, Filter, Hash Probe, ...
- **Source** emits data
  - Table scan, aggregate HT scan, ORDER BY scan, etc

## ● Operator Interface

```
OperatorResultType Execute(  
    ExecutionContext &context,  
    DataChunk &input,  
    DataChunk &chunk,  
    OperatorState &state);
```

- **Execute** takes an input chunk, and outputs another chunk

- **Projection** is straightforward

```
void Projection::Execute(DataChunk &input, DataChunk &result)
{
    executor.Execute(input, result);
}
```

- **Hash Probe** seems straightforward...

```
void HashJoin::Execute(DataChunk &input, DataChunk &result) {  
    Probe(input, result);  
}
```

- How do we handle multiple matches per tuple?
  - 1 input entry can lead to many output entries...
- Operators need a way of signalling they are not done processing the input

- **OperatorResultType** is used for this

```
enum OperatorResultType {
    NEED_MORE_INPUT,
    HAVE_MORE_OUTPUT,
    FINISHED
};
```

- **NEED\_MORE\_INPUT**: Operator will be called with a new input chunk
- **HAVE\_MORE\_OUTPUT**: Operator will be called with the same input chunk
- **FINISHED**: The operator will not be called again, terminates the pipeline



```
enum OperatorResultType {
    NEED_MORE_INPUT,
    HAVE_MORE_OUTPUT,
    FINISHED
};
```

- **FINISHED** required to interrupt execution
  - Happens naturally in a pull-based model
  - e.g. **LIMIT** in pull-based simply stops pulling
- In push-based, we need to signal to the execution loop that we finished early

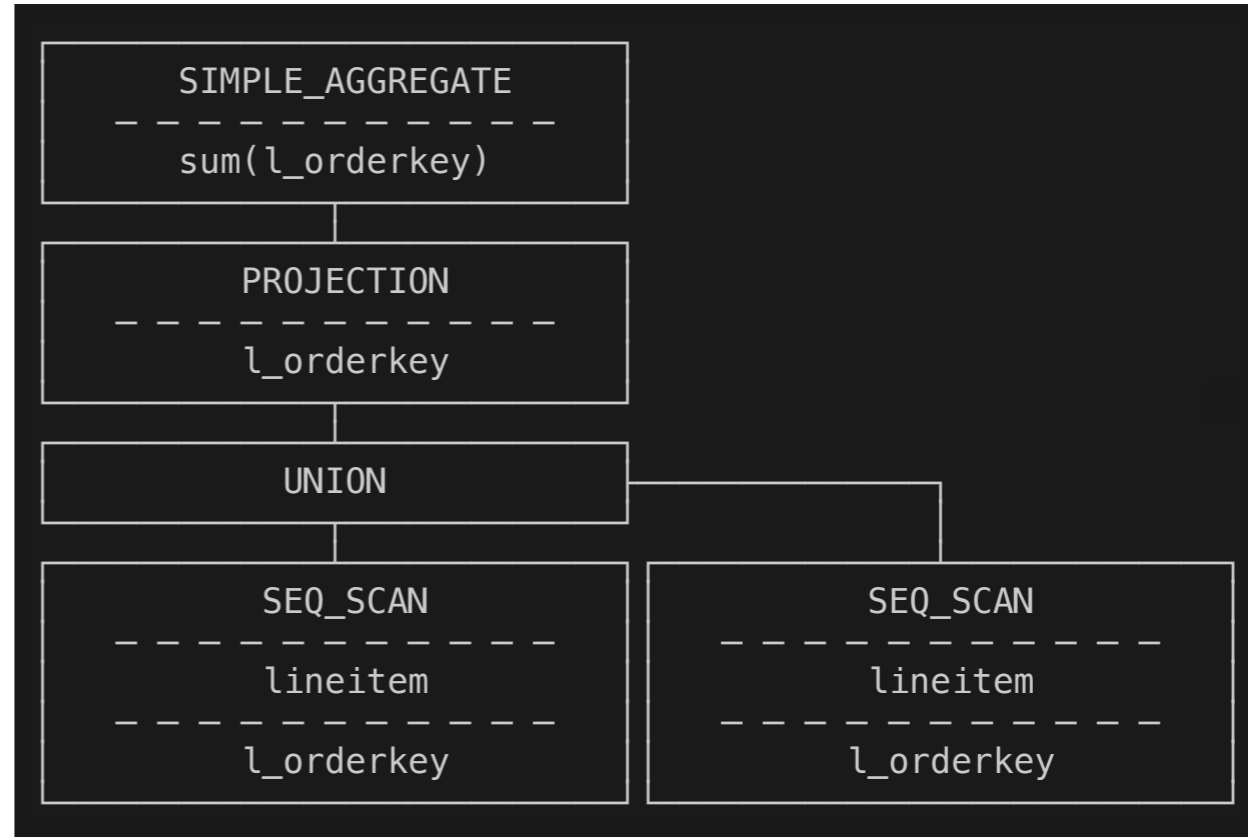
- **Source Interface**
- Similar to **Sink** interface
- **Global** and **local** states
- **GetData** is called until no more data remains
  - Or pipeline is cancelled earlier

```
void GetData(  
    ExecutionContext &context,  
    DataChunk &chunk,  
    GlobalSourceState &gstate,  
    LocalSourceState &lstate);
```

# Pipeline Events

```

SELECT SUM(l_orderkey)
FROM
(
  SELECT *
  FROM lineitem
  UNION ALL
  SELECT *
  FROM lineitem
)
    
```



