E0-261

Database Management Systems

Project #08 DuckDB

Akash Maji [24212]

Utkarsh Sharma [24116]

Agenda

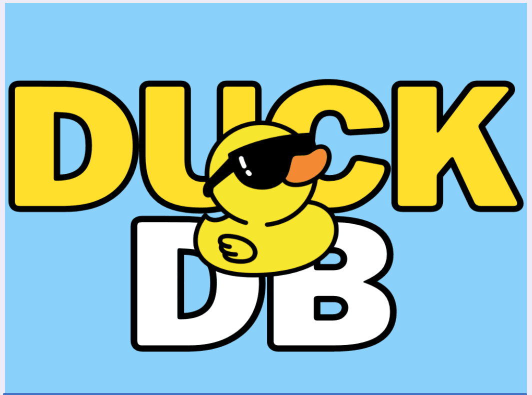
1. About DuckDB
2. Project Overview
3. DuckDB Architecture
4. Execution Model
5. New Join Operator
6. New GroupJoin Operator
7. Test & Experiments
8. Summary & Future Work

DuckDB

Let's Dive In!

About DuckDB

- **DuckDB** is an open-source analytical (OLAP) database system.
- It runs as a single-process without needing server connection.
- The entire database can sit in a *file* with .db or .duckdb extension.



I am so cool, ain't I ?

Features:

- Fast, Reliable, Portable, Open-Source
- Analytical Support
- Columnar, Compressed, Block-Based Storage

Project Overview

In this project,

- we aim to explore the interface provided by DuckDB,
- and implement certain operators.

1. **Firstly**, we will discuss the implementation of a new 'Join' operator.
2. **Secondly**, we change the query planner and optimizer to pick our join operator.
   - whenever a predetermined condition is met.
3. **Thirdly**, we implement a new 'GroupJoin' operator.
   - which will be invoked if a query containing a 'Join' followed by a 'Group By' is handed to the engine.

