Implementing the GroupJoin Operator in DuckDB

Akash Maji

M.Tech CSE, Department of Computer Science and Automation
Database Systems Lab
Indian Institute of Science, Bangalore, India
akashmaji@iisc.ac.in

Abstract—**DuckDB is a high-performance, open-source, free analytical database system designed for efficient in-process data analytics. Its architecture emphasizes speed, reliability, and ease of integration. GroupJoin is a new database operator that fuses two operators into one to improve performance.**

First, we performed a simple experiment to demonstrate the baseline comparison of GroupJoin against the currently used HashJoin-Aggregate logic, to analyze the conceptual efficacy of GroupJoin.

Next, we implemented and evaluated a new physical database operator, GroupJoin, aimed at optimizing query execution patterns where a JOIN operation is immediately followed by a GROUP BY. Instead of executing the join and aggregation as separate steps, the GroupJoin operator combines both phases into a single, tightly coupled operator. This optimization can reduce intermediate data materialization and improve cache locality, leading to potential performance gains. We integrated the GroupJoin operator into the DuckDB query execution engine, modified the planner to identify candidate patterns, and evaluated its impact. GroupJoin offers both conceptual simplicity and performance benefits in scenarios common to analytical query processing. Our initial implementation, although inefficient compared to DuckDB's native operators, demonstrates the potential for performance benefits.

Index Terms—**DuckDB, Query Optimizer, Logical Plan, Physical Plan, Physical Operator, GroupJoin, Baseline Comparison, TPC-H, Restricted Implementation**

I. INTRODUCTION

Modern analytical query engines are increasingly optimized to exploit common query patterns and to reduce unnecessary computation and data movement. One such ubiquitous pattern in analytical workloads is a JOIN operation followed immediately by a GROUP BY clause. This combination often appears in star-schema queries and reporting-style queries, where data from multiple tables is joined and then aggregated (e.g., summarizing sales per region, per product, etc.). The traditional query execution engines handle these as two separate operators — a JOIN followed by a GROUP BY — which may involve materializing large intermediate results and performing redundant computations.

To address this inefficiency, we explored the design and implementation of a new physical operator, **GroupJoin** [6], in DuckDB [1] — a lightweight, high-performance, in-process analytical database system. The *GroupJoin* operator fuses the join and aggregation phases into a single, unified operator. This integration allows the system to bypass the creation of full intermediate join results and directly aggregate on-the-fly during the join process, reducing memory overhead and improving performance.

A. Formal Definition of GroupJoin

Although **GroupJoin** is defined in [6] (Sec. 2.5), we redefine it here for convenience:

$$e_1 \bowtie_{A_1 \theta A_2 : g; f} e_2 := \{y \circ g : G \ || \ y \in e_1, \dots \}_b$$

This definition describes an operator that joins two relations, e_1 and e_2, by grouping the matching rows from e_2 and applying an aggregation function to them. The result contains all tuples from e_1 extended with a new attribute g holding the aggregated result.

TABLE I
COMPONENT BREAKDOWN OF THE GROUPJOIN OPERATOR

Symbol	Description
e_1, e_2	The two input relations (tables).
$\bowtie_{\dots}$	The groupjoin operator symbol.
$A_1 \theta A_2$	The **join predicate**, where A_1, A_2 are attributes and θ is a comparison operator ($=$, $! =$, etc.).
$g; f$	Defines the **aggregation step**, where f is an aggregation function (e.g., `COUNT`) and g is the new attribute name.
$y \in e_1$	Iterates through each tuple y in the left relation e_1.
$\{x \mid \dots \}_b$	The core grouping mechanism; collects all matching tuples x from e_2 into a **bag** (multiset).
$G = f(\dots)_b$	Applies the aggregation function f to the entire bag, yielding a single value G.
$y \circ g : G$	The output tuple, formed by extending tuple y with the new attribute g holding the value G.

II. RELATED WORK

The concept of pushing aggregation through joins has been explored in prior literature, notably in works such as the DBToaster project [2], which uses incremental view maintenance to optimize such patterns, and in fused operators proposed for query compilation engines like HyPer [3], where join-aggregate fusion improves cache locality and reduces CPU cycles. Similarly, techniques like late materialization and operator fusion in systems such as MonetDB [4] and Vectorwise [5] have shown that tightly integrated operator pipelines are effective for analytical performance.

GroupJoin as a concept has been studied more directly by Moerkotte and Neumann [6], who formalized the idea

of combining *joins* and *group-by aggregations* into a single algebraic operator. They showed that such a transformation can reduce the complexity of query plans and lead to substantial performance benefits in analytical settings. Extending this work, Fent and Neumann [7] presented a practical implementation of GroupJoin and nested aggregates within a compiled query engine, demonstrating its viability and benefits in real-world workloads.