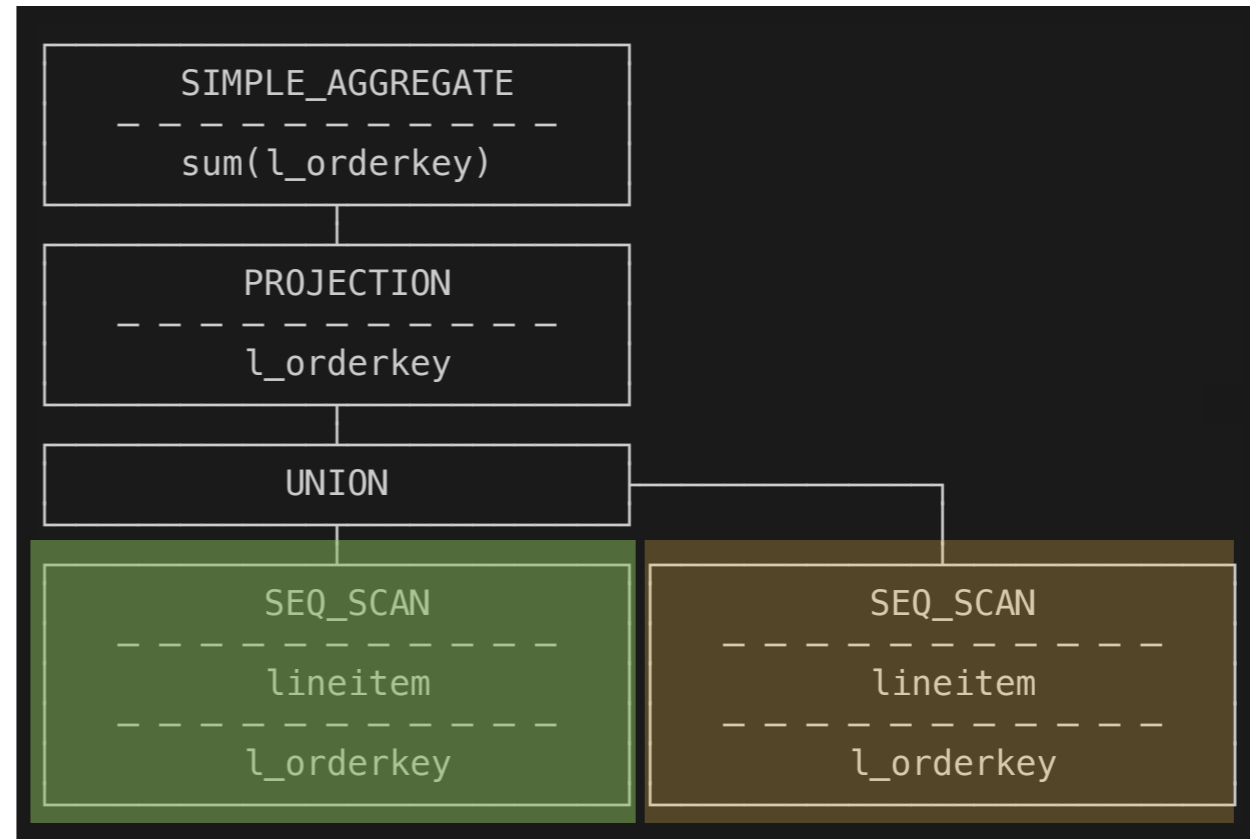
- **UNION nodes**
- How do we execute unions?

- **Pull-Based:** Easy, we control the flow

```
void Union::GetChunk(DataChunk &result) {  
    if (!left_done) {  
        left_child->GetChunk(result);  
        if (result.size() > 0) {  
            return;  
        }  
        left_done = true;  
    }  
    right_child->GetChunk(result);  
}
```

- How do we do it **push-based**?

```
SELECT SUM(l_orderkey)
FROM
(
  SELECT *
  FROM lineitem
  UNION ALL
  SELECT *
  FROM lineitem
)
```



●<sub>D</sub> **Push-Based Union**

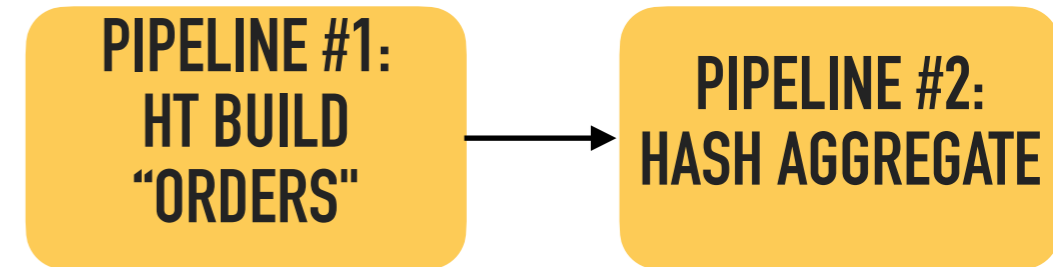
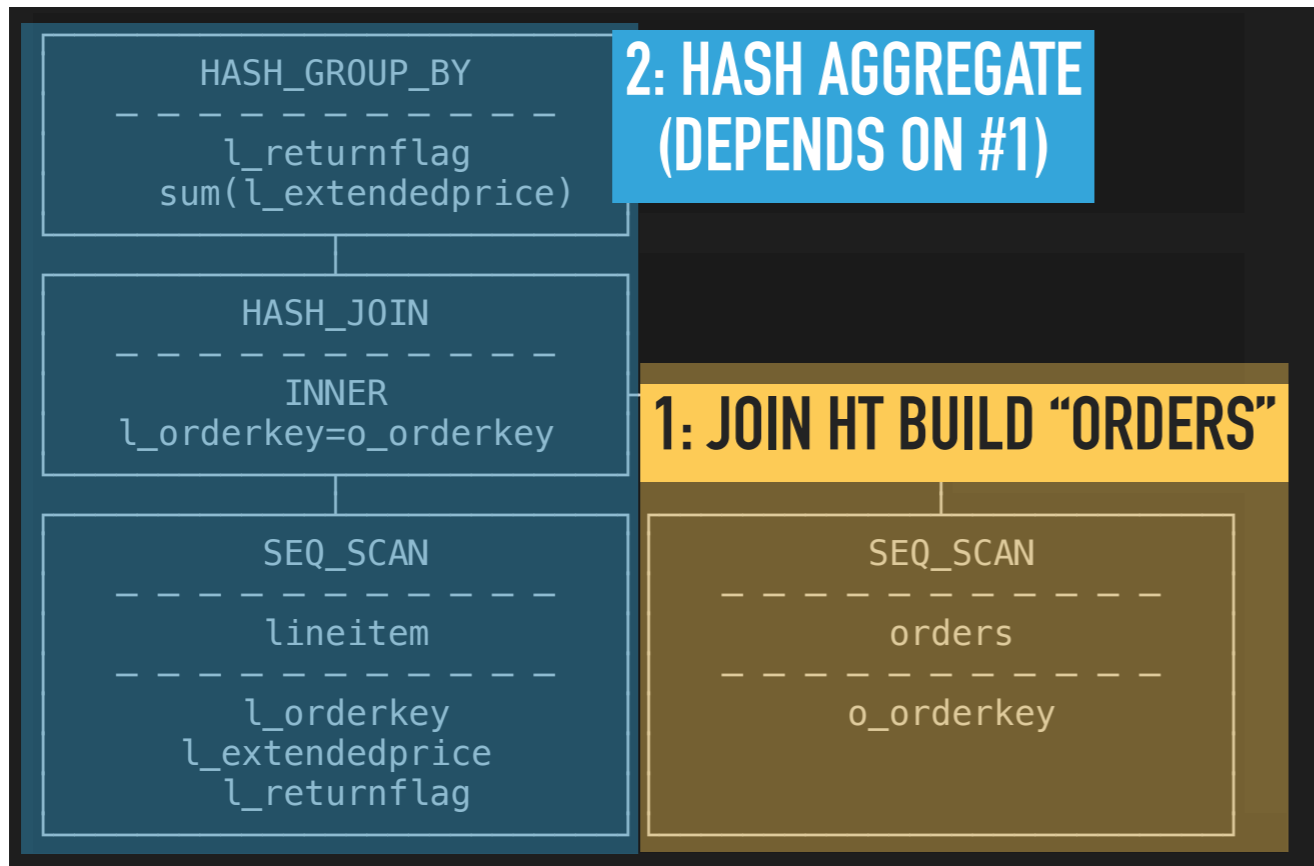
●<sub>D</sub> Create two pipelines with same sink

●<sub>D</sub> Or more, if there are more unions

●<sub>D</sub> **Sink::Finalize** only after **all** pipelines are done!