DuckDB Architecture

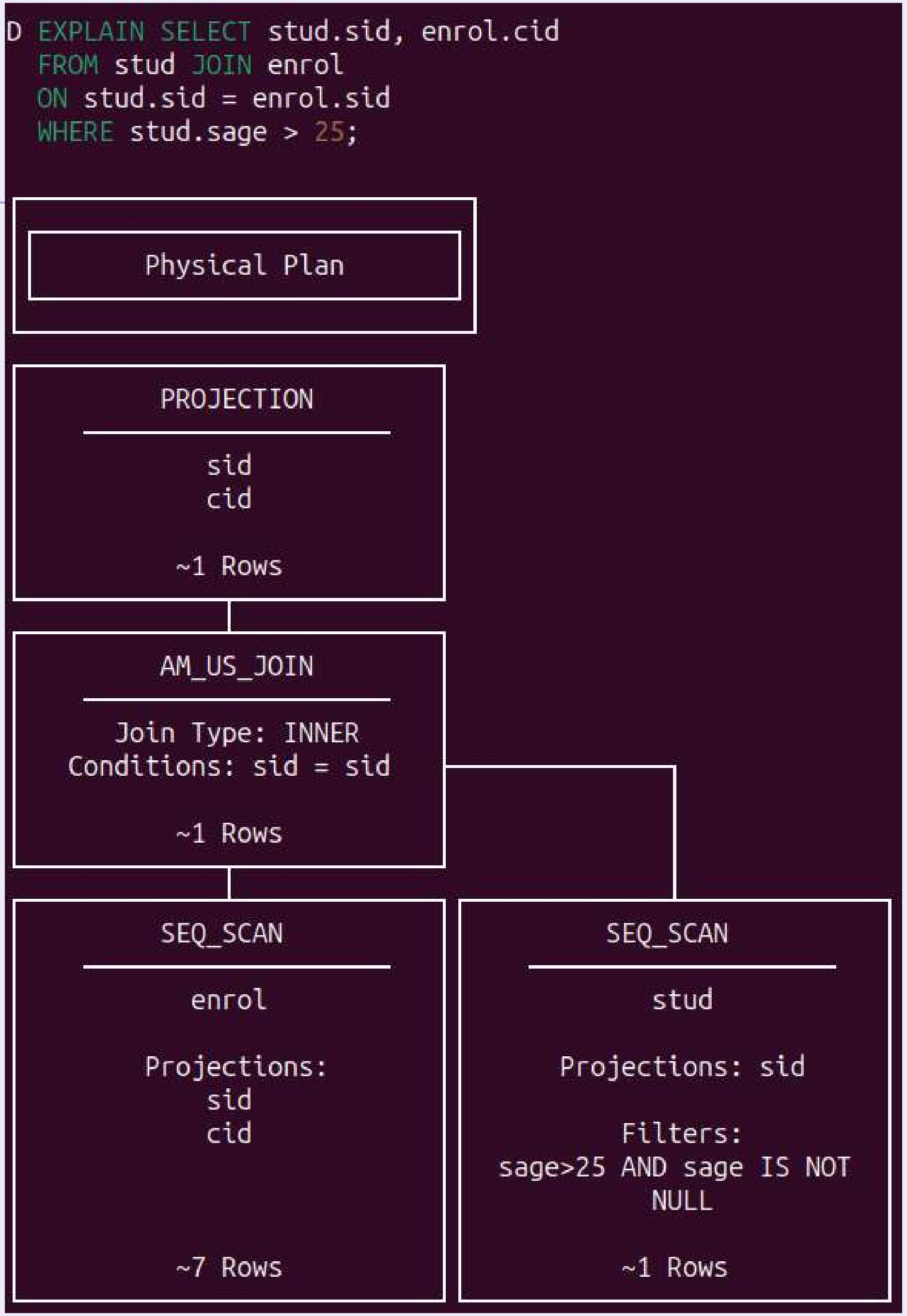
- ***SQLStatement***: query statement in DuckDB is represented as this
- ***Binder***: responsible for binding tables and columns to actual physical tables and columns in the catalog.
- ***Planner***: has a *CreatePlan*() method that creates the logical plan tree from the AST.
- The choice of actual algorithm is left to the optimizer based on **estimated cardinality.**
- A ***Value*** is a unit that holds a single value of arbitrary type.
- A **Vector** is the smallest unit of data handling holding values in a column.
- A **DataChunk** is a set of Vectors serving as the unit of data processing in the pipeline through operators.
- A **Vector** can hold up to a fixed number of values, defined by STANDARD_VECTOR_SIZE (power of 2)

Table	
101	25
102	23
103	31
104	28
105	30
106	27

Chunk1	
101	25
102	23
103	31
104	28

Chunk2	
105	30
106	27

DuckDB Execution Model

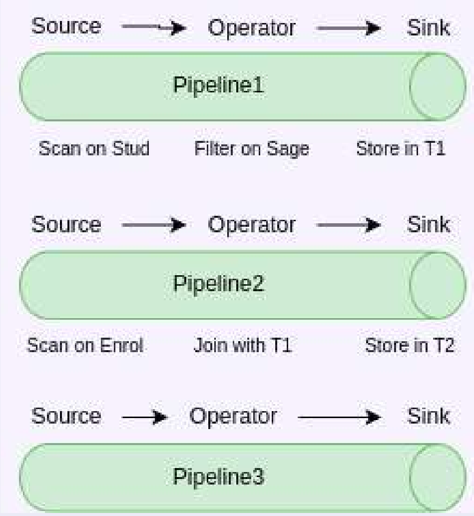


Push-Based, Pipelined

- Operators **process** the data (e.g. Join) and **push** the data to the next operator (e.g. filter) in the pipeline.
- A long execution sequence is broken into multiple pipeline events and executed possibly in *parallel*.

Each pipeline has *three* interfaces: **Source**, **Operator** and **Sink**.

