Master Thesis

Optimizing Database Joins

Cost Models and Benchmarking for CPU and GPU Systems

Marko Matušovič

DELFT UNIVERSITY OF TECHNOLOGY

TU Delft

DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Optimizing Database Joins: Cost Models and Benchmarking for CPU and GPU Systems

Author:
Marko MATUŠOVIČ

Supervisors:
Wenbo SUN
Rihan HAI
Christoph LOFI

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

25th April 2024

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science

Software Technology

Master of Science

Optimizing Database Joins:
Cost Models and Benchmarking
for CPU and GPU Systems

by Marko MATUŠOVIČ

Optimizing SQL query execution through effective cost models is a critical challenge in database management systems (DBMS). This thesis introduces a modular benchmarking system for cost models, with a pluggable architecture for both cost models and execution engines, enabling comprehensive benchmarking across various scenarios. Accompanied by a detailed methodology for the empirical measurement of cost model performance across different execution engines, a standardized approach is established, ensuring consistent and reproducible benchmarks. Furthermore, as a showcase of the developed system's capabilities, an analysis of key features influencing join-order optimization performance in both CPU and GPU systems is presented. This analysis demonstrates the system's utility in developing more effective cost models and optimizers. These contributions pave the way for future research in DBMS optimization, providing a research platform for the accelerated development of new cost models.

keywords: Database Management Systems, SQL, Query Optimization, Cost Models, Benchmarking, Research Platform, Performance Analysis

Acknowledgements

I express my sincere gratitude to my dedicated supervisors Professor Rihan Hai and Wenbo Sun for their invaluable guidance and support throughout the development of this thesis. Their expertise and encouragement have been instrumental in shaping the research.

I am particularly grateful to Professor Christoph Lofi, my main advisor, whose insightful perspectives have illuminated the path forward and significantly contributed to the direction of this study.

I extend heartfelt thanks to my friends for the moments of laughter and camaraderie, providing a necessary balance during the intense phases of academic work.

My deepest appreciation goes to my family for their support and the opportunity to pursue higher education. Their encouragement and belief in my abilities have been the foundation of my academic journey.

Part of the cover page uses a modified image generated by Microsoft Bing Designer.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Methodology & Contributions . . . 3
1.4 Thesis Overview . . . 4

2 Background 5
2.1 Query Optimization . . . 5
2.1.1 Heuristic Optimization . . . 5
2.1.2 Cost-Based Optimization . . . 6
2.1.3 Additional Optimization Techniques . . . 7
2.2 Cost Model . . . 7
2.2.1 Analytical Cost Models . . . 7
2.2.2 Learned Cost Models . . . 8
2.2.3 Features & Metrics . . . 9
2.2.4 Cardinality Estimation . . . 11
2.3 Query Processing . . . 12
2.3.1 Execution Engines . . . 12
2.3.2 Data Frames and Processing . . . 12
2.4 CPU and GPU architectural differences . . . 13

3 System Design 15
3.1 Requirements . . . 15
3.2 Architecture Overview . . . 16
3.3 Use cases . . . 20
3.4 Component Details . . . 22
3.4.1 Schema Parsing . . . 23
3.4.2 Query Parsing . . . 23
3.4.3 Mapping of Operations . . . 25
3.4.4 DataFrame Interface . . . 25
3.4.5 DataFrame Framework Placeholder . . . 26
3.4.6 Profiling . . . 26
3.4.7 CSV Loaders . . . 28
3.4.8 Logical Optimization . . . 28
3.4.9 Join Order Injection . . . 29
3.4.10 Cardinality Estimation . . . 30
3.4.11 Feature Collection . . . 30
3.4.12 Cost Model Placeholder . . . 32

3.4.13 Plan Enumeration & Search . . . 32
3.5 Implementation . . . 33

4 Methodology for Benchmarking **35**
4.1 Benchmarking Metrics . . . 35
4.2 Data Collection & Processing . . . 36
4.3 Datasets . . . 37
4.4 Query Selection . . . 37

5 Experiment & Analysis **39**
5.1 Motivation . . . 39
5.2 Methodology . . . 39
5.3 Experiment Setup . . . 40
5.4 Evaluation Plan . . . 42
5.5 Results . . . 43
5.5.1 Collecting Data . . . 43
5.5.2 Benchmarking Cost Models . . . 46
5.5.3 Comparing the Scores . . . 51
5.6 Analysis . . . 53
5.6.1 Significant Features on CPU . . . 53
5.6.2 Significant Features on GPU . . . 53
5.6.3 Proposed Explanation . . . 54

6 System Evaluation **55**
6.1 Requirements Evaluation . . . 55
6.2 Evaluating Use Cases . . . 57
6.3 Limitations . . . 58
6.3.1 Designed system & Benchamrking methodology . . . 58
6.3.2 Experiment . . . 59
6.4 Future Work . . . 59

7 Conclusion **61**
7.1 Research Contributions . . . 61
7.2 Addressing the Research Questions . . . 62
7.3 Limitations & Future Work . . . 62
7.4 Concluding Remarks . . . 62

Bibliography **63**

A Selected Queries **67**

B Detailed Correlation Results **73**

List of Figures

2.1 Taxonomy of Cardinality Estimation Techniques from [15] 12
2.2 Comparison between a CPU (AMD EPYC 7702P) and a GPU (NVIDIA ampere A100). [25] 14

3.1 General Components and Flow of SQL DBMS 17
3.2 Detailed Overview of Architecture of the Benchmarking System 18
3.3 Component Highlight of Use Case 1 21
3.4 Flow-chart of Use Case 1 21
3.5 Component Highlight of Use Case 2 21
3.6 Flow-chart of Use Case 2 21
3.7 Component Highlight of Use Case 3 22
3.8 Flow-chart of Use Case 3 22
3.9 SQL Parsing Extract of the System Architecture 23
3.10 Execution Engine Extract of the System Architecture 24
3.11 Sotrage Management Extract of the System Architecture 27
3.12 Query Optimization Extract of the System Architecture 28

5.1 Stages of Query JOB 2A 44
5.2 Stages of Query JOB 12A 44
5.3 Stages of Query SSB 43 45
5.4 Stages of Query TPC-DS 50 45
5.5 Various Join-orders for Query JOB 2A (CPU) 47
5.6 Various Join-orders for Query JOB 2A (GPU) 47
5.7 Various Join-orders for Query TPC-DS 50 (CPU) 48
5.8 Various Join-orders for Query TPC-DS 50 (GPU) 48
5.9 Various Join-orders for Query JOB 2A (CPU) + Cost Model 49
5.10 Various Join-orders for Query JOB 2A (GPU) + Cost Model 49
5.11 Various Join-orders for Query TPC-DS 50 (CPU) + Cost Model 50
5.12 Various Join-orders for Query TPC-DS 50 (GPU) + Cost Model 50
5.13 Correlation of Various Cost Models with CPU and GPU Execution Engines 51
5.14 Correlation Gain of Various Cost Models with CPU and GPU Execution Engines (Bar Chart View) 52
5.15 Correlation Gain of Various Cost Models with CPU and GPU Execution Engines (Table View) 52

B.1 Feature Comparison, with Detailed Correlation for Each Query, for CPU 74
B.2 Feature Comparison, with Detailed Correlation for Each Query, for GPU 75

List of Abbreviations

Abbreviation	Definition
CE	**C**ardinality **E**stimation
CM	**C**ost **M**odel
CNF	**C**onjunctive **N**ormal **F**orm
CPU	**C**entral **P**rocessing **U**nit
DB	**D**ata**B**ase
DBMS	**D**ata**B**ase **M**anagement **S**ystem
DF	**D**ata **F**rame
GPU	**G**raphical **P**rocessing **U**nit
IMDb	**I**nternational **M**ovie **D**ata**b**ase
JO	**J**oin **O**rder
JOB	**J**oin **O**rder **B**enchmark
JOO	**J**oin **O**rder **O**ptimization
ML	**M**achine **L**earning
RQ	**R**esearch **Q**uestion
SE	**S**electivity **E**stimation
SIMT	**S**ingle **I**nstruction **M**ultiple **T**hreads
SM	**S**treaming **M**ultiprocessors
SSB	**S**tar **S**chema **B**enchmark
SQL	**S**tructured **Q**uery **L**anguage
QO	**Q**uery **O**ptimization

Chapter 1

Introduction

This chapter serves as the foundation for the thesis, introducing the background and setting the stage for the exploration of join-order optimization in relational databases. It then outlines the problem statement, and defines the research questions, together with methodology and contributions. The chapter concludes with an overview of this thesis.

1.1 Motivation

The advancement of Database Management Systems (DBMS) is a testament to the extensive research and development efforts dedicated to solving the complex challenges of data storage, retrieval, and processing. This area has been well-studied, leading to the creation of numerous DBMS solutions, each designed to meet specific requirements and performance criteria. Relational DBMS were among the first to be developed and remain the most widely used systems due to their robustness, flexibility, and well-understood theoretical underpinnings.

A fundamental strategy to make database management systems more efficient is query optimization. Research into this field has been ongoing since as early as 1975 [29]. Each DBMS employs its unique query optimizer, tailored to leverage the system's specific characteristics and capabilities. The role of the query optimizer is to determine the most efficient way to execute a given query, a task that involves a deep understanding of the data structure, query syntax, and the underlying hardware.

One of the most crucial [17, 22] and persistent [32] challenges in database query optimization is join-order optimization. This optimization process involves determining the most efficient order of joining tables in a query. A well-optimized join order can significantly minimize execution time, resulting in faster query execution and more efficient resource utilization.

Central to join-order optimization is a cost model. Cost models are designed to predict the total cost of executing a specific execution plan and thus a specific join order. The estimated cost and thus the optimal join order may change depending on the data distribution or other factors such as physical operators, making it difficult to determine which join order will produce the best query plan in advance [22].

A fundamental aspect of a cost model is its logical connection to the execution engine, which is responsible for carrying out the operations specified in a query. These execution engines are diverse, reflecting the wide range of DBMS architectures.

Alternative research efforts in the past decade have focused on exploiting the capabilities of general-purpose GPUs to enhance database performance. Some studies

have demonstrated significant speed-ups, ranging from 20 to 70 times faster than traditional CPU-based database architectures [3, 26]. Additionally, in some situations they lead to a reduction of I/O costs by up to 68% [5]. These findings underscore the potential benefits of incorporating GPUs into database systems, opening avenues for further investigations into their viability and advantages. However, due to the close link between the cost model and the underlying execution engine, these new databases based on GPUs also require specialized cost models.

Many DBMS perform join-order optimization, and their cost models are often integrated within these systems, and highly coupled with the execution engines. This integration makes the cost models difficult to access, understand, and evaluate. The challenge lies in the differences between cost models and their applicability in different hardware and software environments. Hardware differences encompass the underlying physical architecture, such as CPUs versus GPUs, while software variations refer to the execution engines' implementation of algorithms. These distinctions necessitate a more nuanced understanding and analysis to effectively apply and evaluate cost models in diverse settings.

1.2 Problem Statement

To better formulate the issue tackled in this thesis, let's revisit and refine the previously introduced concepts:

- There are many DBMS, with different cost models and execution engines.
- Cost models are highly coupled to execution engines.
- Cost models are difficult to design and lack reusability.
- Developers want to have a clear way of evaluating existing and new cost models in their systems.

This situation presents a distinctive challenge. There is an abundance of different cost models and execution engines. However, due to the tight coupling between them, it is difficult to reuse any of the components and conduct benchmarks, or research into either. The complexity of these systems and their interdependencies make it challenging to assess the performance of cost models across different execution engines effectively.

RQ1: How can a system be designed, to measure the performance of any cost model across different execution engines?

Answering this question is particularly interesting for two main scenarios.

Firstly, in the scenario where multiple cost models are evaluated on a single execution engine, this approach allows us to identify effective cost models from the existing ones. By evaluating various cost models in a controlled environment, models that provide the best performance can be determined. Those can be either used directly or can be used as a basis for the design of the new cost model.

Secondly, evaluating a single cost model across multiple execution engines is advantageous too. This approach allows for the evaluation of the universality of cost models, revealing their performance and adaptability across different systems. This is crucial for understanding how a cost model performs in a specific environment.

The core focus of this thesis revolves around RQ1, which guides the development and utilization of a system designed to measure the performance of various cost models across different execution engines. However, a further subquestion is required.

RQ1.1: How can the performance of a cost model be measured empirically?

Addressing this question requires the development of a comprehensive methodology, essential for providing a structured approach to utilize the system proposed in RQ1. This methodology is vital as it ensures consistency and objectivity in evaluating cost models across different execution engines. With the methodology established, it sets the stage for its practical application, demonstrating the system's capabilities. This progression naturally leads to RQ1.2, which focuses on a specific use case of the system.

RQ1.2: What are the key features that impact the performance of cost models in CPU and GPU systems?

The exploration of this question serves as the foundation for a subsequent experiment designed to showcase the potential of the system. Understanding input features is essential, as they constitute the foundational building blocks of every cost model, influencing their performance and effectiveness. By identifying and analyzing the key features that influence cost model performance in CPU and GPU systems, this exploration provides valuable insights into the adaptability and effectiveness of cost models across different hardware architectures.

1.3 Methodology & Contributions

To answer the defined research questions, the following steps will be taken:

1. Examine previous work and select a set of compatible engines for the benchmarking system. The selected engines must at least cover execution on CPU and GPU.
2. Develop a benchmarking environment with interchangeable engines. This system must be able to execute an SQL query with any join order.
3. Describe a methodology for a comprehensive benchmarking of cost models in the developed system.
4. Design and carry out an experiment to reveal the key features of cost models in CPU and GPU systems.

This section outlines the key contributions of this thesis:

- **Design and development of a benchmarking system for cost models:** This modular system has a pluggable slot for a cost model and a slot for an execution engine. It is capable of executing SQL queries, performing optimization, and profiling the execution engine. Its modular architecture allows for a comprehensive evaluation of cost models across different scenarios, providing insights into the performance characteristics of cost models and execution engines.
- **Methodology of evaluating the performance of cost model:** This methodology will detail the steps and criteria for empirically measuring the performance of various cost models across different execution engines. Establishing

a clear and standardized approach ensures that the system's capabilities are fully utilized, enabling a consistent and reproducible evaluation of cost models.

- **Analysis of key features:** This contribution involves an examination of the key features that influence the performance of join-order optimization in CPU and GPU systems. The analysis aims to identify the most critical factors affecting the efficiency of cost models and the execution of join orders. The findings from this analysis will contribute to the development of more effective cost models and optimizers for both CPU and GPU systems.

1.4 Thesis Overview

This section provides a structured overview of the thesis's content, outlining the chapters and their respective contributions to the research. It serves as a roadmap for the reader, facilitating a comprehensive understanding of the thesis's scope and the methodological approach taken to address the research questions.

Chapter 2: **Background.** Discusses the theoretical foundations and previous work. It provides an overview of cost-based query optimization, and cost models in the context of database management systems. Also, the differences between CPU and GPU architectures are outlined.

Chapter 3: **System Design.** Details the design and development of the cost model benchmarking system, including system requirements, its architecture, modular components, and how it integrates with different execution engines.

Chapter 4: **Methodology for Benchmarking.** Describes the methodology for benchmarking cost models using the developed system, including the setup of the benchmarking environment and the empirical measurement of cost model performance.

Chapter 5: **Experiment and Analysis.** Utilizes the benchmarking system and methodology to design and conduct an experiment, aiming to identify key features that influence cost model performance in CPU and GPU systems. This chapter includes the experiment setup, results, and analysis.

Chapter 6: **System Evaluation.** Presents the evaluation of the benchmarking system itself, the benchmarking methodology and the experiment, discussing its capabilities and performance.

Chapter 7: **Conclusion.** This chapter concludes the thesis by stating the contributions alongside addressing research questions, discussing limitations, and suggesting future work

Chapter 2

Background

This chapter provides a foundation for the thesis by outlining key concepts and reviewing the state-of-the-art in relevant areas. It prepares the reader for the detailed exploration of the designed system, as well as the experiment carried out.

2.1 Query Optimization

Query optimization is an essential process in DBMS aimed at determining the most efficient way to execute a given query. The goal of query optimization is to minimize the query execution time and resource consumption, thereby enhancing the overall performance of the DBMS. This section explains the two most important query optimization concepts and outlines the area as a whole. Heuristic optimizations identify known query patterns and employ predefined rules on them, removing inefficiency, while cost-based optimization estimates the execution costs of various plans, selecting the one with the lowest cost [9].

2.1.1 Heuristic Optimization

Heuristic optimization employs a set of predefined rules or heuristics to transform the original query into an optimized, logically equivalent query, that is expected to execute more efficiently. These rules are grounded in general principles and best practices derived from empirical evidence and theoretical analysis.

