Cost Model for Heterogeneous Architectures

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Mistry Kathan Bhargavkumar

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2025

Declaration of Originality

I, **Mistry Kathan Bhargavkumar**, with SR No. **04-04-00-10-51-23-1-22804** hereby declare that the material presented in the thesis title

Cost Model for Heterogeneous Architectures

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2024-25**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: 21/06/2025 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof.Jayant R. Haritsa Advisor Signature

DEDICATED TO

My family and to all the elevators,

for always lifting me up

Acknowledgements

I would like to express my deepest gratitude to **Prof. Jayant R. Haritsa**, my advisor at the Database Systems Laboratory, Indian Institute of Science, Bengaluru, for his insightful guidance, constant encouragement, and unwavering support throughout the course of this work.

I am also profoundly thankful to **Dr. Harish Doraiswamy** of Microsoft Research for his valuable feedback, constructive suggestions, and the many enriching discussions that helped shape the direction of this thesis.

My heartfelt thanks go to my lab mates and friends—**Himanshu Devrani, Suprit Chafle, Alaap Surendran, Bhavya Choudhary, Shweta Shukla, Dhruvin Chaudhari, Dev Gandhi**, and **Sakshi Sindhwal**—for their camaraderie, constant motivation, and the countless brainstorming sessions that made this journey both productive and enjoyable.

I am grateful to the **DuckDB Discord community** for generously answering my questions and helping me overcome technical hurdles, and to **Kuntal Ghosh** for sharing his invaluable knowledge of PostgreSQL.

Abstract

Join and GroupBy operations are among the most computationally intensive processes in database query execution, often serving as critical performance bottlenecks in data-intensive applications. Leveraging the massive parallelism inherent in hardware accelerators such as FPGAs, GPUs, and TPUs presents a promising avenue to accelerate these operations. Although existing research has explored various Join and GroupBy operations on these architectures, no database engine to date integrates both CPU-based and accelerator-based operators in a hybrid fashion to maximize performance. In this study, we introduce GPU-based implementations of Join and GroupBy operators designed which deliver efficient performance. We propose a cost model designed for resource-limited environments that reliably estimates the execution time of these operations. A major contribution is the integration of GPU-based Join and GroupBy operators into DuckDB, a leading vectorized database system. By enabling these operators, our work demonstrates the practical viability of GPU acceleration within DuckDB.

Contents

List of Figures

Chapter 1

Introduction

Join and GroupBy operations are crucial and often among the most time-intensive tasks in database systems. To better understand the performance characteristics, we conducted CPU profiling in DuckDB[24, 27] on five random queries from the TPC-H and TPC-DS benchmark suites [2, 3]. For TPC-H, we profiled Q4, Q9, Q13, Q18, and Q21, while for TPC-DS, we profiled Q18, Q35, Q40, Q82, and Q98. The reported times for each operator represent the total wall-clock time spent completing its work, including actual computation, I/O operations, and any waiting for resources or thread synchronization. In addition, each operator's reported time is the sum of all occurrences of that operator within the query. This reflects the real-world execution time of the operator in query processing. We conducted experiments for scale factor 100.

As shown in Figure 1.1, the Join and GroupBy operators constitute a significant portion of the overall execution time in database queries(Each bar contains thin white lines that separate the individual Join and GroupBy operations.). For these standard queries with the default parameters from the TPC-H and TPC-DS benchmark suites, analyzing the query plans shows that the optimizer frequently selects hash joins as one of its preferred algorithms, underscoring their importance.

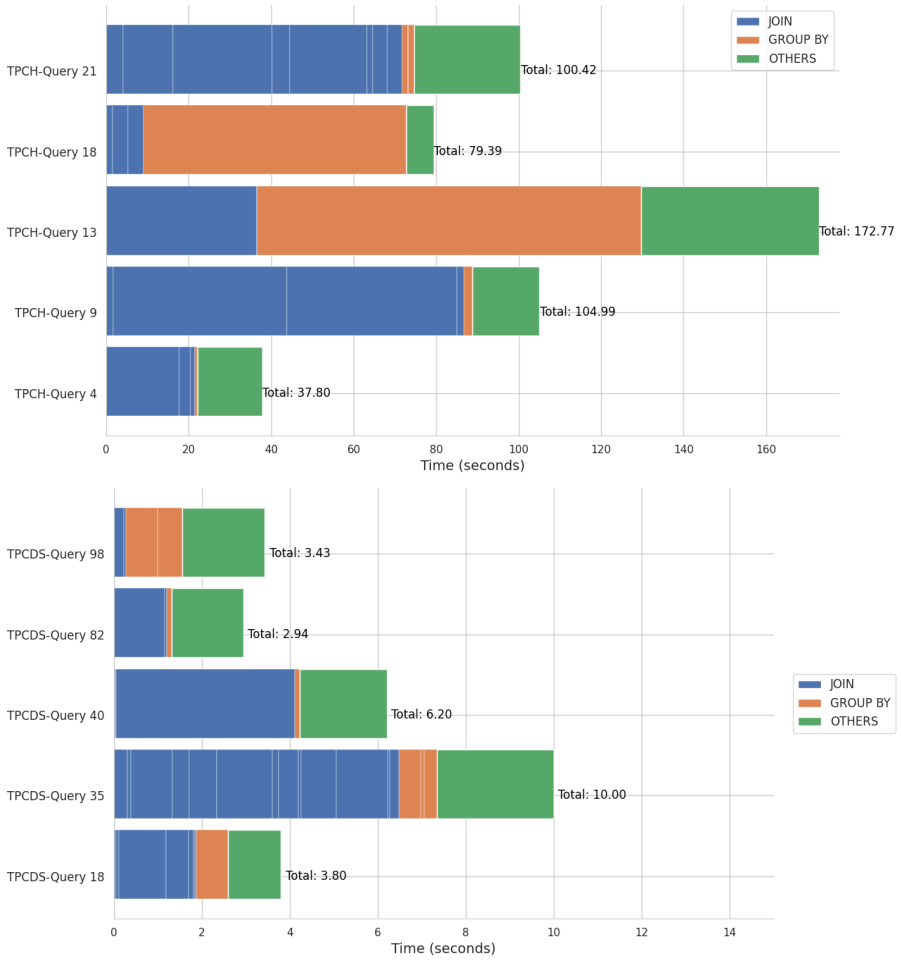

Figure 1.1: Query Profiling on CPU

Efforts to optimize the hash Join and GroupBy operators have led to the exploration of advanced hardware platforms, including FPGAs, GPUs, and TPUs, as alternatives to traditional CPU-based execution. Modern multicore architectures have demonstrated superior performance for hash Join and GroupBy operations compared to traditional CPUs [10, 8, 30, 17].

However, current research predominantly focuses on executing these operations entirely on specialized hardware, to achieve better performance. However, for some workloads the overhead of transferring data to the device can outweigh the gains from accelerated execution, so performing the hash join on the CPU remains the more efficient option. On the other hand, for smaller tables, CPU-based nested loop Joins often outperform hash Joins, while sort-merge Joins can be more efficient on CPUs when the data is pre-sorted or can be sorted effectively [5]. The same trade-offs apply to GroupBys.

Despite these advancements, existing solutions lack a hybrid execution strategy that lever-

ages the strengths of different architectures to achieve potentially better performance across a wide range of workloads.

In this work, recognising that query planners depend on accurate cost estimates, we design a dedicated model for GPU-accelerated Join and GroupBy. Incorporating this model allows the engine to choose the fastest execution path for a given workload and hardware profile, as illustrated in Figure 1.2.

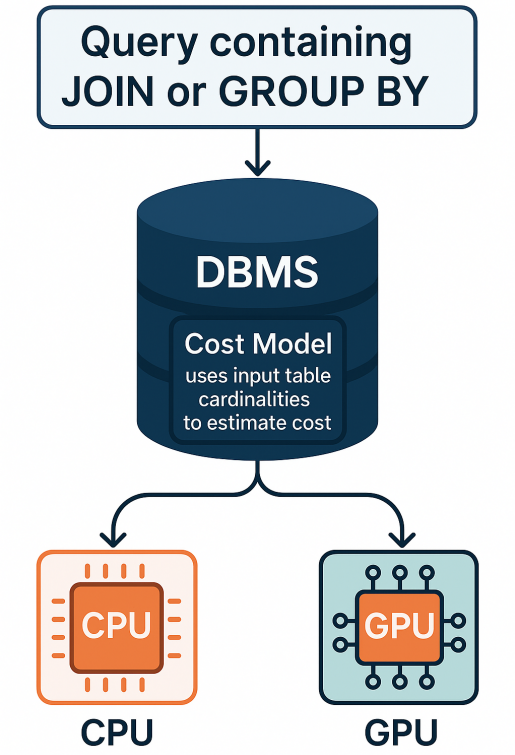

Figure 1.2: Overview of the system

Chapter 2

Background

2.1 Cost-model

In a relational database engine, a cost model is the component of the query optimizer that estimates the "cost" of executing a given query plan. Formally, it is a function that maps a proposed execution plan (and the current database state) to a numerical cost value representing the expected resource usage [20]. In these systems, the optimizer generates many logically equivalent plans for a SQL query, uses the cost model to estimate the cost of each plan, and then chooses the plan with the lowest cost. Importantly, cardinality estimates—the number of tuples(rows) an operator processes—are a crucial input to cost models.

Existing work on cost modelling can be grouped into two streams [15, 13]: (a) *analytic models with parameter tuning* and (b) *black-box ML models*.

The former focuses on adjusting the cost parameters within the predefined cost functions of the existing models. In contrast, the latter involves training ML models on a set of executed query plans to predict the execution time for new queries. While the whitebox tuning approach achieves only limited accuracy and does not question the fundamental assumptions of the cost functions, the black-box ML models are data-intensive, lack explainability, and may perform poorly on queries that differ significantly from those in the training set.

Because no published analytic model exists for GPU operators—especially for **Joins** and **GroupBys**—there is nothing to retune. We therefore design a new analytic model whose terms capture GPU-specific effects (PCIe transfers, kernel launches, hash-table operations, *etc.*) and then fit its coefficients from measurements of representative workloads.

As highlighted by Lan et al., errors in cardinality estimation—the estimated number of tuples produced by an operator—can propagate through the cost model and amplify into deviations of many orders of magnitude, often yielding severely sub-optimal execution plans [21].

In our work, we therefore assume perfect cardinality information (that is, every intermediate and final cardinality is known exactly) so as to isolate and evaluate the intrinsic accuracy and robustness of our cost model without conflating it with cardinality-induced variance.

2.2 DuckDB

DuckDB(DDB) is an in-process analytical database management system (often described as "SQLite for analytics"). Internally, DDB is designed for OLAP (online analytical processing) workloads, using a columnar storage format and a vectorized execution engine to efficiently handle large scans and complex queries.

One of DDB's core design features is its execution engine, which processes data in chunks (vectors) rather than one row at a time. Instead of the classical "tuple-at-a-time" Volcano model[14], DDB operates on batches of values (Typically 2048 values in a vector) for each operation. Query operators produce and consume entire vectors/chunks rather than individual rows.

DuckDB employs a **push-based**(data-flow) execution engine: operators consume input vectors, process them, and immediately push the produced chunks downstream, all the way to the pipeline's sink. Unlike a Volcano-style pull model—where the root repeatedly calls 'GetChunk()' on its children—each DuckDB operator drives its successors proactively, enabling tight, cache-friendly pipelines and minimizing function-call overhead.

It supports SOTA CPU implementations of GroupBy and Joins[30, 27]. DDB's extensible and modular architecture makes it ideal for integrating custom operators, such as our GPU operators, allowing seamless integration into its execution pipeline. Additionally, DDB's ability to handle SQL queries with minimal setup ensures a user-friendly interface for testing and benchmarking our operators.

Each column chunk in DDB appears as a compact *UnifiedVectorFormat*(UVF) triple: `data` (pointer to values), `validity` (NULL bitmap), and a `SelectionVector` $\langle s_0, s_1, \dots \rangle$ that maps logical row i to `data[`s_i`]`. We keep the pointer even for dense chunks ($s_i = i$) so every operator sees the same branch-free "pointer + offset" interface across plain, dictionary, or run-length storage.