Building upon these ideas, our work implements a specialized **GroupJoin** operator directly within DuckDB's vectorized execution engine. We aim to evaluate the correctness and performance implications of this transformation on representative analytical queries. We begin by performing a baseline comparison to support the conceptual and theoretical efficacy of *GroupJoin*.

III. BASELINE COMPARISON

This section details an empirical study comparing the performance of two fundamental database query processing strategies: a traditional post-aggregation hash join (Hash-Join then Aggregate) and a more optimized pre-aggregation strategy (Group-Join) outside of DuckDB. The next section emphasizes the *GroupJoin* implementation in DuckDB.

A controlled simulation environment was developed using C++ to implement and time these algorithms. Synthetic datasets of varying sizes and key uniqueness were generated using *Python* scripts to facilitate a comprehensive benchmark. The results, visualized using *matplotlib* and *seaborn* libraries, conclusively demonstrate that the pre-aggregation approach offers a considerable performance and scalability advantage. The code repository is at this *GitHub link*.

A. Tools and Technologies

The following system, tools and technologies were used for the baseline comparison and implementing *GroupJoin* in DuckDB:

System Specifications:
- **OS:** Ubuntu Linux 25.04 (Linux 6.14.0-15-generic)
- **CPU:** AMD Ryzen 7 5800H (8 cores, 16 threads)
- **RAM:** 24 GB DDR4

Tools and Build Setup:
- **Visual Studio Code (VSCode)**: Used as the primary code editor, offering features like IntelliSense, debugging support, and Git integration.
- **CMake**: Used for configuring the DuckDB build system. It simplifies the compilation and linking of large C++ projects with modular components.
- **GCC (GNU Compiler Collection)**: Used as the C++ compiler for building the DuckDB source code.
- **Make**: Used to orchestrate the compilation process through CMake-generated Makefiles.
- **GDB and LLDB**: Used for debugging and step-wise inspection of the DuckDB execution engine during development and testing.

In this section, we compare simulations of a simple query with the following form:

```sql
SELECT A.k, SUM(A.v) AS summ
    FROM A JOIN B
    ON A.k = B.k
    GROUP BY A.k;
```

Listing 1. Query for Baseline Benchmarking

B. Data Generation

A Python script ('data_gen.py') was developed to generate synthetic data for two tables, A and B as A.txt and B.txt.

- **Table A**: Contains two integer columns, '(k, v)'.
- **Table B**: Contains a single integer column, '(k)'.

The script accepts command-line arguments to control the number of rows in each table and the degree of key(k) uniqueness across both tables. Our key k is both the **join** key and **group-by** key. This key-uniqueness allows for benchmarking performance under various data distributions, from scenarios with many duplicate keys (low uniqueness) to those where most keys are unique.

C. Benchmark Implementation

A C++ program ('combined_compare.cpp') was implemented to simulate the database operations. It reads the two tables from CSV files ('A.txt', 'B.txt') into memory and executes the target query using two distinct algorithms.

1) Algorithm 1: Hash-Join then Aggregate (Post-Aggregation): This method follows the traditional simple Hash-Join algorithm [8] followed by aggregation:

1) **Join Phase (Build + Probe)**: A hash table is built in memory on the key 'k' from Table A. Table B is then streamed, and for each row, the hash table is probed. For every match found, a new row is materialized in a large intermediate 'JoinedRow' vector.
2) **Aggregation Phase**: The program iterates through the entire intermediate 'JoinedRow' vectors, computing the final 'SUM(v)' for each key 'k' using a second hash map.

2) Algorithm 2: Group-Join (Pre-Aggregation): This method avoids the creation of the large intermediate table by summarizing data first:

1) **Pass 1**: The program scans Table A and computes a partial aggregate, storing the sum of 'v' for each distinct key 'k' in a hash map: $Map_A(k \rightarrow \sum v)$.
2) **Pass 2**: The program scans Table B and computes the count for each distinct key 'k' in a second hash map: $Map_B(k \rightarrow \text{count}(*))$.
3) **Final Join Phase**: The program iterates through one of the small summary maps (e.g., Map_B) and probes the other. For each matching key, the final aggregate is calculated by multiplying the pre-calculated sum from Map_A with the count from Map_B.

D. Results and Analysis

The following two tables provide a granular view of the raw performance data collected from the C++ benchmark program for two tables (e.g., 1M rows, 200M rows). The keys (k) were generated randomly for both tables with uniqueness percentages ranging from 10 to 100 in steps of 10. We measured the times and relative speedups.