Examples of heuristic optimizations include:

- **Selection Push-Down**: This optimization involves moving selection operations closer to the data source in the query execution plan. By applying filters as early as possible, the size of intermediate results is reduced, leading to lower data processing and transfer costs, in some cases. [12]
- **Selection Pull-Up**: In certain scenarios, particularly when the selection operation utilizes an expensive filter, it is advantageous to pull the selection operation up in the query execution plan. Executing this operation after other selections or joins, especially if they significantly reduce the size of intermediate results, can lead to more efficient query execution by minimizing the computational overhead on larger datasets. [12]
- **Early Projections**: Similar to selection push-down, performing projections early closer to the data source is advantageous. This reduces the amount of data that needs to be processed and transferred in subsequent operations. [28]

Heuristic optimization offers a quick and straightforward method to improve query performance without the necessity for detailed cost analysis.

2.1.2 Cost-Based Optimization

Cost-based optimization is a sophisticated approach that evaluates multiple execution plans for a given query to select the one with the lowest estimated cost. This method is distinguished by its reliance on a comprehensive cost model, which estimates the resource usage (e.g., CPU, I/O, memory) of executing a query plan. The cost model incorporates the statistical properties of the data, such as table sizes, data distribution, and index availability, to make informed decisions. One of the earliest and most influential implementations of cost-based optimization is found in the System-R optimizer [6], which introduced a mathematical model to estimate the costs associated with different query execution strategies, thereby selecting the most efficient execution plan based on those cost estimates.

Cost Model The cornerstone of cost-based optimization is the cost model, which provides a quantitative measure of the resources required to execute a query plan. It evaluates various factors, including CPU time, disk I/O, and memory usage, to estimate the cost of different operations such as scans, joins, and sorts. The accuracy of the cost model is critical, as it directly influences the optimizer's ability to select the most efficient execution plan. Enhancements in cost modeling techniques, including the incorporation of machine learning algorithms, have been explored to improve the precision of cost estimates. Cost models are further explained in section 2.2

Cost models use statistical data, such as initial cardinality, data distribution, and other statistics collected to perform cost estimation. To ensure the accuracy of a cost model, it is important that the statistical data accurately represent the database's current state. This dependency necessitates the regular collection and analysis of database statistics, a process that, despite its potential overhead, is crucial for the model's reliability.

Plan Enumeration Plan enumeration involves generating a comprehensive set of feasible execution plans for a given query. This step is crucial as it lays the foundation for the optimizer to evaluate and compare the costs of different plans. The number of possible plans can significantly increase, particularly for complex queries with multiple joins and operations, as the enumeration process must account for a broad spectrum of strategies.

Searching Once a set of potential execution plans has been enumerated, the optimizer must search through this space to identify the plan with the lowest estimated cost. Techniques such as dynamic programming are commonly employed to systematically explore the space of possible plans to find an optimal or near-optimal plan.

In summary, cost-based optimization represents a comprehensive approach to query optimization, with the cost model playing a pivotal role in estimating the resources required for different execution plans. Through the processes of plan enumeration and searching, the optimizer can select the most efficient plan, leading to potentially more effective and resource-conscious query execution.

2.1.3 Additional Optimization Techniques

While the focus of this thesis is on cost-based optimizations, it is important to acknowledge additional optimization techniques that also play a crucial role in query optimization. These include:

- **Physical Optimization**: Focuses on selecting the most efficient physical operators and access paths for executing the query, based on the physical layout of the data and available indexes. This can also be part of cost-based optimization. [6]
- **Adaptive Query Optimization**: Adjusts the query execution plan based on actual runtime statistics and data access patterns, enabling the DBMS to react to unexpected conditions. [2]
- **Parallel Query Optimization**: Concentrates on decomposing a query into subqueries that can be executed concurrently across multiple processors or nodes, optimizing for parallel execution. [1] Similarly, specialized hardware, such as GPUs enhance parallel execution capabilities due to their ability to run many threads simultaneously, speeding up data processing tasks. [18]

In summary, query optimization encompasses a variety of strategies aimed at enhancing the performance of DBMS. This thesis concentrates on heuristic and cost-based optimizations, laying the groundwork for exploring more advanced optimization techniques in future research endeavors.

2.2 Cost Model

This section introduces cost models. It discusses analytical and learned cost models, their methodologies, and applications. Analytical cost models rely on established formulas and statistical data to estimate costs, whereas learned cost models leverage machine learning techniques to forecast costs based on patterns observed in past query executions. The section also covers features that describe data and query characteristics, such as table sizes and data distribution and are used in cost models as input parameters. Additionally, it outlines cardinality estimation and its importance in the accuracy of cost predictions. The aim is to provide a clear understanding of cost models and their role in query optimization.

2.2.1 Analytical Cost Models

Analytical cost models utilize a combination of hand-crafted mathematical formulas and statistical information about the database to evaluate the cost of an execution plan. Crafting these formulas demands extensive knowledge about the underlying execution engine, as the models must accurately represent the consumption of computational resources, such as processing time, disk I/O operations, and memory usage, across various operations within a query plan.

Analytical models are closely linked to the specifics of the underlying execution engine, making this dependency a critical factor in their design and effectiveness. However, this close linkage also means that changes in the execution engine or its environment can render a cost model outdated.

Analytical cost models are indispensable in the optimization processes of relational database management systems (RDBMS), offering a methodical way to choose efficient query plans. For instance, the model utilized by PostgreSQL as of 2018 demonstrates the model's complexity and effectiveness. It calculates query costs by considering various factors, including the CPU time needed to process individual tuples, the CPU time required for each index entry beyond its I/O cost, and the CPU time used by each database operation, such as arithmetic and comparison functions [28].

To ensure the accuracy of a cost model, the statistical data must also accurately represent the database's current state. This requirement may lead to the regular collection and analysis of database statistics, a process that, despite its potential overhead, is crucial for the model's reliability.

$$\text{cost}_{\text{join}} = \text{cost}_{\text{scan}(R)} \times \text{cost}_{\text{scan}(S)} \times \text{cost}_{\text{comparison}} \tag{2.1}$$

The example shown in Equation 2.1 represents the cost function of a join operation between two tables, `(R)` and `(S)`. It calculates this cost as the product of three factors: the cost of scanning table `(R)` $\text{cost}_{\text{scan}(R)}$, the cost of scanning table `(S)` $\text{cost}_{\text{scan}(S)}$, and the cost of comparing tuples from the two tables to determine if they satisfy the join condition $\text{cost}_{\text{comparison}}$. This is an example of a cost function that could be part of a simplistic cost model.

2.2.2 Learned Cost Models

Learned cost models employ machine learning techniques to forecast the cost associated with executing SQL queries, thus tackling the complexities inherent in modern data-intensive applications. Both regression and classification methodologies necessitate extensive training data, which generally comprises features derived from query execution plans and database statistics. This training data, collected from past execution profiling, is crucial for the models to learn and accurately predict future costs. The primary similarity between these approaches lies in their reliance on this collected training data, enabling them to make informed predictions about the costs of future queries with varying levels of detail and accuracy.

Regression-based Models Regression-based learned cost models operate similarly to analytical cost models but leverage machine learning techniques to predict the cost associated with a query's execution plan. These models, designed to predict a continuous value, utilize techniques such as linear regression, polynomial regression, or more sophisticated approaches like neural networks and reinforcement learning. The outcome is a numerical estimate of the resources required for executing a query, such as execution time or memory consumption. Neural networks [15] and reinforcement learning [36, 19] are particularly favored for developing regression-based cost models. Their ability to capture complex input-output relationships and model nonlinear patterns in data makes them adept at handling the intricate aspects of database query optimization.

Classification-based Models Conversely, classification-based learned cost models assign query plans to discrete categories based on their anticipated performance. These categories might range from simple labels such as "fast", "medium", and "slow", to more nuanced classifications based on thresholds of resource usage. Decision trees, support vector machines, or neural networks like Tree-LSTM [19, 36, 30] and

Relational SPNs [13] can be utilized to construct these models. Classification models are especially beneficial for swiftly eliminating inferior query plans or when an exact cost estimation is less crucial than understanding the relative performance of various execution strategies.

Model Type	Techniques	References
Regression	Neural Networks	[15, 14]
	Reinforcement Learning	[15, 36]
Classification	Tree-LSTM	[19, 36, 30]
	Relational SPN	[13]

TABLE 2.1: Overview of Selected Learning-Based Query Optimization Methods

Learning-based models that work as a black box have been measured multiple times to have worse prediction accuracy [35]. Furthermore, black box models have little contribution to the understanding of relationships between the query, data, and execution performance.

Both regression and classification models can significantly improve the query optimization process by offering insights beyond those available through traditional cost estimation methods. Nonetheless, their success is heavily contingent upon the quality and representativeness of the training data, as well as the selected machine learning algorithm's capacity to generalize from historical observations to future queries.

2.2.3 Features & Metrics

Features, or the input metrics, play a crucial role in the performance of any cost model. Selecting the features used in the predictive learned model is a non-trivial task. Selecting too many can result in a high cost in prediction. On the other hand, omitting key features will also result in poor predictions. Furthermore, when designing an analytical cost model, knowing the features available and understanding their importance and relations is essential, to design an accurate cost model.

Given the pivotal role that features play in the accuracy and efficiency of cost models, it is imperative to carefully select those that have a significant impact on prediction outcomes. The subsequent lists outline features that have been widely adopted in the domain of query optimization. These features have been categorized based on their application at different levels of the cost model, starting with the foundational parameters used by PostgreSQL's cost models. [35]

Features used for plan-level models: [35]

- Estimated number of output tuples.
- Estimated average size of an output tuple (in bytes).
- Number of query operators in the plan.
- Estimated total number of tuples input and output to/from each operator.
- Estimated total size (in bytes) of all tuples input and output.
- The number of operators in the query.
- The total number of tuples output from operators.

Features used for operator-level models: [35]

- Estimated I/O (in number of pages).
- Estimated number of output tuples.
- Estimated number of input tuples (from left/right child operator).
- Estimated operator selectivity.
- Start-time of left/right child operator.
- Run-time of left/right child operator.

Hardware features (to increase the accuracy of the cost model): [35]

- I/O cost to sequentially access a page.
- I/O cost to randomly access a page.
- CPU cost to process a tuple.
- CPU cost to process a tuple via index access.
- CPU cost to execute an operation such as hash or aggregation.

The selection and understanding of features play a pivotal role in the design of accurate cost models. These features, as previously discussed, serve as inputs to cost models, enabling them to predict the costs associated with different execution plans. However, it is important to distinguish these input features from the metrics used to evaluate the actual performance of execution plans, such as measured execution time and resource utilization. Unlike the input features, these performance metrics are outcomes that are only observable after the execution of a query and are not known at the time of cost calculation.

Pearson Correlation Evaluating the effectiveness of a cost model requires comparing its predictions against these actual performance metrics. To facilitate this comparison, Pearson correlation is employed as a statistical measure. This method is used for measuring the performance of cost models [27, 34]. Pearson correlation quantifies the linear relationship between the predicted costs (the output of the cost model) and the observed metrics (measured execution time, resource utilization, etc.), providing insights into the accuracy of the cost model's predictions.

$$r = \frac{n(\sum x_i y_i) - (\sum x_i)(\sum y_i)}{\sqrt{[n\sum x_i^2 - (\sum x_i)^2][n\sum y_i^2 - (\sum y_i)^2]}} \tag{2.2}$$

The Pearson correlation coefficient (r) is calculated as shown in Equation 2.2, where x_i and y_i represent individual observations of the predicted costs and the actual metrics, respectively. A high r value, close to 1, indicates a strong positive linear relationship, suggesting that the cost model effectively predicts the impact on resource utilization and execution time. This evaluation process is crucial for assessing the cost model's performance and its ability to guide the optimization process toward minimizing resource consumption and execution time.

2.2.4 Cardinality Estimation

Understanding cardinality estimation is recognized as a fundamental aspect of query optimization and a crucial component of the baseline for this thesis. To establish a robust baseline, state-of-the-art techniques, known for their reliability and effectiveness in estimating the cardinality of intermediate results, are utilized. The exploration of these established methods aims to lay a solid foundation for the research, ensuring that the baseline is built upon sound and well-established principles. This section explains the concept of cardinality estimation, highlighting well-established techniques and methodologies.

Cardinality estimation is the process of predicting the number of tuples generated by an operator. It is a critical input for calculating operators' cost within a cost model. While the cost model may introduce errors of up to 30%, cardinality estimation can introduce errors in many orders of magnitude, especially for multi-join queries [15].

Cardinality and Selectivity Cardinality and selectivity are two closely related concepts in the realm of database management and query optimization. Cardinality refers to the number of rows or tuples produced by an operation in a database query, while selectivity deals with the proportion of rows that meet a specified condition or filter within that operation. Knowing the cardinality of intermediate results is sufficient for calculating the selectivity of the related operations. Intermediate cardinalities can also calculated with the initial table cardinality and selectivity of operations. In a way, cardinality and selectivity are interchangeable, since they refer to similar information.

Taxonomy Lan et al. slots different cardinality estimation techniques into 3 categories, synopsis-based methods, sampling-based methods, and learning-based methods [15]. Figure 2.1 illustrates the full taxonomy. Synopsis-based techniques introduce new data structures for storing statistical information, such as histograms and sketches among the common structures. Sampling-based techniques gather a set of samples from tables and subsequently execute the query on these samples to estimate cardinality. Leaning-based methods are among the newer, they are based on machine learning.

Learning-based methods Recently there are growing proposals for learning-based cardinality estimation as a replacement for conventional cardinality estimators. While the learned models exhibit superior accuracy, they are burdened by high training and inference costs. Moreover, their suitability in environments with frequent data updates is questionable [33, 10]. For these reasons, they will not be used in the proposed thesis.

Robust Baseline Since the main object of research of this thesis is the cost model, cardinality estimation must be a robust method. Synopsis-based techniques have been in research for the longest time, and are the most predominant in commercial DBMS applications [15]. Histograms are a fundamental technique in cardinality estimation, they provide a compact representation of the distribution of values within an attribute. This found distribution can be utilized for selectivity estimation of filtering and join operations.

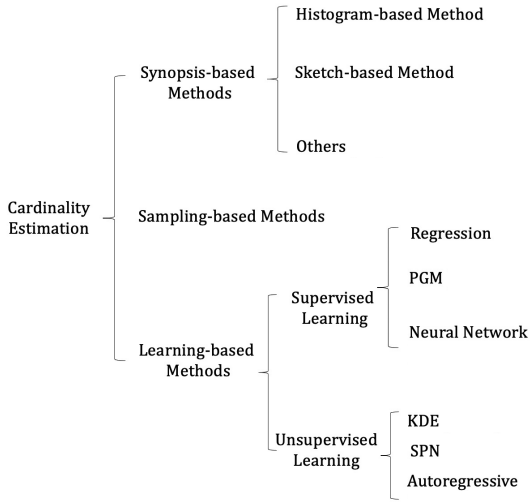


FIGURE 2.1: Taxonomy of Cardinality Estimation Techniques from [15]

2.3 Query Processing

This section introduces the concept of query processing within DBMS, focusing on the role of execution engines and the use of DataFrames (DFs) as a simulation mechanism for these engines.

2.3.1 Execution Engines

Execution engines are critical components of DBMS, tasked with the physical execution of queries. They include a range of physical operations such as data retrieval, sorting, aggregation, and join operations, all orchestrated to satisfy the demands of an SQL query. Fundamentally, execution engines handle memory allocation, data storage, and the efficient execution of operations, ensuring the optimal use of system resources. Understanding the details of execution engines is not the focus of this thesis.

The benchmarking system designed in this thesis measures the performance of how well the cost model predicts different execution plans. For this reason, it must also be able to execute and measure any execution plan. This assumes two requirements: firstly, the execution engine needs to be able to execute a custom execution plan, therefore the database must be injectable. Secondly, the database must be able to provide detailed information about the execution profile, the time taken and resource usage of all operations. Table 2.2 shows the comparison of features of relevant state-of-the-art databases. None of the relevant databases provide the option of injecting a custom execution plan. Therefore, to implement a modular benchmarking system, an alternative approach must be identified.

2.3.2 Data Frames and Processing

Data Frames are tabular data structures that are central to data analysis and manipulation in many programming environments. They are designed to store and operate on heterogeneous types of data, where each column can have a different type, similar to relational tables or spreadsheets. Libraries that implement DFs, such as Pandas

Name	System	Reads CSV files	Parses SQL	Query Opimization	Custom Execution Plan	Operation Timing
PostgreSQL [23]	CPU	-	✓	✓	-	✓
RateupDB [16]	hybrid	-	✓	✓	-	✓
BlazingSQL (RAPIDS) [4]	hybrid	-	✓	✓	-	-
Spark (RAPIDS) [37]	hybrid	✓	✓	✓	-	✓
HeavyDB [11]	hybrid	✓	✓	✓	-	✓
DuckDB [8]	CPU	✓	✓	✓	✓	✓
Pandas [21]	CPU	✓	-	-	✓	✓
cuDF (RAPIDS) [7]	GPU	✓	-	-	✓	✓

TABLE 2.2: Feature Comparison of SQL and DF Engines

in Python [21], provide a rich set of operations that can be used to perform complex data manipulations, aggregations, and transformations efficiently.