A one-off `ToUnifiedFormat()` call performs any decoding *before the critical inner loop*—the tight per-tuple loop in joins and aggregations—so a NULL check degenerates to `validity.RowIsValid(s)` and avoids branchy `if(null)...` code.

Output is zero-copy: create a new `SelectionVector` and call `vector.Slice(sel,count)`, an $\mathcal{O}(1)$ pointer swap that *re-wraps* the same buffer (i.e., re-wrapping here means building a new view over the identical data, with no copying). During hash-join build/probe we then

materialise only the selected, non-NULL keys into contiguous host arrays—already ideal for a single bulk transfer if GPU support is added later.

Thus, **UVF + SelectionVector** is our uniform abstraction for every "gather" (key extraction) and "scatter" (chunk emission) step, removing encoding differences, minimising data movement, and keeping the critical loops branch-less and cache-friendly.

Strings are encoded using UTF-8 encoding[31] in DDB.

2.3 In-memory Join and GroupBy on GPUs

In this work, we assume that all inputs and outputs fit entirely within GPU memory. The GroupBy operator processes a single relation

$$\mathcal{R}(k_1, k_2, \ldots, a_1, \ldots, a_m),$$

where $k_1, k_2, \ldots$ are the grouping key(s) (each of type `INT64` or `VARCHAR`), and $a_1, \ldots, a_m$ are the payload (non-key) columns.

For the Join, we focus on the most common inner *equi*-join— specifically, the many-to-many case where duplicate keys may appear on either side. It takes two input relations

$$\mathcal{R}(k, a_1, \ldots) \quad \text{and} \quad \mathcal{S}(k, b_1, \ldots),$$

matches tuples on the single join key k (either `INT64` or `VARCHAR`), and produces the output relation

$$\mathcal{O}(k, a_1, \ldots, b_1, \ldots).$$

Here $\mathcal{R}.k$ and $\mathcal{S}.k$ are the join keys, and $a_1, \ldots, b_1, \ldots$ denote the payload (non-key) attributes carried into the result.

2.4 Existing join and groupby algorthms

Several GPU-based algorithms have been developed for executing Joins and GroupBy aggregations efficiently. For Join operations, notable methods include Garuda's graphics-driven query execution approach[8] and the hardware-conscious hash join algorithm[9]. For GroupBy operations, existing solutions include the grahics driven method, and the Efficient GPU-Accelerated GroupBy Aggregation algorithm[11] published recently.

However, due to practical constraints, we could not directly utilize these existing implementations. Specifically, permission to use the Garuda implementation was not obtained, and the hardware-conscious hash Join algorithm contains known unresolved bugs that would require sig-

nificant debugging time. Consequently, we developed our own algorithms for Join and GroupBy operations tailored specifically for this research. While these algorithms may not represent the absolute state-of-the-art, they are effective, functional, and reliably meet the requirements of our cost modeling experiments.

Chapter 3

Experimental setup

3.1 Hardware

We ran all experiments on a workstation powered by a 12th Gen Intel Core® i9-12900K processor (x86-64, 24 threads) and 125 GiB of RAM. We used an NVIDIA GeForce RTX 4090 GPU with 24 GiB of dedicated memory. All code was compiled using CUDA 12.5 and GCC 11.4.0. To ensure consistent performance measurements, we cleared the system cache before each test and restarted DDB prior to every query execution.

3.2 Workload Description

We evaluated performance on TPC-H[23](skewed versions with zipf= 0.1, 0.5 and 1.0 to understand perfornace with the skewness of data) and TPC-DS benchmarks using scale factors of 1, 10, 50, and 100. We were not able to scale more than this because our system was not able to generate the data because of high data movements. We also use a data generator, which is used in several studies[9] to benchmark performace.

Each benchmark was run multiple times, and reported runtimes correspond to the average execution time across iterations.

All the testing and training queries are available here:[19]. Queries feature keys of varying types—both integer and string (`CHAR`/`VARCHAR`)—and every aggregate payload column is a 64-bit integer (`INT64`). The workload combines keys with low, medium, and very-high cardinality alongside diverse aggregate functions (`COUNT(*)`, `SUM`, `MIN`, `MAX`, `AVG`).

[1]We refer to NVIDIA hardware and terminology in this work.

Chapter 4

GroupBy

4.1 GPU Group By overview

To start with, we developed a hash GroupBy operator specifically for this research.

Design overview. Our GroupBy forwards 64-bit integer keys unchanged, but encodes each `VARCHAR` key as a 64-bit FNV-1a hash—a fast, non-cryptographic function with a low collision rate[12]—while maintaining a hash → string table for later decoding. The stand-alone implementation then runs a three-kernel pipeline that keeps ALL intermediate state resident on the GPU:

1. **Row-hash prepass** (`kernel_row_hash`). A single thread per input row constructs a 64-bit composite hash by Murmur-mixing every key column,

$$h \leftarrow \operatorname{mix}\big(h \oplus k_0\big) \ \ldots \ \operatorname{mix}\big(h \oplus k_{K-1}\big),$$

   where mix(·) is the 64-bit MurmurHash3 finalizer [28]. MurmurHash3 was selected for its speed and efficiency in generating uniformly distributed hash values, which is crucial for reducing collisions. Hashes are `row_hash` buffer for perfectly coalesced reads in the next phase.

2. **Lock-free build & aggregation** (`kernel_insert`). Each thread linearly probes a power-of-two table of capacity $C = 2\,\lceil 2N \rceil_2$ until it can claim a bucket:

   2.a. *Reservation.* `atomicCAS(&ht_keys[0][pos], HT_EMPTY, HT_BUSY)` swaps an `HT_EMPTY` sentinel for `HT_BUSY`, guaranteeing that exactly one thread performs the first write.

2.b. *Commit.* The winning thread fills the remaining key columns, executes `__threadfence()`, and publishes the real key with `atomicExch`, flipping the bucket from *busy* to *full*.

2.c. *Collision resolution.* Non-owners compare every key column in registers; mismatches advance the probe $\texttt{pos} \leftarrow (\texttt{pos} + 1) \bmod C$.

Once the bucket is known, the thread updates the requested aggregates in place: `atomicAdd` for COUNT/SUM and typed `atomicMin`/`atomicMax` for extrema. No host intervention or coarse locks are required, so thousands of inserts progress fully in parallel.

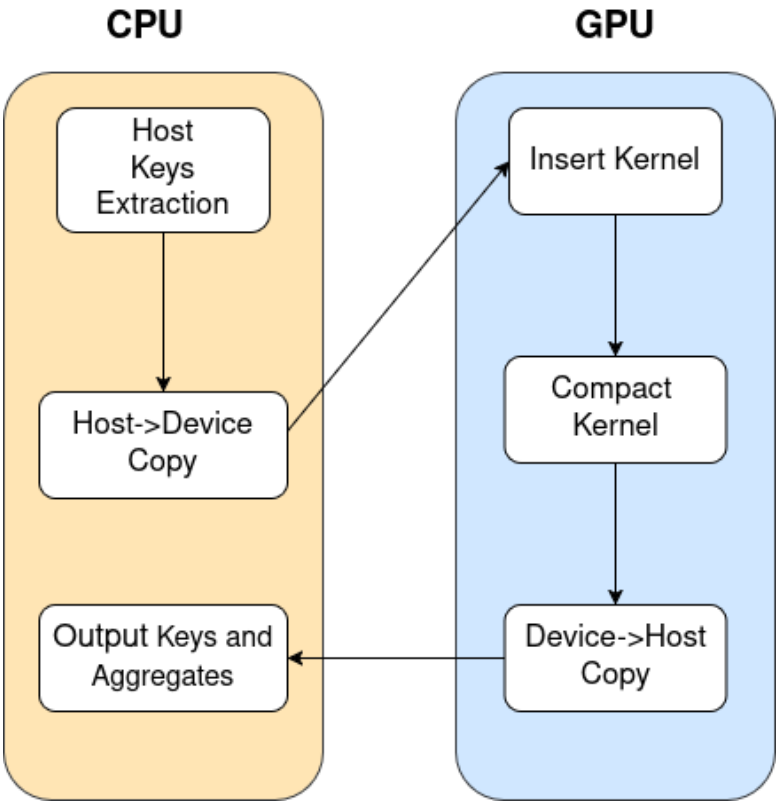

Figure 4.1: Working of GPU-Groupby

3. **Parallel compaction** (`kernel_compact`). A final sweep assigns dense output positions using a single `atomicAdd(d_size,1)`, copies occupied buckets into column-major result arrays, and leaves $|\text{groups}| = d\_\text{size}$ in device memory. The host performs exactly *two* bulk DMA transfers (table $\rightarrow$ dense, dense $\rightarrow$ host), independent of the number of groups.

Overall, we implement our GROUP BY operator as illustrated in Figure 4.1.

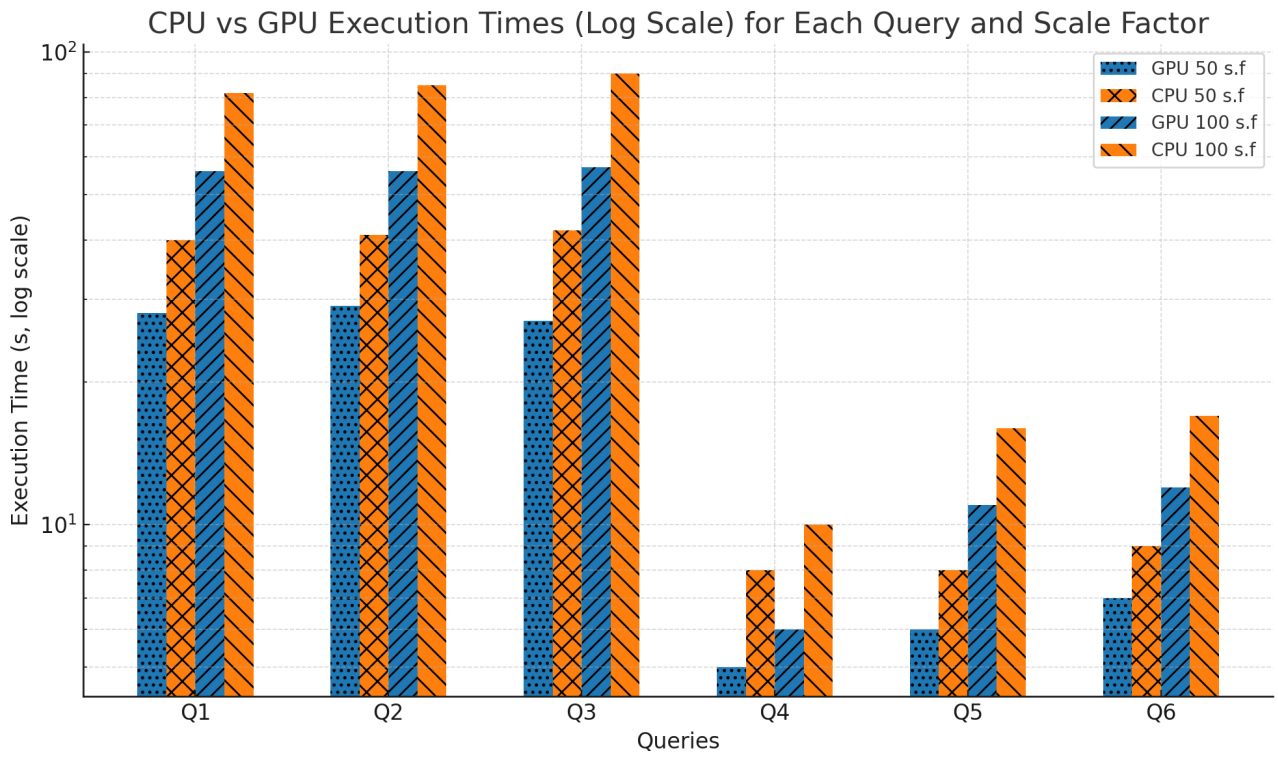

Figure 4.2: CPUvGPU

To evaluate our GPU GroupBy performance, we implemented a comparable single-core CPU hash GroupBy using `std::unordered_multimap`. This approach avoids the FNV-based VARCHAR→64-bit conversion used in our GPU code—relying instead on direct hashing of strings—thereby reducing per-row overhead on the CPU.