SELECT stud.sid, enrol.cid FROM stud JOIN enrol
ON stud.sid = enrol.sid WHERE stud.sage > 25;



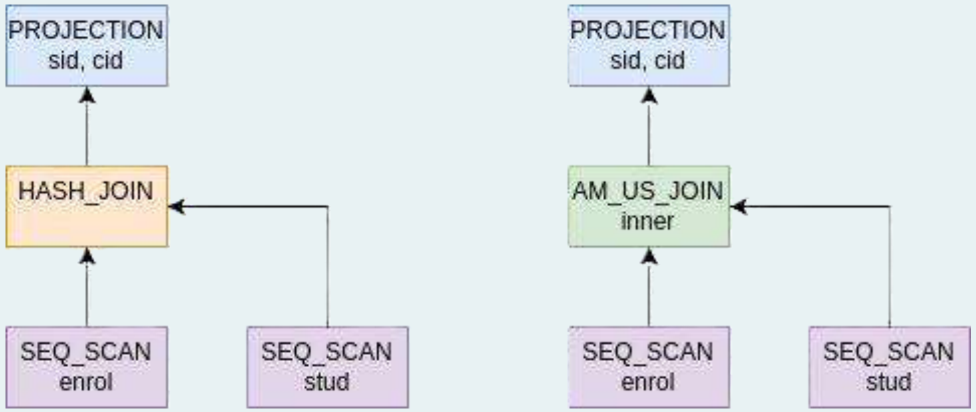
DuckDB New Join Operator (AM_US_JOIN)

- **Operator Definition**: The "**AM_US_JOIN"** physical operator is associated with the **"LOGICAL_COMPARISON_JOIN**" logical operator and functions as a nested loop join.
- **Operator Selection:** DuckDB's query planner selects join operators based on the estimated cardinality of tuples in participating relations.
- **Threshold Configuration:** The client configuration is tweaked to set a cardinality threshold (e.g., 100 tuples), which is tunable as needed.
- **Physical Plan Generator:** A cardinality check is added to the physical plan generator to ensure input relations are within the threshold.
- **Automatic Selection**: The optimizer automatically chooses "AM_US_JOIN" for small relations when detected as child nodes in the physical plan.

Working (Contd)

```
if(can_do_physical_amus_join(client_config, left, right)){
    plan  = make_uniq<PhysicalAmUsJoin>(op, std::move(left), std::move(right), std::move(op.conditions),
                                        op.join_type, op.estimated_cardinality);
    std::cout << "AM_US_JOIN Physical Plan Taken\n";
    return plan;
}
```

```
bool can_do_physical_amus_join(ClientConfig& client_config,
                               LogicalComparisonJoin& op,
                               unique_ptr<PhysicalOperator>& left,
                               unique_ptr<PhysicalOperator>& right){
    bool can_use = false;
    if(PhysicalAmUsJoin::IsSupported(op.conditions, op.join_type)){
        if (left->estimated_cardinality <= client_config.am_us_join_threshold &&
        right->estimated_cardinality <= client_config.am_us_join_threshold) {
            can_use = true;
        }
    }
    return can_use;
}
```



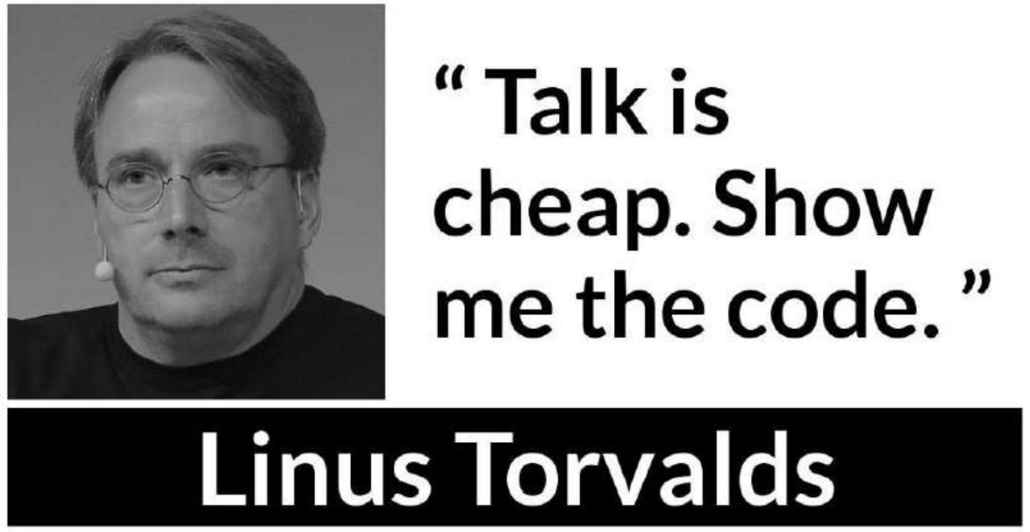

```
private:
    OperatorResultType ResolveComplexJoin(ExecutionContext &context, DataChunk &input, DataChunk &chunk,
                                          OperatorState &state) const;
```