The significance of DFs extends beyond mere data storage; they embody a versatile abstraction for data manipulation. This versatility is underpinned by the extensive suite of operations they support, which include filtering, grouping, and joining of data, among others. These operations are not only fundamental to data analysis tasks but also closely resemble the operations performed by an execution engine within a DBMS.

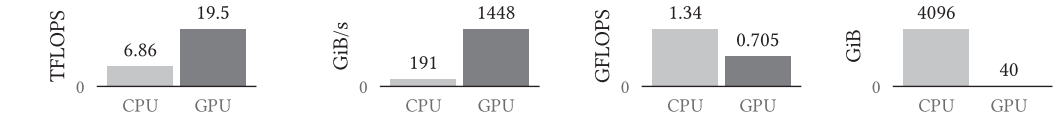
In the context of this thesis, DFs are not only of interest for their data storage capabilities but also for their potential to simulate the behavior of an execution engine. By leveraging the operations provided by DF libraries, it is possible to mimic the execution of SQL queries. This approach offers a unique avenue for exploring query optimization techniques in environments where traditional execution engines might not be readily modifiable or accessible for experimentation.

The mapping of DF operations to those of an execution engine is not trivial and requires careful consideration of the semantics and performance characteristics of both domains. This mapping process is crucial for ensuring that the simulated execution engine behaves in a manner that is both accurate and efficient. The specifics of how DF operations are mapped to execution engine operations, including the challenges and strategies involved in this process, are discussed in detail in subsection 3.4.3.

2.4 CPU and GPU architectural differences

Understanding the architectural differences between CPUs and GPUs is crucial for designing accurate cost models. CPUs are traditionally used for a wide range of computing tasks, including complex query processing that requires significant memory due to their larger memory capacity. GPUs, with their high data throughput capabilities, are better suited for parallel processing tasks. The efficiency of GPUs in DBMS can be impacted by the overhead of data transfer between the CPU and GPU.

Notable differences between CPU and GPU performance and resource statistics are illustrated in Figure 2.2.



(a) Aggregate performance (FP32) (b) Memory bandwidth (c) Serial performance (FP32) (d) Memory size

FIGURE 2.2: Comparison between a CPU (AMD EPYC 7702P) and a GPU (NVIDIA ampere A100). [25]

Core Architecture and Parallelism GPU architectures are divided into Streaming Multiprocessors (SMs), each with cores and registers, supporting the execution of many threads in thread blocks. These blocks are divided into warps of usually 32 threads, following the Single Instruction Multiple Threads (SIMT) model, enhancing parallel processing efficiency. [26]

Memory Architecture The GPU's memory architecture is hierarchical, global memory is the largest, but furthest from threads, and so the slowest. Each SM has shared memory for quick data exchanges, with memory requests going through an L2 cache shared by all SMs and an optional L1 cache per SM, crucial for performance in memory-limited operations. [26]

Execution Model and Use Cases The GPU's execution is shaped by its memory and core architecture, using thread blocks and warps to optimize memory bandwidth use. Programmers can allocate global and shared memory, with shared memory offering higher bandwidth but limited capacity. Registers are the fastest memory layer, with spillover to global memory affecting performance. [26]

Chapter 3

System Design

This chapter outlines the design of the system capable of evaluating cost models across different environments. The design process starts with requirements, proceeds with architecture diagrams, and later explains all components in detail.

3.1 Requirements

This section outlines the requirements for a benchmarking system capable of evaluating any cost model for join-order optimization in relation to any execution engine. The requirements are divided into functional and non-functional categories.

Functional Requirements

Functional requirements detail the essential operations, behaviors, and functionalities your benchmarking system must execute, particularly focusing on SQL query optimization and evaluation of cost models for join-order optimization. These are pivotal for defining the system's core capabilities.

FR 1 **SQL Parsing**: The system must be able to parse SQL queries and db schema, supporting various selection operations, and equijoins.

FR 2 **Cost Model Integration**: The system must allow for the integration of various cost models for join-order optimization.

FR 3 **Execution Engine Compatibility**: The system must be able to execute a query with different execution engines, including CPU and GPU-based systems.

FR 4 **Input Feature Support**: The system must collect DB statistics and perform cardinality estimation, to expose a wide set of input features to the cost model to support diverse cost models.

FR 5 **Cost Prediction Collection**: The system must provide a method of collecting cost predictions for any specified join order, as a step in evaluating the performance of different cost models against various execution engines.

FR 6 **Execution Profiling**: The system must be capable of profiling the execution time of queries, to collect detailed characteristics of the execution.

FR 7 **Dataset Management**: The system must support the use of different CSV datasets and SQL queries for testing and evaluation purposes.

FR 8 **Result Reporting**: The system must generate detailed reports on the benchmarking results, in the form of tables and plots.

Non-Functional Requirements

Non-functional requirements encompass the system's quality attributes, such as performance, scalability, and usability, crucial for a benchmarking system analyzing SQL queries across different execution engines. These attributes are vital for ensuring efficiency, reliability, and user satisfaction.

NFR 1 **Portability**: The system should be simple and fast to configure and start on different platforms.

NFR 2 **Usability**: The system should have a user-friendly interface for various user interactions.

NFR 3 **Plugability**: The system should support a plug-and-play architecture where cost models, and execution engines, can be easily added, removed, or modified.

NFR 4 **Extensibility**: The system should be designed with extensibility in mind for future enhancements and new features, such as supporting any execution plan, or more profiling metrics.

NFR 5 **Consistency & Reproducibility**: The system should be reliable, each experiment should have consistent results, and experiments should be easy to reproduce.

3.2 Architecture Overview

This section presents the architecture of the benchmarking system, designed to assess the efficacy of different cost models across different environments. An architecture diagram is provided to illustrate the system's structural design, highlighting its key components and their interactions.

Before explaining the architecture overview, it is essential to understand the foundational structure of an SQL database. Refer to Figure 3.1 for a flow chart overview. This flow typically consists of three core components: SQL parsing, query optimization, and the execution engine.

The process begins with SQL parsing, where the input SQL query is parsed and converted into a naive relational algebra expression. This expression serves as a formal representation of the query, devoid of syntactic sugar, making it easier for the system to manipulate. The next stage is query optimization, where this relational algebra expression is transformed into an optimized execution plan. This plan is a detailed blueprint of how the database will execute the query, including which algorithms to use for joins and the order of operations, aiming to minimize resource usage and execution time. Finally, the execution engine takes this optimized plan and interacts with the database to perform the actual data retrieval and manipulation, outputting the SQL results.

The designed system mimics the same flow. On top of it, it includes various pointcuts to input components and extract information valuable for analysis. Consider Figure 3.2, which shows the components of the three main blocks.

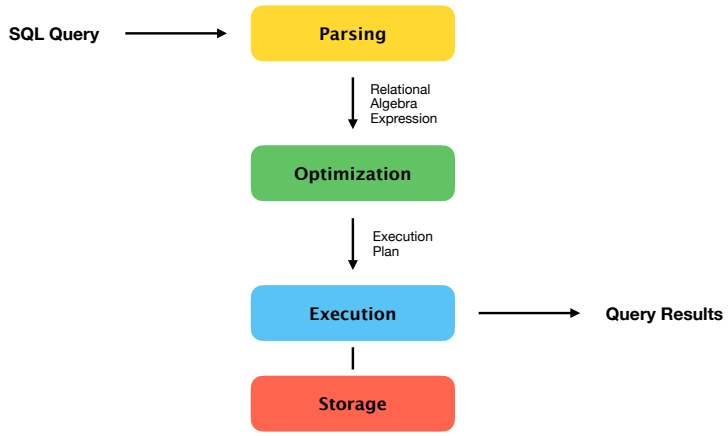


FIGURE 3.1: General Components and Flow of SQL DBMS

Short Description of Components

This section offers a short description of the components in the architecture diagram as shown in Figure 3.2. A full design and implementation details of the components can be found in section 3.4.

SQL Parsing

- **Schema Parsing**: Involves parsing of the DB schema from `CREATE TABLE` command. This component is essential for understanding the structure and types of data within the database, directly supporting the requirement for SQL parsing (FR 1) and enabling accurate data representation in the system.
- **Query Parsing**: Involves parsing the SQL query, focusing on `WHERE` clauses. This process is crucial for decomposing the query into a format that can be optimized and executed by the system, addressing the functional requirement for SQL parsing (FR 1) and serving as a foundation for query optimization processes.

The SQL Parsing component is crucial for interpreting SQL queries and database schemas, enabling the system to understand and manipulate database structures and queries effectively. This component directly supports FR1 by ensuring the system can parse and interpret SQL queries and schemas accurately, laying the groundwork for subsequent optimization and execution processes.

Query Optimization

- **Logical Optimization**: Pushes down filtering operations to reduce data processed later, enhancing performance and supporting heuristical optimizations outside plan enumeration, aligning with extensibility (NFR 4).
- **Join Order Injection**: Injecting the optimization process with a specified join order. This component allows for the evaluation of different join orders, enabling the requirement for cost (FR 5) and profile (FR 6) collection for various join orders.
- **Cardinality Estimation**: Involves predicting the size of intermediate results for each part of the query. This estimation is critical for selecting the most

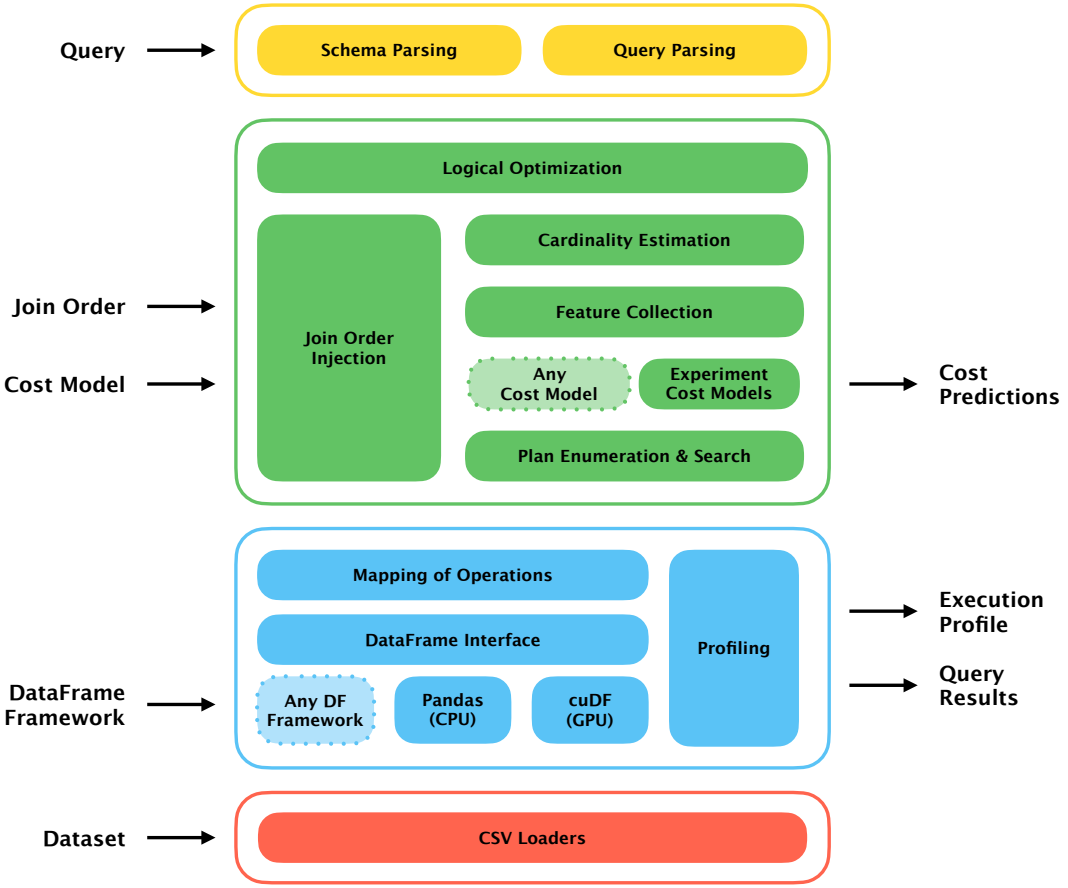


FIGURE 3.2: Detailed Overview of Architecture of the Benchmarking System

efficient query plan, as it impacts the cost model's ability to accurately predict execution costs, addressing the requirement for input feature support (FR 4).

- **Feature Collection**: This process gathers various statistics and characteristics of the data and query, which are used by the cost model to estimate execution costs. It directly supports the system's requirement for input feature support (FR 4) by providing the necessary data.
- **Cost Model Placeholder**: A placeholder for integrating different cost models that estimate the total cost to execute a join order. This component is central to the system's design, allowing for the evaluation of various cost models against the execution engine, and fulfilling the requirement for cost model integration (FR 2) and pluggability (NFR 3).
- **Plan Enumeration & Search**: Involves generating and evaluating multiple query plans to select the one with the lowest estimated cost. This component is essential for exploring the space of possible query plans.

The Query Optimization component provides a wrapping of a modular cost model. This component can be used to feed cost models input features, and collect the cost predicates, or use the cost predicates to optimize the query execution plan. It supports FR2, FR4, FR5, FR6, NFR3, and NFR4, by integrating various cost models for join-order optimization, collecting input features for cost prediction, and allowing

for the evaluation of different join orders. This component is pivotal in ensuring efficient query execution strategies are selected.

Execution Engine

- **Mapping of Operations**: Translates the optimized query plan into operations that can be executed by a DataFrame framework. This translation is crucial for executing queries in the chosen DataFrame framework, addressing the requirement for execution engine compatibility (FR 3).
- **DataFrame Interface**: A connector for DataFrames, this layer is used to exchange between different DataFrame Frameworks. It ensures the system's flexibility and extensibility in supporting various DataFrame libraries, fulfilling the requirement for pluggability (NFR 3) and extensibility (NFR 4).
- **DataFrame Framework Placeholder**: A placeholder for integrating different DataFrame processing libraries to execute operations. This component allows the system to adapt to different execution environments, directly supporting the requirement for execution engine compatibility (FR 3) and extensibility (NFR 4).
- **Profiling**: Involves measuring the execution time of queries to gather performance data. This process is essential for evaluating the performance of query execution, directly contributing to the system's ability to profile execution times (FR 6).

The Execution Engine component is responsible for executing optimized queries using DataFrame operations. It addresses FR3, NFR4, and NFR4 by ensuring compatibility with different execution engines and FR6 by profiling execution times. This component is essential for translating optimized queries into actions performed by the DataFrame framework, enabling the actual data retrieval and manipulation.

Storage Management

- **CSV Loaders**: Responsible for loading datasets from CSV files. This component ensures that data stored in CSV format is accurately imported into the system for processing and analysis, directly supporting the requirement for dataset management (FR 7) by enabling the use of diverse datasets for testing and evaluation.

Storage Management handles the loading and management of datasets, crucial for the system's ability to process real-world data. It directly supports FR7 by enabling the system to manage and utilize various datasets for benchmarking and evaluation, ensuring the system's applicability to diverse data scenarios.

Inputs

- **Dataset**: A collection of data tables on which the SQL queries will be executed.
- **Query**: The SQL query or set of queries to be optimized and executed by the system.
- **Join Order**: Specifies the sequence in which joins are to be executed in the query.

- **Cost Model**: The model used by the system to estimate the cost of executing a query plan.
- **DataFrame Framework**: The underlying framework or library used for data manipulation and execution.

The Inputs component defines the necessary data for the system to operate, including datasets, queries, join orders, cost models, and DataFrame frameworks. This setup is foundational for the system's operation, enabling it to process and optimize SQL queries across different environments and configurations.

Outputs

- **Cost Predictions**: The estimated costs of executing a specific join order, as determined by the cost model. This fulfills FR5.
- **Execution Profile**: Detailed metrics on the execution of the query, including the execution time of various processes in the query execution pipeline. This fulfills FR6.
- **Query Results**: The final results of executing the SQL query.

The Outputs component specifies the results produced by the system, including cost predictions, execution profiles, and query results. These outputs are crucial for evaluating the performance of SQL queries and the effectiveness of different optimization strategies, directly supporting FR8 by providing detailed benchmarking results.

3.3 Use cases

Following the architectural overview, this chapter will delve into three distinct use cases, each accompanied by flow charts. These use cases exemplify the system's capabilities in collecting cost-model predictions, measuring join execution times, and functioning as an SQL engine, thereby showcasing the system's versatility and comprehensive approach to SQL query optimization.

1. Measuring Execution Times
2. Generating Cost Predictions
3. Getting Query Results

Measuring Execution Times Showcased in Figure 3.3 and 3.4. In this use case, the objective is to measure the actual execution times of SQL queries to gather performance data. The inputs required are a dataset, the SQL query, a specified join order, and the DataFrame Framework (e.g., Pandas for CPU or cuDF for GPU execution). The system first performs some rudimentary logical optimization. Then the query is executed according to the provided join order using the specified DataFrame Framework. The output is an execution profile that includes detailed metrics on the execution time of the query, with details on the execution times of individual components, providing valuable data for performance analysis and optimization.

