E0-261 Database Management Systems

Project P8: DuckDB

Akash Maji
SR No.: 24212
akashmaji@iisc.ac.in

Utkarsh Sharma
SR No.: 24116
utkarsh2024@iisc.ac.in

05 December 2024

Abstract

DuckDB is a modern, open-source analytical (OLAP) database system. It is designed to be fast, reliable, portable, easy to use, and it provides a rich SQL dialect. DuckDB supports arbitrary and nested correlated subqueries, window functions, collations, complex types (arrays, structs, maps), and several extensions designed to make SQL easier to use.

Introduction

In this project, we aim to explore the interface provided by DuckDB and implement certain operators. Firstly, we will discuss the implementation of a new 'Join' operator. Secondly, we change the query planner and optimizer to pick our join operator whenever a predetermined condition is met. Thirdly, we implement a new 'GroupJoin' operator, which will be invoked whenever a query containing a 'Join' followed by a 'Group By' is handed to the engine.

Database Architecture

DuckDB is a columnar database, using its custom *DuckDB Storage Format*, which is a compressed, columnar, block-based format designed for efficient storage and query execution. The entire database, including data, metadata, and indexes, is stored in a single, self-contained file for ease of use and portability. The database can sit in a file with a *.db* or *.duckdb* extension.

A query statement in DuckDB is represented via an *SQLStatement* class instance. The *Binder* class is responsible for binding tables and columns to actual physical tables and columns in the catalog. The *Planner* class has a *CreatePlan*() method that creates the logical plan tree from the AST (Abstract Syntax Tree) obtained using *Binder class*.

When the query is submitted to the engine, it obtains a logical plan tree. This tree contains logical operators as nodes. For example, a LOGICAL_COMPARISON_JOIN is a logical operator that can be replaced by an appropriate physical operator like PHYSICAL_HASH_JOIN, PHYSICAL_SORT_MERGE_JOIN, PHYSICAL_NL_JOIN etc. The choice of actual algorithm is left to the optimizer based on estimated cardinality in DuckDB.

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*. This parameter is configurable and allows it to cater for different compute-environments, where it may not be possible to keep vector size large. We have set this as 4 in our experiments. It is to be noted that DuckDB only allows vector size to be powers of two. The choice of four is to keep the demonstration simple.

We show below how a sample table can be broken down into two chunks each with two vectors. It is to be noted that the chunks materialized only when needed and are stored in memory on the fly after scanning from associated tables and columns.

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

Figure1: A sample table and its possible *datachunks*

Execution Model

The execution model used in DuckDB is currently push-based, which is a revamp from the earlier pull-based model. This means, the operators in DuckDB process the data (e.g. Join) and push the data to the next operator (e.g. filter) in the pipeline. This allows for massive parallelism, reduced latency, and simplified dataflow. The execution happens in a pipelined mode, meaning a long execution sequence is broken down into multiple pipeline events, and executed possibly in parallel. Each pipeline event has one or more operators in their dependency order, and the operations in the pipeline are done in that order. Each pipeline has three interfaces: Source, Operator and Sink. Source produces data for the Operator, which processes the data, and Sink stores the processed results for the next pipeline.

For example, a typical query Q1 can get us the following pipelines as shown in Figure2.

```
Q1:  SELECT Stud.sid, Enrol.cid
     FROM Stud JOIN Enrol ON Stud.sid = Enrol.sid;
     WHERE Stud.sage > 25;
```

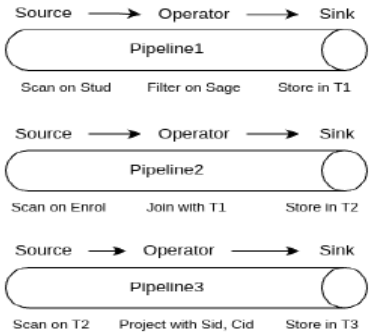


Figure2: A sample pipeline demonstration

Communication between operators, sources, and sinks is central to data flow in DuckDB's query execution pipeline. For this, certain signals are used by the three interfaces as stated under:

<u>OperatorResultType</u>: Used by regular (non-source/sink) operators to indicate how data should flow:

- NEED_MORE_INPUT: Operator is ready to process more input.
- HAVE_MORE_OUTPUT: Operator still has more output to produce from the current input.
- FINISHED: Operator has completed its work; no further calls will occur.
- BLOCKED: Operator is paused (e.g., waiting for async I/O).