Figure 4.2 compares the total execution time (i.e., the end-to-end time from query submission to receiving the final result) of six queries from our GroupBy query suites on CPU and GPU implementations.

We approximately get 2x speed up for every benchmark.

4.2 Extending DDB

4.2.1 Implementating New GroupBy Operator

We began by exploring the codebase of the DDB database engine. To integrate our GPU GroupBy operator, we defined a new physical operator named `Physical_GPU_GROUPBY` and an associated logical operator named `GPU_GROUP_BY`. To ensure that DDB chooses our operator during execution, we initially modified the `plan_aggregate.cpp` file to hardcode the selection of `Physical_GPU_GROUPBY`.

As an example, consider the following simple database schema and query:

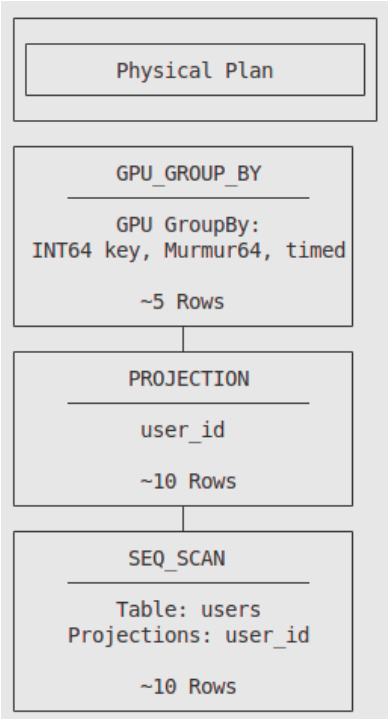

Figure 4.3: Example query plan of DDB

```
TABLE users (user_id BIGINT, enrollment_number BIGINT);

SELECT user_id AS key, COUNT(*) AS cnt
FROM users
GROUP BY user_id;
```

For this query, DDB generates a physical query plan, which is depicted in Figure 4.3.

DDB does not natively support files with a `.cu` extension, which are required for CUDA programming. To enable support for `.cu` files, we made several modifications to the `CMake` configuration files in the DDB codebase. These changes allow the integration of GPU-specific components within the existing DDB framework.

The `PhysicalGPUGROUPBY` class extends the `PhysicalOperator` class in DDB. This class implements key methods to manage the GroupBy operation, as detailed below:

- `Sink`
  - Appends every incoming `DataChunk` to a host-side "build buffer".

- `Finalize`
  - *Host preparation*
    - Flattens all key and value vectors.
    - Encodes each `VARCHAR` key to a 64-bit FNV-1a hash, while keeping a hash → string map for later recovery.
    - Collects INT64 payload columns (one per aggregate that is not `COUNT(*)`).
  - *Host → Device transfer*
    - Allocates device buffers for row keys, payloads, and an auxiliary pointer array for each.
    - Copies the flattened host columns into these device buffers.
  - *Key hashing on the GPU*
    - Launches `kernel_hash_keys` to compute a composite 64-bit Murmur-style hash for every row (supports up to 16 key columns).
  - *Hash-table build & aggregate update*
    - Creates a power-of-two, open-addressing hash table whose capacity is 2× the surviving row count.
    - Runs `kernel_insert_multi`: each thread inserts (or locates) its key and updates aggregate slots with atomic operations for `SUM`, `MIN`, `MAX`, and the running total used by `AVG`.
  - *Compaction*
    - Allocates dense output arrays for keys, counts, and aggregate accumulators.
    - Executes `kernel_compact_multi` to scan the sparse hash table and pack live entries contiguously.
  - *Device → Host transfer*
    - Downloads the compacted key arrays, group counts, and aggregate results.
  - *VARCHAR rebuild*
    - Uses the stored hash → string maps to recover original UTF-8 keys, resolving any collisions if they occurred.
- `GetData`
  - Streams the compacted result in batches of `STANDARD_VECTOR_SIZE`.

Overall workflow of the GroupBy in duckdb is as figure 4.4:

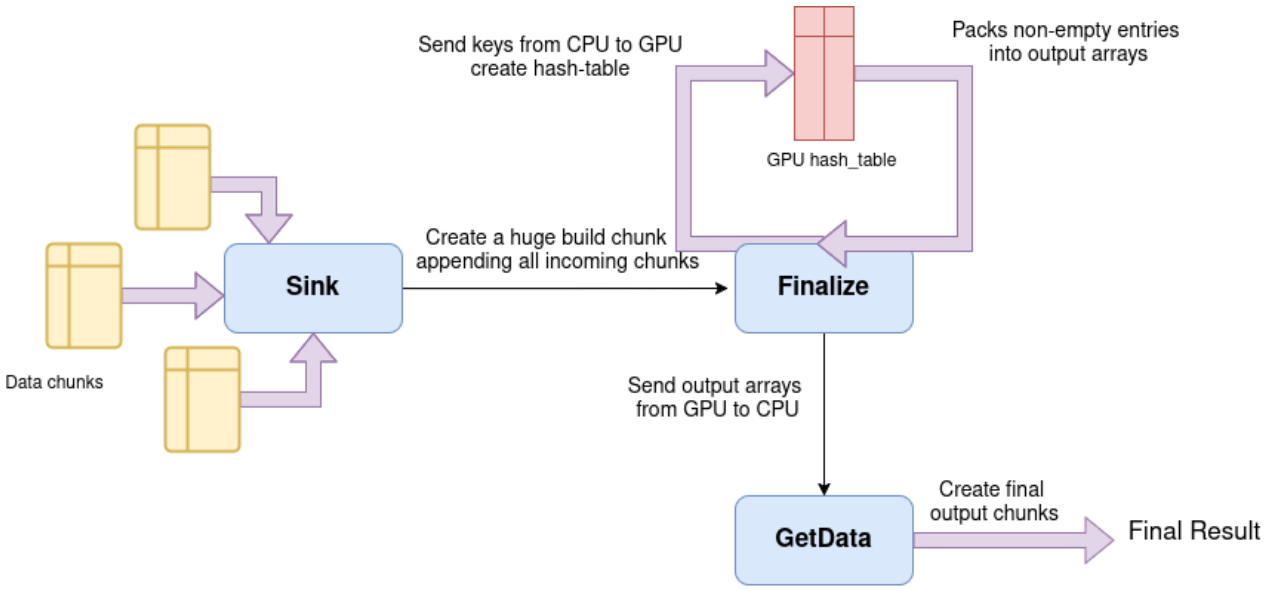

Figure 4.4: `Physical_GPU_GROUP_BY` implementation

Among the three main stages—`Sink`, `Finalize`, and `GetData`—DuckDB can dispatch multiple CPU threads for the `Sink` and `GetData` stages, but only a single thread for `Finalize`.

When using multiple threads for `Sink`, each thread produces a partial build chunk that must later be merged via a Combine step into one large chunk. We implemented both approaches (single-threaded `Sink` versus multi-threaded `Sink` + Combine) and consistently observed that the single-threaded `Sink` outperformed the multi-threaded `Sink` + Combine variant. So, we chose to stick with single threaded sink version.

Regarding `GetData`, its execution time is very small relative to the overall query runtime (to the point that we exclude it from our final cost model). As a result, parallelizing `GetData` across all available CPU threads yields negligible benefit. In summary, although DuckDB supports parallel `Sink` and `GetData` stages, in practice a single-threaded `Sink` is faster (due to Combine overhead), and parallelizing `GetData` does not meaningfully reduce total execution time.

The implementation of the described approach has been made available online. The complete codebase, including all relevant files and documentation, can be accessed at the following repository [18].

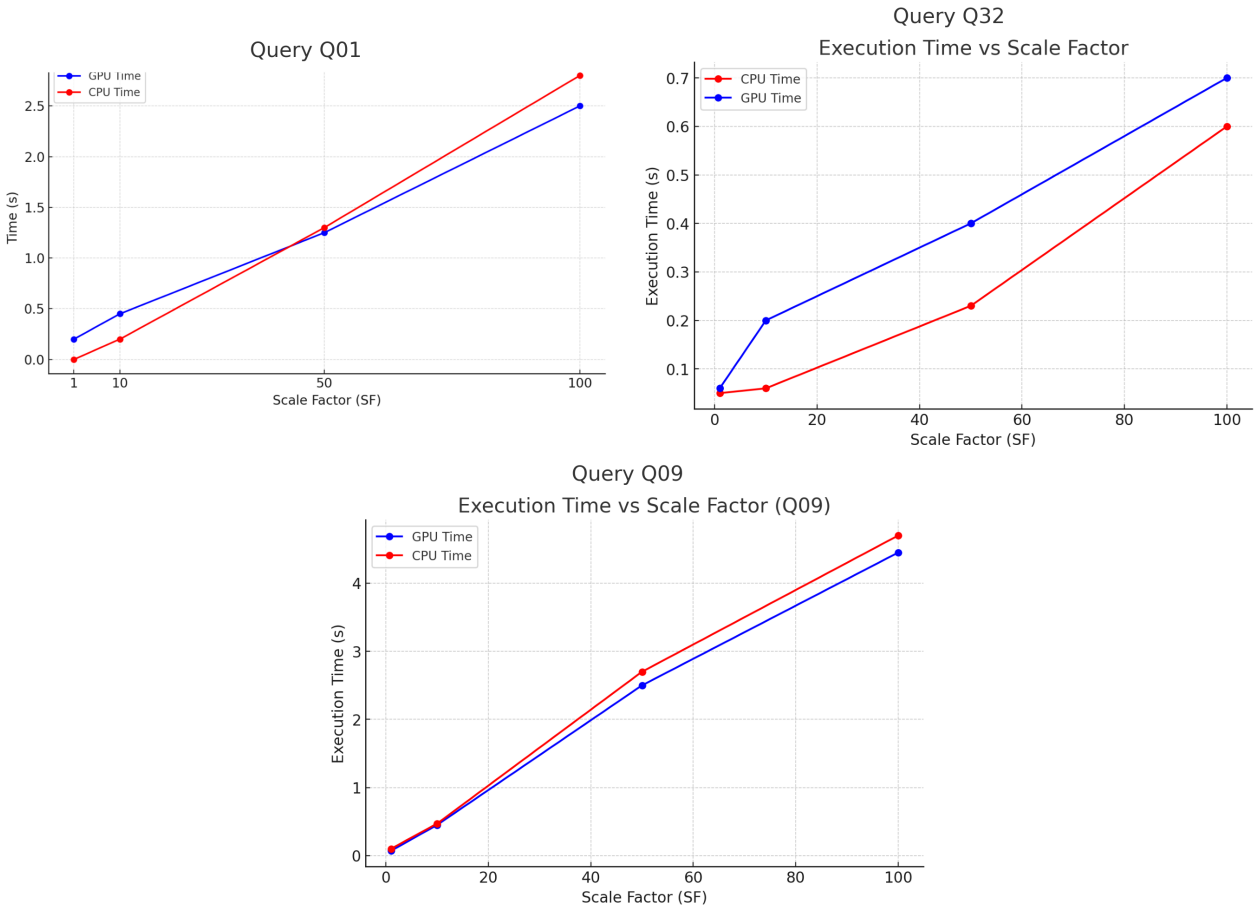

Figure 4.5: DDB's `Physical_Hash_AGGREGATE` vs GPU GroupBy

Figure 4.5 shows that at small scale factors, the CPU GroupBy outperforms the GPU version; as the data size grows, the GPU GroupBy runtime converges to the CPU's. We expect that large scales (e.g., 500 GB or 1 TB), the GPU's massive parallelism will overcome the CPU's increasing stall time, yielding clear GPU speedups.

4.3 Cost-model

4.3.1 Model Fitting Methodology

To determine the coefficients of the cost model, we perform a **separate multiple linear regression** for each execution sub-stage(Section 4.3.3's term-by-term breakdown of the cost model shows why a linear formulation is adequate.)