TABLE II
PERFORMANCE RESULTS FOR TABLE SIZE: 1M

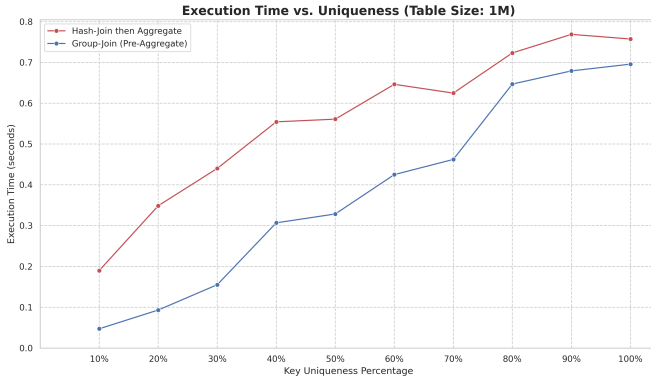| Uniqueness	Hash-Join (s)	Group-Join (s)	Speedup
10%	0.1899	0.0472	4.02x
20%	0.3488	0.0932	3.74x
30%	0.4403	0.1552	2.84x
40%	0.5545	0.3069	1.81x
50%	0.5611	0.3287	1.71x
60%	0.6465	0.4251	1.52x
70%	0.6250	0.4624	1.35x
80%	0.7234	0.6471	1.12x
90%	0.7689	0.6794	1.13x
100%	0.7576	0.6959	1.09x

Fig. 1. Execution Time v/s Key Uniqueness

TABLE III
PERFORMANCE RESULTS FOR TABLE SIZE: 200M

Uniqueness	Hash-Join (s)	Group-Join (s)	Speedup
10%	86.38	45.47	1.90x
20%	130.09	66.52	1.96x
30%	168.98	93.65	1.80x
40%	184.21	112.37	1.64x
50%	201.80	135.32	1.49x
60%	224.79	183.21	1.23x
70%	236.50	201.35	1.17x
80%	244.85	223.87	1.09x
90%	256.34	244.13	1.05x
100%	259.19	254.11	1.02x

Fig. 2. Execution Time v/s Key Uniqueness

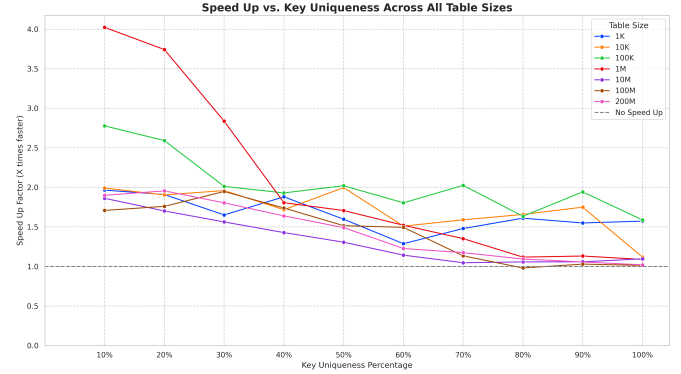We now generate visualization plots for comparing the runtimes and speedups of varying configurations.

Fig. 3. Speed Up v/s Key Uniqueness

The generated plots consistently show that the *GroupJoin* (Pre-Aggregation) algorithm outperforms the *Hash-Join-then-Aggregate* approach across all tested data sizes and uniqueness levels. This is for the simplest query type (Listing-1).

The key observation is that the performance gap widens dramatically as key uniqueness decreases (i.e., as the number of duplicate keys increases). This is because low uniqueness causes the size of the intermediate table in the post-aggregation method to explode, leading to high memory pressure and processing overhead. In contrast, the pre-aggregation method's memory footprint is dependent only on the number of *distinct* keys, making it far more scalable. The speedup factor, which quantifies this difference, is highest at low uniqueness values and gradually decreases as uniqueness approaches 1.0, though the Group-Join method remains faster in almost all scenarios.

E. Conclusion of Baseline Comparison

The results of this study provide strong quantitative evidence supporting the initial hypothesis that *GroupJoin* would perform better. The pre-aggregation (Group-Join) strategy is demonstrably superior in performance and scalability to the naive post-aggregation (Hash-Join then Aggregate) strategy for queries involving *joins* and *group-by aggregations* even for

the simple query where materialization cost for each matched row is very little. The findings underscore the importance of query optimization techniques that minimize the size of intermediate data structures, a core principle in the design of efficient database systems. We believe that for more complex scenarios, the execution time for the traditional approach would be higher due to increased materialization costs, while the *GroupJoin* operator's time would be comparatively less.

IV. METHODOLOGY

This section focuses on the *GroupJoin* implementation in DuckDB and the methodology used.

A. Development Setup and Tools

The implementation of the GroupJoin operator was carried out using the open-source DuckDB codebase, version 1.1.0, available at https://github.com/duckdb/duckdb/tree/v1.1.0. All development was performed on a *Linux* machine which provides a stable environment for systems programming and database development. The code repository is at this *GitHub link*.

DuckDB was compiled from source with development flags enabled to facilitate debugging and instrumentation. We worked with three build modes, such as: *release*, *debug*, and *release with debug*. Custom logging and profiling utilities were added to trace query plan construction, physical operator selection, and execution flow for queries matching the join-group-by pattern. The performance comparison in section VIII was done with *release* mode.

