



Summer Project Presentation

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

Agenda

1. Introduction: The Problem & The Proposed Solution
2. Part 1: Baseline Comparison (Outside DuckDB)
3. Part 2: GroupJoin Implementation in DuckDB
4. Key Concepts
5. Methodology
6. Implementation Details
7. Results & Analysis
8. Initial Performance vs. Native DuckDB
9. Performance vs. "Restricted" DuckDB
10. Conclusion & Future Work

Introduction

The World of Analytical Databases

- **DuckDB:** A high-performance, open-source analytical database system designed for efficient in-process data analytics.
- **Analytical Queries:** Often involve joining multiple tables and then aggregating the results (e.g., summarizing sales per region).
- **Common Pattern:** A **JOIN** operation immediately followed by a **GROUP BY** clause is a ubiquitous pattern in these workloads.



The Core Problem: Inefficiency in Traditional Execution

Traditional query engines handle
JOIN and **GROUP BY** as two separate, sequential operators.

Major Drawbacks:

- **Large Intermediate Results:** The JOIN can create a massive intermediate table that must be fully materialized (written to memory).
- **High Memory Overhead:** Storing this intermediate result consumes significant memory and can lead to high memory pressure.
- **Wasted I/O & Cache Misses:** Reading and writing this large temporary result is inefficient.

The Proposed Solution: GroupJoin Operator



- **Concept:** A new physical operator that **fuses** the JOIN and GROUP BY operations into a single, tightly coupled step.
- **How it works:** It bypasses the creation of the full intermediate join result and performs aggregation directly on-the-fly during the join process.

How it helps: It saves wasted I/O and materialization costs

This is the high-level idea

We will see the implementation details later

Previous Work

- This work builds upon established ideas in database research:
- **Moerkotte and Neumann** : First formalized the GroupJoin operator, showing its potential to reduce query plan complexity and improve performance in analytical settings.
- <https://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf>
- **Fent and Neumann**: Presented a practical implementation of GroupJoin in a compiled query engine, demonstrating its real-world viability.
- <https://vldb.org/pvldb/vol14/p2383-fent.pdf>

Project Goal & Phased Approach

*

The project was executed in two main phases:

Phase 1: Baseline Comparison

- **Goal:** To demonstrate the conceptual and theoretical efficacy of the GroupJoin strategy outside of a full-featured database system.
- **Method:** A controlled C++ simulation comparing a pre-aggregation (GroupJoin) vs. post-aggregation (Join-then-Aggregate) approach.

Phase 2: DuckDB Implementation

- **Goal:** To implement *GroupJoin* as a new physical operator within the DuckDB query engine and evaluate its performance.
- **Method:** Worked with open-source DuckDB codebase and tested on TPC-H style queries at various scaling factors.

Phase 1: Baseline Comparison

- **Goal:** To demonstrate the conceptual and theoretical efficacy of the GroupJoin
- **Environment:** A C++ program to simulate the two query execution strategies.
 - Algorithm 1: Hash-Join-Then-Aggregate (Traditional)
 - Algorithm 2: GroupJoin
- **Data:** Synthetic datasets generated by a Python script, allowing control over table size and key uniqueness. Here, key k is both join key and group by key.
- **Key Metric:** Execution time, measured across varying levels of key uniqueness (from 10% to 100%) and table sizes from 10K to 100M by 10x jumps.
- **Query:** A simple but representative analytical query:

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

Baseline Comparison - The Two Algorithms

- Algorithm 1: Hash-Join-Then-Aggregate (Traditional)

Join Phase: Build a hash table on Table A, probe with Table B, and materialize every matching row into a large intermediate vector.

Aggregate Phase: Iterate through the entire large intermediate vector to compute the final $\text{SUM}()$ using a second hash map.

- Algorithm 2: Our GroupJoin (Early Aggregation)

Pass 1: Scan Table A to compute a partial aggregate: $\text{map}_A(k \rightarrow \text{SUM}(v))$.

Pass 2: Scan Table B to compute counts for each key: $\text{map}_B(k \rightarrow \text{COUNT}(*))$.

Final Join: Join the two *small* summary maps to calculate the final result.

Baseline Comparison

Results & Key Finding

- Result:

GroupJoin (Pre-Aggregation) consistently and significantly performs better.

- Key Finding:

The performance gap widens dramatically as key uniqueness decreases.

Why?