This use case demonstrates the system's capability to accurately measure the execution times of SQL queries, directly supporting FR6. By profiling the performance of queries across different join orders and execution engines, this use case validates the system's effectiveness in performance analysis and optimization.

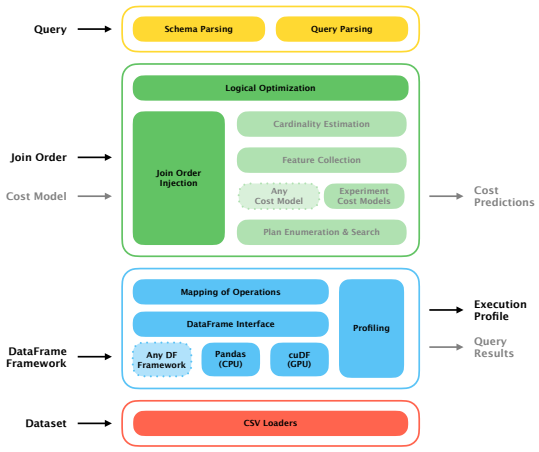


FIGURE 3.3: Component Highlight of Use Case 1

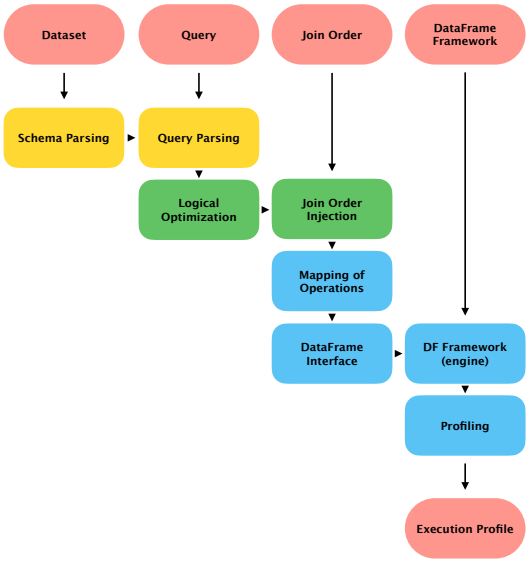


FIGURE 3.4: Flowchart of Use Case 1

Generating Cost Predictions Showcased in Figure 3.5 and 3.6. This use case focuses on the generation of cost predictions, for executing a given SQL query with a specified join order. This use case does not employ any components in the entire execution engine block. The inputs for this process include a dataset, the SQL query to be optimized, a predefined join order, and the cost model to be used for prediction. The cost model utilizes the input features derived from the dataset, query, and join order to estimate the cost of executing the query with that specific join order. The output of this use case is a set of cost predictions, which provide insights into the expected execution time and performance implications of the query execution plan. These predictions are crucial for evaluating the efficiency of different join orders and selecting the most optimal one for query execution. These theoretical predictions, complement the measured execution times from previous use cases, enabling an analysis and comparison of the two.

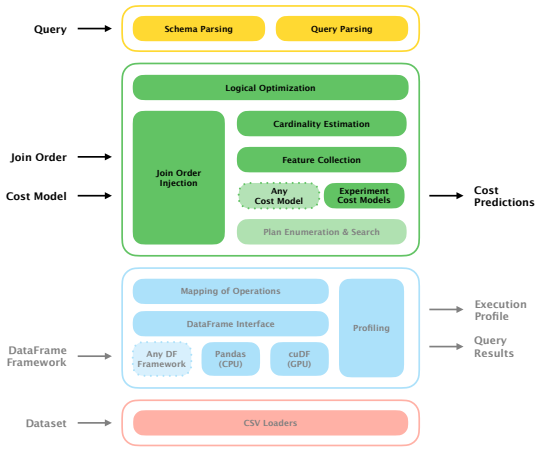


FIGURE 3.5: Component Highlight of Use Case 2

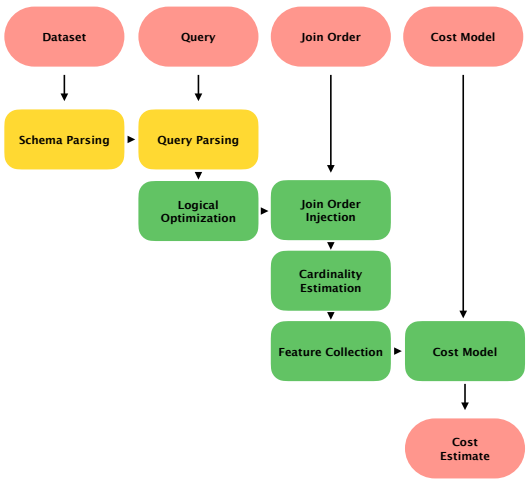


FIGURE 3.6: Flowchart of Use Case 2

Focused on generating cost predictions for different join orders, this use case validates FR2 and FR5 by showcasing the system's ability to integrate and evaluate various cost models. It emphasizes the system's analytical capabilities, enabling the comparison of theoretical cost predictions with actual execution metrics.

Getting Query Results Showcased in Figure 3.7 and 3.8. Diverging from the analytical nature of the first two use cases, this scenario explores the system's capability to function akin to a real Database Management System (DBMS), albeit not its primary intended use. While the initial use cases—generating cost predictions and measuring execution times—are tools designed to generate data for analysis, this use case focuses on executing an SQL query to retrieve actual query results. The inputs remain the same: a dataset, the SQL query, a cost model, and the DataFrame Framework. Here, the cost model aids in determining an efficient join order to minimize execution costs. Following this, the query is executed using the selected DataFrame Framework, handling data manipulation and execution tasks akin to a traditional DBMS. The output, in this case, is the query results—the final data retrieved by executing the SQL query. To facilitate further analysis and sharing, these results are saved in a CSV file. Although not the system's intended behavior, supporting this use case allows for the verification of result correctness, showcasing the system's versatility beyond its primary analytical functions.

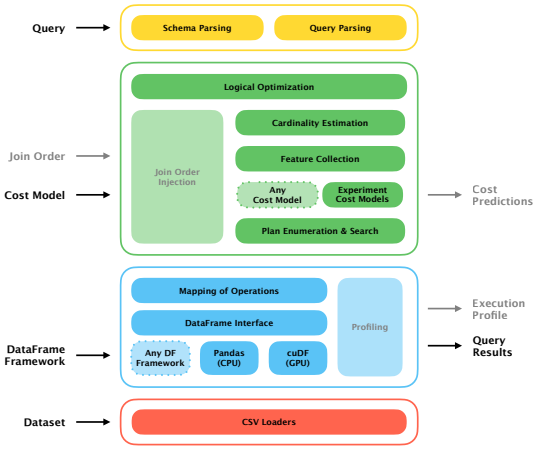


FIGURE 3.7: Component Highlight of Use Case 3

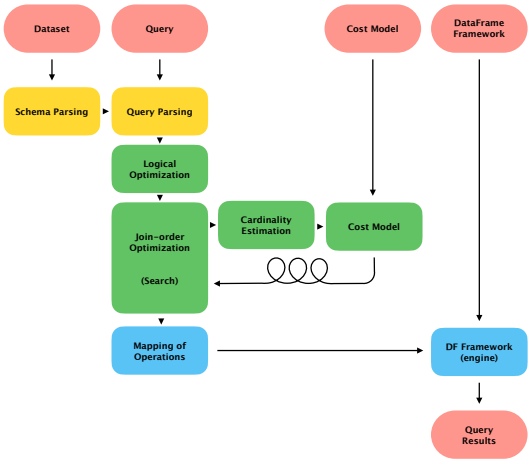


FIGURE 3.8: Flowchart of Use Case 3

This use case explores the system's functionality beyond analytics, demonstrating its ability to execute SQL queries and retrieve actual results, akin to a traditional DBMS. While not its primary function, this capability can be extended in the future to produce a usable DBMS.

3.4 Component Details

This section includes a detailed description of all the components in the designed system. The description includes motivation for the design decisions and implementation details. The section is organized into three parts following the three blocks of the architecture, parsing, optimization, and execution.

SQL Parsing

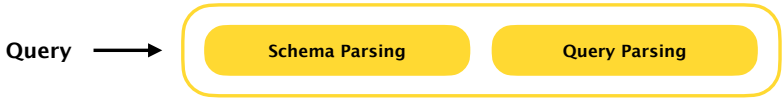


FIGURE 3.9: SQL Parsing Extract of the System Architecture

3.4.1 Schema Parsing

Parsing a schema from an SQL CREATE TABLE command is a crucial step in understanding the structure of CSV files used as data storage in the designed system. Since CSV files are not guaranteed to provide headers, and the type is not explicitly stated, parsing the schema externally through SQL commands allows the system to interpret the CSV data correctly. By parsing the schema, the system gains essential information about the table name, column names, data types, and primary keys. The format of schemas supported is shown in Listing 3.1.

```
CREATE TABLE {{table}} (
    {{column}} {{data_type}} [[ {{constraints ignored}} ]] [[
       PRIMARY KEY ]],
    ... # more columns
    [[ PRIMARY KEY ({{column}} [[, {{column_n}} ]]) ]],
);
```

LISTING 3.1: Suppoted SQL schema format

Notice that ignoring the constraints since the system is not intended to be used as a database with create, update, or delete operations. And leave the integrity of the datasets up to the dataset designers.

3.4.2 Query Parsing

To be able to process many queries, a simple SQL parser is included. The implementation of the SQL parser in the proposed experimental system focuses on handling uncomplicated queries with a focus on equi-joins, and basic filters to support a variety of scenarios. The following structure is supported:

```
SELECT {{columns}}
FROM {{tables}}
[[ WHERE {{where_clauses}} ]]
[[ GROUP BY {{ignored}} ]]
[[ ORDER BY {{ignored}} ]]
[[ LIMIT {{ignored}} ]]
;
```

LISTING 3.2: Suppoted SQL query format

This parser is tailored to process basic SQL statements by extracting essential components such as the `SELECT` clause, the `FROM` clause with table aliases, and the `WHERE` clause, which includes filtering conditions and joins. While its scope is limited to simple queries, the parser effectively dissects and organizes the query's critical elements, enabling further analysis or subsequent processing.

Implementing operations such as `GROUP BY`, `ORDER BY`, and `LIMIT` is out of scope, therefore also the parser ignores those operations.

One of the more interesting parsing problems is the `WHERE` clause. Following is a list of supported operations:

Operation	Description
= (as a join)	Equi-join on columns from different tables
= (as a filter)	Equality filter on a column
`!=`	Not equal to filter on a column
`IN`	Checks if a column's value is in a list of values
`IS NULL`	Checks for null values in a column
`IS NOT NULL`	Checks for non-null values in a column
`>`	Greater than filter on a column
`>=`	Greater than or equal to filter on a column
`<`	Less than filter on a column
`<=`	Less than or equal to filter on a column
`IS BETWEEN`	Lower and upper bound filter on a column
`LIKE`	Pattern matching filter on a column
`NOT LIKE`	Inverse pattern matching filter on a column

TABLE 3.1: Supported SQL Operations with Descriptions

The parser is designed to support `WHERE` clauses that adhere to the Conjunctive Normal Form. This means that the `WHERE` clause should be structured as a series of conditions connected by `AND` operators, where each condition can be a simple comparison or a set of comparisons connected by `OR` operators. This is a general enough format to support a wide variety of queries, while simple enough to parse. For an example of a clause in CNF, see Listing 3.3.

```
WHERE (occupation = "Developer" OR occupation = "Designer")
    AND age IS BETWEEN 20 AND 25
    AND (city LIKE "New␣%" OR city LIKE "Paris")
    AND (education = "Bachelor's" OR education = "Master's")
```

LISTING 3.3: Example of a WHERE clause highlighting the CNF

Execution Engine

Before looking into the optimization block, let's explain the execution. It introduces some concepts, such as operations interface, that are used also in components of the optimization block.

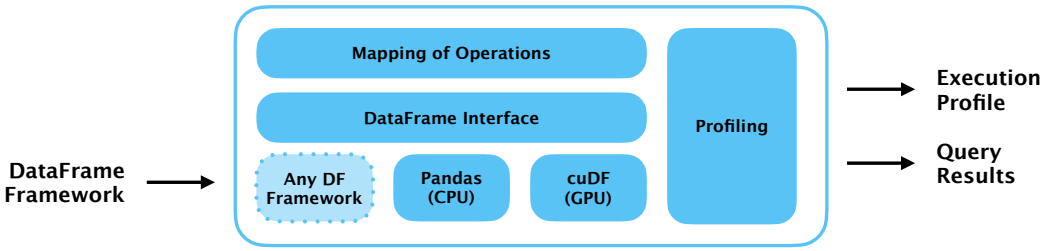


FIGURE 3.10: Execution Engine Extract of the System Architecture

3.4.3 Mapping of Operations

The system must be able to execute all instructions, namely the join, with various execution engines. The choice of the execution engine is any `DataFrame` framework. `DataFrame` include operations such as `merge` and `loc`, which can be used to achieve joins and filtering.

First, this section defines the `Operations Interface`, which defines all supported operations. You can see this interface in Listing 3.4.

```
TVal: TypeAlias = str | int | float | bool

class Operations(ABC, Generic[I, O]):
    @abstractmethod
    def from_tables(self, db_path: str, db_name: str, tables:
        list[str], aliases: list[str] = []) -> Callable[[], I]:
         ...

    @abstractmethod
    def join_fields(self, field_name_1: str, field_name_2: str)
        -> Callable[[I], O]:
         ...

    @abstractmethod
    def filter_field_eq(self, field_name: str, values: TVal |
        list[TVal]) -> Callable[[I], O]:
         ...

    # Similar declaration of all other filter_field_ methods
        hidden
```

LISTING 3.4: Operations Iterface

This interface is implemented by us to provide real operations with `DataFrame` operations. The use of `DataFrame` operations is shown in Listing 3.5

```
# Filters =, !=, <, >, <=, >=
table.loc[table[field_name] == value]

# Pattern filter LIKE
table.loc[table[field_name].str.contains(pattern)]

# Equi-Joins
table_1.merge(table_2, how="inner", left_on=field_name_1,
    right_on=field_name_2)
```

LISTING 3.5: Mapping of SQL to DF operations

3.4.4 DataFrame Interface

The flexibility to switch between different DataFrame processing engines is a crucial aspect of the system, enabling it to operate seamlessly across both CPU and GPU environments. This capability is particularly important for evaluating the performance and efficiency of SQL query execution and join-order optimization strategies under varying hardware conditions. To facilitate this, the system employs a dynamic engine swapping mechanism, shown in Listing 3.6.

```
def set_engine(name: str):
    global engine
    if name == "cpu":
        import pandas
        engine = pandas

    elif name == "gpu":
        import cudf
        engine = cudf

def get_engine() -> Any:
    global engine
    return engine
```

LISTING 3.6: Code Snippet showing swap functionality of DataFrame frameworks

Note that this swapping can be extended in the future with any compatible DataFrame implementation. Also, note that the system uses Python's parser benefits and only loads the DF framework after it's selected, thus reducing system requirements.

This approach offers several advantages. Firstly, it abstracts away the specifics of the DataFrame processing libraries, allowing the rest of the system to interact with the data through a unified interface. Secondly, it provides the flexibility to adapt to different hardware configurations without requiring significant changes to the system's architecture or the underlying code.

3.4.5 DataFrame Framework Placeholder

The system will be implemented with two data processing libraries, `pandas` [21] and `cudf` [7]. The important difference is that `pandas` runs on CPU and `cudf` runs on GPU. Switching between the two libraries therefore ensures that the proposed system can be used to run the same operations on different hardware.

3.4.6 Profiling

Profiling in the context of the designed benchmarking system is crucial for understanding the performance characteristics of different components. To achieve this, point-cuts are introduced at strategic locations within the system's workflow. These point-cuts serve as markers, between which measurements of execution time for operations to complete. This method allows us to gather detailed performance data, which is essential for identifying bottlenecks and optimizing the system's efficiency.

The following point-cuts have been identified for profiling within the system:

1. **SQL Parsing**: Measures the time taken to parse the database schema from SQL `CREATE TABLE` commands, and the duration of parsing SQL queries.
2. **Overhead (Loading CSV Files)**: Times the overhead associated with loading CSV files into the system.
3. **Filters**: Profiles each filter operation individually.
4. **Joins**: Similar to filters, each join operation is profiled individually.
5. **Results Materialization**: Measures the time taken to save the results back to a CSV file.