B. Experimental Datasets

We use the following datasets for evaluation:

- **TPC-H Benchmark:** Datasets generated at scale factors 1, 5, 10, 30, 50 using DuckDB's built-in extensions (e.g., `tpch-sf10.db`, `tpch-sf30.db`, etc.). These datasets simulate real-world analytical workloads with complex join and aggregation queries, commonly used for benchmarking database systems. We use these datasets for performance comparison.
- **Synthetic Uniform Datasets:** Custom-generated datasets with two tables (`A` and `B`) at increasing scales — 1K, 1M, and 10M rows — saved as `small.db`, `medium.db`, and `big.db` respectively. These were used to validate the correctness and scalability of the *GroupJoin* operator during **development phase** in controlled settings.

C. Evaluation Metrics

To judge our implementation, we focused on these metrics with *primary* focus on first:

- **Accuracy:** Does the query produce the correct number of rows and the correct values on specified dataset as compared to the expected results.
- **Latency:** How long does it take, on average over few runs, to process one query on a given dataset by our *GroupJoin* implementation and their default execution.

D. Evaluation Queries

We use the following custom TPC-H style queries on our TPC-H datasets, call them **Q1**, **Q2** and **Q3** respectively.

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

Listing 2. Lineitem-Orders GroupJoin Query

```sql
SELECT c_custkey, SUM(c_acctbal) AS
    total_balance
FROM customer
JOIN orders
  ON customer.c_custkey = orders.o_custkey
GROUP BY c_custkey;
```

Listing 3. Customer-Orders GroupJoin Query

```sql
SELECT ps_partkey, SUM(ps_supplycost)
FROM partsupp
JOIN part
  ON ps_partkey = p_partkey
GROUP BY ps_partkey;
```

Listing 4. Partsupp-Part GroupJoin Query

V. IMPLEMENTATION OVERVIEW

Our initial implementation focuses on having a working GroupJoin operator in DuckDB that works with existing operators and interface in the codebase, and working with suitable queries targetted by the operator. For now, our operator works with simpler types like INTEGER/BIGINT and FLOAT/DECIMAL for the join columns and group by keys. It can be extended to generic cases with not much effort. Also, for now we explore only the cases where the join column matches the group by key of one relation among the participating two relations. We check the working for these three aggregate variants: SUM(), COUNT(), and AVG() only.

VI. KEY TERMS AND CONCEPTS

A. 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* extension.

B. Query Planning

When a 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_NL_JOIN* etc. The choice of actual algorithm is left to the optimizer based on estimated cardinality in DuckDB.

C. Internal Structures

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 we keep it at 2048 as it is standard in DuckDB.

D. Query Execution

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.

VII. Implementation Details

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. 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_COMPARISON_JOIN* operator somewhere in the child nodes. If such a child node exists, we replace the parent with our *GROUPJOIN* physical operator and return the generated plan. The query planner is extended to detect join-group-by patterns and replace them with the GroupJoin operator whenever applicable.

For example, when we run the following query in Listing-4 to see the physical plan, we get the modified physical plan as shown in Fig-5 from original plan in Fig-4.

```
explain
select l_orderkey, sum(l_extendedprice)
    from lineitem JOIN orders
    on l_orderkey = o_orderkey
    group by l_orderkey;
```

Listing 5. Chcking Physical Plan using Explain

Algorithm 1 CANREPLACEBYGROUPJOIN1(LogicalOperator &op)

1: **if** op.type $\neq$ LOGICAL_AGGREGATE_AND_GROUP_BY **then**
2:     **return false**
3: **end if**
4: $groupby \leftarrow op$.Cast<LogicalAggregate>()
5: **if** $groupby$.groups.size() > 0 **then**
6:     **if** $groupby$.children[0] $\neq$ null **then**
7:         **if** $groupby$.children[0].type = LOGICAL_COMPARISON_JOIN **then**
8:             **return true**
9:         **end if**
10:     **else if** $groupby$.children[0].children[0] $\neq$ null **then**
11:         **if** $groupby$.children[0].children[0].type = LOGICAL_COMPARISON_JOIN **then**
12:             **return true**
13:         **end if**
14:     **end if**
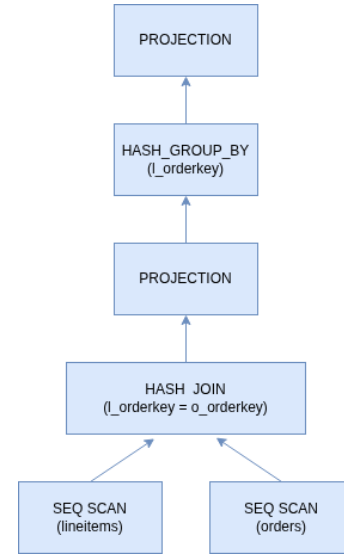15: **end if**
16: **return false**

Fig. 4. Physical Plan Before replacing with GroupJoin

Fig. 5. Physical Plan After replacing with GroupJoin

To support **GroupJoin** as a physical operator in DuckDB, we implemented a new class `PhysicalGroupJoin` derived from the base `PhysicalJoin` operator. This operator is designed to hold two children, namely left and right table, from where data will be scanned. `PhysicalJoin` inherits from `PhysicalOperator` operator representing the base of all operators in the database.