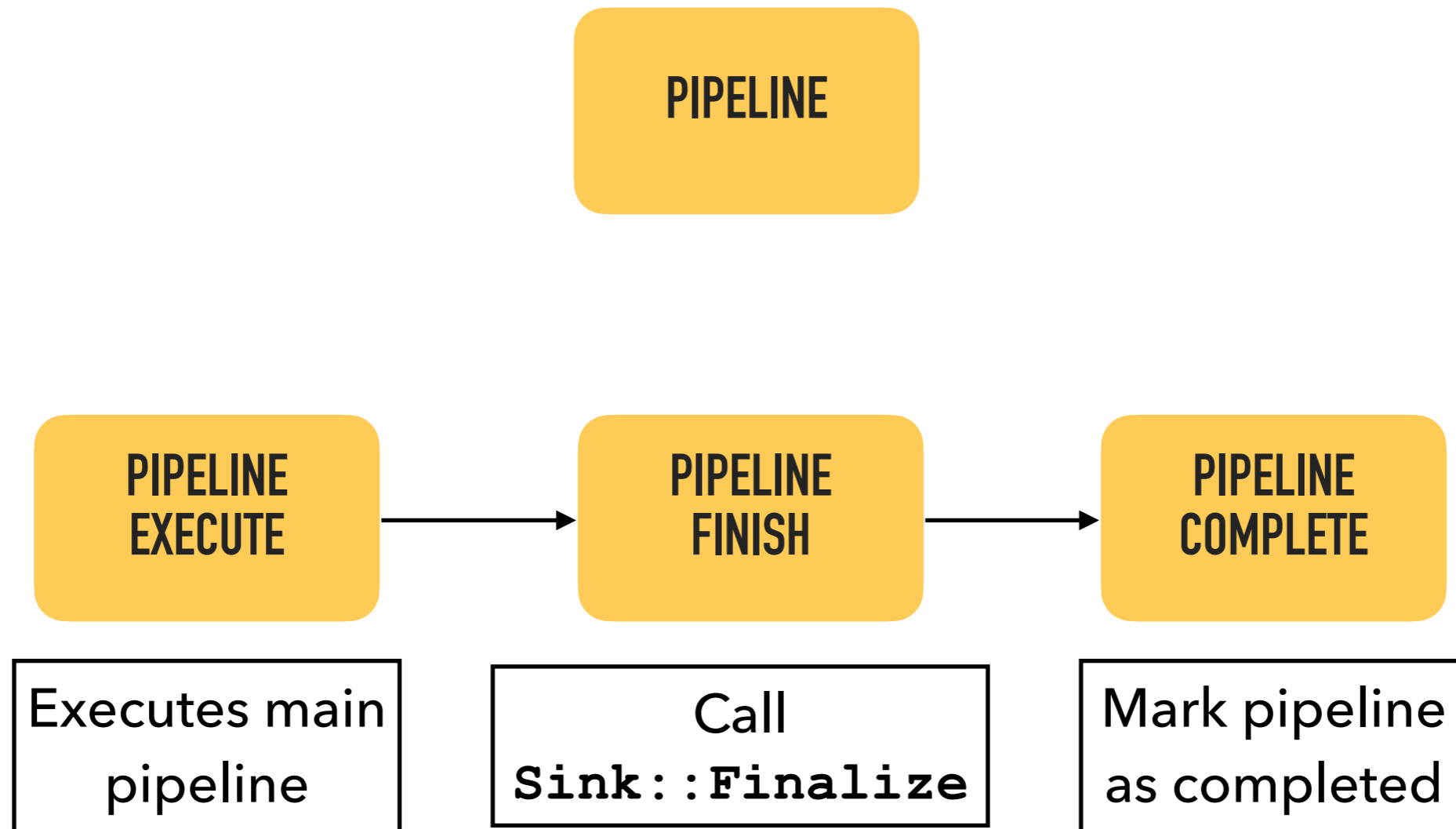
# ● Pipeline Scheduling

```
SELECT SUM(l_extendedprice)
FROM lineitem
JOIN orders
ON (l_orderkey=o_orderkey)
GROUP BY l_returnflag;
```



● How do we handle the **Union** case here?