Data Storage and Analysis Profiling data is saved into CSV files as raw data. This approach allows for the preservation of granular performance metrics across different profiling sessions. The system includes tools designed to extract this data from the CSV files, enabling researchers to easily access and analyze the profiling information.

```
# Lines are wrapped for display convenience
TIMESTAMP;DB_SET/QUERY;DEVICE;TYPE_OF_RUN;JOIN_PERMUTATION
    ;EXIT_CODE;PARSING;OVERHEAD
    ;FILTERS
    ;JOINS
[2024-02-21T03:37:56];job/2a;gpu;0;0,1,2,3,4
    ;200;0.00304;3.67675
    ;0.03445,0.00742
    ;0.00728,0.01504,0.01821,0.02343,0.00509
...
```

LISTING 3.7: Example of profiling results

The data extraction tools are complemented by data summarization utilities, which aggregate the raw profiling data into meaningful summaries. These summaries highlight key performance metrics and trends, making it easier to identify potential areas for optimization.

Furthermore, the system provides plotting tools that leverage the summarized data to generate visual representations of the profiling metrics. These plots offer a clear and intuitive way to understand the performance characteristics of the system, facilitating a deeper analysis of the implications of different join orders and cost models on query execution performance.

Extensibility of Profiling While the initial focus of profiling is on measuring execution times, the framework is designed to be extensible. Future enhancements can introduce additional metrics such as resource usage (CPU, GPU, memory), which are critical for a comprehensive performance analysis. By incorporating these metrics, the system can provide a more holistic view of the performance implications of different join orders and cost models, facilitating a deeper understanding of their behavior across various execution environments.

This profiling mechanism, with its focus on execution time and potential for extension, forms a foundational component of the benchmarking system. It enables the rigorous evaluation of cost models and the optimization of SQL query execution, aligning with the overarching goals of this research.

Storage Management

FIGURE 3.11: Sotrage Management Extract of the System Architecture

3.4.7 CSV Loaders

The CSV Loaders component is responsible for loading tables stored in CSV format, converting them into individual DataFrames for subsequent processing within the execution block. This conversion leverages the `read_csv()` function from the DataFrame processing engine chosen by the system. Depending on whether pandas or cuDF is selected, data is loaded into CPU or GPU memory, respectively. This ensures that the data loading process is optimized for the computational resources in use.

The configuration file for the CSV Loaders component is essential for defining how tables stored in CSV format are loaded into DataFrames. This file contains configurations for each dataset, specifying parameters that dictate the interpretation and loading process of CSV data. The structure of the configuration file allows for easy extension, enabling the addition of configurations for new datasets as required. Below is an example of how a dataset configuration is defined within this file:

```
{
    "default_db": {
        "file_suffix": "csv",
        "column_sep": ","
    }
    // Additional datasets can be configured similarly
}
```

LISTING 3.8: Example datasets configuration file

As the initial step in the system's data management workflow, the CSV Loaders component is critical. It facilitates the efficient loading of data into the designated memory space, paving the way for the data's processing in subsequent phases.

Query Optimization

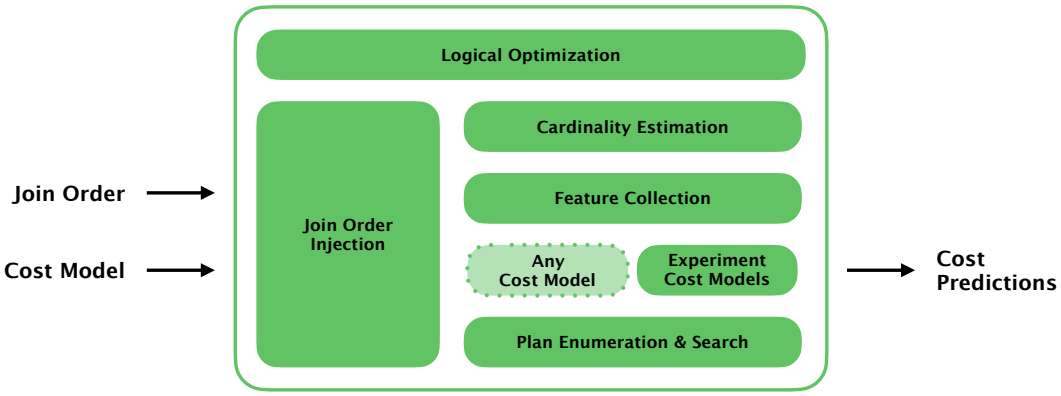


FIGURE 3.12: Query Optimization Extract of the System Architecture

3.4.8 Logical Optimization

The focus of the designed system is join-order optimization. For this reason, the cost-based optimization only changes the order of joins. To produce a more realistic situation, the system has a layer of logical optimization that performs some basic heuristical optimization. The only optimization implemented is the selection push-down. This means that all selection, and filtering operations are performed before

the joins. This optimization reduces the size of intermediate data, which lowers the costs of subsequent operations, in this case, joins. It is important to note that the system only implements selection pushdown because the system supports simple filters, eliminating the need for selection pull-up. For more details on heuristic optimization, refer to subsection 2.1.1 in the background chapter.

The only optimization implemented is the selection pushdown. This means that all selection, and filtering operations are performed before the joins. This optimization reduces the size of intermediate data. Which lowers the costs of subsequent operations, in this case, joins.

3.4.9 Join Order Injection

To evaluate the cost model for different join orders, the system must collect statistical information on different join orders. Those statistics include cost model predictions and measured execution times for different join orders. The system achieves this by enabling the injection of arbitrary join orders in place of the join order optimization algorithm. The use case can be selected (either predicting the cost or measuring the execution time), and a specific join order can be specified to then collect statistical information about that join order.

To specify a join order and inject it into a system, a way of encoding a join order in a determined order must be designed. In standard relational DBMS, the query is parsed into a relational algebra expression, which is a tree. It is important to note that multiple join orders can be represented by a single execution tree. Depicted in Listing 3.9 is a query in which four tables are combined with three joins. Consider two different join order `[1,2,0]` and `[2,1,0]`. They both compile into the same tree structure as shown in Listing 3.10.

However, the exact order in which joins are executed significantly impacts the overall query performance due to various underlying factors, such as data locality and cache utilization. The execution tree provides a structural representation of how tables are joined, but it does not inherently encode the sequence of join operations. This lack of specificity means that while two different join orders might compile into the same tree structure, their execution costs can vary dramatically.

```
SELECT *
FROM student, cohort, course, exam_result
WHERE student.id = exam_result.student_id   # JOIN ID: 0
    AND cohort.id = student.cohort_it       # JOIN ID: 1
    AND course.id = exam_result.course_id   # JOIN ID: 2
```

LISTING 3.9: Example of an SQL query with 3 joins

```
                                0
                              /   \
                            /       \
                          /           \
                        /               \
                      1                   2
                    /   \               /   \
                  /       \           /       \
              cohort      student  exam_result  course
```

LISTING 3.10: Execution tree of query in Listing 3.9

Therefore, while the execution tree provides a high-level view of the join operations, the specific order in which those joins are executed plays a pivotal role in optimizing query performance, underscoring the importance of sophisticated join order optimization algorithms in database systems. For this reason, the join orders are also encoded in a linear format, similar to the example in Listing 3.9 where joins are indexed, starting at 0, and the join order is encoded into an ordered list containing all indexes.

3.4.10 Cardinality Estimation

Cardinality estimation plays a critical role in database query optimization and execution by providing estimates of the number of distinct values in a particular column or set of columns within a table.

The process of cardinality estimation involves several steps, focusing on collecting and utilizing statistical information about the database tables involved in a query. This information is typically calculated before query execution and cached for efficiency. In operational databases, these statistics can be recalculated during idle times to ensure they remain accurate as the data changes.

Collect Initial Statistics The first step involves gathering basic statistics about each table, such as the total number of rows (table cardinality) and the range of keys. This information provides a baseline for further estimations.

Use of Histograms Histograms are a widely employed technique for achieving accurate cardinality estimates. Histograms capture the underlying data distribution and frequency of values, enabling more precise estimations of distinct value counts. [15] Where possible, equi-width histograms [28] are used to estimate the selectivity of joins and other filtering operations. Histograms divide the range of a column's values into buckets, with each bucket containing a count of the number of values that fall within its range. This structure allows for estimating the distribution of data values, which is essential for predicting the result size of operations like joins and filters.

Hashmap for High Repetitive Columns In cases where histograms are not feasible or practical, especially for columns with a high number of repeated values (where the total length of the column is significantly greater than the number of unique values), a hashmap with counts for each unique value is used. This approach enables the calculation of selectivity for operations involving these columns by directly referencing the count of specific values.

Fallback to Standard Rules When neither histograms nor hashmaps can be applied, the system falls back on standard rules of thumb for cardinality estimation. These rules are based on general assumptions about data distribution and are less accurate than methods based on actual data statistics.

3.4.11 Feature Collection

The process of feature collection is a critical component in the development of cost models for join-order optimization. This step involves gathering a comprehensive set of data features that can significantly influence the performance of SQL queries,

especially when executed on different platforms such as CPUs and GPUs. The primary goal of feature collection is to capture a wide array of database statistics that can be used by the cost model. Since the system does not know the model plugged in, the feature collection must be rigorous and support a wide range of features. The provided features can be classified into three primary categories: Table-Level Features, Column-Level Features, and Join-Level Features.

Table-level Features Table-level features pertain to the characteristics and statistics of individual tables involved in the join operations. These features provide a foundational understanding of the data volume and distribution at the table level.

- **Table Length (`t_length`)**: The total number of rows in a table.
- **Table Unique (`t_unique`)**: The number of unique values in the join column(s) of a table.
- **Table ID Size (`t_id_size`)**: The size of the identifier or primary key column.
- **Table Row Size (`t_row_size`)**: The size of a row in the table.
- **Table Cache Age (`t_cache_age`)**: The age of the data in the cache, indicating how frequently the table's data is accessed. Higher values indicate higher age. If the value is too high, this might mean that the data was dropped from the cache.
- **Table Cluster Size (`t_cluster_size`)**: The size of table cluster. If this is a native unjoined table, this is 1, when tables join, their cluster sizes sum.
- **Table Bounds (Low, High, Range) (`t_bounds_low`, `t_bounds_high`, `t_bounds_range`)**: The lower bound, upper bound, and range of key values in the table, providing insights into data distribution.

Join-level Features Join-level Features are directly related to the characteristics of the join operations themselves. These features are instrumental in predicting the performance and cost of executing specific join orders.

- **Result Length (`c_len_res`)**: The length of the result column after the join.
- **Result Maximum Length (`c_len_possible_max`)**: The maximum possible length of a column after the join, this is the same as a result of the cross product between two tables.
- **Result Maximum Unique Length (`c_len_unique_max`)**: The maximum possible length of a column if taken cross product between the unique values.
- **Result Selectivity (`c_selectivity`)**: The selectivity of the join condition, indicating the fraction of rows that satisfy the join predicate.
- **Result Cluster Size (`c_cluster_size`)**: The size of clusters formed by the join operation.
- **Result Cluster Overlap (`c_cluster_overlap`)**: The degree of overlap between data clusters in the join operation.

Relative Features Relative features are similar to the table features, but provide a better insight into the relationship between min and max values of table features in a join.

- **Table Ratios (`c_tbl_ratio_length`, `c_tbl_ratio_unique`, etc.)**: Ratios comparing the characteristics of the result set to the original tables, including length, uniqueness, row size, cache age, and bounds range.
- **Table Minimums and Maximums (`c_tbl_min_length`, `c_tbl_max_unique`, etc.)**: The minimum and maximum values for table characteristics post-join, providing bounds for data volume and distribution.

3.4.12 Cost Model Placeholder

The Cost Model Placeholder serves as a critical component within the system's architecture, designed to facilitate the seamless integration and evaluation of diverse cost models for join-order optimization. This flexibility is paramount for a benchmarking system that aims to assess the efficacy of cost models across different execution environments, such as CPU and GPU platforms.

Integration with Operations Interface At the core of the Cost Model Placeholder's design is its compatibility with the Operations Interface, as detailed in Listing 3.4. The Operations Interface defines a contract for essential database operations, including joins and filters, which any cost model must implement to be integrated into the system. This design choice ensures that the system can accommodate a wide range of cost models, each potentially employing different methodologies for cost estimation, without necessitating modifications to the system's core architecture.

Cost Prediction Mechanism The primary function of the Cost Model is to predict the execution cost of each operation defined by the Operations Interface. To achieve this, the placeholder invokes the cost estimation methods provided by the integrated cost model. These methods leverage the extensive set of features collected during the Feature Collection phase, as described in subsection 3.4.11, encompassing Table-Level, Column-Level, and Join-Level features. The richness of these features allows the cost model to make informed predictions about the execution cost of operations, taking into account factors such as data distribution, selectivity, and the characteristics of the execution environment.

Extensibility and Flexibility The design of the Cost Model Placeholder emphasizes extensibility and flexibility, allowing researchers to experiment with different cost models. By abstracting the integration of cost models through the Operations Interface and providing a rich set of features for cost prediction, the system facilitates a dynamic exploration of cost estimation techniques. This design approach not only supports the ongoing evolution of cost models but also contributes to the broader goal of optimizing SQL query execution across diverse hardware platforms.

3.4.13 Plan Enumeration & Search

The proposed system is not meant to be used as a functional DBMS, therefore the plan enumeration and search are not within the priority. However, a simple implementation of a search algorithm is provided, to showcase the potential. This section outlines the approach taken in plan enumeration and the search strategy employed to identify optimal join orders.

Plan Enumeration Involves generating all possible permutations of join orders for a given set of joins. The enumeration process generates all permutations of the sequence `[0, 1, ..., n]`, where `(n)` represents the number of joins. Each permutation represents a potential join order to be evaluated. This exhaustive enumeration serves as the foundation for exploring the search space of join orders, enabling the identification of the most cost-effective execution plan.

Search Strategy To efficiently explore the vast space of potential join orders generated through plan enumeration, a genetic algorithm integrated via the PyGAD Python library [24] is employed. This approach utilizes an inversion mutation technique to introduce variability while preserving the relative ordering of joins, alongside an ordering-preserving crossover method to ensure the validity of offspring join orders. The genetic algorithm iteratively refines a population of join orders, leveraging these genetic operators to navigate toward optimal solutions. This strategy allows for an effective balance between exploration and exploitation of the enumerated join order space, identifying efficient join orders by evaluating their fitness based on estimated execution costs.

3.5 Implementation

The system design detailed in this chapter has been implemented in Python as the Join Benchmark Framework. This choice of programming language was motivated by its rich ecosystem of libraries and frameworks, which significantly facilitated the tasks of SQL parsing, optimization, and execution across both CPU and GPU platforms.

The implementation is hosted on GitHub[1], making it publicly accessible for review, use, and contribution by the broader research and development community.

It is accompanied by comprehensive `README` files that document the setup process, running benchmarks, and integrating new database sets in detail. These guides ensure that users can effectively leverage the framework to its full potential.

The repository structure is thoughtfully organized, containing directories for the core benchmarking system itself, dataset generation tools, and a runtime environment.

The repository encompasses tools for generating datasets for `JOB`, `SSB`, and `TPC-DS` benchmarks, facilitating the testing and evaluation of the system across different scenarios and data volumes.

A dedicated runtime environment, provided as a `Dockerfile`, simplifies the setup process, enabling quick and flexible system deployment on both CPU and GPU platforms.

This implementation, therefore, stands as a practical realization of the system design, offering a versatile toolkit for exploring SQL query optimization, cost model evaluation, and the performance implications of different join orders in diverse execution environments.

[1] https://github.com/marko-matusovic/join-benchmark-toolkit

Chapter 4

Methodology for Benchmarking

This chapter presents a methodology used for the empirical benchmarking of cost models across diverse execution engines and establishes a framework for the subsequent analysis of the collected measurements. It outlines the systematic approach for the empirical measurement of cost model performance and the analysis thereof. The methodology is designed to ensure a rigorous and reproducible evaluation process, enabling a nuanced understanding of cost model efficacy in varying computational contexts.

The primary focus of this methodology is to benchmark cost models, which entails quantitatively determining the performance of a cost model. High-performing cost models can accurately predict costs, which can then be successfully used to search through different execution plans. To select an effective cost model, it is crucial to compare them based on a performance score. This chapter establishes a methodology for calculating this performance score, facilitating the comparison of cost models.

4.1 Benchmarking Metrics

Benchmarking plays a pivotal role in the empirical evaluation of cost models. The primary objective of benchmarking is to determine the performance of cost models in a manner that is both consistent and comparable across different scenarios and systems. This ensures that the findings are robust, and can be reproduced.

Execution time is an essential metric for evaluating the performance of cost models, chosen for two primary reasons. First, it has a direct application to the end-user experience; lower execution times translate to faster query responses, significantly enhancing user satisfaction. Second, execution time is a metric general enough to be applicable across various systems, allowing for the collection and comparison of performance data in diverse computational environments. This universality makes it a good starting measure for assessing the effectiveness of cost models in predicting the most efficient order of operations for join queries.