A. Class Structure

The *PhysicalGroupJoin* class overrides key interfaces from the base physical operator hierarchy (*PhysicalJoin* and *PhysicalOperator*) to support the three core phases of DuckDB's execution engine: **sink**, **operator**, and **source**.

The following metadata are stored in the operator's state:
- `condition` — stores the join condition (= or !=, etc.)
- `groups`, `aggregates` — expressions representing the group-by keys and aggregate functions.
- `aggregation_map`, `final_results` — hash maps used to accumulate intermediate and final group-aggregated values.

B. Sink Phase

In our implementation, the `Sink()` method is responsible for buffering the input data rather than performing aggregation immediately. During this phase, incoming `DataChunks` from the left and right children are copied and stored into the custom global sink state.

No aggregation or join logic is performed at this stage. This design choice simplifies parallelism and decouples compute-heavy logic from the ingestion phase. Remember that *Sink()* can be called in parallel for the two datasources(two tables) are they are independent pipelines having same operator(GroupJoin) where they *sink* their data.

This method is called by the *pipeline executor* multiple times for each operator, until all *datachunks* have been *sunk*.

C. Finalize Phase

The core logic of join and aggregation is executed during the `Finalize()` method. Once all input data has been collected, we iterate over the stored chunks to perform on-the-fly aggregation. This is known as early-aggregation as we know that we are into this *Physical Plan* seeing the query and having decided *GroupJoin* is possible.

Depending on the join condition, one of the following routines is called:
- `PerformEqualityAggregation()` for equality joins (=)
- `PerformInEqualityAggregation()` for inequality joins (!=, etc.)

These routines traverse the buffered left and right chunks, compare tuples according to the join condition, and update the aggregation hash maps (such as `final_results`) in-place. This two-phase approach makes the operator modular and easier to debug.

This method is called by the *pipeline executor* only once for each pipeline(table scan), so we must prevent double aggregation.

D. Source Phase

The `GetData()` method emits the final aggregated results, packaged as `DataChunks` to the consumer of the pipeline, which is the *Projection* operator at the top of the query plan. This allows GroupJoin to act as a full producer of data, similar to an aggregation operator.

This method is called by the *pipeline executor* multiple times for each consumer, till all *datachunks* have been *sourced*.

E. Planner Integration

To integrate GroupJoin into the query planner, we detect patterns of the form:

```sql
SELECT k, SUM(v)
FROM A JOIN B ON A.k = B.w
GROUP BY k;
```

Listing 6. Supported Query Template

Such patterns are identified in the logical plan, and replaced with a custom logical operator node that is later transformed into a `PhysicalGroupJoin` during physical plan generation.

F. Operator Characteristics

Our PhysicalGroupJoin operator has the following characteristics:
- **Pipeline aware:** Implements both sink and source interfaces, making it usable as a standalone pipeline.
- **Non-parallel:** The current implementation is single-threaded, with `ParallelSink()` and `ParallelSource()` both returning false.
- **Order sensitive:** Preserves output ordering, indicated via `SourceOrder()` and `OperatorOrder()` returning `FIXED_ORDER`.
- **Custom State:** We also define custom sink and source states to hold per-thread and global accumulation data. These are derived from DuckDB's `GlobalSinkState` and `GlobalSourceState` interfaces.

G. Pipeline Construction

In DuckDB, each physical operator can contribute one or more execution pipelines depending on whether it acts as a source, sink, or intermediate operator. In the case of `PhysicalGroupJoin`, the operator acts as a sink and source, so it creates a pipeline that performs the final aggregation in the `Finalize()` phase.

The `BuildPipelines()` method is responsible for defining the dependencies between the GroupJoin pipeline and its child pipelines. DuckDB builds and schedules these pipelines based on such dependencies.

The pipeline building process proceeds as follows:
- The current pipeline is marked as the finalization pipeline for the join, by calling `SetPipelineSource()`.
- Two child meta-pipelines are created — one for the left child and one for the right child — using `CreateChildMetaPipeline()`.
- These meta-pipelines recursively build their own pipelines to scan the input tables.
- Finally, the current (GroupJoin) pipeline is marked as dependent on the left and right base pipelines. This ensures that the data is fully scanned and buffered before the GroupJoin's finalization runs.