Working

- ***PhysicalAmUsJoin*** class extending the ***PhysicalComparisonJoin***
- Actual join is performed by the *ResolveComplexJoin*() method by calling *Execute*() internally.
- The join starts with getting the left *datachunk*, and operating it will all *datachunks* on the right.
- Then we go to next *datachunk* in left and continue the process until all *datachunks* are exhausted in left.
- For each pair of left and right *datachunk*, there are tuple markers 'left' and 'right' which progressively move during operation, and the rows are marked first which matches the Join condition.
- The matching positions are kept in a *match_vector*.
- The actual join happens between the left and right *datachunks* using the *Perform*() method.
- If matches are **found**, the matched tuples are **sliced** from the input and stored in the output *datachunk*.

Working

- ***PhysicalAmUsJoin*** class extending the ***PhysicalComparisonJoin***
- Actual join is performed by the *ResolveComplexJoin*() method by calling *Execute*() internally.
- The join starts with



tachunks on the right.
- Then we go to next *tachunks* are exhausted in left.
- For each pair of left d 'right' which progressively move during operation, and the r n.
- The matching positi
- The actual join happens between the left and right *datachunks* using the *Perform*() method.
- If matches are **found**, the matched tuples are **sliced** from the input and stored in the output *datachunk*.

ResolveComplexJoin Algorithm

Algorithm

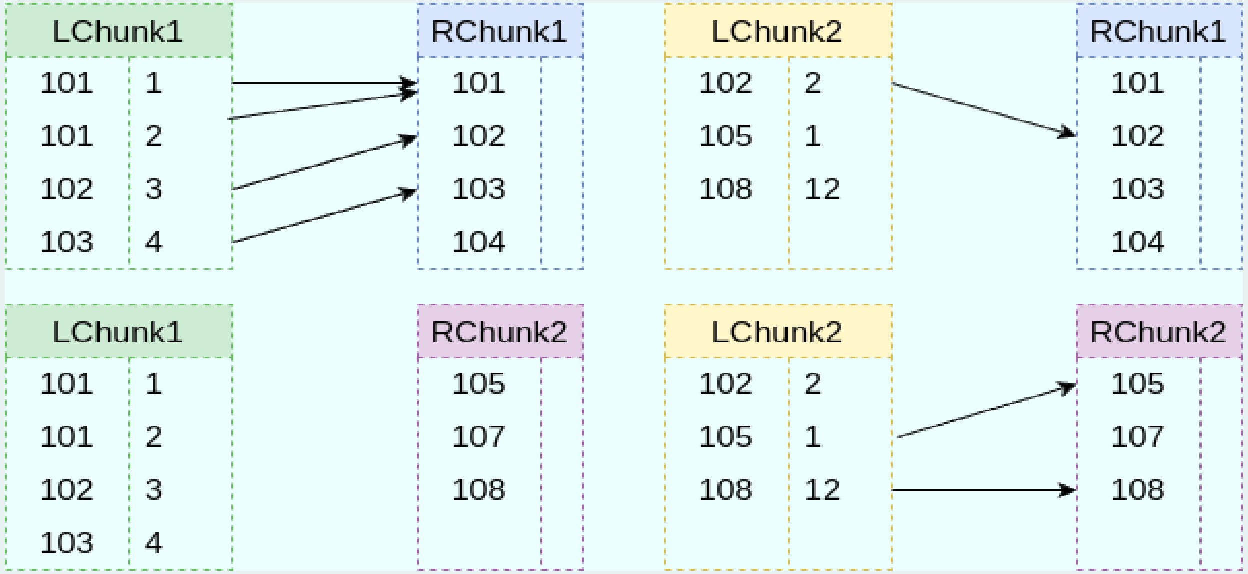
```
Algorithm ResolveComplexJoin(context,input,chunk,state){

  // Cast state and global state to specific types
  state = cast state to PhysicalAmUsJoinState
  gstate = cast sink_state to AmUsJoinGlobalState

  // Initialize match_count
  match_count = 0

  // Loop until a match is found
  do{
    if state.fetch_next_right is true{

     / If right chunk is exhausted,
     // move to the next chunk on the right
     state.left_tuple = 0
     state.right_tuple = 0
     state.fetch_next_right = false

     // Check if there are more right conditions
     if some right conditions exist:
     // If conditions exist,
     // scan the right payload data
     gstate.right_payload_data.Scan(state.right_payload)

     else:
     // If all right conditions are exhausted,
     // move to the next left chunk
     state.fetch_next_left = true

     // Handle left join: output unmatched rows
     state.left_outer.ConstructLeftJoinResult(input, chunk)

     return NEED_MORE_INPUT
  }
  if state.fetch_next_left is true{

     // Resolve the left condition for the current chunk
     state.lhs_executor.Execute(input,state.left_condition)

     // Reset tuples and scan conditions on the right side
     state.left_tuple = 0
     state.right_tuple = 0

     gstate.right_payload_data.Scan(state.right_payload)
     state.fetch_next_left = false
     // Now perform the actual join
     // between the left and right chunks
     left_chunk = input
     right_payload = state.right_payload

     // Verify chunks
     left_chunk.Verify()
     right_payload.Verify()

     // Perform the actual join
     // using the left and right chunks
     lvector = SelectionVector(STANDARD_VECTOR_SIZE)
     rvector = SelectionVector(STANDARD_VECTOR_SIZE)

     match_count = Perform(state.left_tuple,
                           state.right_tuple,
                           state.left_condition,
                           right_condition,
                           lvector, rvector)

     // If matches are found
     if match_count > 0{

     // Set matches for both left and right sides
     state.left_outer.SetMatches(lvector,
                                 match_count)
     gstate.right_outer.SetMatches(rvector,
                                   match_count)

     // Slice the input chunks and store
     chunk.Slice(input, lvector, match_count)
     chunk.Slice(right_payload, rvector, match_count)
     }

     // If no matches are found in this iteration,
     // continue to the next iteration
     if state.right_tuple >= right_condition.size():
       state.fetch_next_right = true

     // Continue until a match is found
     } while (match_count == 0)

     // Return that more output is available

     return HAVE_MORE_OUTPUT
   }
```

Demonstration

```
SELECT stud.sid, enrol.cid
FROM stud JOIN enrol
ON stud.sid = enrol.sid;
```

Table Stud		
101	A	25
102	B	26
103	A	27
104	B	23
105	A	30
107	D	30
106	C	25
sid	sname	sage