In the realm of join order optimization, the relative ordering of costs predicted by cost models holds more significance than the absolute values of those costs. This is because the ultimate goal is to identify the most efficient sequence of joins, rather than to determine the absolute execution time of each possible join order. Consequently, this research focuses on evaluating the cost models' ability to accurately rank different join orders in terms of their expected performance. This approach aligns with the practical requirements of query optimization, where the selection of the optimal join order is paramount.

To quantitatively assess the accuracy of cost model predictions in terms of their ordering, Pearson correlation is employed as the scoring mechanism. Pearson correlation is explained in subsection 2.2.3. Predictions that highly correlate with real execution times can be used to accurately search the possible join orders, thereby proving the cost model effective. This makes Pearson correlation a good choice for evaluating the predictive quality of cost models in the context of join order optimization.

Moreover, the utilization of Pearson correlation offers a significant advantage in the context of this benchmarking methodology, which aims to accommodate a broad spectrum of cost models. One of the primary benefits of correlation over accuracy in this scenario is its independence from the units of measurement. Accuracy assessment necessitates that the cost predictions and the measured metric, in this case, execution time in milliseconds, be expressed in the same units, and the same scale. However, Pearson correlation circumvents this requirement by evaluating the strength and direction of a linear relationship between two variables, regardless of their units, or order of magnitude. This characteristic of correlation allows for the inclusion and benchmarking of diverse cost models, including those that quantify cost in abstract or non-standard units. Consequently, employing Pearson correlation as the evaluation metric enhances the methodology's flexibility, enabling the benchmarking of a wider array of cost models without necessitating uniformity in their output units.

4.2 Data Collection & Processing

A systematic approach to data collection is essential for an accurate evaluation of a cost model's performance. The system outlined in chapter 3 plays a crucial role by providing two key functionalities: measuring the real execution times of some join order and estimating the corresponding cost prediction. This dual capability facilitates the comprehensive data acquisition necessary for the analysis.

The process for collecting this data and subsequently deriving the performance score of a cost model encompasses several steps:

1. Initially, for a predefined set of join orders, both the actual execution times and the cost model predictions are collected. This step forms the foundation by supplying the raw data essential for the analysis.
2. The collected data is then sorted according to the actual execution times of the join orders. This organization is critical as it reflects the true performance of each join sequence, establishing a benchmark for evaluating the predictive accuracy of the cost model.
3. With the data sorted, the Pearson correlation between the cost predictions and the actual execution times is calculated. This coefficient quantifies the linear correlation between the two datasets, with a high value indicating that the cost model's predictions closely mirror real-world performance. This correlation serves as a measure of the model's effectiveness in predicting efficient join orders.
4. This methodology is applied consistently across a diverse set of queries to ensure a comprehensive and robust evaluation of the cost model's performance.

5. Finally, a weighted average of the Pearson correlation coefficients is calculated, considering the number of join orders in each query. This step ensures that the overall performance score reflects the complexity and variability of different queries, offering a nuanced view of the cost model's predictive accuracy.

This structured methodology enables a quantitative assessment of cost models in predicting the most efficient join orders, providing valuable insights into their effectiveness in optimizing query execution.

4.3 Datasets

To ensure a comprehensive evaluation of the cost model's performance, it is important to utilize representative datasets that encompass a wide variety of scenarios. This diversity is crucial for obtaining a benchmark score that accurately reflects the cost model's efficacy across different data processing contexts. While obtaining real-world datasets covering diverse scenarios can be challenging due to constraints like data privacy and availability, leveraging existing benchmark datasets can be advantageous. This thesis will use three datasets.

Join Order Benchmark (JOB) [17] The JOB uses real-world data from IMDb, offering a complex query workload that includes analytical queries. Its diverse range of join operations and intricate schema make it ideal for testing join-order optimization strategies and evaluating the cost model's performance, particularly in scenarios that involve complex analytical queries derived from real-world data.

Star Schema Benchmark (SSB) [20] The SSB, with its synthetically generated data, is designed to test the performance of star schema joins common in analytical processing within data warehousing. It is based on TPC-H, which is known for its focus on decision support systems, thereby allowing for controlled scalability testing and exploration of cost model behavior across various data volumes. This makes SSB particularly relevant for analytical queries in a data warehousing context.

TPC-DS [31] TPC-DS, also utilizing synthetic data, complements SSB by providing a broad spectrum of query types and data models, including those relevant to analytical queries in a simulated real-world business environment. Its diverse query set and varied data distribution patterns offer a comprehensive testing ground for join-order optimization techniques and for evaluating the cost model's adaptability to different analytical scenarios.

4.4 Query Selection

In the pursuit of optimizing join order, it became imperative to judiciously select queries from the Join Order Benchmark (JOB), Star Schema Benchmark (SSB), and TPC-DS datasets. Given the broad focus of these datasets, especially the TPC-DS which encompasses over 100 queries aimed at a wide array of database operations, a comprehensive inclusion of all queries was neither feasible nor aligned with the thesis's objectives. The selection criteria were meticulously designed to ensure that the chosen queries are representative of scenarios where join order optimization could significantly impact performance.

The primary criteria for query selection were as follows:

- **Supported Operations:** Only queries that involve operations supported by the cost models under evaluation were considered. This ensures that the findings are directly applicable to the optimization techniques being tested.
- **Absence of Sub-queries:** To maintain focus and simplicity in the evaluation process, queries containing sub-queries were excluded.
- **Presence of Filters and Joins:** Queries selected needed to include a mix of filtering operations and joins. This mix is crucial for evaluating the cost model's ability to optimize join orders in the presence of varying conditions and constraints.
- **Equi-joins Only:** The selection was limited to queries that perform equi-joins and where joins are explicitly defined in the WHERE clause. This criterion was chosen because equi-joins are not only fundamental to join order optimization but also highly representative of common query patterns in real-world applications. Their prevalence in typical database operations makes them essential for a comprehensive evaluation of optimization strategies.

Based on these criteria, the selected queries[1] for evaluation are:

- **SSB:** 21, 31, 41, 42, 43
- **JOB:** 2A, 6F, 9D, 12A, 14A
- **TPC-DS:** 27, 50

This focused selection ensures the evaluation targets scenarios where join order optimization is both applicable and beneficial, allowing for a concentrated examination of the cost models' performance in optimizing join operations, crucial in query execution times for complex analytical queries derived from real-world data.

[1] You can find full queries in Appendix A.

Chapter 5

Experiment & Analysis

Building upon the System Design in chapter 3 and the Methodology for Benchmarking in chapter 4, this chapter gives an example of their practical application. It describes an experiment, conducted as a demonstration of the potential and capabilities of the system. The experiment identifies key features for join-order optimization cost models and highlights the differences in their significance between CPU and GPU systems. This chapter is structured to first introduce the experiment, detailing the setup and the methodology employed. Following this, the results of the experiment are presented and analyzed.

5.1 Motivation

Cost models utilize a set of input metrics, known as features, to estimate the total cost of executing a query. The significance of investigating these key features in cost models across different environments arises from several critical aspects.

Firstly, the accuracy of cost models is dependent on the selection and application of these features. They encapsulate the characteristics of both the execution environment and the operations being performed, serving as the foundation for any cost prediction. A deep understanding of how these features impact cost predictions is crucial for the design of both analytical and learned cost models. For analytical models, designers must have a detailed comprehension of the execution engine's inner workings to discern which features most significantly influence costs. Similarly, for learned cost models, choosing an appropriate set of input features is the first important step. Choosing a set of known good features helps to avoid the "garbage in, garbage out" principle.

Analyzing key features in cost models across CPU and GPU environments is crucial for optimizing query execution. Through comparative analysis, the experiment in this chapter illustrates how architectural variances influence operational costs, aiding in the formulation of more accurate and suitable cost models. Such insights enable database designers to significantly improve system performance across various hardware environments.

5.2 Methodology

In this section, the methodology employed to carry out the experiment is explained. The approach is structured into three primary stages, each designed to progressively uncover the significance of different features in estimating the cost of SQL query execution. The stages are as follows:

1. A baseline cost model is established, utilizing cardinality estimates as its foundational metric. The performance of this model is then evaluated across both CPU and GPU systems to establish a baseline for comparison.
2. Following the baseline model, a series of comparison cost models are constructed. Each model pairs the baseline metric with one distinct additional feature. Then each pair is evaluated independently. This methodical pairing and subsequent evaluation allow for the precise assessment of the impact each individual feature exerts on the model's performance, without the confounding effects of feature accumulation. The performance of these pair-based models is assessed in a manner akin to the baseline, enabling direct comparisons.
3. The concluding stage encompasses a thorough comparison between the pair-based models and the baseline model. This comparative analysis is aimed at discerning the significance of each additional feature within the context of cost models for CPU and GPU systems. Through the examination of performance disparities among the models, insights are gleaned regarding which features significantly enhance the accuracy of predicting query execution costs across varied hardware environments.

This methodology provides a structured framework for assessing the efficacy of various features in cost models, thereby enabling the identification of key factors that influence the accuracy of cost predictions in SQL query execution.

5.3 Experiment Setup

This section explains the setup for the experiment. It references the features used in the cost models, the design of the baseline cost model and well as all comparison cost models. Moreover, it gives insights into the hardware used and the software environment used.

Features

In this experiment, all collectible features by the system designed in this thesis were utilized, as detailed in subsection 3.4.11. These features are organized into three categories: table-level features, join-level features, and relative features.

In this experiment, the table-level features now reduce into the sum of both tables involved in a join. The distinction between table features and relative features proves beneficial, especially given the model's constraint on the number of input features. Relative features, in particular, offer a deeper understanding of individual statistics, underscoring their importance in this study.

Cost Model Design

Cost models in this experiment are structured to sum the predicted costs for each operation within a query. The baseline cost model is formulated on the principle that the cost of an operation can be directly inferred from a single feature.

Baseline Cost Model The feature used in the baseline cost model is the `t_length` feature, representing the sum of the lengths of both tables involved in a join operation. The choice of `t_length` as the foundational feature is grounded in its prevalent

use in existing cost models, attributed to its direct correlation with the computational effort required for processing table joins. Consequently, in the baseline model, the cost is equated to the sum of values of the `t_length` feature for both tables.

Comparison Cost Models In contrast to the baseline model, the comparison cost models introduce additional features alongside the baseline `t_length` feature. These models are constructed by pairing `t_length` with another feature, and the cost of each operation is determined by a linear function of these input features. The formula for calculating the cost in comparison models is encapsulated in Equation 5.1, where `feature_1` is `t_length`, and `feature_2` represents the additional feature paired with `t_length` for each model. This approach allows for better estimation of costs by considering the combined effects of multiple features on the computational requirements of query operations.

$$\text{cost}_{\text{operation}} = \text{coef}_1 \times \text{feature}_1 + \text{coef}_2 \times \text{feature}_2 \tag{5.1}$$

Each feature was systematically paired with `t_length` to generate a series of comparison cost models. This methodology facilitates a comprehensive evaluation of how additional features when considered alongside the baseline feature, influence the accuracy of cost predictions. Through this comparative analysis, the models aim to identify which combinations of features yield the most precise estimations of query execution costs.

Hardware & Software

This section documents the experiment setup. It explains the hardware configurations employed at various stages of this research, including development, benchmarking, and evaluation, and also outlines the software used with a focus on the ease of reproducibility of the experiment.

Hardware The selection of hardware is critical, as it directly influences the reproducibility of results and the applicability of the proposed cost models across different computational environments. The following hardware platforms were utilized:

- **Personal MacBook Pro 13" 2018** Used as primary device for development and testing. Due to limited performance, data collection and benchmarking were not performed on this device. Also, this device does not contain a dedicated GPU, so cuDF which requires an NVIDIA GPU was not available.
- **TUDelft st4 GPU server** was utilized specifically for generating cost model predictions. Given its shared nature, this server did not offer a consistent benchmarking environment, which was a non-issue for tasks that did not require uniform execution times. Its primary advantage lies in the ease and speed of access for generating results, particularly benefiting from the server's NVIDIA GPUs for compute-intensive tasks.
- **Amazon EC2 G5.xl** was chosen for the precise measurement of execution times. The selection was based on Amazon EC2's ability to provide a highly consistent and widely comparable benchmarking environment. This consistency is essential for the accurate and reliable comparison of execution times across different configurations and runs, ensuring the integrity of the collected execution time data.

Each platform offers unique characteristics in terms of CPU and GPU capabilities, memory, and processing power, providing a varied landscape for evaluating the proposed methodologies. The subsequent sections provide detailed specifications and the rationale behind the selection of each hardware configuration.

Software The runtime environment, accessible via the project's GitHub repository as described in section 3.5, is encapsulated within a Docker image. This image comprises all essential Python libraries and drivers necessary for supporting both Pandas and cuDF frameworks. The provision of this Docker image facilitates the swift setup of the experimental environment, thereby enhancing the reproducibility of the experiment. Additionally, the repository includes scripts designed for the generation of datasets used during the benchmarking process, further aiding in the reproducibility of results. Automated scripts for the collection, processing, and visualization of experimental data are also provided, streamlining the analysis process.

5.4 Evaluation Plan

The objective of this evaluation plan is to identify the significant features of cost models for execution engines in CPU and GPU environments and to understand the differences in those key features. The plan is structured into three main steps:

1. **Collecting Data:**
   All data is collected using the designed system, and its intended use cases 1 and 2, as detailed section 3.3.

   (a) *Use Case 1:* Execution times for SQL queries are generated on both CPU and GPU systems to establish a baseline for the actual performance of query execution across the two types of hardware. These execution times are critical for evaluating the accuracy of cost predictions made by the various cost models.

   (b) *Use Case 2:* Cost predictions are collected for all cost models, including the baseline and comparison models. This step focuses on generating predicted costs for executing SQL queries using the different cost models designed within the system. The predictions from these models are then prepared for comparative analysis against the actual execution times obtained from use case 1.

2. **Benchmarking Cost Models:** This step follows the benchmarking methodology, as described in chapter 4, to calculate the correlation scores. Specifically, this process entails computing the correlation between the cost predictions generated by each cost model (including both the baseline model and its variants) and the actual execution times recorded for SQL queries. These calculations are performed separately for each engine profile, namely CPU and GPU, resulting in a distinct correlation score for every combination of cost model to engine profile. The derived correlation score for each pairing serves as a quantitative measure, indicating the precision with which a given cost model can predict the execution costs associated with SQL queries on the specified hardware environment.

3. **Comparing the Scores:**

(a) *Percentage Gain:* For each hardware environment, the scores of all comparison cost models are compared to the baseline cost model score. The percentage gain in accuracy is calculated by determining the difference in correlation scores between the comparison models and the baseline model, expressed as a percentage of the baseline score. This metric highlights the improvement in prediction accuracy achieved by the comparison models over the baseline model.

(b) *Cross-Hardware Comparison:* The percentage gain for the same cost model is compared between CPU and GPU environments. This comparison aims to identify how the effectiveness of a given cost model varies across different hardware environments, providing insights into the model's adaptability and performance under varying computational conditions.

Through this structured approach, the evaluation plan aims to provide a detailed analysis of the system's capability in accurately predicting SQL query execution costs, thereby facilitating the optimization of query execution across diverse hardware environments. The significant features of cost models for execution engines in both CPU and GPU environments are identified, along with the differences in those key features.

5.5 Results

This section presents the outcomes derived from executing the methodology and following the evaluation plan. It details the collected data, emphasizing the performance of various features within the cost models across CPU and GPU systems. Insights into the effectiveness of these features, based on empirical data, are aimed at enhancing the understanding of SQL query execution optimization through the proposed system, without delving into the analysis of these results.

5.5.1 Collecting Data

First, data must be collected. Execution times and cost predictions are collected for all queries as defined in the benchmarking methodology in section 4.4. And all data is included in Appendix B. However, for explanation purposes, predominantly two queries will be included in the next section in detail. Those queries are JOB query 2A and TPC-DS query 50.

Use Case 1: Execution Times

In this step, the execution times for different join orders must be collected. First, detailed data of one join order is explained, and then a summary is abstracted showing multiple join orders.

Consider the following examples of execution times for different stages in query execution, as shown in Figure 5.1, 5.2, 5.3, and 5.4. These four bar charts show the difference in execution time for various stages in the query execution. The stages are ordered in the same order as they are executed. For plotting purposes the Y axis is logarithmic since the "overhead" stage takes comparably longer. In this stage, the CSV files are parsed and read in memory. The charts show two series, these are the execution times for the query on CPU and GPU.