The following code snippet shows the C++ implementation:

```cpp
// Construct pipelines for PhysicalGroupJoin
void PhysicalGroupJoin::BuildPipelines(Pipeline &
    current, MetaPipeline &meta_pipeline) {
  D_ASSERT(children.size() == 2); // binary join

  auto &left_child = children[0];
  auto &right_child = children[1];

  // has Finalize(), so mark it as the sink
  auto &state = meta_pipeline.GetState();
  state.SetPipelineSource(current, *this);

  // Recursively build meta-pipelines for children
  MetaPipeline &left_meta = meta_pipeline.
      CreateChildMetaPipeline(current, *this);
  left_meta.Build(*left_child);

  MetaPipeline &right_meta = meta_pipeline.
      CreateChildMetaPipeline(current, *this);
  right_meta.Build(*right_child);

  // Obtain shared_ptrs to the pipelines
  shared_ptr<Pipeline> current_ptr = current.
      shared_from_this();
  shared_ptr<Pipeline> &left_ptr = left_meta.
      GetBasePipeline();
  shared_ptr<Pipeline> &right_ptr = right_meta.
      GetBasePipeline();

  // Add dependency on both child pipelines
  current_ptr->AddDependency(left_ptr);
  current_ptr->AddDependency(right_ptr);
}
```

Listing 7. Pipeline construction in `PhysicalGroupJoin`

H. Aggregation Logic

We primarily focus on equality aggregation as it is amenable to the Hash-Join-Aggregate algorithm. The `PerformEqualityAggregation()` method in the `PhysicalGroupJoin` class implements a hash-based aggregation strategy for queries with equality join conditions followed by a `GROUP BY`. This implementation avoids the overhead of materializing the full join result and instead performs aggregation using independent scans of two tables.

- **Step 1: Pre-Aggregate Left Table (A)**
  The left input table (referred to as `left_data`) is scanned once. For each row, the grouping key and value column are extracted, and the partial sum is computed and stored in a hash map: `left_sums`. This map has the structure: Key $\rightarrow$ SUM(value).

- **Step 2: Count Matching Keys in Right Table (B)**
  The right input table (`right_data`) is scanned, and the grouping key is extracted from each row. A second hash map `right_counts` is maintained to count the number of occurrences of each key: Key $\rightarrow$ COUNT(*).

- **Step 3: Final Aggregation**
  For every key in `left_sums`, the corresponding count from `right_counts` is retrieved (if it exists), and the final result is computed as:

  $$final\_results[key] = left\_sum \times right\_count$$

  This result is stored in the global `final_results` map. If a key from the left table is not present in the right table, it is ignored. Refer Listing-8 for details.

The following code snippet shows the C++ implementation for the equality-based aggregation:

```cpp
void
duckdb::PhysicalGroupJoin::
    PerformEqualityAggregation(
  duckdb::GroupJoinGlobalSinkState &global_state,
  std::unordered_map<int, double>& final_results)
      const {

  std::cout << "Inside PerformEqualityAggregation()"
      << std::endl;

  std::unordered_map<int, double> left_sums;
  std::unordered_map<int, long long int>
      right_counts;

  duckdb::DataChunk lscan_chunk;
  duckdb::DataChunk rscan_chunk;

  // Scan left table and
  // compute SUM(value) grouped by key
  global_state.left_data.InitializeScan();
  while (global_state.left_data.Scan(lscan_chunk)){
    for (size_t i = 0; i < lscan_chunk.size(); ++i){
        int key;
        double value;
        try {
            key = lscan_chunk.data[0].GetValue(i).
                GetValue<int>();
            value = lscan_chunk.data[1].GetValue(i).
                GetValue<double>();
        } catch(const std::exception& e) {
            std::cout << e.what() << std::endl;
            continue;
        }
        left_sums[key] += value;
    }
  }

  // Scan right table
  // and count matching keys
  global_state.right_data.InitializeScan();
  while (global_state.right_data.Scan(rscan_chunk)){
    for (size_t i = 0; i < rscan_chunk.size(); ++i){
        int key;
        try {
            key = rscan_chunk.data[0].GetValue(i).
                GetValue<int>();
        } catch(const std::exception& e) {
            continue;
        }
        right_counts[key]++;
    }
  }
```

```cpp
    // Now the hasmaps are ready, and we do
        accumulate

    // Final aggregation: sum * count
    for (const auto &left_entry : left_sums) {
        int key = left_entry.first;
        double sum = left_entry.second;

        long long int matching_rows =
        right_counts.count(key)? right_counts[key]: 0;
        if (matching_rows > 0) {
            final_results[key] = sum * matching_rows;
        }
    }
}
```

Listing 8. Equality Aggreagtion Logic in `PhysicalGroupJoin`

The `PerformInEqualityAggregation()` method in the `PhysicalGroupJoin` class implements similar hash-based aggregation strategy for queries with inequality join conditions followed by a `GROUP BY`. The core logic is similar; we use non-matching rows to compute the *final_results* map.

VIII. RESULTS AND ANALYSIS

We generated the TPC-H datasets for various scaling factors using the in-built DuckDB extensions as follows:

```
INSTALL tpch;
LOAD tpch;
```

Listing 9. Dataset generation

TPC-H datasets were generated for scale factors of 1, 5, 10, 30, and 50, as shown in table IV.