Each sub-stage is modeled as a linear combination of relevant features:

$$T = \sum_{i=1}^{d} c_i \cdot x_i = \mathbf{x}^\top \mathbf{c}$$

where:

- T is the measured execution time for the sub-stage (in nanoseconds),
- $\mathbf{x} = (x_1, x_2, \ldots, x_d)^\top$ is the feature vector,
- $\mathbf{c} = (c_1, c_2, \ldots, c_d)^\top$ is the coefficient vector to be estimated.

Given n observed measurements across multiple queries, we construct a design matrix:

$$X = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(n)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}^\top \in \mathbb{R}^{n \times d}$$

$$\text{and the response vector} \quad \mathbf{t} = \begin{bmatrix} T^{(1)} \\ T^{(2)} \\ \vdots \\ T^{(n)} \end{bmatrix} \in \mathbb{R}^n$$

We solve for the least-squares estimate of the coefficients:

$$\hat{\mathbf{c}} = \arg\min_{\mathbf{c}} \|X\mathbf{c} - \mathbf{t}\|_2^2$$

The solution is given by the closed-form expression:

$$\hat{\mathbf{c}} = (X^\top X)^{-1} X^\top \mathbf{t}$$

Once each sub-stage is fitted independently, the **total execution time** is modeled as the sum of all sub-stage models:

$$\begin{aligned} T_{\text{total}} \approx\ & T_{\text{sink}} + T_{\text{host-prep}} + T_{\text{H2D}} + T_{\text{hash}} + T_{\text{insert}} + T_{\text{compact}} \\ & + T_{\text{D2H}} + T_{\text{rebuild}} + T_{\text{getdata}} \end{aligned}$$

This process ensures high fidelity of the overall model by capturing the fine-grained contributions of each component stage.

$$
\begin{aligned}
T_{\text{total}} \approx\ & \underbrace{0.7375\,N\,w}_{\text{Sink()}} + \underbrace{\left(6.002\,K_{\text{int}}\,N + 1.590\,K_v\,B\,N + 2.855\,S_v\,N + 155.5\,K_v\,G_v\right)}_{\text{host-prep}} + \underbrace{0.63848\,(K+S_v)\,N}_{\text{host}\to\text{device}} \\
& + \underbrace{0.01043\,K\,N}_{\text{composite hash}} + \underbrace{\left(0.08505\,N + 0.1561\,S_v\,N + 0.4859\,G + 0.2055\,K\,G\right)}_{\text{insert kernel}} \\
& + \underbrace{\left(0.02988\,N + 0.01191\,K\,G + 0.03549\,G + 0.01952\,S_v\,G\right)}_{\text{compact kernel}} \\
& + \underbrace{2.4336\,(K+S_v+1)\,G}_{\text{device}\to\text{host}} + \underbrace{\left(0.2204\,K_v\,G + 5.29\,K_v\,B\,G\right)}_{\text{VARCHAR rebuild}} \\
& + \underbrace{\left(5.509\times10^{-5}\,(K_{\text{int}} + K_v\,B)\,G + 8.194\times10^{-4}\,S\,G\right)}_{\text{GetData()}}
\end{aligned}
\tag{4.1}
$$

4.3.2 Symbol legend

Symbol	Meaning
N	Surviving rows after the NULL-key filter
w	Input bytes *per* row shipped to `Sink()`
K_{int}	Number of integer key columns
K_v	Number of VARCHAR key columns
K	Total keys ($K_{\text{int}} + K_v$)
B	Avg. UTF-8 length of a VARCHAR key
S	Total number of aggregates
S_v	"Value" aggregates (SUM/MIN/MAX/AVG)
G_v	Number of distinct string hashes inserted (obtain this by first running SELECT COUNT(DISTINCT key) FROM table)
G	Final output groups (cardinality)

4.3.3 Cost equation

Equation 4.1 presents the complete cost model incorporating all execution stages, while Equation 4.2 shows a simplified version obtained by omitting terms with minimal impact on the

$$
\begin{aligned}
T_{\text{total}} \;\approx\;& 0.7375\,N\,w + \Big(6.002\,K_{\text{int}} + 1.590\,K_v\,B + 2.855\,S_v\Big)\,N \;+\; 155.5\,K_v\,G_v \\
&+\; 0.63848\,(K + S_v)\,N + \Big(0.08505 + 0.1561\,S_v\Big)\,N \;+\; \Big(0.4859 + 0.2055\,K\Big)\,G \\
&+\; 2.4336\,(K + S_v + 1)\,G + 5.29\,K_v\,B\,G
\end{aligned}
\tag{4.2}
$$

overall runtime. All costs are expressed in nanoseconds.

The following models detail each execution stage, outlining the contributing terms, their influencing factors, and the corresponding fitted coefficients.

1. Sink() ($0.7375\,N\,w$)

- During `Sink()` the executor *rowises* every surviving tuple: it loads contiguous elements from columnar vectors.
- The work therefore scales with the *exact byte volume* copied, $N\,w$ bytes.

2. Host preparation

- $6.002\,K_{\text{int}}N$
  - Each **integer key** is simply read from the vector and written into the staging buffer.
- $1.590\,K_v\,B\,N$
  - For a **VARCHAR key** we first follow its pointer, then hash the string (FNV-1a) into 64 bits.
  - Cost grows with the average byte length B and the number of such keys K_v.
- $2.855\,S_v\,N$
  - Every **value aggregate** (SUM/MIN/MAX/AVG) causes one extra column fetch.
- $155.5\,K_v\,G_v$
  - Each *distinct* string is inserted into a host-side `unordered_map`.

3. Host → Device copy ($0.63848\,(K + S_v)N$)

- All key columns and all payload columns needed by value-aggregates are transferred from host to device.

4. Composite-hash kernel ($0.01043\,K\,N$)

- A composite key is created from all the K-64 bit keys.

5. Insert kernel

$$0.08505\,N \;+\; 0.1561\,S_v\,N \;+\; 0.4859\,G \;+\; 0.2055\,K\,G$$

- $0.08505\,N$: one atomicAdd(&ht_cnt) for every row,
- $0.1561\,S_v\,N$: one atomic (ADD / MIN / MAX) for each aggregate payload column,
- $0.4859\,G$: successful atomicCAS + atomicExch to claim the slot,
- $0.2055\,K\,G$: copying the remaining K key columns into the hash-table row .

6. Compact kernel

$$0.02988\,N \;+\; 0.01191\,K\,G \;+\; 0.03549\,G \;+\; 0.01952\,S_v\,G$$

- Scans every slot ($0.02988\,N$) and copies survivors to dense arrays (remaining terms in G).

7. Device → Host copy ($2.4336\,(K + S_v + 1)G$)

- Keys, counts and payloads form $(K + S_v + 1)$ columns of 64-bit values are transferred from device to host.

8. VARCHAR rebuild

$$0.2204\,K_v\,G \;+\; 5.29\,K_v\,B\,G$$

- If $K_v > 0$ we map group hashes back to the original strings (first term) and, for materialisation, memcpy the bytes into DDB's string heap (second term, proportional to B).

9. GetData()

$$5.509 \times 10^{-5}\,(K_{\text{int}} + K_v B)\,G \;+\; 8.194 \times 10^{-4}\,S\,G$$

- CPU materialises a `DataChunk`: one scalar assignment per output element (keys, counts, aggregates).

4.4 Group by cost model is good fit

4.4.1 Checking for Overfitting

To ensure that our models are not overfitted to the training data, we performed the following diagnostic checks. Here, R^2 denotes the *coefficient of determination*, defined as

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2},$$

where y_i are the true values, $\hat{y}_i$ are the model predictions, and $\bar{y}$ is the mean of the observed values. Intuitively, R^2 measures the fraction of variance in the response variable explained by the model.

1. **Train–Test Gap.** We split the data into an 80% training set and a 20% hold-out test set. Let

   $$R^2_{\text{train}} \quad \text{and} \quad R^2_{\text{test}}$$

   denote the coefficients of determination on the training and test splits, respectively. A large difference $\left(R^2_{\text{train}} - R^2_{\text{test}}\right)$ would indicate that the model is memorizing noise in the training set rather than capturing generalizable patterns.

2. **k-Fold Cross-Validation ($k = 5$).** We performed fivefold cross-validation, refitting the model five times—each time holding out one-fifth of the data. We then computed the mean and standard deviation of

   $$R^2_{\text{cv}}.$$

   If $\overline{R^2_{\text{cv}}}$ is close to R^2_{train} and the standard deviation $\sigma_{R^2_{\text{cv}}}$ is small, this indicates good generalization and low variance.

3. **Learning Curves.** For a representative model, we plotted R^2_{train} and R^2_{val} against the fraction of training data used (from 10% up to 100%). If the model were overfitting, these two curves would diverge as the training fraction increases. Conversely, underfitting would manifest as both R^2_{train} and R^2_{val} remaining low.

4. **Residual Plots.** We examined scatterplots of residuals $r_i = y_i - \hat{y}_i$ versus fitted values $\hat{y}_i$. Overfitted models tend to exhibit erratic residual patterns (wild swings or clusters), while underfitted models show clear systematic structure (e.g., curvature or funnel shapes). A well-fitted model produces a tight, featureless cloud of points around zero.

Results. Across all submodels—host → device, retrieve, flatten, host_prep_probe, getdata, etc.— each diagnostic indicated strong generalization:

- *Train/Test and 5-Fold CV.* We observed that

$$R^2_{\text{train}} \quad \text{and} \quad R^2_{\text{cv}}$$

were nearly identical, so the train–test gap was negligible.

- *Learning Curves.* For each representative model, the curves for R^2_{train} and R^2_{val} remained nearly overlapping. For an example see the insert kernel's cost model's plot figure 4.6.

- *Residual Plots.* Residuals formed a tight, structureless cloud around zero, with no visible bends, funnels, or systematic deviations(see figure 4.6) .

These diagnostics represent the most widely used methods for detecting overfitting, and in the next section we also present experiments conducted with different datasets. Because all independent checks concur, we conclude that the fitted models are *not* overfitted.

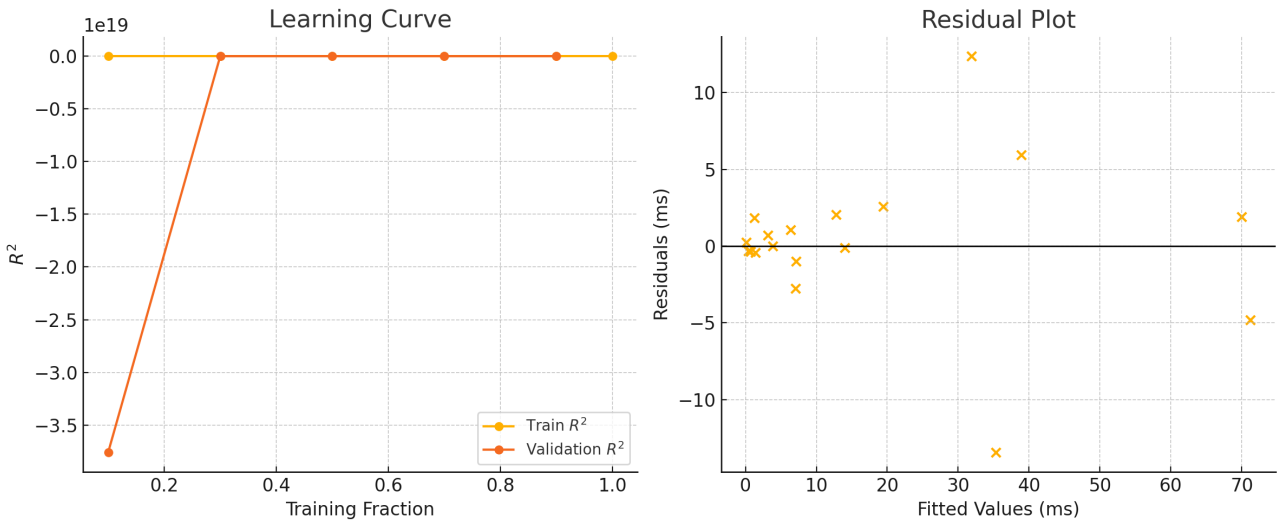

Figure 4.6: Learning curve and residual plot for insert kernel cost model

4.4.2 All stages predictions

Figure 4.7 shows the comparison between the real execution time and the predicted time for GroupBy Query 10 at scale factor 100. It demonstrates that all model predictions are accurate, and the mispredictions do not cancel each other out.