- Split up **Pipeline** into **Events**
- Schedule those **Events**

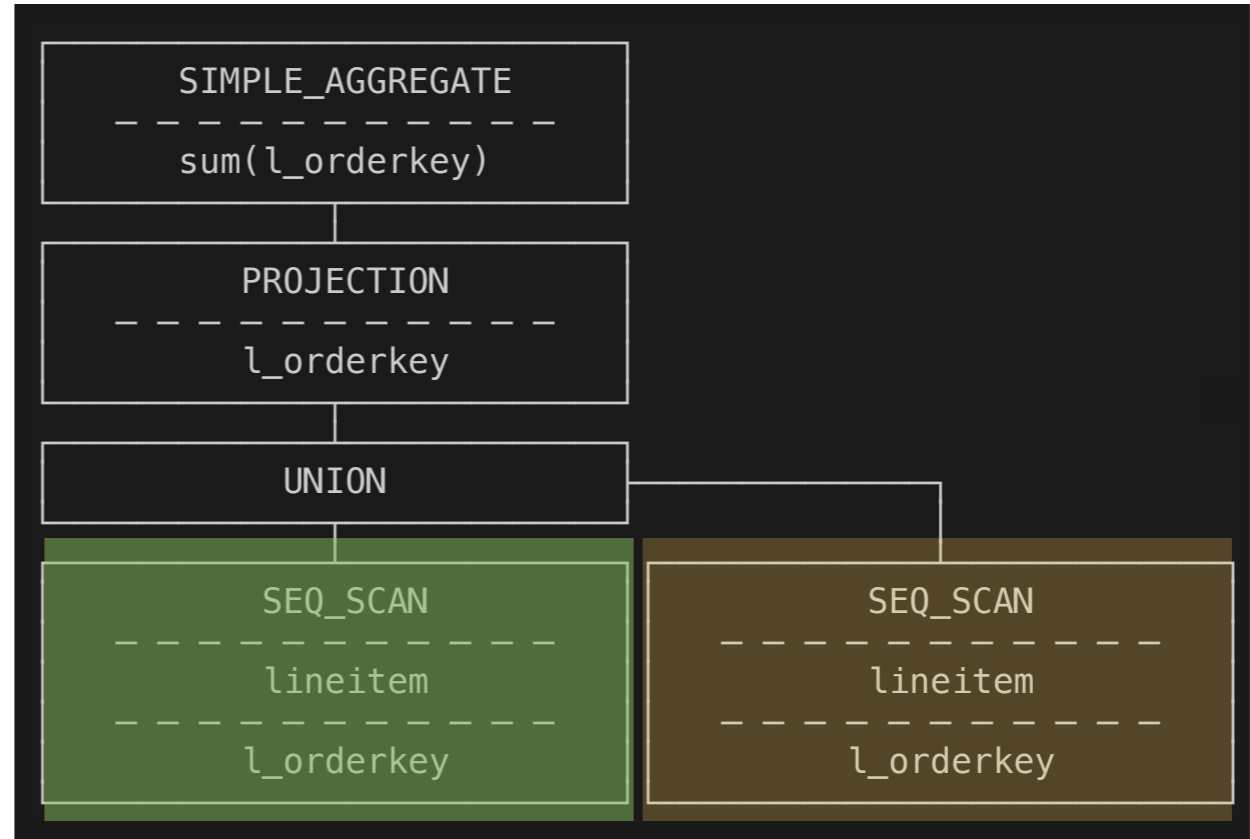




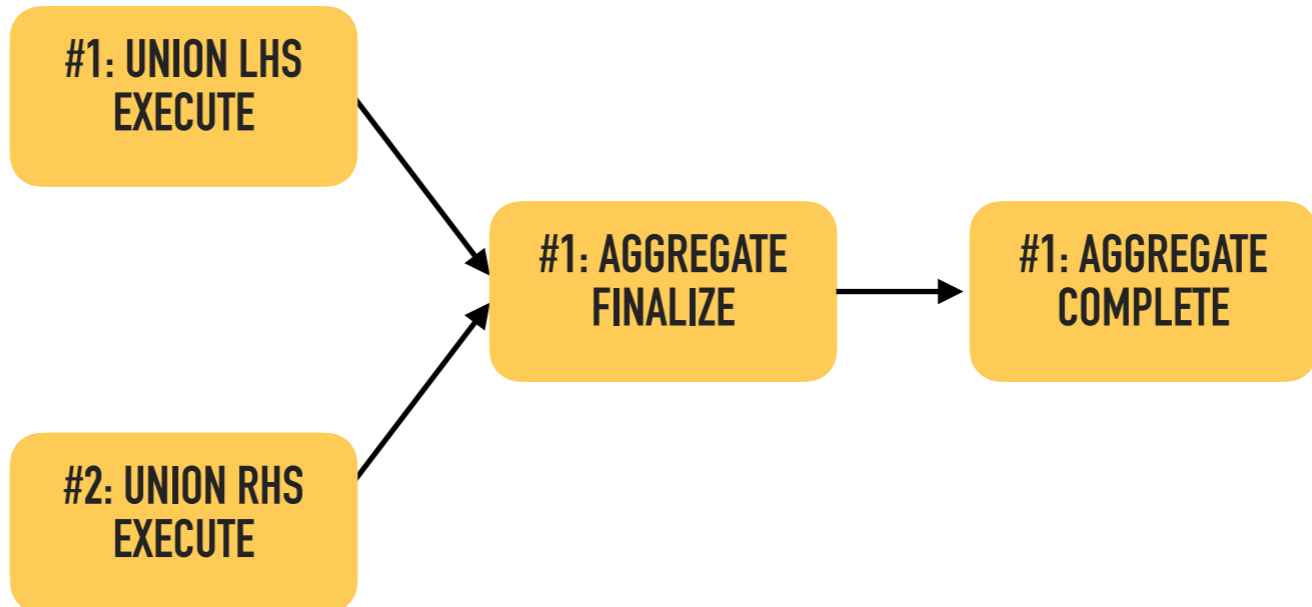
```

SELECT SUM(l_orderkey)
FROM
(
  SELECT *
  FROM lineitem
  UNION ALL
  SELECT *
  FROM lineitem
)

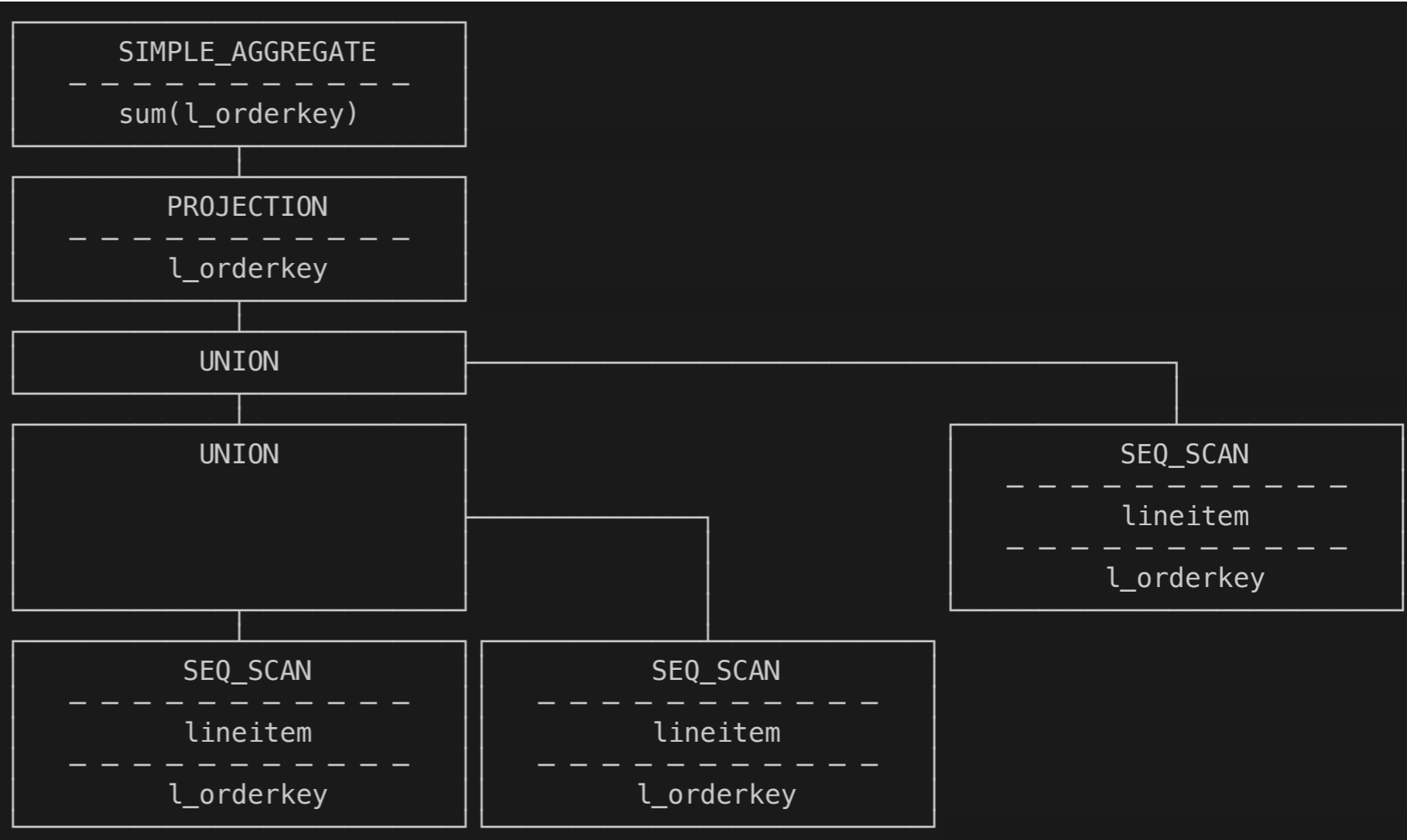
```