TABLE IV
TPCH DATABASE FILE SIZES AND GENERATION COMMANDS

Command	Scale Factor	Size on Disk
`CALL dbgen(sf = 1);`	SF=1	0.27 GB
`CALL dbgen(sf = 5);`	SF=5	1.3 GB
`CALL dbgen(sf = 10);`	SF=10	2.7 GB
`CALL dbgen(sf = 30);`	SF=30	8.2 GB
`CALL dbgen(sf = 50);`	SF=50	13.7 GB

The following relations are involved in the three queries **Q1**, **Q2** and **Q3** which are given in section IV.

TABLE V
CARDINALITY OF TPC-H TABLES ACROSS DIFFERENT SCALE FACTORS

Table Name	SF = 1	SF = 5	SF = 10	SF = 30	SF = 50
`customer`	150K	750K	1.50M	4.50M	7.50M
`lineitem`	6.00M	30.00M	59.99M	180.01M	300.01M
`orders`	1.50M	7.50M	15.00M	45.00M	75.00M
`partsupp`	800K	4.00M	8.00M	24.00M	40.00M
`part`	200K	1.00M	2.00M	6.00M	10.00M

A. Initial Results

The performance results, given in tables VI through X, consistently show that our proof-of-concept *GroupJoin* implementation is significantly slower than DuckDB's native hash-join-aggregate pipeline across all tested scale factors. The performance ratio, calculated as DuckDB's execution time divided by our operator's time from start to end of query execution, ranges from approximately 0.08x to 0.27x. It is to be noted that DuckDB uses cache-conscious radix-partitioned hash-join [8](multiple hash tables instead of one) followed by aggregate as the two operators, while we use a single hash-table inside the *GroupJoin* operator. We used the clean DuckDB release *(v1.1.0)* to compare against our implementation of GroupJoin on DuckDB *(v1.1.0)* for the tables VI through X.

TABLE VI
QUERY PERFORMANCE COMPARISON (SF=1)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	0.596	4.211	1.50M	0.14x
Q2	0.040	0.517	200K	0.08x
Q3	0.064	0.248	100K	0.26x

TABLE VII
QUERY PERFORMANCE COMPARISON (SF=5)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	2.857	15.775	7.50M	0.18x
Q2	0.212	2.552	1.00M	0.08x
Q3	0.441	1.964	500K	0.22x

TABLE VIII
QUERY PERFORMANCE COMPARISON (SF=10)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	5.965	41.459	15.00M	0.14x
Q2	0.525	5.339	2.00M	0.10x
Q3	1.317	4.928	1.00M	0.27x

TABLE IX
QUERY PERFORMANCE COMPARISON (SF=30)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	17.443	90.710	45.00M	0.19x
Q2	1.970	15.428	6.00M	0.13x
Q3	5.073	25.207	3.00M	0.20x

TABLE X
QUERY PERFORMANCE COMPARISON (SF=50)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	33.756	204.800	75.00M	0.16x
Q2	3.885	21.990	10.00M	0.18x
Q3	7.224	30.466	5.00M	0.24x

B. Analysis

The primary architectural difference and a major contributor to the performance gap is our operator's blocking nature versus DuckDB's fully streaming execution model. Our naive implementation first consumes and buffers the entire relations into an in-memory state. Only after this is complete does it begin probing and computing the final results. But DuckDB streams the *datachunks* without buffering them all.

The second fundamental reason for the performance disparity is parallelism. Our naive GroupJoin operator is implemented as a single-threaded algorithm and uses a single hash-table, resulting in costly lookups and inserts. DuckDB, on the other hand, has its hash-join and aggregation operators heavily parallelized, capable of partitioning the workload across all available CPU cores. It uses cache-conscious radix-partitioned hash-join, making it faster. This intra-operator parallelism dramatically accelerates both the construction of the hash table and the subsequent probing and aggregation steps, leading to the orders-of-magnitude performance advantage observed in our results.

C. Performance under restrictions

To establish a fair basis for comparison, we evaluate our *GroupJoin* operator against a restricted version of the native DuckDB (v1.1.0) implementation. We will call this **restricted** native implementation. By restricted, we mean limiting the number of threads to one, removing the parallelism of the *Source*, *Sink* and *Operator* interfaces of all operators (namely **HashJoin** and **Aggregate**). We will schedule the *pipeline events* one after another like *table scans*. Also, we will restrict the algorithm to use only one hash-table (by setting partitions to 1). These modifications are intended to align the execution model of the native engine with our proof-of-concept *GroupJoin* implementation, which does not leverage parallelism, thus enabling a more direct comparison. We limit the number of threads to one for fair comparison as:

```
PRAGMA threads=1;
```