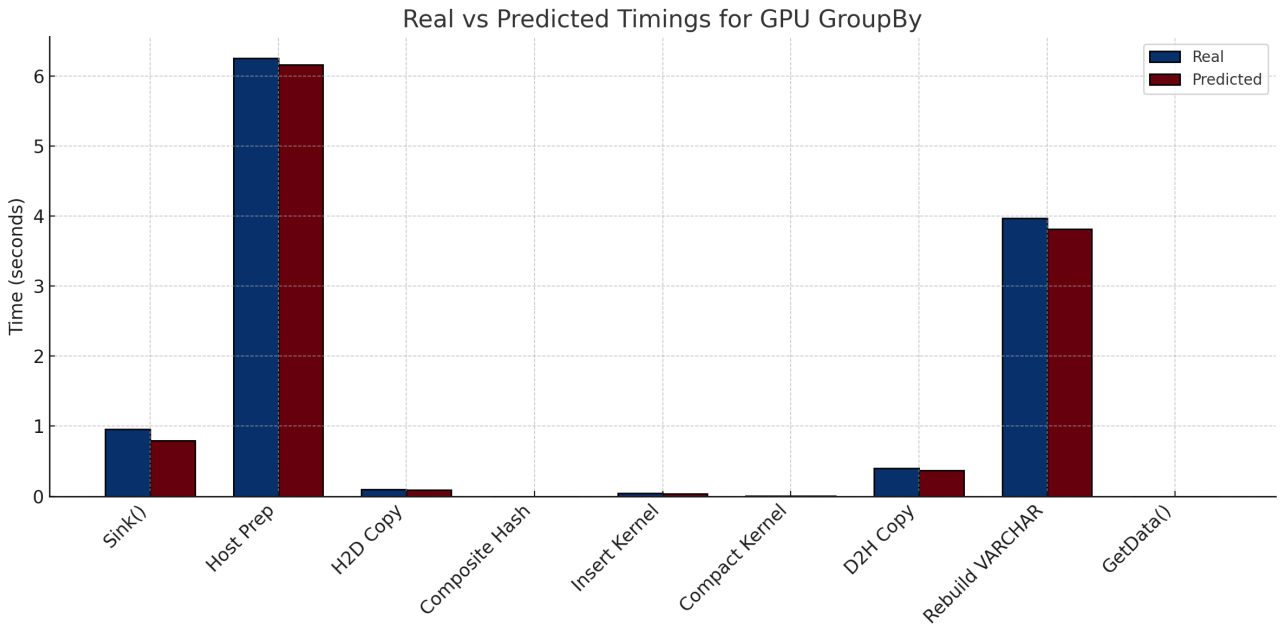

Figure 4.7: All stages predictions vs real time

4.4.3 All train and test queries prediction

Figure 4.8 displays a scatter plot comparing the predicted execution times with the actual times for all training and testing queries.

To assess the performance of our regression model for execution time prediction, we evaluate it using the following standard metrics:

Mean Absolute Error (MAE): MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. It is defined as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the total number of observations.

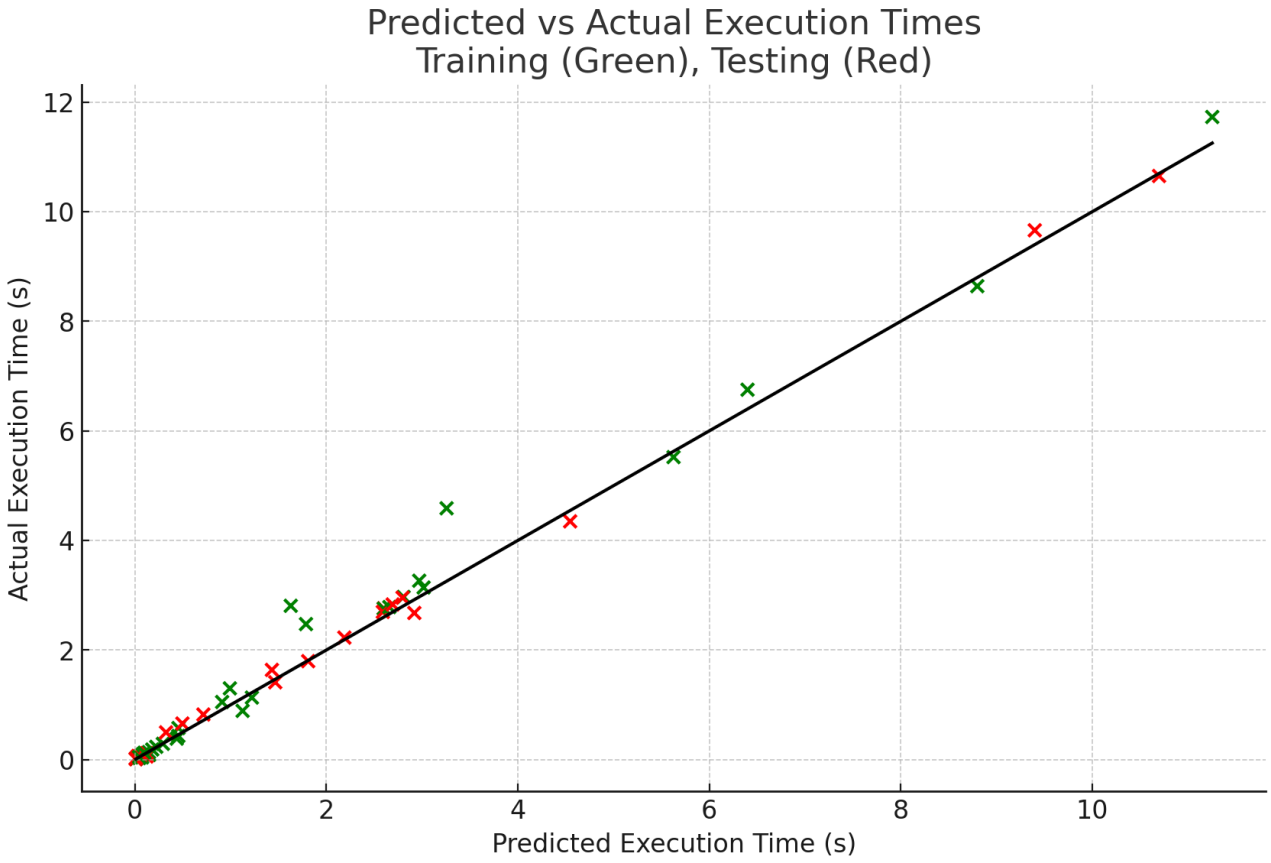

Figure 4.8: All test and training data split

Root Mean Squared Error (RMSE): RMSE is the square root of the average of squared differences between prediction and actual observation. It is more sensitive to large errors than MAE:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Evaluation Results

- **MAE: 0.153** seconds
- **RMSE: 0.292** seconds
- R^2**: 0.989**

These values indicate excellent predictive accuracy, with very small average and root-mean-square errors, and a near-perfect fit.

Chapter 5

Join

5.1 GPU-Join Algorithm

We developed our own GPU hash Join algorithm for this research.

Design overview.

1. **Host-side key normalisation.**

   - `BIGINT` keys are losslessly cast to 64-bit words.
   - `VARCHAR` keys are condensed into 64-bit scalars with the branch-free FNV-1a hash

   The original strings are cached in host memory so that any rare 64-bit hash collisions can be detected later.

2. **GPU build phase.** All scalarised keys from relation A are inserted, in a *single kernel launch*, into a multi-value hash table provided by the WARPCORE library [6, 7]. Warpcore internally re-hashes each 64-bit key with *MurmurHash3* to choose a bucket; duplicates are chained in a device-side value array. Using a target load factor $\lambda = 0.9$ the table capacity is $\lceil |A|/\lambda \rceil$ and requires only one contiguous device allocation.

3. **GPU probe phase.** Relation B is processed by two kernels:

   3.1. a *dry-run* call produces, for every probe row i, the interval $[\,\mathsf{begin}[i],\, \mathsf{end}[i])$ that delimits its matching build rows and returns the global match count M;

   3.2. a second call materialises the flat buffer $\mathsf{matchIds}[0{:}\,M-1]$ containing all build-side row identifiers.

4. **Host reconstruction and collision filter.** For each probe row i we iterate over its slice of matchIds, compare the stored key string of A with that of B to eliminate the vanishingly rare FNV collisions, and finally emit the joined tuple $\langle A[\mathsf{id}_A], B[\mathsf{id}_B]\rangle$..

Figure 5.1 shows how our GPU Join works.