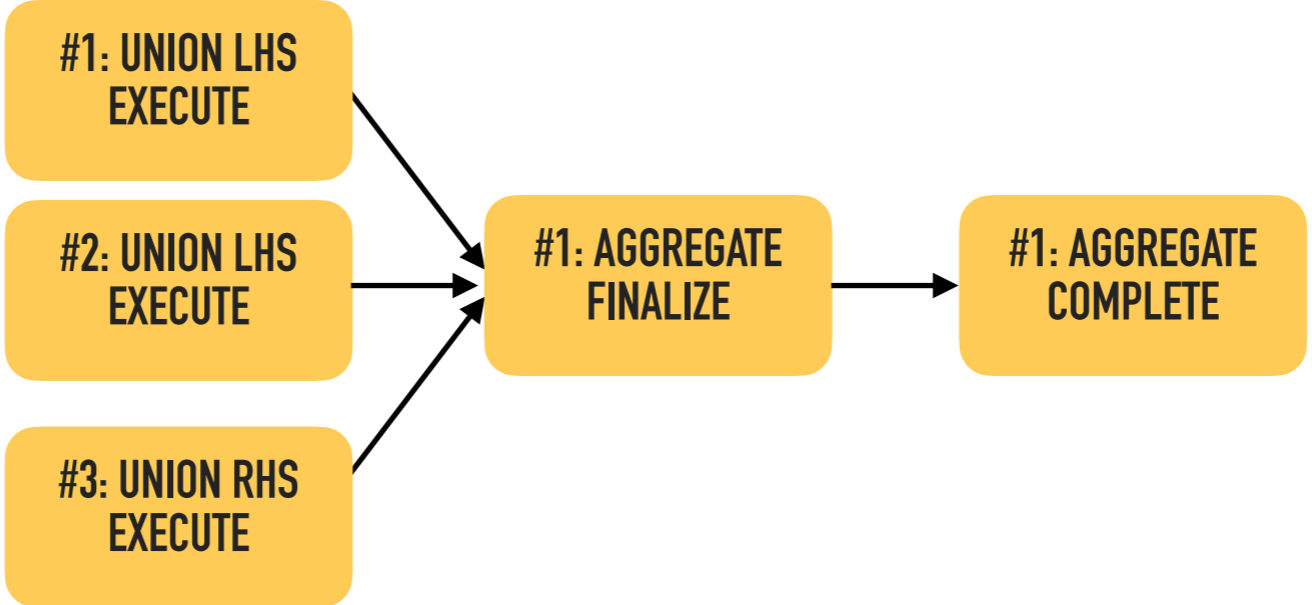
Now we can schedule multiple unions that will call Finalize once



```
SELECT SUM(l_orderkey)
FROM
(
  SELECT *
  FROM lineitem
  UNION ALL
  SELECT *
  FROM lineitem
  UNION ALL
  SELECT *
  FROM lineitem
)
```