Table Enrol	
101	1
101	2
102	3
103	3
102	2
105	1
108	12
sid	cid

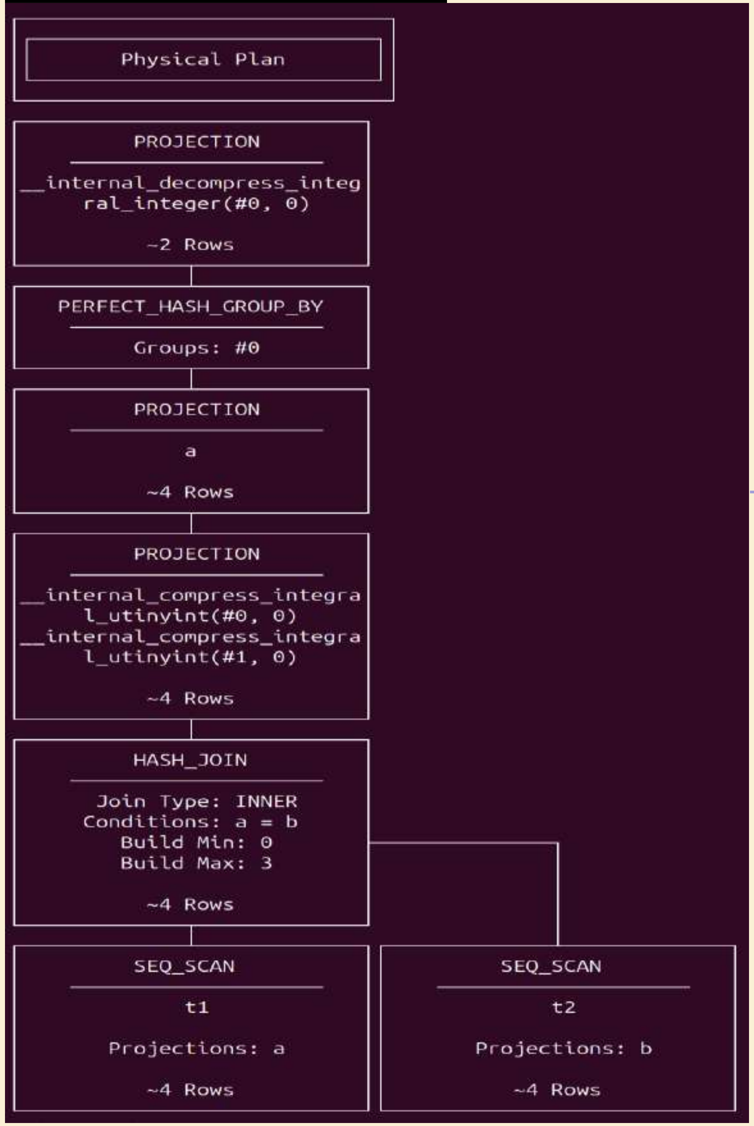


The choice of left and right tables is done by DuckDB based on approximate cardinality estimation.

DuckDB New GroupJoin Operator (GROUPJOIN_GROUPBY)

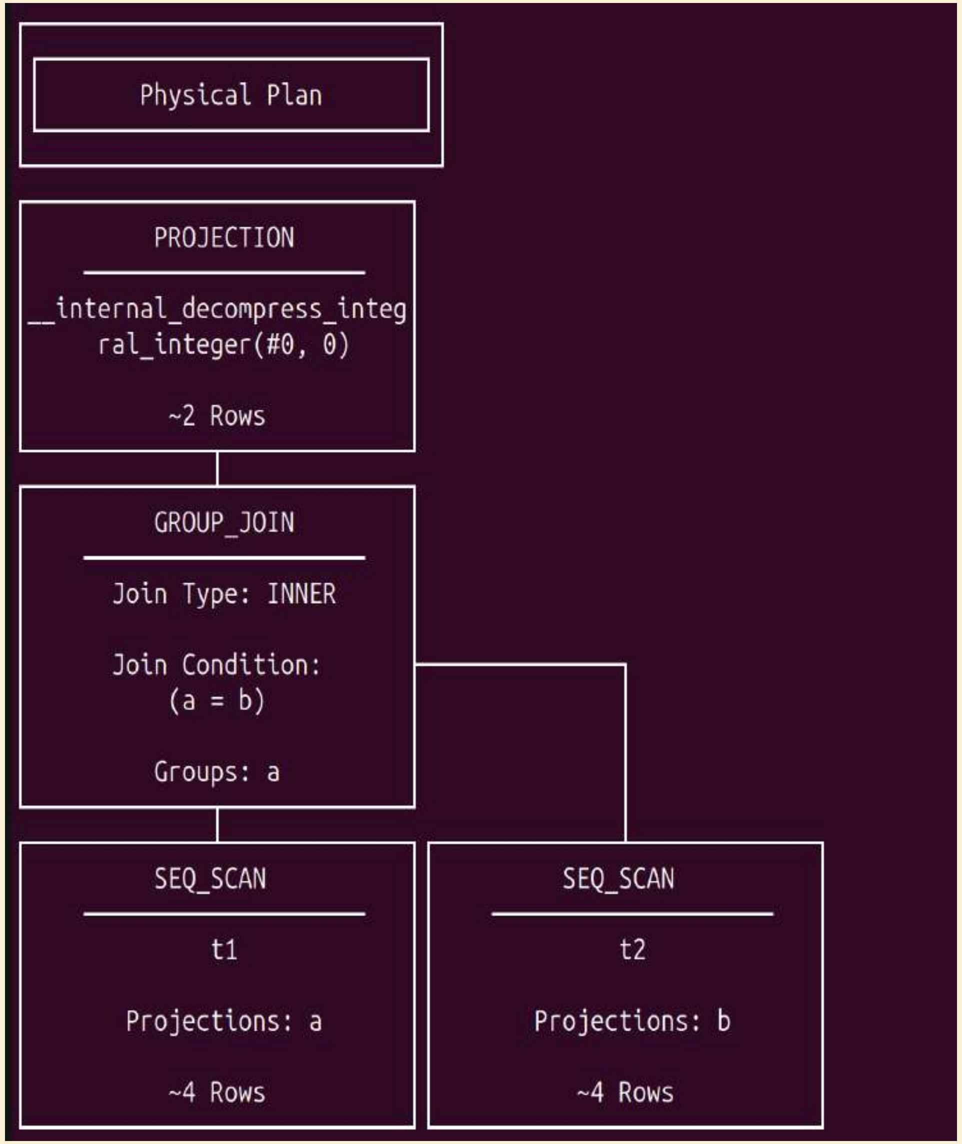
- When we find a ‘Join’ operator as a child under the ‘Group By’ node in the logical plan tree of the query.
- The Join operator can be any logical operator or physical operator, not necessarily AMUS_JOIN.
- The logical plan generated is based on the logical operators, which has to be replaced with actual physical operators.
- To implement this, whenever we see a **LOGICAL_GROUP_BY** operator, we check to see if there is a **LOGICAL_JOIN** operator somewhere in the children node.
- If such a child node exists, we replace the parent with our **GROUPJOIN_GROUP_BY** physical operator and return the generated plan.
- For this, we implemented the interface for our 'GroupJoin' physical operator (i.e. GROUPJOIN_GROUP_BY).
- For aggregation, it uses hashing similar to what is being done in the currently existing 'Group By' aggregation logic.