- Low uniqueness causes the intermediate table in the traditional method to explode in size, leading to high memory pressure and processing overhead.
- The GroupJoin memory footprint only depends on the number of *distinct* keys, making it far more scalable.

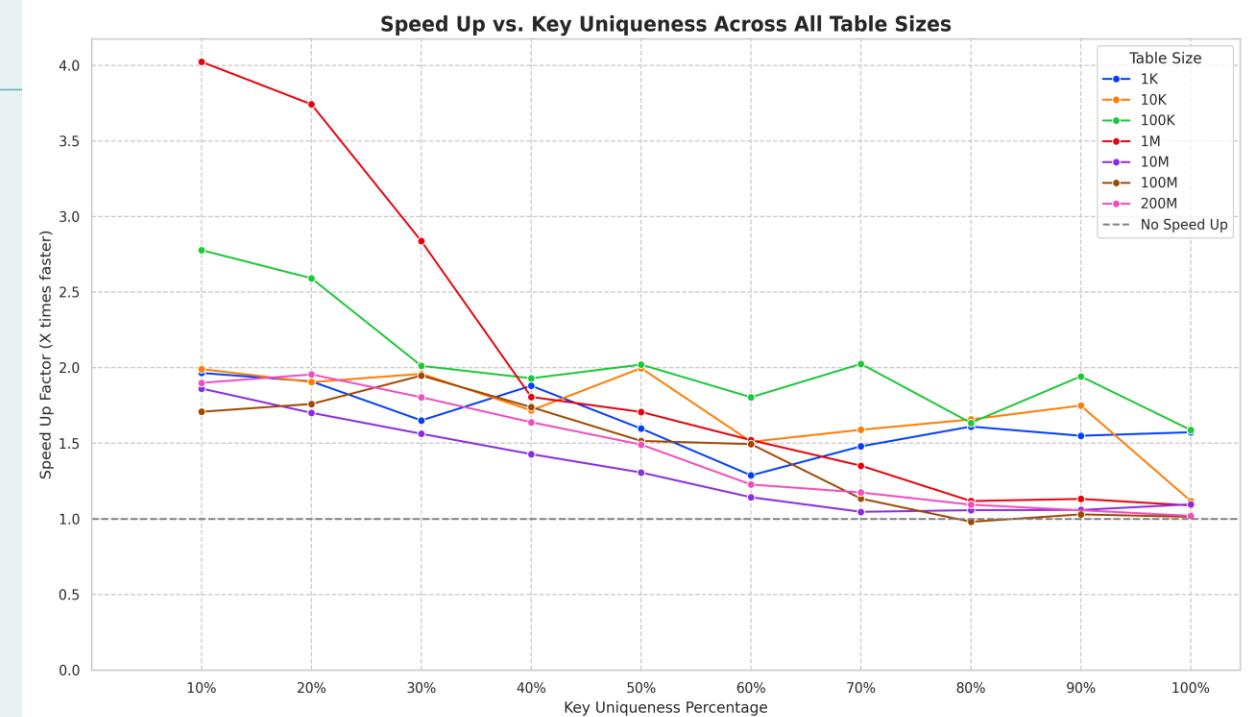


TABLE II
PERFORMANCE RESULTS FOR TABLE SIZE: 1M

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

Phase 2: Implementation in DuckDB - Overview & Scope

Goal: Integrate a new [PhysicalGroupJoin](#) operator into the DuckDB (v1.1.0) execution engine

Methodology:

- Modified query planner to recognize the JOIN-GROUPBY pattern.
- Implemented the operator logic within DuckDB's [PhysicalOperator](#) framework.
- Evaluated for accuracy and latency with [TPC-H](#) datasets.

Initial Scope:

- **Types:** INTEGER/BIGINT and FLOAT/DECIMAL.
- **Condition:** Join key must match the group-by key.
- **Aggregates:** SUM(), COUNT(), and AVG().