Can stack multiple unions

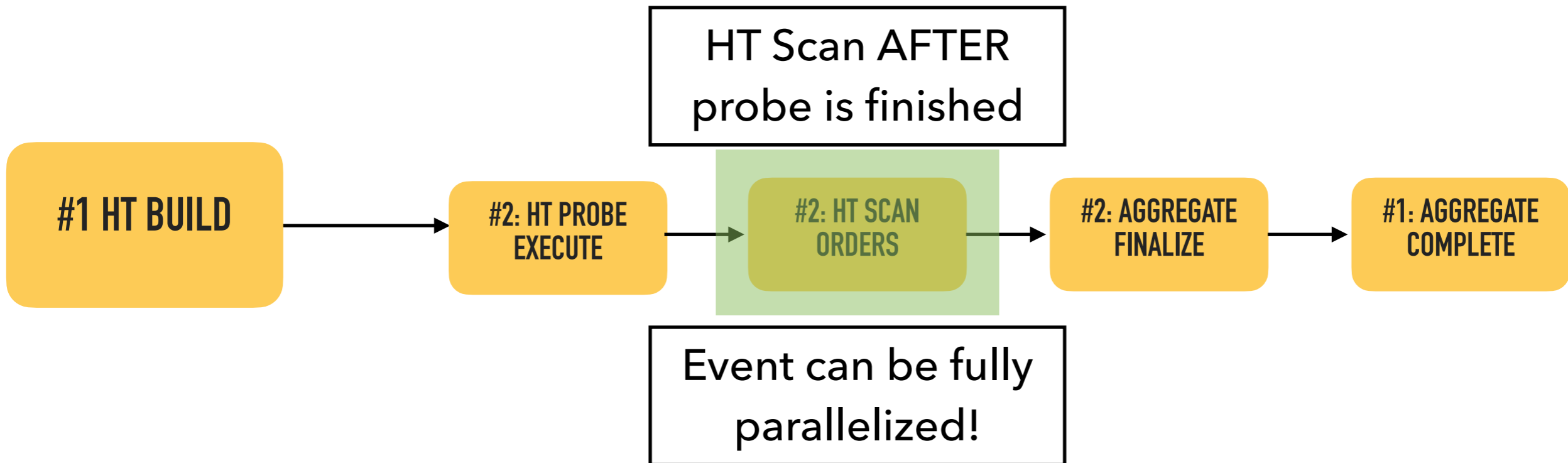
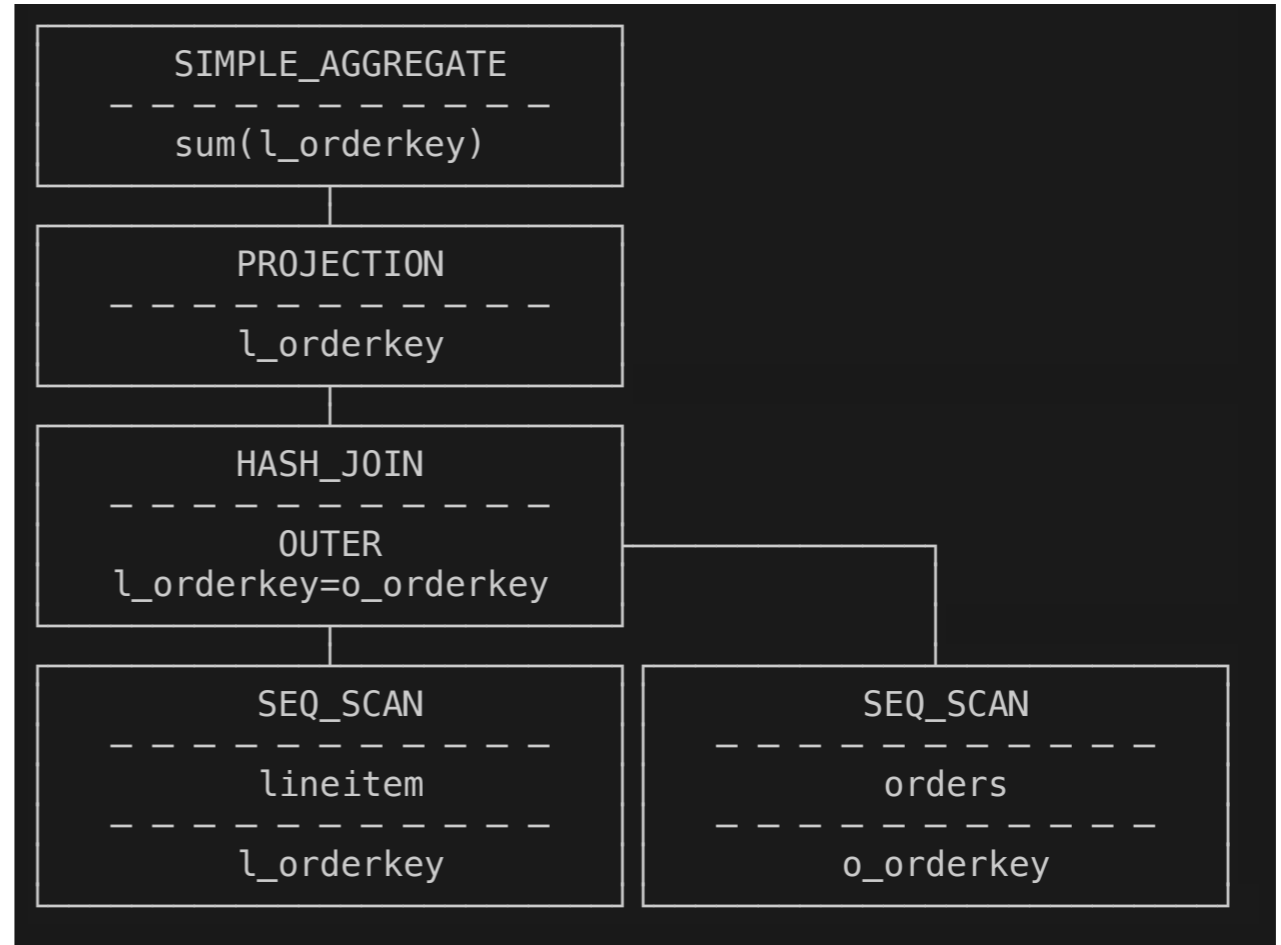


- <sup>D</sup> **Full/Right Outer Joins** have similar challenge
- <sup>D</sup> Three phases:
  - <sup>D</sup> Build HT
  - <sup>D</sup> Probe HT
  - <sup>D</sup> Scan HT (after **ALL** probing is finished)