New GroupJoin Operator (GROUPJOIN) -- WIP



After replacing the
Group – Project - Join
With
GroupJoin Operator

(in Physical Plan)



DuckDB New GroupJoin Operator (GROUPJOIN) -- WIP

We want to combine the Join-Project-GroupBy logic into one single operator (GROUP_JOIN)

GROUPJOIN operator will *directly* read in data from tables, do the necessary joining, and subsequent projection, so that group-by can be done

Our **GROUPJOIN_GROUPBY** can be used for aggregation.

We want to extend the logic of *GetData()* method in **PhysicalGroupJoin** operator

```
SourceResultType PhysicalGroupJoin::GetData(ExecutionContext &context, DataChunk &chunk,
                                            OperatorSourceInput &input) const {

    // Set the source and sink interfaces
    // Read in Data from Tables
    // Do the in-memory join
    // Set the data to sink
    return SourceResultType::FINISHED;
}
```

DuckDB New GroupJoin Operator (GROUPJOIN) -- WIP

```
bool canReplaceByGroupJoin(LogicalOperator &op){

   // If not a groupby, nothing to replace
   if(op.type != LogicalOperatorType::LOGICAL_AGGREGATE_AND_GROUP_BY) return false;
   auto &groupby = op.Cast<LogicalAggregate>();

   // Check if there are groups and has a join as a child
   if(groupby.groups.size() > 0 && groupby.children[0]->children[0]->type == LogicalOperatorType::LOGICAL_COMPARISON_JOIN){
      return true;
   }

   return false;
}
```

- Basic conditions to replace the Group-Project-Join with **PhysicalGroupJoin** Operator
- Additional checks based on the join conditions and group by attributes added in **PhysicalGroupJoin::IsSupported()**

DuckDB GroupJoin--WIP

Pseudocode for GetData() Function

```
SourceResultType PhysicalGroupJoin::
                 GetData(ExecutionContext &context,
                 DataChunk &chunk,
                 OperatorSourceInput &input) {

    // Set the source and sink interfaces
    // Read in Data from Tables
    // Do the in-memory join
    // Set the data to sink
    return SourceResultType::FINISHED;
}
```