DuckDB Internals: A Primer

- **Query Plans:** Queries are transformed from a **Logical Plan** (what to do) into a **Physical Plan** (how to do it). Our goal is to replace a **LOGICAL_COMPARISON_JOIN** followed by a **LOGICAL_AGGREGATE** in logical plan with our single **PHYSICAL_GROUPJOIN** in physical plan.
- **Data Processing Unit:** Data flows through operators in **DataChunks**, which are sets of columnar vectors. The standard vector size is 2048 values.
- **Execution Model:** DuckDB uses a push-based, pipelined model. Operators are chained together, and data is pushed through the pipeline from a **Source** to a **Sink**

DuckDB Internals: Planner Integration

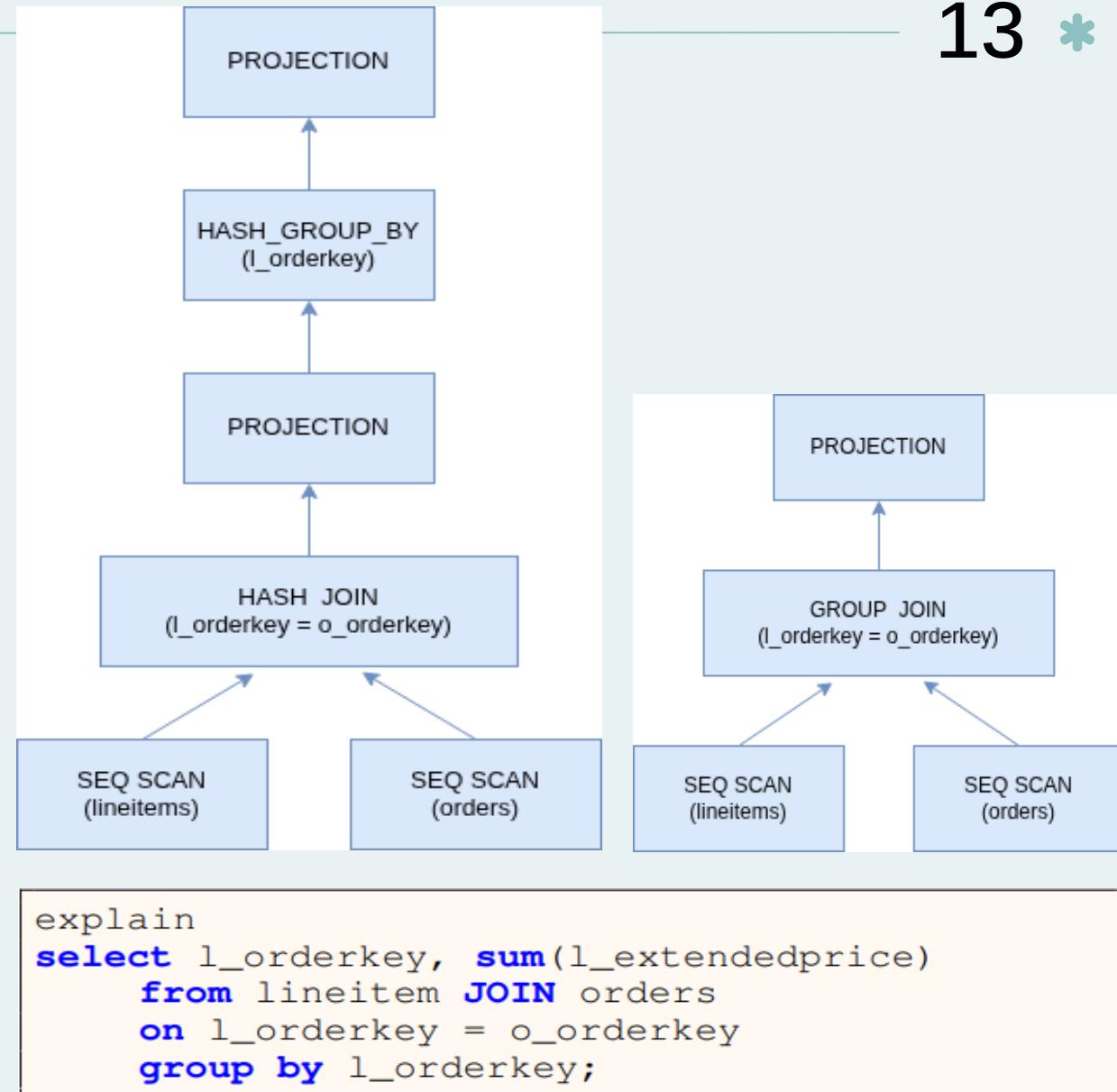
We extended the query planner to automatically replace the standard **two-operator** plan with our fused operator.