<u>OperatorFinalizeResultType</u>: Used to indicate whether an operator has finished flushing cached results:

- HAVE_MORE_OUTPUT: Operator still has more cached data to flush.
- FINISHED: Operator has completed flushing.

<u>SourceResultType</u>: Indicates the result of pulling data from a source:

- HAVE_MORE_OUTPUT: Source has more output; data is returned.
- FINISHED: Source is exhausted.
- BLOCKED: Source is paused (e.g., waiting for async I/O).

<u>SinkResultType</u>: Indicates the result of pushing data into a sink:

- NEED_MORE_INPUT: Sink requires more input to process.
- FINISHED: Sink is complete; additional input won't change the result.
- BLOCKED: Sink is paused (e.g., waiting for async I/O).

<u>SinkCombineResultType</u>: Indicates the result of combining sink data:

- FINISHED: Combination is complete.
- BLOCKED: Combination is paused (e.g., waiting for async I/O).

<u>SinkFinalizeType</u>: Indicates the result of a Finalize call on a sink:

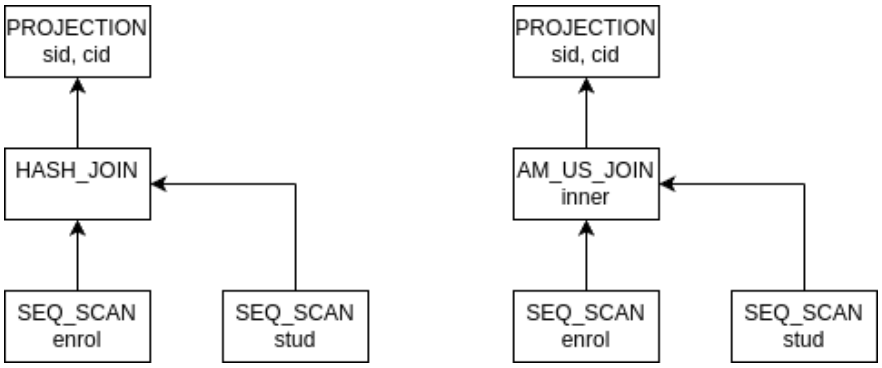
- READY: Sink is ready for further processing.
- NO_OUTPUT_POSSIBLE: Sink won't provide output, and related pipelines can be skipped.
- BLOCKED: Finalization is paused (e.g., waiting for async I/O).

Implementing New Join Operator

We first begin exploring the interface provided by DuckDB to implement a new JOIN operator. We call this physical operator "AM_US_JOIN". The associated logical operator is "LOGICAL_COMPARISON_JOIN." This join operator is similar to a plain nested loop join operator. The query planner and optimizer in DuckDB picks the appropriate join operator based on estimated cardinality of the tuples involved in the participating relations. We therefore tweak the 'client configuration' file to set the threshold of utmost 100 tuples of relations, and we add the predetermined condition to check the cardinality of input relations to be within the threshold limit, in the physical plan generator. Note that the choice of 100 tuples is arbitrary and can be tuned as per need. The optimizer/physical plan generator then automatically chooses our join operator, when it sees two small relations as child nodes.

For example, we run the query Q2 and get the following physical plan, as shown in Figure3.

Q2:
```
EXPLAIN SELECT Stud.sid, Enrol.cid
FROM Stud JOIN Enrol
ON Stud.sid = Enrol.sid;
```



(3a) Before (3b) After

Figure3: Change in generated physical plan

Implementation Details and Working

We begin by implementing the *PhysicalAmUsJoin* class extending the *PhysicalComparisonJoin* class. It contains Source, Sink and Operator interfaces as mentioned in the execution model. The Source method is *GetData()* which scans a *datachunk* which is used in the join operation. Actual join is performed by the *ResolveComplexJoin*() method by calling *Execute*() internally. We can do all types of Joins (INNER, LEFT, RIGHT, OUTER) using *ResolveComplexJoin().* We have a *Combine()* method that combines results from different threads and puts them into Sink. At the end, *Finalize()* method is called when the Operator is done producing results, and Sink is not expecting any more results being given.