Pseudocode for checker Function

```
bool canReplaceByGroupJoin(LogicalOperator &op){
        // If not a groupby, nothing to replace
        if(op.type != LogicalOperatorType::
            LOGICAL_AGGREGATE_AND_GROUP_BY) return false;
        auto &groupby = op.Cast<LogicalAggregate>();
        // Check if there are groups and has a join as a
            child

        if(groupby.groups.size() > 0 && groupby.children
            [0]->children[0]->type == LogicalOperatorType::
            LOGICAL_COMPARISON_JOIN){
        return true;
    }
    return false;
 }
```

Pseudocode for Plan Generator Function

```
unique_ptr<PhysicalOperator> PhysicalPlanGenerator::
    PlanGroupJoin(LogicalAggregate &op) {
    // Visit the children
    auto &join = op.children[0]->children[0]->Cast<
        LogicalComparisonJoin>();
    idx_t lhs_cardinality = join.children[0]->
        EstimateCardinality(context);
    idx_t rhs_cardinality = join.children[1]->
        EstimateCardinality(context);
    auto left = CreatePlan(*join.children[0]);
    auto right = CreatePlan(*join.children[1]);
    left->estimated_cardinality = lhs_cardinality;
    right->estimated_cardinality = rhs_cardinality;
    D_ASSERT(left && right);

    if (join.conditions.empty()) {
        // No conditions: insert a cross product
        return make_uniq<PhysicalCrossProduct>(op.types, std
            ::move(left), std::move(right), op.
            estimated_cardinality);
    }
    unique_ptr<PhysicalOperator> plan;

    for (auto &cond : join.conditions) {
        RewriteJoinCondition(*cond.right, left->types.size()
            );
    }
    auto condition = JoinCondition::CreateExpression(std::
        move(join.conditions));
    std::cout << "Group Join Everytime" << std::endl;
    // Pass the grouping and aggregate expressions
    plan = make_uniq<PhysicalGroupJoin>(op, std::move(left),
            std::move(right),  std::move(condition),
            join.join_type, op.estimated_cardinality,
            op.groups, op.expressions);
    return plan;
}
} // namespace duckdb
```

DuckDB Tests and Experiments

To verify the implementations of **AM_US_JOIN** and **GROUPJOIN_GROUPBY aggregation only** for their correctness,

1

We created and used a small database consisting of two relations each of cardinality 8, to keep demonstration simple, as shown:

1. **Stud (Sid: INT, Sname: VARCHAR, Sage: INT)**
2. **Enrol (Sid:INT, Cid: INT)**

Table Stud

sid	sname	sage
101	A	25
102	B	26
103	A	27
104	B	23
105	A	30
107	D	30
106	C	25

Table Enrol

sid	cid
101	1
101	2
102	3
103	3
102	2
105	1
108	12

2

We also created a database with 3 big tables each with cardinality 75, to test out on larger dataset:

- **users (user_id, first_name, last_name, address, email);**
- **products (product_id, product_name, description, price);**
- **orders (order_id,user_id,product_ordered,total_paid);**

+

We evaluated the results of these 4 queries producing ~5180, ~383320, ~70, ~5625 rows respectively.
We found results consistent with the expected results
The two databases reside in *small.db* and *big.db* respectively in *$ROOT/myduckdb/sql_files*

Summary

- reviewed the ecosystem and interfaces provided by open-source DuckDB
- explored the implementation of a simple nested loop join, calling it AM_US_JOIN
- tweaked the query planner and optimizer to pick our version of the Join
- explored the implementation of aggregation using GROUPJOIN_GROUPBY (simple version)

Future Work

- explored how we can efficiently combine both 'Join' and 'Group By' into a single 'GroupJoin' operator

DuckDB

Thank You

Akash Maji

akashmaji@iisc.ac.in

Utkarsh Sharma

utkarsh2024@iisc.ac.in

DuckDB

Link to stable version:
https://github.com/akashmaji946/myduckdb/tree/main