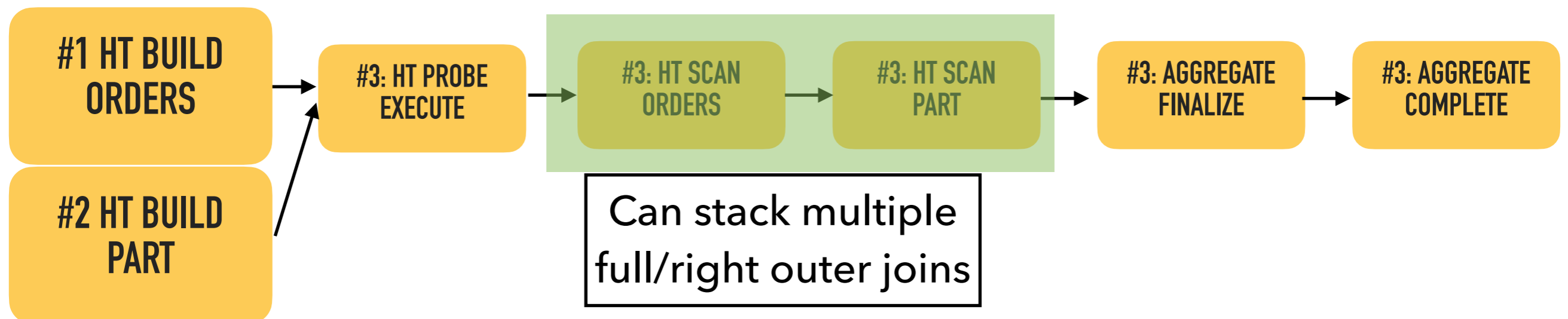
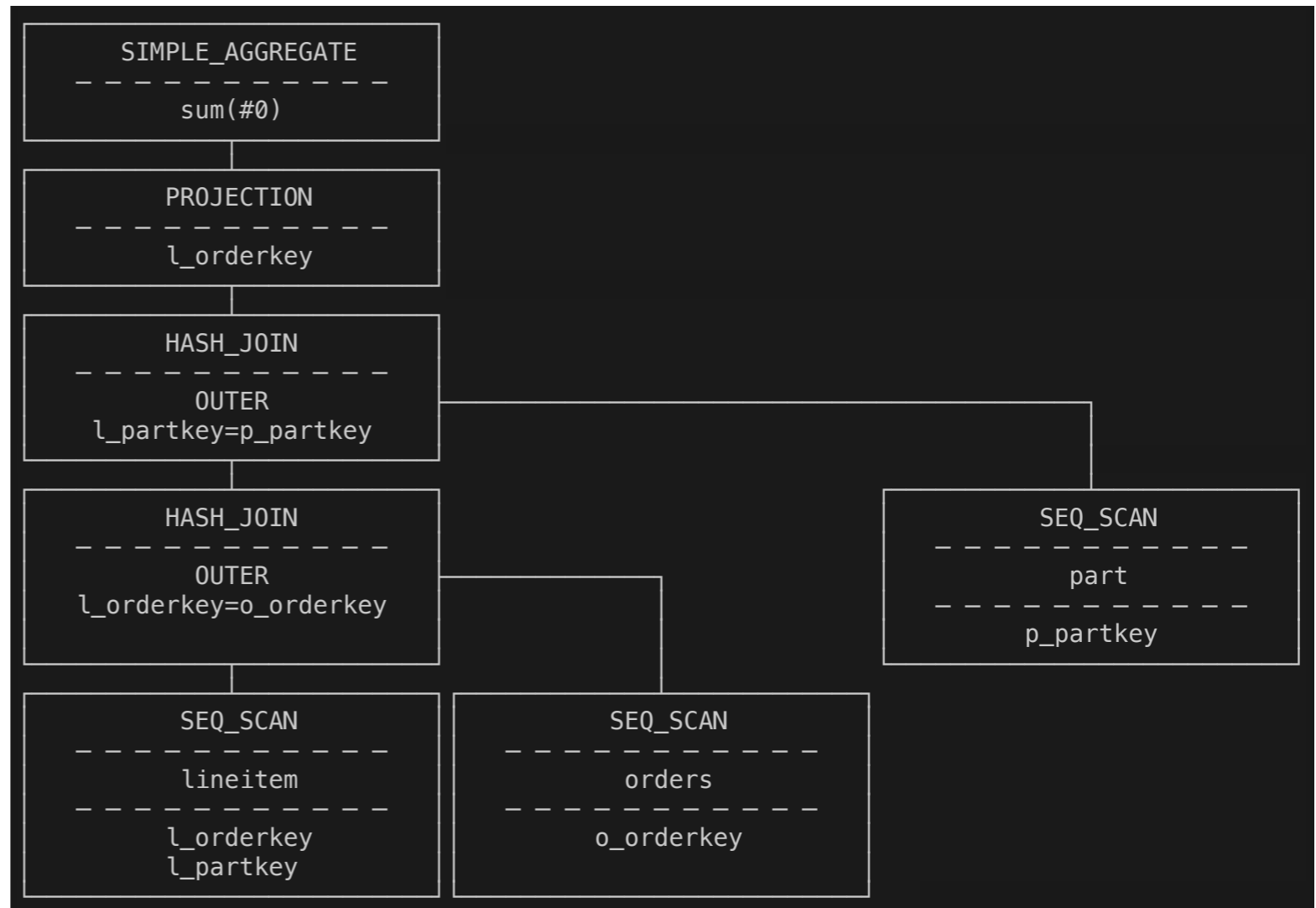
```

SELECT sum(l_orderkey)
FROM lineitem
FULL OUTER JOIN orders
ON (l_orderkey=o_orderkey);

```

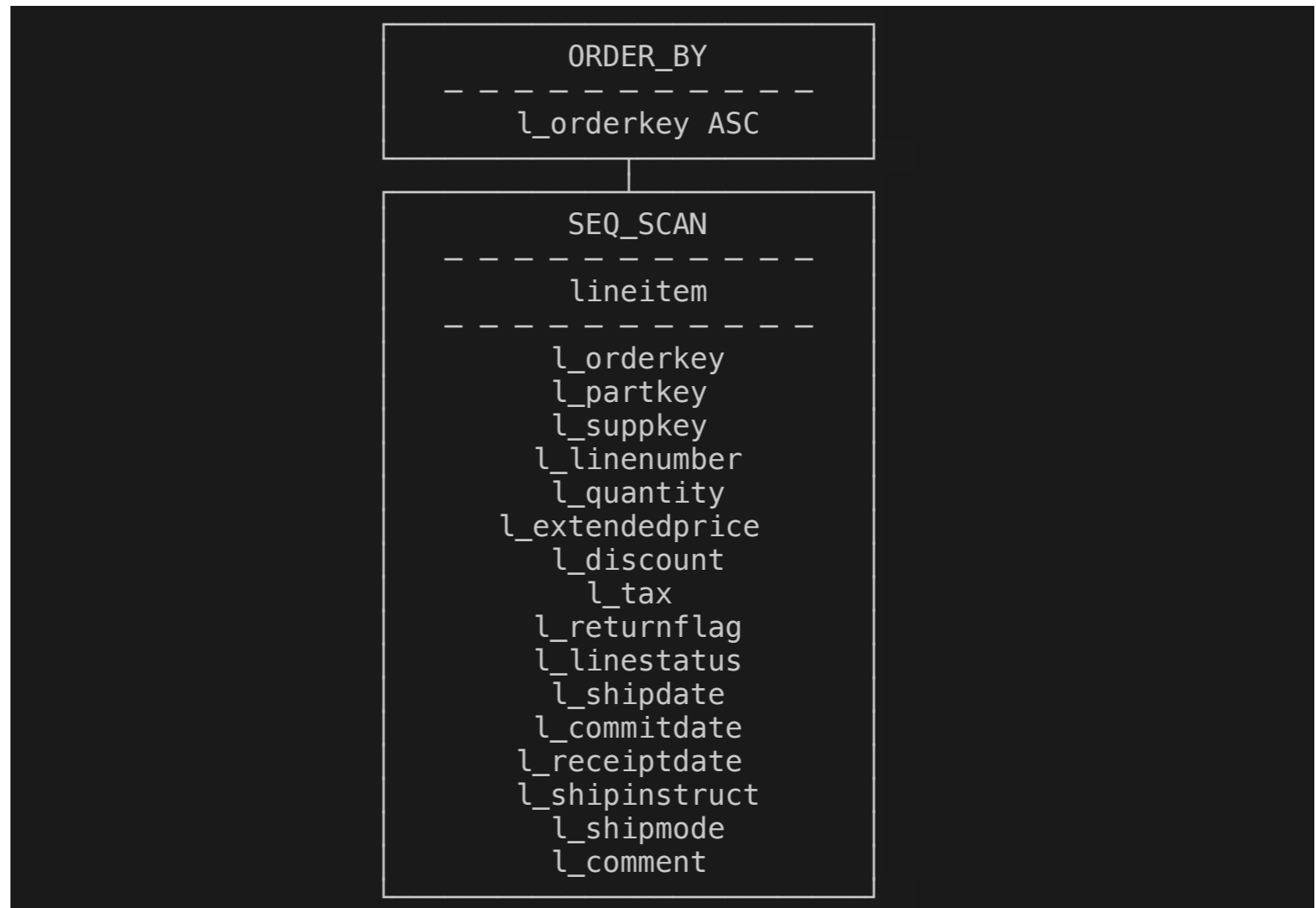


```
SELECT sum(l_orderkey)
FROM lineitem
FULL OUTER JOIN orders
  ON (l_orderkey=o_orderkey)
FULL OUTER JOIN part
  ON (l_partkey=p_partkey);
```