Algorithm 1 CANREPLACEBYGROUPJOIN1(LogicalOperator &op)

```

1: if op.type ≠ LOGICAL_AGGREGATE_AND_GROUP_BY then
2:   return false
3: end if
4: groupby ← op.Cast<LogicalAggregate>()
5: if groupby.groups.size() > 0 then
6:   if groupby.children[0] ≠ 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] ≠ 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

```



DuckDB GroupJoin: Implementation Details

- **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 → SUM(value)`.

- **Step 2: Count Matching Keys in Right Table (B)**

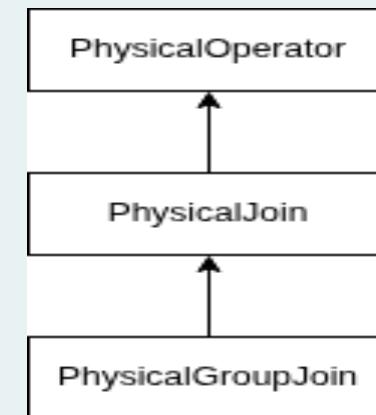
The right input table (`right_data`) is scanned, and the grouping key is extracted from each row. A second hash map `right_counts` is maintained to count the number of occurrences of each key: `Key → COUNT(*)`.

- **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:

$$\text{final_results}[\text{key}] = \text{left_sum} \times \text{right_count}$$

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.



DuckDB Internals: Experimental Setup

```
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

```
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

```
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

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

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

Initial Results: Slower than Native DuckDB

- **Result:** Our initial naive implementation was **significantly slower** than DuckDB's highly optimized native pipeline.
- **Speedup:** The performance ratio ranged from a poor **0.08x to 0.27x**. This means our operator was roughly 4x to 12x slower.

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

Initial Results: Slower than Native DuckDB

Analysis: Why Was It Slower?

The performance gap was expected and is attributable to two main factors:

- **Blocking vs. Streaming Execution:**
 - **Our Naïve Operator:** Is a "blocking" operator. It consumes and buffers *a//* data from both tables before producing a single output row.
 - **DuckDB Native:** Uses a fully "streaming" (pipelined) model, processing data in small chunks, which allows for better resource utilization and lower latency.
- **Single-Threaded vs. Parallelism:**
 - **Our Naïve Operator:** Implemented as a single-threaded algorithm using one simple hash table.
 - **DuckDB Native:** The hash join and aggregate operators are heavily parallelized. They use modern techniques like cache-conscious, radix-partitioned hash tables to maximize multi-core CPU performance.

A Fairer Comparison: The "Restricted" Experiment

To prove the *conceptual* advantage of the **GroupJoin** fusion, we needed a fair comparison.

- **Idea:** to manually "de-optimize" the native DuckDB engine to match the limitations of our implementation.
- **Restrictions Applied to Native DuckDB:**
 - **Disabled Parallelism:** Forced the engine to run on a single thread (`PRAGMA threads=1;`).
 - **Disabled Radix Partitioning:** Forced the native hash join to use only a single partition, mimicking our simple hash table.
 - **Making Interfaces and Pipeline Non-Parallel:** We forced the Source, Sink and pipeline events to run in serial fashion and mimic our implementation.

This creates a fair, single-threaded showdown between

the two-operator (HashJoin-Aggregate) approach and our fused (GroupJoin) operator approach.

Restricted Results: GroupJoin Wins!

To prove the *conceptual*/advantage of the **GroupJoin**, we needed a fair comparison.

Under the restricted, single-threaded execution model, the results reversed dramatically.

- **Result:** Our GroupJoin operator now consistently and substantially outperforms the restricted native pipeline.
- **Speedup:** The observed speedup ranges from 1.22x to 4.55x.

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

```
// 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 implementation



Conclusion

We successfully implemented an end-to-end new
PhysicalGroupJoin operator in DuckDB.

While slower than the fully optimized native engine,
our operator proved to be significantly faster
in a controlled, single-threaded environment.

This validates the core hypothesis:
Fusing join and aggregation reduces overhead and can lead to
superior performance.

Future Work



Introduce a Streaming Model:

Re-architect to process data chunk-by-chunk instead of buffering everything.

Introduce Parallelism:

Partition the operator's state and workload to leverage multi-core CPUs.

Optimize Memory Access:

Implement a more efficient, hash table (e.g., radix-partitioned).



Akash Maji

✉ akashmaji@iisc.ac.in

🌐 akashmaj.me

Report: https://akashmaj.me/reports/GroupJoin_Report_Akash.pdf