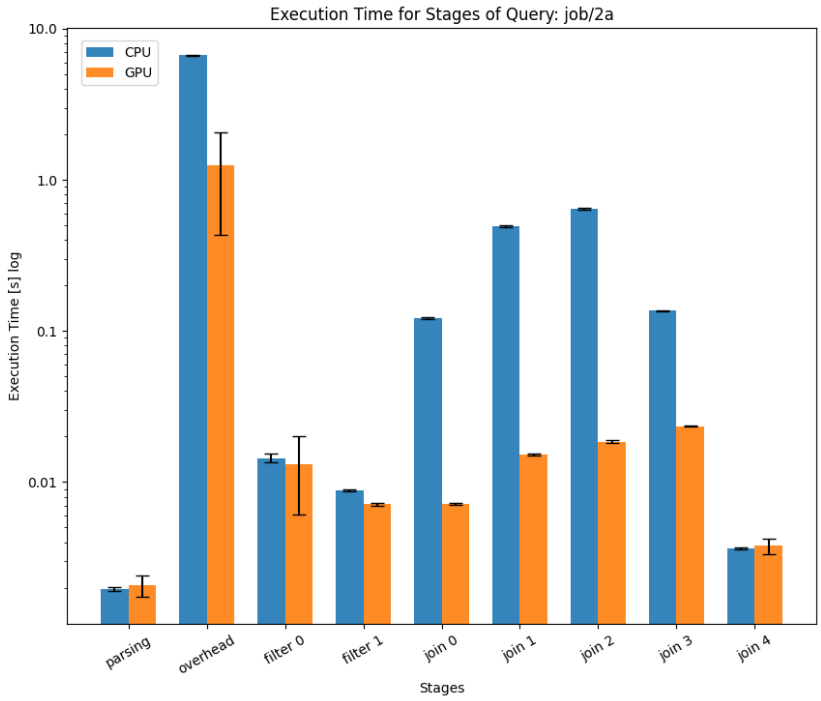


FIGURE 5.1: Stages of Query JOB 2A

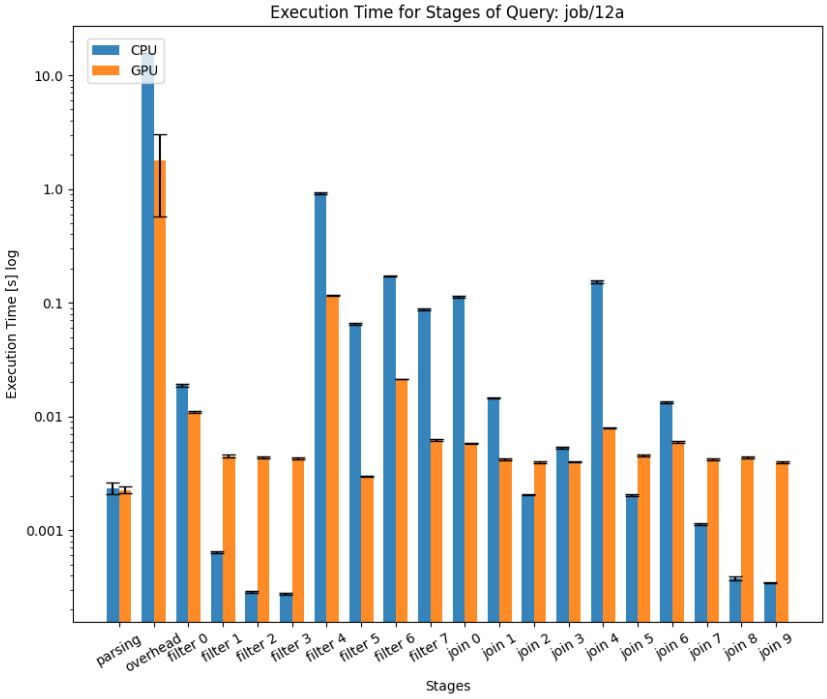


FIGURE 5.2: Stages of Query JOB 12A

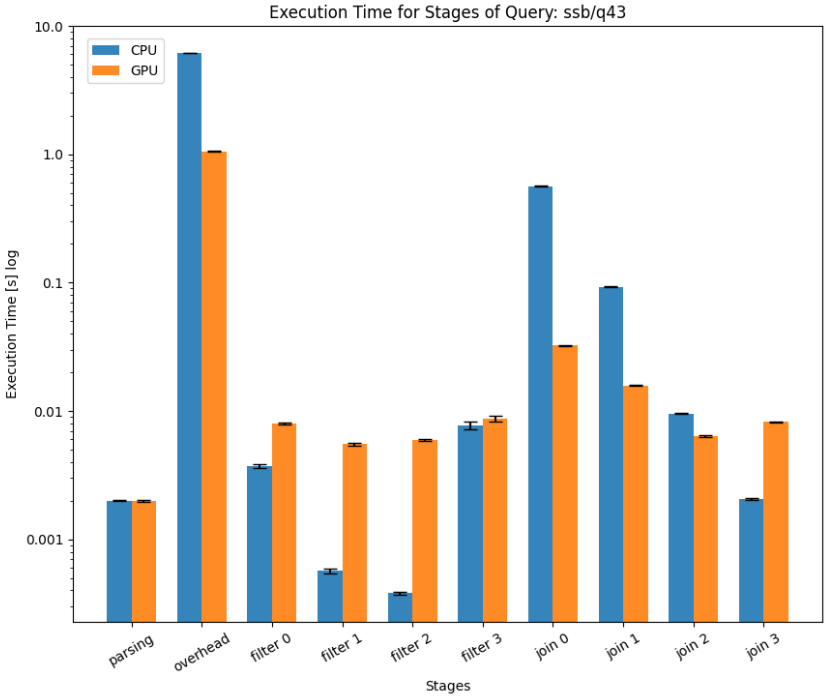


FIGURE 5.3: Stages of Query SSB 43

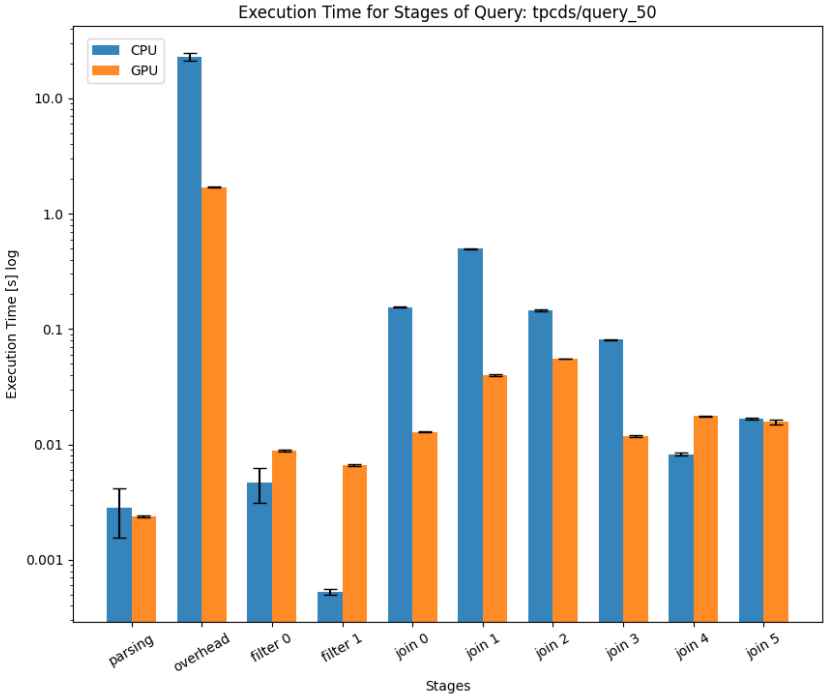


FIGURE 5.4: Stages of Query TPC-DS 50

While these charts alone, do not provide much information, there are a couple of interesting observations. In Figure 5.3, notice that the execution time of consequent joins is going down on CPU, however on GPU, this cost never drops below a certain level. An explanation here is that executing operations on GPU, such as joins, requires a certain overhead, and while joins of small tables are fast on the CPU, because of this overhead, the join takes longer on GPU.

The focus of this experiment is cost models for join order optimization, therefore only the join times are relevant. Consider the charts in Figure 5.5, 5.6, 5.7, and 5.8. These charts go one abstraction level above the previous charts. Each chart shows the variance in the total execution time of joins for one query on some execution engine. The variance is spread through different join orders. Two different execution engines are shown in separate figures, this is because the join orders are ordered by the total execution time. And because the ordering is different for CPU and GPU, the charts cannot be shown together. Note the total times are much lower on GPU than on CPU, this shows how the GPU DataFrame library cuDF is targeting faster execution times.

Use Case 2: Cost Predictions

In this step, the measured execution times of various join orders must be complemented with predicted execution costs. Consider Figure 5.9, 5.10, 5.11, and 5.12, where cost predictions are added alongside the execution times from the previous section. The predictions in these figures are calculated using the baseline cost model, as described in section 5.3.

Additionally, the Pearson correlation coefficient is calculated, shown at the top left in each figure, to measure how closely the predicted and actual costs match. A coefficient close to one indicates a strong match, showing the cost model's effectiveness in estimating the computational costs of different join orders. Correlation coefficients for all

Observing a single chart in isolation provides limited insights. However, comparing multiple charts enables a deeper understanding of the impacts of changing the cost model or the execution engine on correlation. For example, cardinality estimates correspond more closely with execution times on the GPU for query JOB 2A, whereas for query TPC-DS 50, the cost predictions correlate stronger with the CPU system than with the GPU. This observation highlights the importance of including a diverse set of queries within the cost model benchmarking methodology, as detailed in chapter 4, to facilitate a comprehensive evaluation of the cost model's overall correlation.

5.5.2 Benchmarking Cost Models

In previous steps, the correlation of execution times to cost predictions was calculated for different queries, cost models, and execution engines. This step determines the benchmark score of cost models given some execution engine. You can find all the calculated correlations for all cost models in Appendix B. Abstracted correlation for each cost model is shown in Figure 5.13.

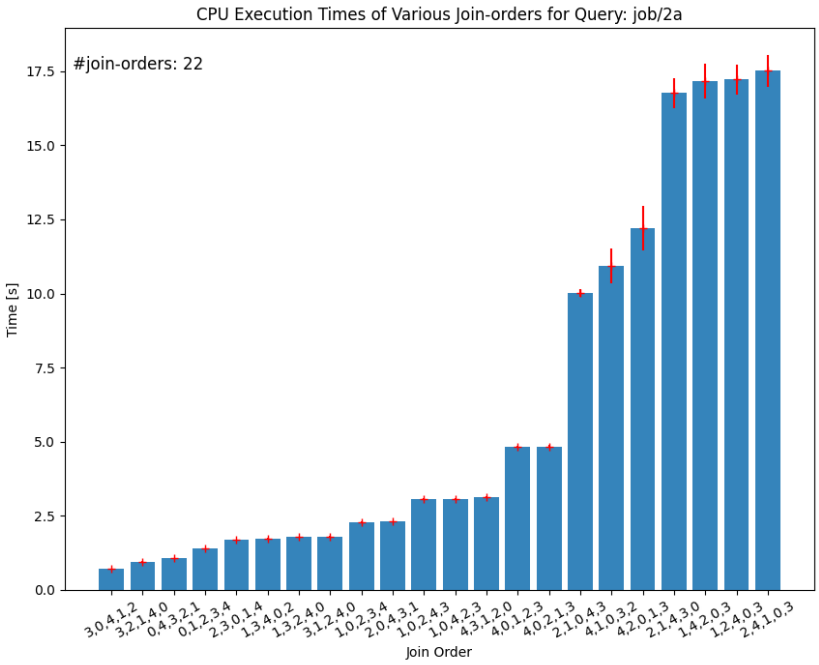


FIGURE 5.5: Various Join-orders for Query JOB 2A (CPU)

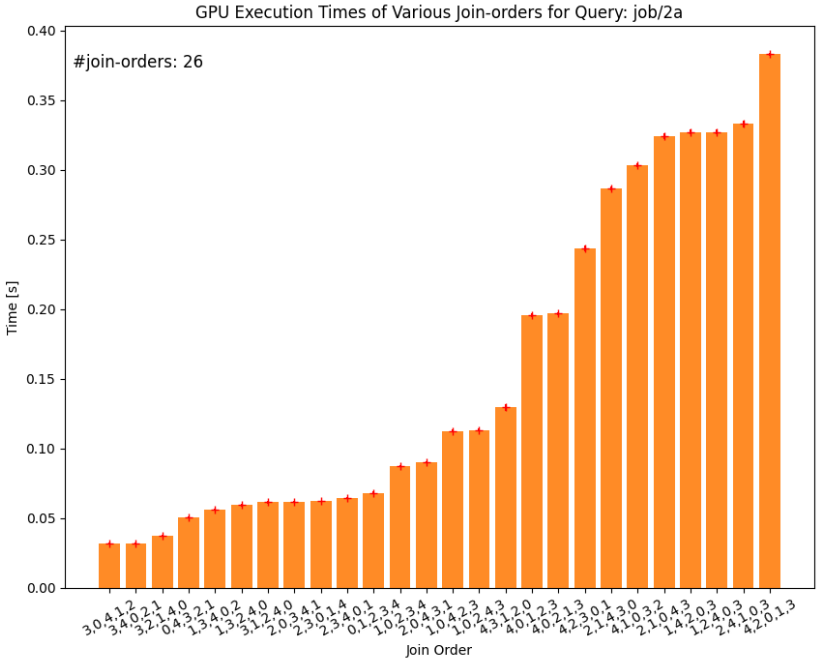


FIGURE 5.6: Various Join-orders for Query JOB 2A (GPU)

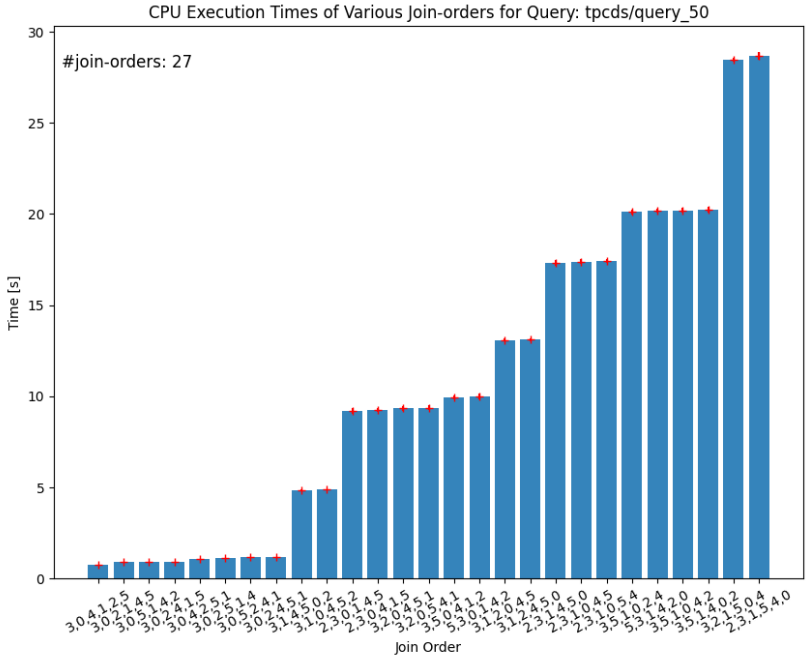


FIGURE 5.7: Various Join-orders for Query TPC-DS 50 (CPU)

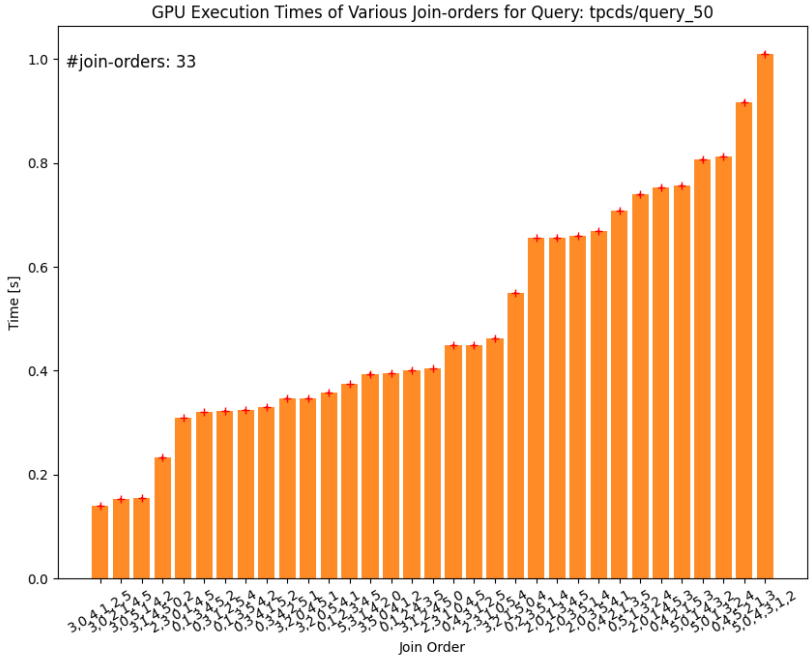


FIGURE 5.8: Various Join-orders for Query TPC-DS 50 (GPU)