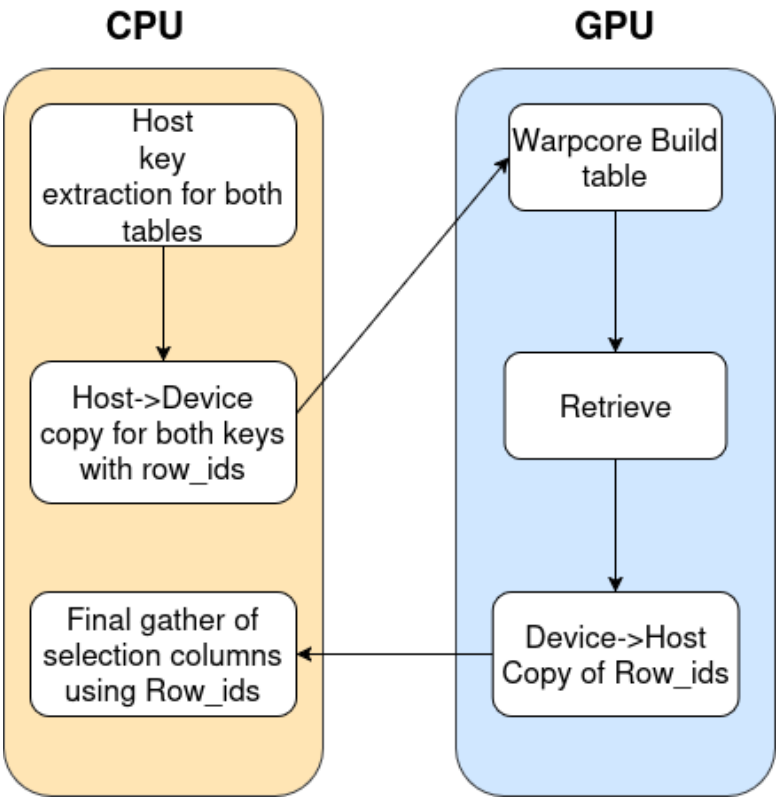

Figure 5.1: Working of GPU-join

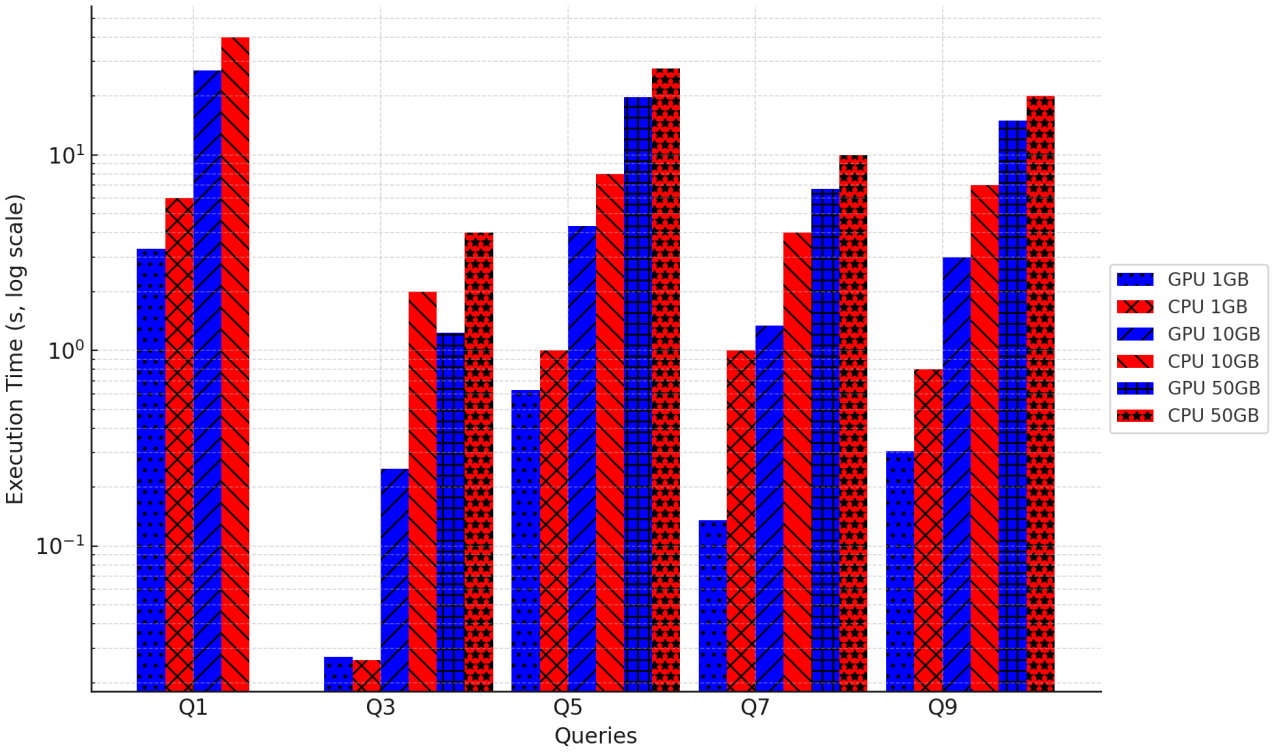

Figure 5.2: GPU vs CPU: Execution Time Comparison

To assess the performance of the GPU-based hash Join, we implemented a comparable single-core CPU hash Join algorithm. Both implementations use the Murmur hash function to ensure a fair comparison. The CPU-based Join algorithm employs the Murmur hash function, as it is also used in the GPU algorithm, ensuring a fair comparison between the two implementations. There are no discernible advantages to using the Murmur hash function in either the GPU or CPU implementations for the Join, as its use does not contribute to any significant performance improvements in this context[22, 4]. To minimize overhead in the CPU version, we omit the FNV-based VARCHAR-to-64-bit conversion used on the GPU side and instead rely on direct string comparisons.

We conducted performance comparisons of GPU-based and CPU-based hash-join algorithms using datasets from the TPC-H. As shown in figure 5.2, the GPU-based hash Join demonstrated performance improvements compared to CPU-based hash Join, and also, when the S.F is higher, we see better speed up.

5.2 Extending the DDB

5.2.1 Implementating New Join Operator

Similar to the GroupBy To integrate our Join operator, we defined a new physical operator named `Physical_GPU_JOIN` and an associated logical operator named `GPU_JOIN`. To ensure that DDB chooses our operator during execution, we initially modified the `plan_comparison_join.cpp` file to hardcode the selection of `Physical_GPU_JOIN`.

As an example, consider the following simple database schema and query:

```
TABLE users (user_id INTEGER, user_name VARCHAR);
TABLE orders (order_value INTEGER, user_id INTEGER);

SELECT *
FROM users u
JOIN orders o ON u.user_id = o.user_id;
```

For this query, DDB generates a physical query plan, which is depicted in Figure 5.3.

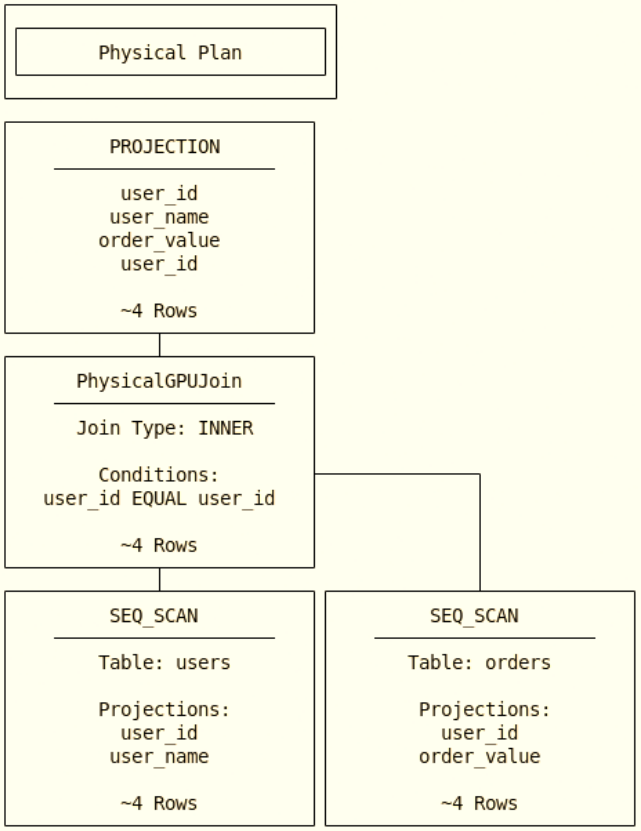

Figure 5.3: Example query plan of DDB

The `PhysicalGPUJoin` class extends the `PhysicalOperator` class in DDB. The overview main method for Join are explained below:

- `Sink`
  - Appends every incoming build-side `DataChunk` to a host-side buffer that stores one contiguous array per column.
- `Finalize`
  - *Host preparation*
    - Linearises the build keys into `h_keys` and their row numbers into `h_vals`.
    - BIGINT keys are copied verbatim; each `VARCHAR` key is transformed into a 64-bit FNV-1a hash.
  - *Host → Device transfer*
    - Allocates two device buffers with `cudaMalloc`.

    - Copies `h_keys` and `h_vals` to the GPU with back-to-back `cudaMemcpy` operations.
  - *GPU hash-table build*
    - Instantiates a `warpcore::MultiValueHashTable` sized so the load factor stays below 0.9.

- `Execute`
  - Buffers probe-side chunks in host memory until the pipeline signals that no more input will arrive.

- `FinalExecute`
  - *Probe host preparation*
    - Copies BIGINT probe keys directly; hashes `VARCHAR` probe keys with the same FNV-1a routine.
  - *Probe Host → Device transfer*
    - Allocates a device buffer for the probe key array and transfers it via `cudaMemcpy`.
  - *Two-pass retrieval*
    - **Pass 1:** calls `retrieve()` with `d_ids`=nullptr to compute per-probe match counts and the total number of matches.
    - **Pass 2:** allocates `d_ids` of the exact total length and calls `retrieve()` again to materialise build row IDs.
  - *Device → Host transfer*
    - Copies the begin/end pointers and the flattened `d_ids` array back to host memory.
  - *Match list flattening*
    - Expands each `[beg,end)` interval into parallel `probe_indices` and `build_indices` vectors.
  - *Chunk emission*
    - Slices the buffered probe chunk and build chunk with the flattened indices, emitting joined rows in batches of `STANDARD_VECTOR_SIZE`.

Overall workflow of the Join in DDB illustrated in figure 5.4:

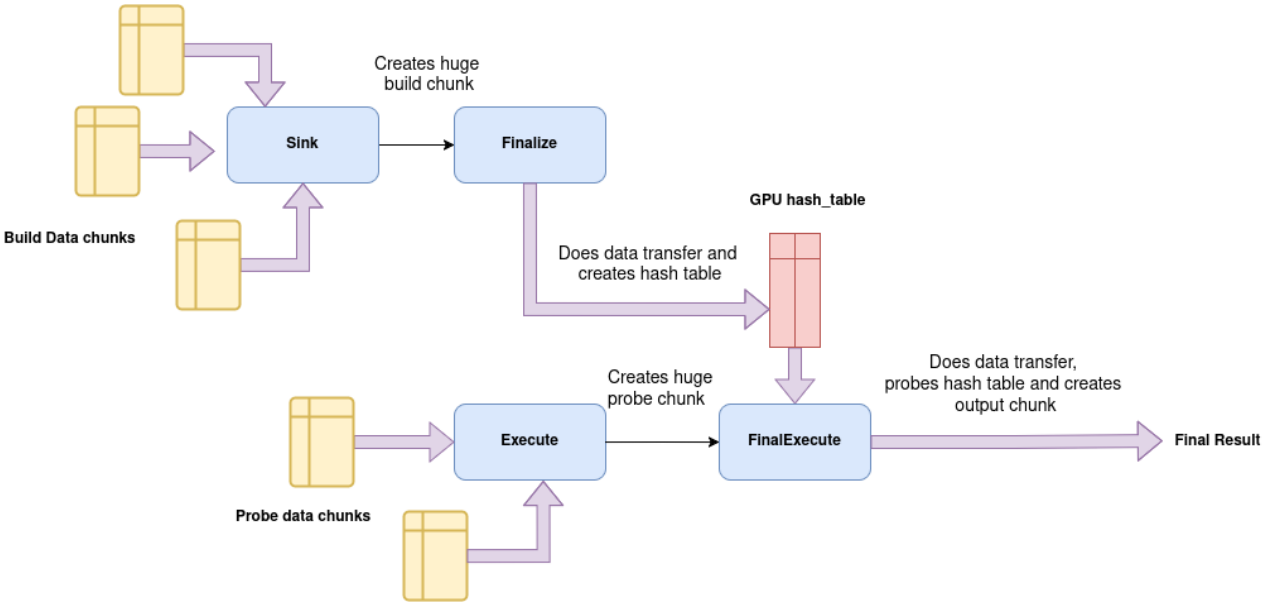

Figure 5.4: `Physical_GPU_JOIN` implementation

Among all the methods, DDB can schedule multiple threads only for the sink method. As discussed in Section 5, a single-threaded sink can outperform a multi-threaded sink + combine operation. In DDB's execution model, the FinalExecute step of a physical operator is deliberately run on a single thread once all of its parallel pipelines have finished. But in our use case, if we could schedule multiple threads, we could see significant performance benefits. However, we haven't done that yet because of the cascading effects on the correctness of other operators.