Listing 10. Limiting threads

```cpp
// make the operator non-parallel
bool ParallelSink() const override {
    return false;
}
bool SinkOrderDependent() const override {
    return true;
}
// set one scheduler task (serial execution)
duckdb::TaskScheduler::SetSchedulerThreads(1);
// force no partitioning
idx_t RadixHTConfig::InitialSinkRadixBits() const {
    return 0;
}
// force no partitioning
idx_t RadixHTConfig::MaximumSinkRadixBits() const {
    return 0;
}
```

Listing 11. Changes made in restricted native implemenation

We measure the query running times (in seconds) using the following inbuilt DuckDB extension:

```
.timer ON
```

Listing 12. Time measurement

TABLE XI
QUERY PERFORMANCE COMPARISON (SF=1)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	11.145	4.211	1.50M	2.65x
Q2	0.630	0.517	200K	1.22x
Q3	1.129	0.248	100K	4.55x

TABLE XII
QUERY PERFORMANCE COMPARISON (SF=5)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	55.758	15.775	7.50M	3.53x
Q2	3.484	2.552	1.00M	1.37x
Q3	7.330	1.964	500K	3.73x

TABLE XIII
QUERY PERFORMANCE COMPARISON (SF=10)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	113.950	41.459	15.00M	2.75x
Q2	15.627	5.339	2.00M	2.93x
Q3	12.584	4.928	1.00M	2.55x

TABLE XIV
QUERY PERFORMANCE COMPARISON (SF=30)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	371.444	90.710	45.00M	4.09x
Q2	54.994	15.428	6.00M	3.56x
Q3	39.349	25.207	3.00M	1.56x

TABLE XV
QUERY PERFORMANCE COMPARISON (SF=50)

Query	Their Time (s)	My Time (s)	Output Size	Speedup
Q1	607.644	204.800	75.00M	2.97x
Q2	74.604	21.990	10.00M	3.39x
Q3	63.236	30.466	5.00M	2.08x

D. Analysis under restrictions

The performance results under the restricted, single-threaded execution environment, presented in tables XI through XV, show a significant and insightful reversal of the previous findings. Our proof-of-concept *GroupJoin* operator now consistently and substantially outperforms the restricted native *Hash-Join-then-Aggregate* pipeline. The observed speedup ranges from approximately 1.22x to a remarkable 4.55x, demonstrating a clear performance advantage in this **restricted** setting.

IX. Conclusion

While our implementation does not achieve performance parity with DuckDB's unrestricted native operators, it successfully serves its primary purpose of correctly implementing the *GroupJoin* logic within the DuckDB ecosystem. However, we observed significant performance improvements when comparing our operator against a restricted version of DuckDB, where parallelism was disabled by making it single-threaded, removing parallelism from the *Source* and *Sink* interfaces, and scheduling pipeline events sequentially.

Our baseline comparison demonstrated that the *GroupJoin* operator can significantly reduce the overhead of intermediate data materialization, leading to superior performance in this controlled setting. The end-to-end integration of our physical *GroupJoin* operator into DuckDB achieved a considerable speedup under these **restricted** conditions, validating its conceptual design.

X. Future Work

This work lays the foundation for future optimization. The clear path forward involves re-architecting the *GroupJoin* operator to bridge the performance gap with the unrestricted DuckDB implementation by:

- **Introducing streaming model:** Processing data chunk-by-chunk instead of buffering them all.
- **Introducing parallelism:** Partitioning the operator state and workload to leverage multi-core architectures.
- **Optimizing memory access:** Implementing a more efficient hash table (like radix-partitioned hash table) to improve cache performance.

References

[1] H. Mühleisen and M. Raasveldt, "DuckDB: An Embeddable Analytical Database," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, ACM, 2019.
https://dl.acm.org/doi/pdf/10.1145/3299869.3320212

[2] C. Koch, Y. Katsis, M. Nikolic, et al., "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views," in *The VLDB Journal*, vol. 23, no. 2, pp. 253–278, 2014.
https://link.springer.com/article/10.1007/s00778-013-0348-4

[3] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," in *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.
https://www.vldb.org/pvldb/vol4/p539-neumann.pdf

[4] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *CIDR*, pp. 225–237, 2005.
https://www.cidrdb.org/cidr2005/papers/P19.pdf

[5] M. Zukowski, N. Nes, and P. A. Boncz, "Vectorwise: A Vectorized Analytical DBMS," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pp. 1349–1350, 2012.
https://ieeexplore.ieee.org/document/6228203

[6] G. Moerkotte and T. Neumann, "Accelerating Queries with Group-By and Join by Groupjoin," in *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 843–854, 2011.
https://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf

[7] P. Fent and T. Neumann, "A Practical Approach to Groupjoin and Nested Aggregates," in *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2383–2396, 2021.
https://vldb.org/pvldb/vol14/p2383-fent.pdf

[8] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 3–14.
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6544839

[9] *Official DuckDB Documentation*
https://duckdb.org/docs/stable/

[10] *Official C++ Reference*
https://en.cppreference.com/index.html