- Sinks often have an expensive Finalize step
  - e.g. **order by** - merging sorted segments
- Need to be executed in parallel

```
SELECT *
FROM lineitem
ORDER BY l_orderkey;
```



**Sink::Finalize** can schedule additional events

# Future Work

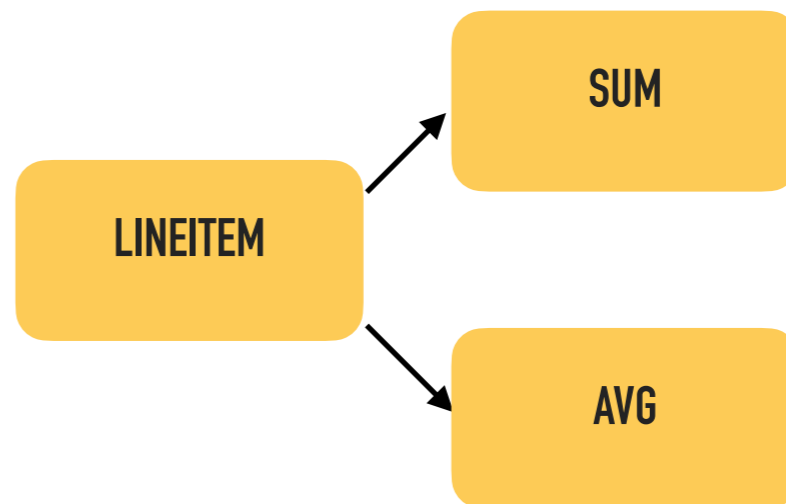


- <sup>D</sup> **Scan Sharing (TODO)**

- <sup>D</sup> Detect pipelines that have the same source

- <sup>D</sup> Scan once, sink into multiple pipelines

```
SELECT SUM(l_orderkey)
FROM lineitem
UNION ALL
SELECT AVG(l_orderkey)
FROM lineitem;
```



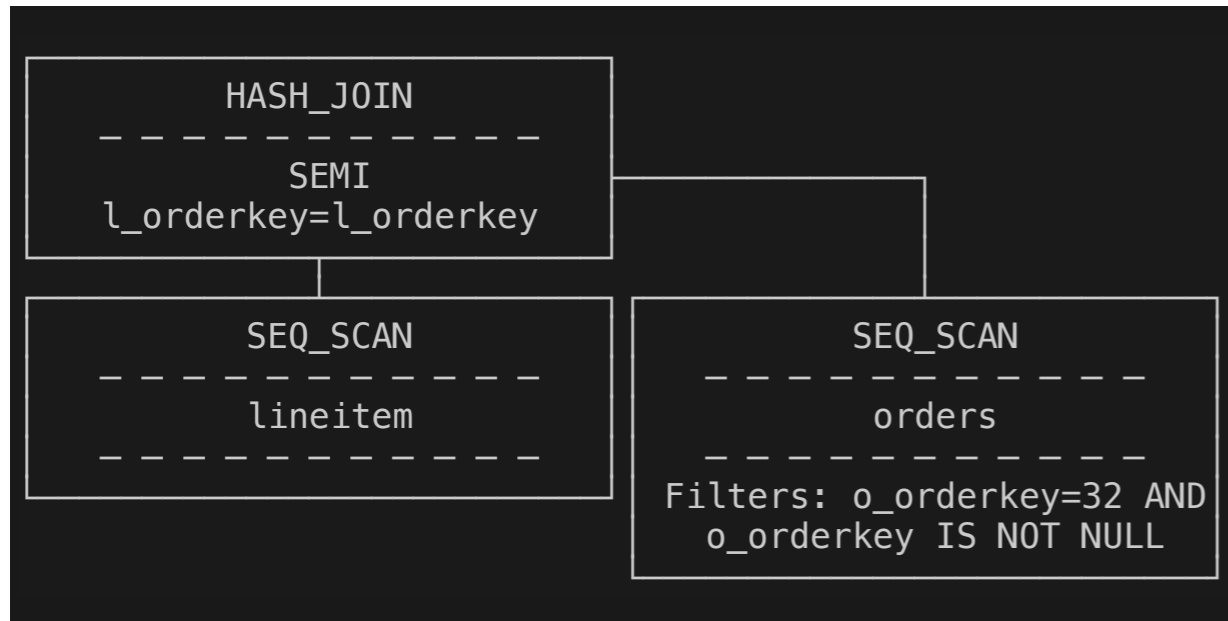
- <sup>D</sup> Complicated by **projection & filter pushdown**

- <sup>D</sup> Disjoint projections -> scan sharing not useful\*

\* Unless we are scanning a row-store

- **Async I/O (TODO)**
- Current scans are still pull-based
- Fine for in-memory data
- Reading from disk/http/etc -> stall on read
- Async I/O solves this by pushing I/O to background threads
- When I/O completes, push data into pipeline

- **Hybrid Early/Late Materialization**
- Async I/O prefetches all required columns
- Early materialization
- Late materialization at times preferable
  - e.g. query with selective predicate on one column



```

SELECT *
FROM lineitem
WHERE EXISTS
(SELECT *
FROM orders
WHERE l_orderkey=o_orderkey
AND o_orderkey=32);
    
```

- <sup>D</sup> This query selects a few rows
  - <sup>D</sup> But reads all columns of entire lineitem table
- <sup>D</sup> **Early materialization:** read entire lineitem table
- <sup>D</sup> **Late materialization:** read l\_orderkey column and few rows from other columns

- <sup>D</sup> **Hybrid Early/Late Materialization**

- <sup>D</sup> Lazy vectors enable hybrid of early/late materialization

- <sup>D</sup> When a vector is first used, fetch data from disk

- <sup>D</sup> Conflicts with Async I/O!

- <sup>D</sup> **Potential solution:** Hybrid Async I/O

- <sup>D</sup> Prefetch with async I/O

- <sup>D</sup> Stop prefetching for a column if we detect column data is not required

**That's all folks!**  
**Thanks for listening!**