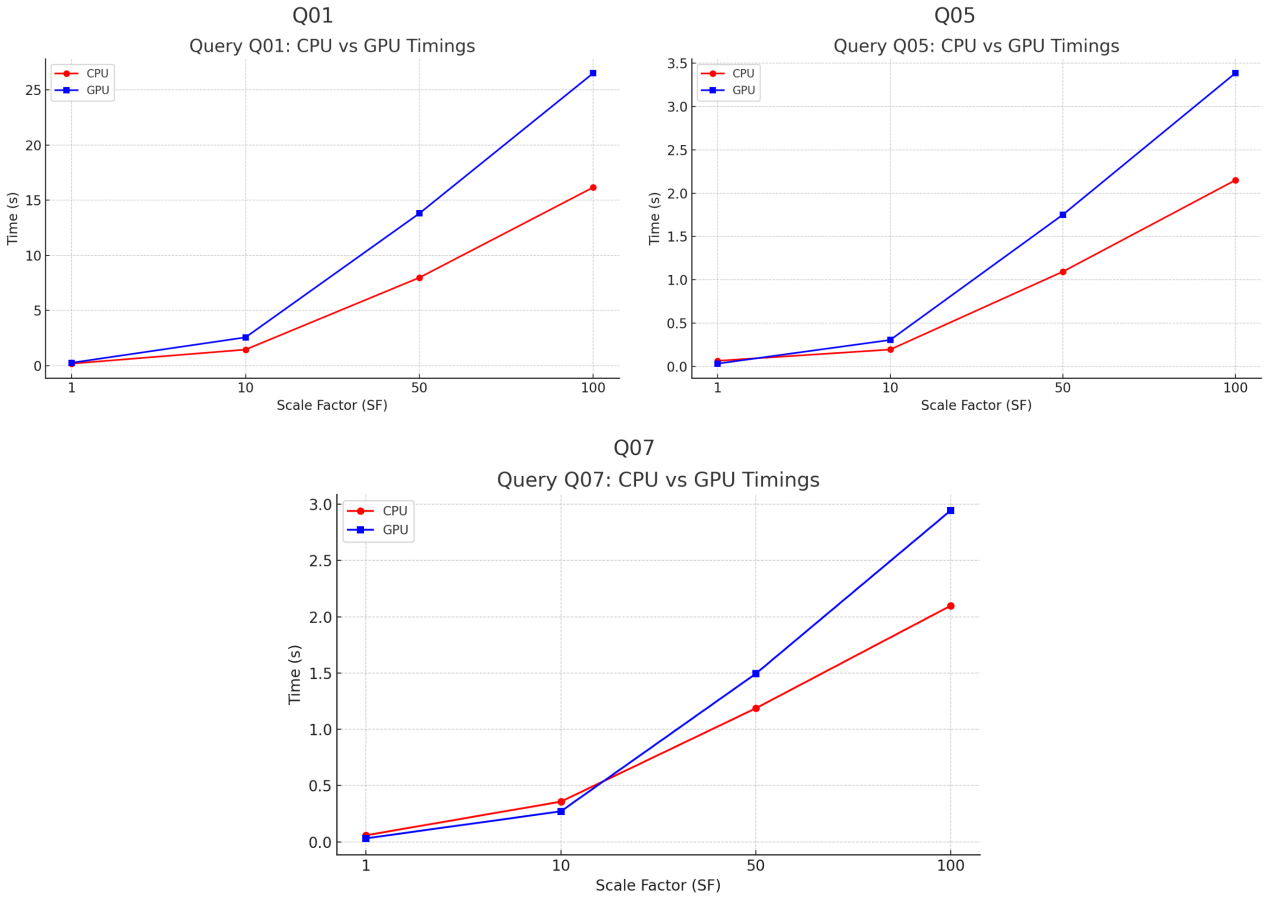

Figure 5.5: DDB's `Physical_Hash_Join` vs GPU Join

One can access the code at the following repository [18].

Figure 5.5 compares our GPU hash join with DuckDB's `Physical_Hash_Join` for mentioned queries from our join query suits. At all tested scale factors, DDB still outperforms the GPU version; nevertheless, enabling parallel `FinalExecute` instances on the GPU would narrow this gap considerably. Looking ahead to terabyte-scale workloads (1 TB and beyond), the per-thread load of DDB join becomes the bottleneck. We therefore anticipate that, with both larger data sizes and concurrent `FinalExecute` support, the GPU hash join will surpass DDB join and yield performance speed-ups.

5.3 Cost Model

We follow the similar separat multiple linear regression method as we used in GroupBy operator.

$$
\begin{aligned}
T_{\text{total}} \approx\ & \underbrace{0.7375\, N_A\, w}_{\text{Sink()}} + \underbrace{(4.0677 + 1.590\, B\, K)\ N_A}_{\text{build host-prep}} + \underbrace{1.3728\, N_A}_{\text{build H2D}} + \underbrace{(0.4698\, N_A + 3940\, N_A/\, G)}_{\text{build hash-table}} \\
& + \underbrace{0.7375\, N_B\, w'}_{\text{Execute()}} + \underbrace{(2.2644 + 1.590\, B\, K)\ N_B}_{\text{probe host-prep}} + \underbrace{0.6621\, N_B}_{\text{probe H2D}} + \underbrace{1.262\, N_B}_{\text{retrieve (pass 1)}} + \underbrace{0.1205\, N_J}_{\text{retrieve (pass 2)}} \\
& + \underbrace{1.262\, N_B}_{\text{D2H beg/end}} + \underbrace{0.67\, N_J}_{\text{D2H ids}} + \underbrace{6.9704\, N_J}_{\text{flatten}} + \underbrace{3.06\, N_J\ C\ + 9.78\, N_B}_{\text{chunk gather}}
\end{aligned}
\tag{5.1}
$$

5.3.1 Input symbol legend

Symbol	Meaning
N_A	Build-side cardinality
N_B	Probe-side cardinality fed to `Execute`
N_J	Number of matches produced ($N_J \leq N_A \cdot N_B$)
G	Total Distinct keys in build table
w	Bytes processed *per build row* inside `Sink`
w'	Bytes processed *per probe row* inside `Execute`
B	Average length (bytes) of the join key when it is VARCHAR
K	Indicator: 1 if the key is `VARCHAR`, 0 if it is `BIGINT`
C	Number of columns copied into the output chunk

So, the final total execution time is modeled as the sum of all sub stage models:

$$
\begin{aligned}
T_{\text{total}} \approx\ & T_{\text{sink}} + T^{(b)}_{\text{host-prep}} + T^{(b)}_{\text{H2D}} + T_{\text{build}} + T_{\text{exec}} + T^{(p)}_{\text{host-prep}} \\
& + T^{(p)}_{\text{H2D}} + T_{\text{retrieve}} + T_{\text{D2H}} + T_{\text{flatten}} + T_{\text{gather}}
\end{aligned}
$$

5.3.2 Cost equation

Equation 5.1 presents the complete cost model incorporating all execution stages. For this operator almost all the terms play significant importance with different selectivity so we have not dropped any terms. All the presented cost models are in ns.

The models below break down every execution stage, identifying the contributing terms, the factors that shape them, and their fitted coefficients.

1.Sink() ($0.7375 N_A w$)

- In the build-side `Sink()` phase, each input row is copied from columnar layout into a staging buffer.
- Cost scales with the raw byte volume: $N_A w$ bytes.

2.Build Host Preparation ($(4.0677 + 1.590BK)\, N_A$)

- $4.0677 N_A$: Fixed overhead per row, including row pointer navigation and per-key vector access.
- $1.590 BK N_A$: For each VARCHAR key, we compute a FNV-1a hash. The cost scales with average byte length B and number of keys K.

3.Build Host→Device ($1.3728 N_A$)

- All build keys and associated row IDs are transferred to GPU memory.
- This term models PCIe transfer cost based on number of rows.

4. Build Hash Table ($0.512\, N_A \;+\; 3.94{\times}10^3 \,\frac{N_A}{G}$ **ns**)

- Each build row is hashed, probed, and written into the WarpCore multi-value hash table.
- **Row cost:** $0.512\,\text{ns} \times N_A$ — one hash, one probe step, one write per row.
- **Collision cost:** $3.94{\times}10^3\,\text{ns} \times \frac{N_A}{G}$ — extra probe steps caused by rows that share the same key (average multiplicity N_A/G).

5.Execute() ($0.7375 N_B w'$)

- The probe-side executor copies input rows for processing.
- Scales with input byte size: $N_B w'$ bytes.

6.Probe Host Preparation ($(2.2644 + 1.590BK)\, N_B$)

- $2.2644 N_B$: Includes vector access and temporary buffer setup.
- $1.590 BK N_B$: VARCHAR probe keys are hashed before transfer.

7.Probe Host→Device ($0.6621 N_B$)

- All probe-side keys are copied to GPU memory.
- Fixed cost per row.

8. Retrieve ($1.262\,N_B\ +\ 0.1205\,N_J$ **ns)**

- **Pass 1 (offset scan)** — one warp per *probe* key counts how many build rows matched and writes the prefix-sum offsets. Work scales with the number of probe rows (N_B), hence the first term 1.262 ns $\times\ N_B$.
- **Pass 2 (materialise list)** — the kernel walks the table again and stores the actual row IDs (or tuple IDs) for every match. The inner loop is executed once per *join match* (N_J), so the second term 0.1205 ns $\times\ N_J$ captures that pure memory-copy cost.

9. D2H beg/end ($1.34\,N_B$ **ns)**

- Two `cudaMemcpy` calls copy the *begin* and *end* offset vectors (N_B 64-bit indices each) from GPU to host.
- Each probe row therefore moves 16 bytes, giving the measured slope 1.34 ns $\times\ N_B$ (12 GB/s sustained PCIe bandwidth).

10. D2H ids ($0.67\,N_J$ **ns)**

- A single `cudaMemcpy` copies the contiguous list of matching build-row IDs (N_J 64-bit values) to the host.
- Only 8 bytes per match are transferred, so the cost is half of the offset copy: 0.67 ns $\times\ N_J$.

11.Flatten ($6.9704N_J$)

- The host flattens variable-length match lists into contiguous arrays of row index pairs.

12. Chunk Gather ($(3.06\,N_J\,C\ +\ 9.78\,N_B)$ ns)

- Builds the final `DataChunk` by filling two `SelectionVector`s (probe IDs, build IDs) and calling `Vector::Slice()` once per output column.
- Cost has two linear parts:
  - $3.06\,\text{ns} \times N_J\,C$ – one probe/index write and one slice-header update per output *cell*.
  - $9.78\,\text{ns} \times N_B$ – cache/TLB misses that grow with the size of the probe buffer (N_B).
- No payload copying occurs; dependence on data type or width is negligible — only the number of cells and the total probe rows determine the latency.

5.4 Join cost model is a good fit

We employed the same techniques—such as plotting residuals and learning curves etc—to verify that the models are not overfitted, similar to the approach used for GroupBy. The results were similarly positive.

5.4.1 All stages prediction

Figure 5.6 presents a comparison between the real execution time and the predicted time for Join Query 13 at scale factor 100. The figure indicates that all model predictions are accurate, and any mispredictions do not offset one another.

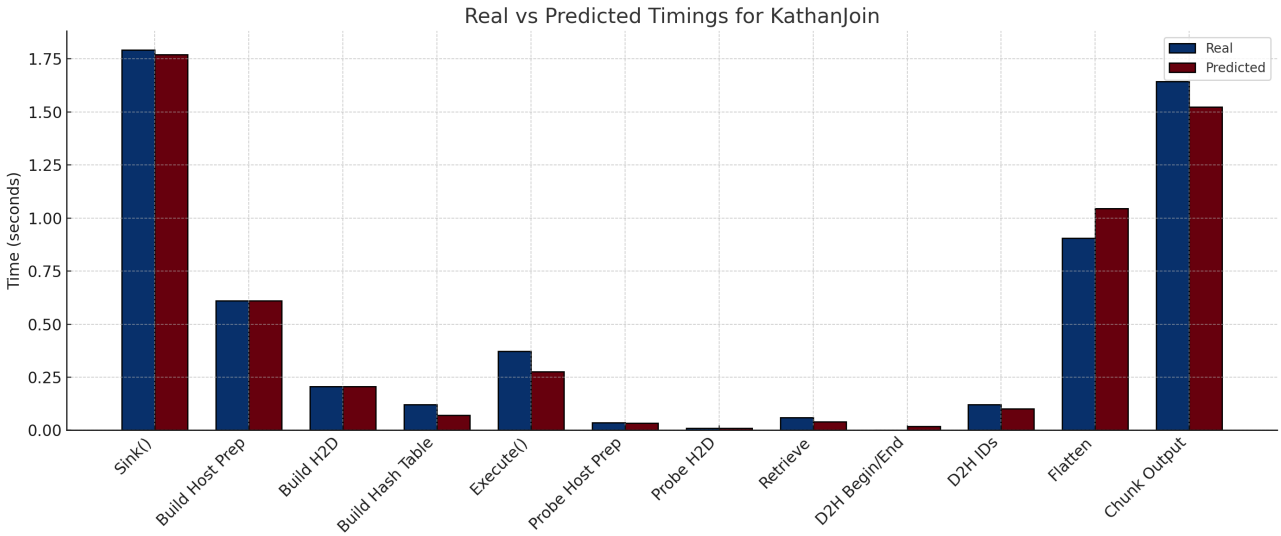

Figure 5.6: All stages predictions vs real time

5.4.2 All train and test queries

Figure 5.7 displays a scatter plot comparing the predicted execution times with the actual times for all training and testing queries.