The join starts with getting the left *datachunk*, and operating it will all *datachunks* on the right. Then we go to the next *datachunk* in left and continue the process until all *datachunks* are exhausted on the 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*. The *lvector* and *rvector* hold the indices of matching rows for the left and right sides, respectively. At the end, the matching rows are sliced into the output *datachunk*. Then we move to the next pair of *datachunks*, and repeat till no more left *datachunk* is left.

Consider two tables ‘Stud’ and ‘Enrol’. When we run the following query Q3, we will get four *chunks* and the operation will be done as shown in Figure4.

Q3: **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

Stud(sid, sname, sage)

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

Enrol(sid, cid)

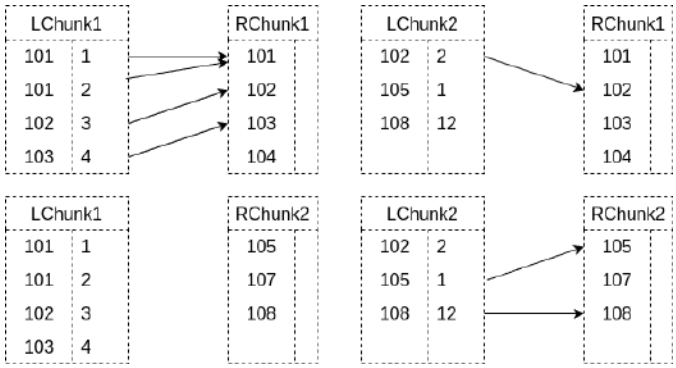


Figure4: Working of AM_US_JOIN

ResolveComplexJoin 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
  }
```

Implementing New GroupJoin Operator

The GroupJoin operator is introduced in the physical query plan 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 is to be replaced with physical operators. To implement this, whenever we see a LOGICAL_AGGREGATE_AND_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.

For example, when we run the query Q4, we see the physical plan as shown in Figure5.

Q4:
```
EXPLAIN SELECT Enrol.cid, COUNT(Stud.sid) AS student_count
FROM Stud JOIN Enrol
ON Stud.sid = Enrol.sid
GROUP BY Enrol.cid;
```

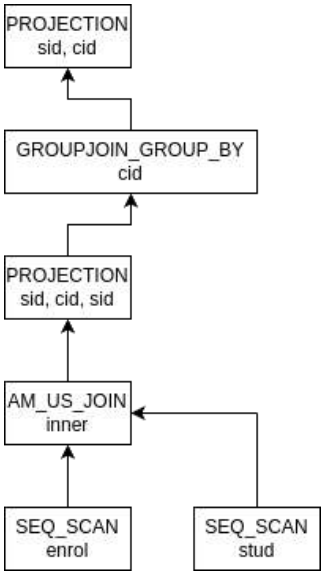


Figure5: Simple GroupJoin shown in physical plan

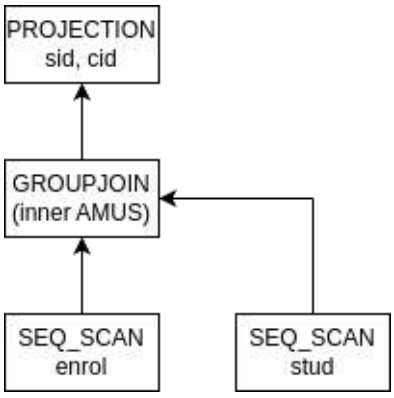


Figure6: Unified GroupJoin

It is to be noted that in this simple variant, GROUPJOIN operator only works on the data output by the already processed JOIN operator in the child. That means, the GROUPJOIN operator is not standalone and needs data relevant for grouping. The standalone GROUPJOIN operator, which will directly operate upon scanned columns to do both in-memory join and grouping will be discussed later. The goal of this simple variant is to be able to introduce our own GROUPJOIN physical operator capable of aggregation, which will be extended later for doing JOIN also, and thus the physical plan tree will be changed as shown in Figure6.

Implementation Details and Working

We begin implementing the *PhysicalGroupJoinAggregate*() class by extending the *PhysicalOperator*() class. It contains Source, Sink and Operator interfaces as a physical operator. The Source interface has a *GetData*() method that produces data from the global sink state. *CreateHT*() method creates a hash table at the time of object instantiation. The *AddChunk*() method adds *datachunk* into the hash table within the *Sink*() interface, thereby performing group-by and aggregation. It can be thought of as the build phase. *Scan*() is called repeatedly to extract all results from the hash table, one batch at a time, into output *datachunks*. It can be thought of as a probe phase. The *Scan*() method is the bridge between the internal representation of aggregated data (stored in the hash table) and the external representation (output *datachunk*).

Tests and Experiments

To verify the implementations of 'AM_US_JOIN' and 'GROUPJOIN_GROUP_BY' for their correctness **only**, 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)

We demonstrated that our 'Join' operation is working as per 3 simple queries [Q1, Q2, Q3]. and we obtained results like what we got before introducing our Join operator. Also, the simple GroupJoin was run using query Q4 that does aggregation using one group, i.e. *cid*. We found results exactly similar to existing aggregation logic for the 'simple GroupJoin' implementation which only does aggregation, on projected and processed data out of Join operation.