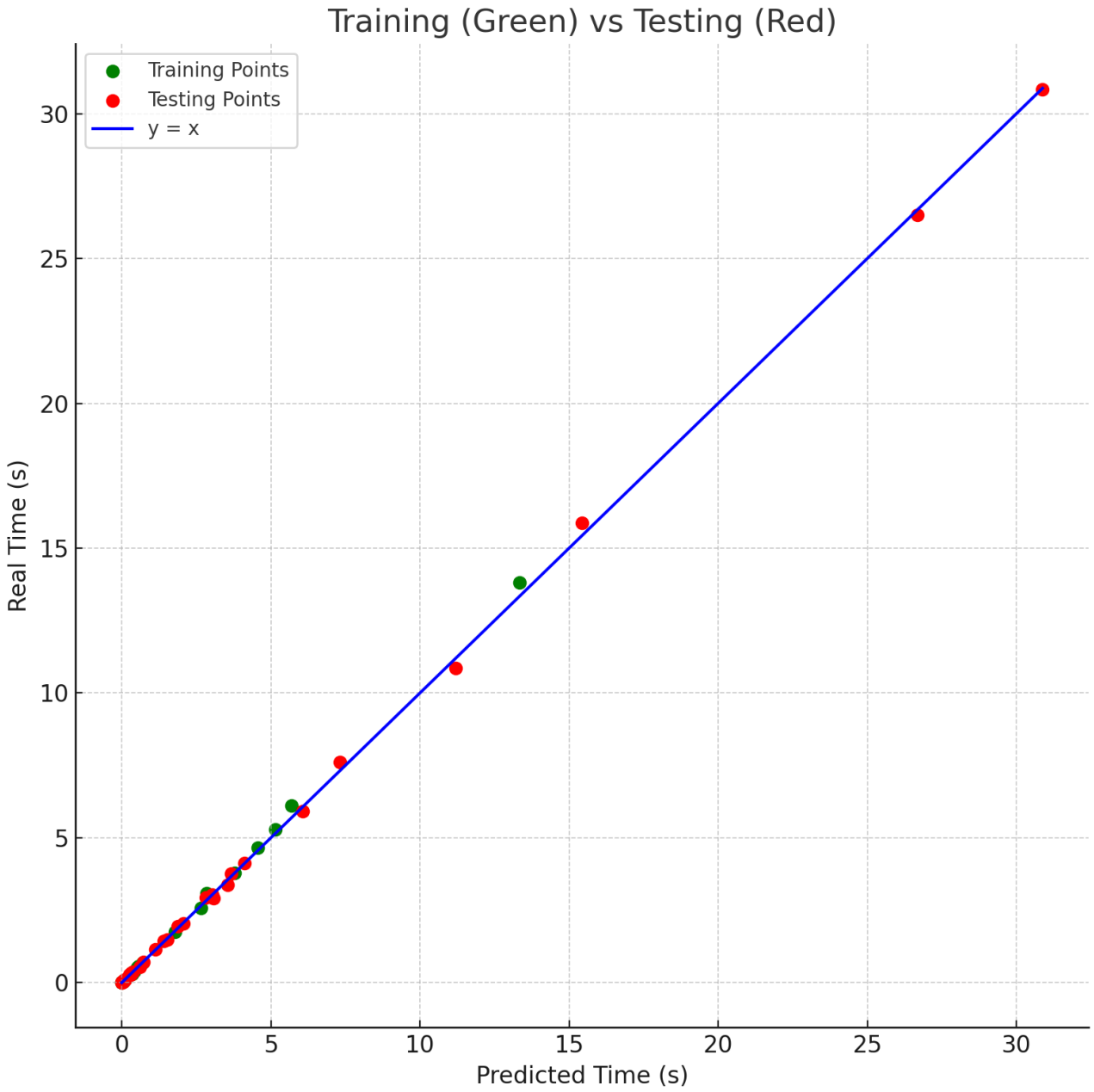

Figure 5.7: All train and test queries GPU Join

To evaluate the accuracy of our join cost model, we compared the predicted execution times against the actual execution times across a diverse set of queries. The following standard regression metrics were computed to assess model performance:

Evaluation Results

- **MAE**: 0.0654 seconds

- **RMSE**: 0.1286 seconds
- R^2: 0.9995 seconds

These results indicate that the join cost model is highly accurate, with very low absolute and squared prediction errors. The R^2 score close to 1.0 confirms an excellent fit between predicted and actual execution times.

Chapter 6

Conclusion and future work

This work addresses the performance bottlenecks of GroupBy and Join operations by developing a GPU-accelerated Join and GroupBy operator. To guide execution in resource-constrained environments, we construct cost models that capture stage-wise execution characteristics. The proposed operators are

integrated into the DDB, demonstrating the potential of GPU acceleration for improving query performance within modern analytical workloads.

There are several ways to extend this work:

a) We currently rely on oracle knowledge of intermediate and final cardinalities. Next, we will plug a realistic *cardinality-estimation (CE)* module in front of the GPU cost model and study how CE error affects runtime prediction. PostgreSQL's MCV/histogram and extended-statistics pipeline serves as a template; re-using its selectivity formulas lets us derive the same inputs $(N, G, B, \dots)$ for our model without perfect knowledge [26].

b) DDB today chooses CPU Join and GroupBy plans via heuristics rather than cost formulas. A natural next step is to derive explicit cost models for the main CPU variants—hash join, perfect hash join, sort–merge join, nested-loop join, hash aggregation, perfect hash aggregation, and sort-based aggregation—and then benchmark those against our existing GPU cost model under the same statistics and workloads.

c) Both our GPU Join and GroupBy kernels assume the entire input fits in device memory. One can introduce an out-of-core, two-pass *radix partitioning* scheme: hash each tuple on the low-order r bits of its join/group key to create 2^r buckets that stream through GPU memory; recursively refine any bucket that is still too large; then (i) run a per-bucket hash

join, or (ii) build a per-bucket aggregation table followed by a host-side merge. Radix partitioning has proven effective for joins [16] and for hash-based aggregations [1].

d) Our present cost model is calibrated only for a single RTX 4090 workstation. We can re-fit the stage coefficients on a variety of GPUs (e.g., RTX 3080, A100, MI210) and host CPUs, then embed a lightweight *auto-tuner* that runs a few micro-benchmarks at start-up to rescale the coefficients—borrowing ideas from GPU kernel auto-tuners such as Kernel Tuner [29] and CLTune [25].

Bibliography

[1] Shaoyu Chen, Bingsheng He, and Xuan Zhou. Accelerating aggregate operators on gpus. In *Proc. VLDB*, pages 1018–1029, 2014. https://doi.org/10.14778/2733085.2733091. 39

[2] The Transaction Processing Performance Council. TPC Benchmark™H (tpc-h). In *Proc. TPC Benchmark H (Decision Support)*, 2005. https://www.tpc.org/tpch/. 1

[3] The Transaction Processing Performance Council. TPC Benchmark™DS (tpc-ds). In *Proc. TPC Benchmark DS (Decision Support)*, 2006. https://www.tpc.org/tpcds/. 1

[4] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. Practical hash functions for similarity estimation and dimensionality reduction. *arXiv preprint arXiv:1711.08797*, 2017. https://arxiv.org/abs/1711.08797. 26

[5] David J. DeWitt and Robert H. Gerber. Multiprocessor hash-based join algorithms. In *Proc. VLDB*, pages 151–164, 1985. http://www.vldb.org/conf/1985/P151.PDF. 2

[6] Daniel Jünger et al. Warpdrive: Massively parallel hashing on multi-gpu nodes. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 441–450, 2018. doi: 10.1109/IPDPS.2018.00054. https://doi.org/10.1109/IPDPS.2018.00054. 24

[7] Daniel Jünger et al. Warpcore: A library for fast hash tables on gpus. In *Proc. IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 11–20, 2020. doi: 10.1109/HiPC50609.2020.00015. https://doi.org/10.1109/HiPC50609.2020.00015. 24

[8] Harish Doraiswamy et al. A case for graphics-driven query processing. *Proc. VLDB Endow.*, 16(10):2499–2511, 2023. doi: 10.14778/3603581.3603590. https://www.vldb.org/pvldb/vol16/p2499-doraiswamy.pdf. 2, 6

[9] Panagiotis Sioulas et al. Hardware-conscious hash-joins on gpus. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 698–709, 2019. doi: 10.1109/ICDE.2019.00068. https://doi.org/10.1109/ICDE.2019.00068. 6, 8

[10] Robert J. Halstead et al. Fpga-based multithreading for in-memory hash joins. In *Proc. CIDR*, 2015. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper12.pdf. 2

[11] Viktor Rosenfeld et al. Performance analysis and automatic tuning of hash aggregation on gpus. In *Proc. DaMoN*, pages 8:1–8:11, 2019. https://doi.org/10.1145/3329785.3329922. 6

[12] Glenn Fowler, Landon Curt Noll, and Kiem Phong Vo. Fowler noll vo hash function. *Wikipedia*, 2025. https://en.wikipedia.org/wiki/Fowler_Noll_Vo_hash_function. 9

[13] Vishal Goel. Think global, model local: A fine-grained approach to query cost estimation with learned parameters, 2020. https://dsl.cds.iisc.ac.in/publications/thesis/vishal.pdf. 4

[14] Goetz Graefe. Volcano—an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. https://doi.org/10.1109/69.273032. 5

[15] Jayant R. Haritsa. Robust query processing: Mission possible. *Proc. VLDB Endow.*, 13 (12):3425–3428, 2020. doi: 10.14778/3415478.3415561. http://www.vldb.org/pvldb/vol13/p3425-haritsa.pdf. 4

[16] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Peter V. Sander. Relational joins on graphics processors. In *Proc. SIGMOD Int'l Conf. on Management of Data*, pages 511–522, 2008. https://doi.org/10.1145/1376616.1376669. 39

[17] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proc. 15th Intl. Workshop on Data Management on New Hardware (DaMoN)*, pages 1–3, 2019. doi: 10.1145/3329785.3329932. https://doi.org/10.1145/3329785.3329932. 2

[18] Kathan. Duckdb gpu join, 2025. https://github.com/kathan3/duckdb_kathan_join/. 14, 31

[19] Kathan. Test queries. 2025. https://github.com/kathan3/Training-and-testing-queries/. 8

[20] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6(1):86–101, 2021. doi: 10.1007/s41019-020-00149-7. http://dx.doi.org/10.1007/s41019-020-00149-7. 4

[21] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6(1):86–101, 2021. doi: 10.1007/s41019-020-00149-7. https://link.springer.com/article/10.1007/s41019-020-00149-7. 4

[22] Hua Luan and Lei Chang. An experimental study of group-by and aggregation on cpu–gpu processors. *Journal of Engineering and Applied Science*, 69(1):54, 2022. doi: 10.1186/s44147-022-00108-1. https://jeas.springeropen.com/articles/10.1186/s44147-022-00108-1. 26

[23] Microsoft. Program for TPC-H data generation with skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430, July 2024. File: TPCDSkew.zip (246 KB); Date Published: July 15, 2024. 8

[24] Hannes Mühleisen and Mark Raasveldt. Duckdb: An embeddable analytical database, 2019. https://duckdb.org/. 1

[25] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *Proc. of the International Conference on Supercomputing (ICS)*, 2015. https://arxiv.org/abs/1703.06503. 39

[26] PostgreSQL Global Development Group. Planner statistics and cost estimation in postgresql. https://www.postgresql.org/docs/current/planner-stats.html, 2025. 38

[27] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proc. of the 2019 International Conference on Management of Data (SIGMOD)*, pages 1981–1984, 2019. doi: 10.1145/3299869.3320212. https://doi.org/10.1145/3299869.3320212. 1, 5

[28] Damien Spaolacci. murmur3: Go implementation of murmurhash3, 2014. https://github.com/spaolacci/murmur3. 9

[29] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019. https://doi.org/10.1016/j.future.2018.08.004. 39

[30] Zhe Wang, Yao Shen, and Zhou Lei. Ega: An efficient gpu accelerated groupby aggregation algorithm. *Applied Sciences*, 15(7):3693, 2025. doi: 10.3390/app15073693. https://www.mdpi.com/2076-3417/15/7/3693. 2, 5

[31] Wikipedia contributors. Utf-8 — wikipedia, the free encyclopedia, 2025. https://en.wikipedia.org/wiki/UTF-8. 6