We also created a database with 3 big tables:

- 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 [QA, QB, QC, QD] producing ~5180, ~383320, ~70, ~5625 rows respectively. The results were consistent with those of original JOINs and GROUP-BYs.

We ran sqllogic tests also to verify the creation and insertion of tuples into the two tables as per DuckDB documentation. The two databases, respectively, reside in two files (small.db and big.db) available at *$ROOT/myduckdb/sql_files/* (where $ROOT is directory of installation).

QA:
```
SELECT users.user_id,  orders.order_id
FROM  users
JOIN orders ON users.user_id != orders.user_id;
```

QB:
```
SELECT u.first_name, u.last_name, p.product_name, o.total_paid
FROM orders o
JOIN users u ON o.user_id != u.user_id
JOIN products p ON o.product_ordered != p.product_id;
```

QC:
```
SELECT o.user_id, o.product_ordered, SUM(o.total_paid) AS total_spent
FROM orders o
GROUP BY o.user_id, o.product_ordered;
```

QD:
```
SELECT u.first_name, u.last_name, p.product_name, SUM(o.total_paid) AS total_spent
FROM orders o
JOIN users u ON o.user_id != u.user_id
JOIN products p ON o.product_ordered != p.product_id
GROUP BY u.user_id, p.product_id, u.first_name,  u.last_name, p.product_name;
```

Q1:
```
SELECT Stud.sid, Enrol.cid
FROM Stud
JOIN Enrol ON Stud.sid = Enrol.sid;
WHERE Stud.sage > 25;
```

Q2:
```
EXPLAIN SELECT Stud.sid, Enrol.cid
FROM Stud
JOIN Enrol ON Stud.sid = Enrol.sid;
```

Q3:
```
SELECT Stud.sid, Enrol.cid
FROM Stud
JOIN Enrol ON Stud.sid = Enrol.sid;
```

Q4:
```
EXPLAIN SELECT Enrol.cid, COUNT(Stud.sid) AS student_count
FROM Stud
JOIN Enrol  ON Stud.sid = Enrol.sid
GROUP BY Enrol.cid;
```

Future Work

We want to combine the **Join-Project-Group By** logic into one single unified operator (GROUP_JOIN). The **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 the *GetData()* method in the **PhysicalGroupJoin** operator. *GetData*() is the Source interface that should do the reading of tables, doing in-memory processing (join) and doing projection, so that GROUPJOIN_GROUPBY (as discussed earlier) can do aggregation.

Additionally, we will need to check whether we can replace GroupBy-Project-Join and if we can use GroupJoin, provided that the children have a Join, and then replace the plan with our **PhysicalGroupJoin** plan.

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;
}
```

The plan generator will generate plans for the two children, attach them as children and return the final GroupJoin plan.

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
```

Summary

In this project, we reviewed the ecosystem and interfaces provided by open-source DuckDB. We first explored the implementation of a simple nested loop join, and saw a basic tuple oriented nested loop join, calling it AM_US_JOIN, using the existing interfaces and classes, and extending the functionality. Next, we tweaked the query planner and optimizer to pick our version of the Join over others when threshold is met. We then explored how we can efficiently combine both 'Join' and 'Group By' Operations into a single GroupJoin operator, that will process the Join first in memory after reading directly from table sources, and use the bulk result to aggregate over it, giving results from one physical GroupJoin operator. This will lead to efficiency over the currently existing pathway, as we will not have to pass the data through many physical operators.

References and Links

GitHub Repository:

https://github.com/akashmaji946/myduckdb/tree/main

GroupJoin VLDB Paper:

https://vldb.org/pvldb/vol14/p2383-fent.pdf

DuckDB Official Documentation

https://duckdb.org/docs/