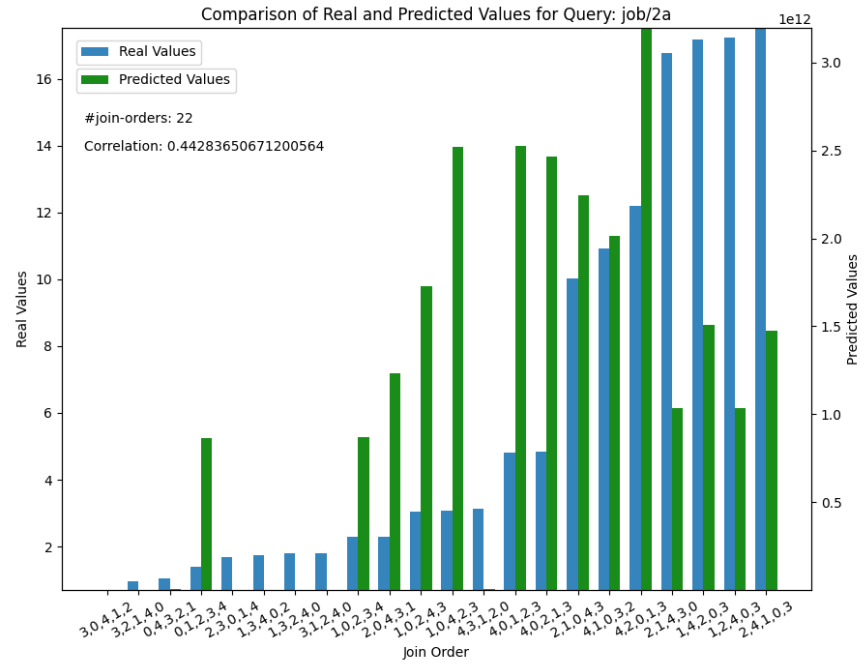


FIGURE 5.9: Various Join-orders for Query JOB 2A (CPU) + Cost Model

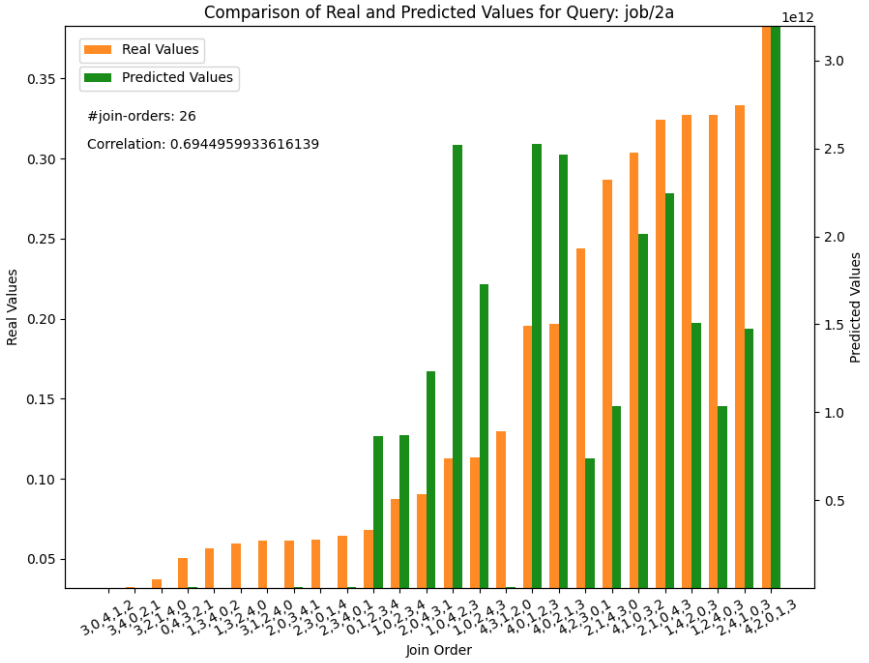


FIGURE 5.10: Various Join-orders for Query JOB 2A (GPU) + Cost Model

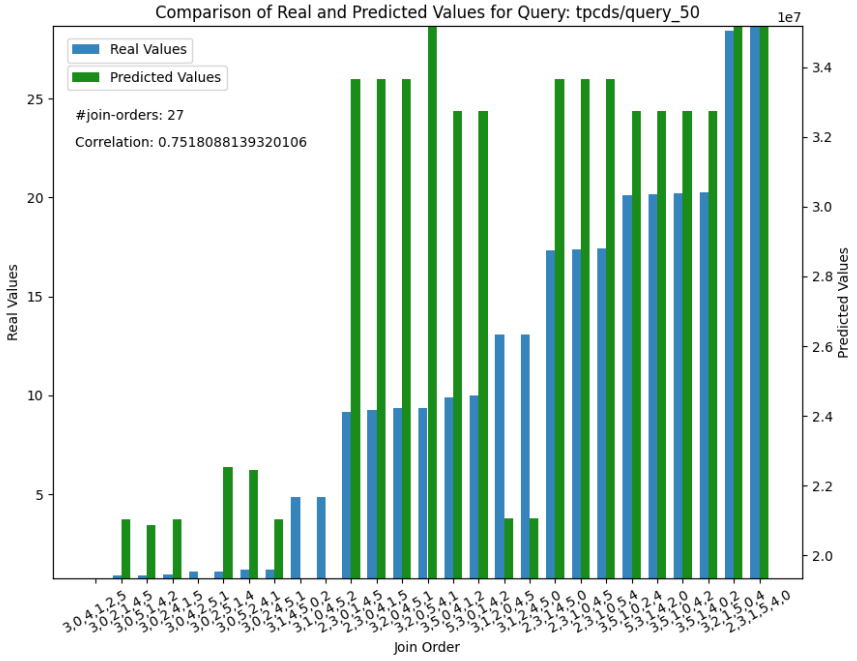


FIGURE 5.11: Various Join-orders for Query TPC-DS 50 (CPU) + Cost Model

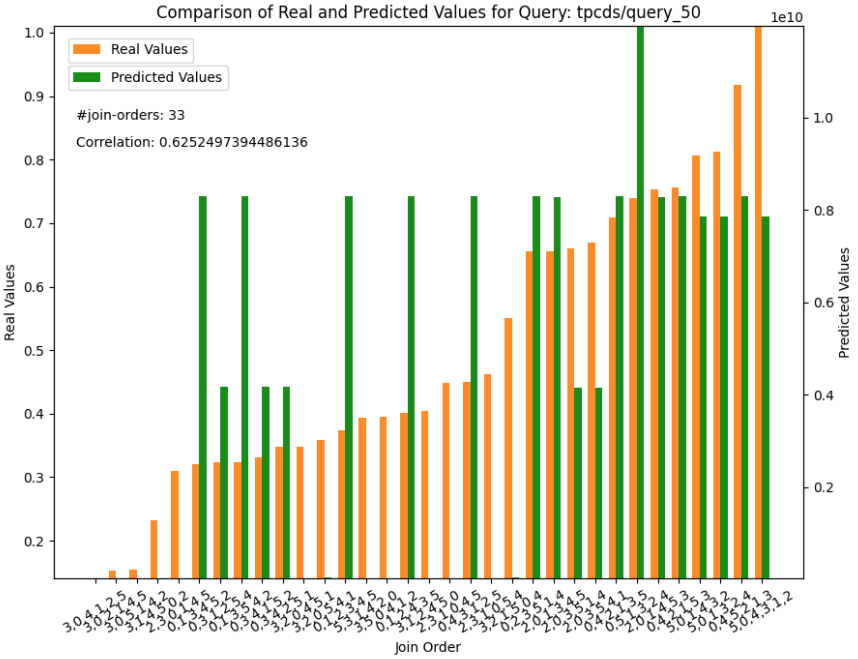


FIGURE 5.12: Various Join-orders for Query TPC-DS 50 (GPU) + Cost Model

	FEATURE	CPU	GPU
	baseline	0,28713868	0,33065836
Table-level	t_unique	0,28749774	0,33368100
	t_id_size	0,28847492	0,33367849
	t_row_size	0,28748704	0,36911600
	t_cache_age	0,28788000	0,33845344
	t_cluster_size	0,28743254	0,34128688
	t_bounds_low	0,29636769	0,33648421
	t_bounds_high	0,29206621	0,33648777
	t_bounds_range	0,28902698	0,33418237
Join-level	c_len_res	0,30049587	0,33116647
	c_len_possible_max	0,31492544	0,33394749
	c_len_unique_max	0,28784896	0,34399719
	c_selectivity	0,30085582	0,35712950
	c_cluster_size	0,28738645	0,33964979
	c_cluster_overlap	0,28956873	0,33199039
Relative	**c_tbl_ratio_length**	0,33839827	0,36722063
	c_tbl_ratio_unique	0,29637753	0,33839162
	c_tbl_ratio_row_size	0,29549365	0,36894394
	c_tbl_ratio_cache_age	0,28847279	0,33192574
	c_tbl_ratio_bounds_range	0,28787822	0,34547797
	c_tbl_min_length	0,30460677	0,33316296
	c_tbl_min_unique	0,28720952	0,33079942
	c_tbl_min_row_size	0,29365939	0,38780454
	c_tbl_min_cache_age	0,28722941	0,33287062
	c_tbl_min_bounds_range	0,28757255	0,33988195
	c_tbl_max_length	0,29986150	0,33116626
	c_tbl_max_unique	0,28714763	0,33107320
	c_tbl_max_row_size	0,28720675	0,37869506
	c_tbl_max_cache_age	0,28722941	0,33287062
	c_tbl_max_bounds_range	0,28757255	0,33988195

FIGURE 5.13: Correlation of Various Cost Models with CPU and GPU Execution Engines

5.5.3 Comparing the Scores

$$\text{gain} = \frac{\text{score(comparison cost model)}}{\text{score(baseline cost model)}} \tag{5.2}$$

To identify comparison cost models, the scores of those cost models must be compared to the baseline cost model. The percentage gain is calculated as shown in Equation 5.2. The comparison of calculated percentage gain in correlation for all comparison cost models is shown in Figure 5.14, and 5.15.

Both of these figures display the same data, however they do so in a different manner. Figure 5.14 shows a simple overview in a bar chart, with the gain for an extra feature on CPU on the left, and gain on GPU on the right. This chart provides a visual overview by employing column size to represent the percentage gain in correlation, where taller columns indicate a larger gain, thereby quickly highlighting the significance of each feature.

Secondly, Figure 5.15 shows the margin gain in a table, with separate columns for CPU and GPU environments. The values are also color-coded for a faster visual representation. Additionally, notice the two columns on the right, which have a filtered list of significant features for CPU and GPU, the selected features have all over +5% increase over the baseline correlation.

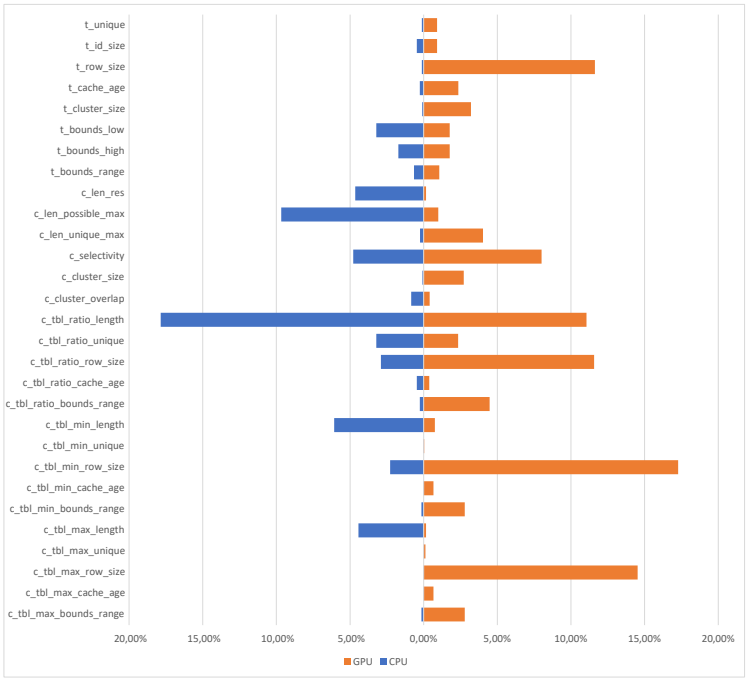


FIGURE 5.14: Correlation Gain of Various Cost Models with CPU and GPU Execution Engines (Bar Chart View)

	EXTRA FEATURE	CPU	GPU	CPU significant	GPU significant
Table-level	t_unique	0,13%	0,91%		
	t_id_size	0,47%	0,91%		
	t_row_size	0,12%	**11,63%**		**t_row_size**
	t_cache_age	0,26%	2,36%		
	t_cluster_size	0,10%	3,21%		
	t_bounds_low	3,21%	1,76%		
	t_bounds_high	1,72%	1,76%		
	t_bounds_range	0,66%	1,07%		
Join-level	c_len_res	4,65%	0,15%		
	c_len_possible_max	**9,68%**	0,99%	**c_len_possible_max**	
	c_len_unique_max	0,25%	4,03%		
	c_selectivity	4,78%	**8,01%**		**c_selectivity**
	c_cluster_size	0,09%	2,72%		
	c_cluster_overlap	0,85%	0,40%		
Relative	c_tbl_ratio_length	**17,85%**	**11,06%**	**c_tbl_ratio_length**	**c_tbl_ratio_length**
	c_tbl_ratio_unique	3,22%	2,34%		
	c_tbl_ratio_row_size	2,91%	**11,58%**		**c_tbl_ratio_row_size**
	c_tbl_ratio_cache_age	0,46%	0,38%		
	c_tbl_ratio_bounds_range	0,26%	4,48%		
	c_tbl_min_length	**6,08%**	0,76%	**c_tbl_min_length**	
	c_tbl_min_unique	0,02%	0,04%		
	c_tbl_min_row_size	2,27%	**17,28%**		**c_tbl_min_row_size**
	c_tbl_min_cache_age	0,03%	0,67%		
	c_tbl_min_bounds_range	0,15%	2,79%		
	c_tbl_max_length	4,43%	0,15%		
	c_tbl_max_unique	0,00%	0,13%		
	c_tbl_max_row_size	0,02%	**14,53%**		**c_tbl_max_row_size**
	c_tbl_max_cache_age	0,03%	0,67%		
	c_tbl_max_bounds_range	0,15%	2,79%		

FIGURE 5.15: Correlation Gain of Various Cost Models with CPU and GPU Execution Engines (Table View)

5.6 Analysis

The conducted experiment has provided valuable insights into the significance of various features in cost models for SQL query execution on CPU and GPU systems. This section analyzes these findings, focusing on the features that have shown a significant impact on the accuracy of cost predictions.

5.6.1 Significant Features on CPU

The feature `c_tbl_ratio_length` emerged as the most significant on CPU systems, alongside other notable features such as `c_len_possible_max` and `c_tbl_min_length`. `c_len_res` and `c_selectivity` also demonstrated potential impact, albeit slightly below the significance threshold.

`c_tbl_ratio_length` complements the baseline cost model by offering insights into the relative sizes of tables in a join. This ratio is crucial for assessing join efficiency, especially when there is a significant size disparity between the tables. Efficient joins are facilitated by scanning the smaller table first, thereby minimizing disk or index accesses when probing the larger table, utilizing the limited cache better. For instance, a join operation starting with a table of 100 rows and then with a table of 2000 rows necessitates fewer index lookups compared to beginning with the larger table. Moreover, in scenarios where the join selectivity is close to 1, leading to an operation akin to a cross-product, the table size ratio becomes even more critical. A 1 to 1 ratio implies a significantly larger result set compared to a disproportionate ratio, such as 1 to 1000, highlighting the importance of `c_tbl_ratio_length` in predicting execution costs by accounting for the dynamics of table size differences.

`c_len_possible_max` and `c_tbl_min_length` are significant due to the CPU's sensitivity to data volume. The maximum possible length of a column after a join (`c_len_possible_max`) represents the worst-case scenario in terms of data volume, directly impacting computational effort. Similarly, the minimum length of tables involved in a join (`c_tbl_min_length`) influences the efficiency of join operations, affecting performance on CPU architectures.

5.6.2 Significant Features on GPU

In the context of GPU systems, `c_tbl_min_row_size` emerged as the most significant feature, with other row size-related features (`t_row_size`, `c_tbl_ratio_row_size`, and `c_tbl_max_row_size`) and `c_selectivity` also demonstrating significant impact.

`c_tbl_min_row_size` is crucial for GPU architectures due to their limited memory capacity compared to CPUs. This feature's importance stems from its impact on join operation efficiency. In cases where both tables in a join have large row sizes, selecting the table with smaller cardinality to optimize cache utilization does not provide much benefit. This directly affects GPU performance in join operations, as handling large data volumes is slower due to memory constraints. Thus, `c_tbl_min_row_size` influences the ability to conduct efficient joins on GPUs by addressing memory capacity limitations.

Other Row Size Features: The size of a row (`t_row_size`) and its other variations (`c_tbl_ratio_row_size` and `c_tbl_max_row_size`) are crucial for GPUs due to their parallel processing capabilities. These features influence the amount of data processed in parallel and the memory bandwidth required, affecting the utilization of GPU cores and overall execution cost.

`c_selectivity` remains significant across both CPU and GPU systems, highlighting its universal importance in determining the efficiency of join operations. For GPUs, the ability to perform massive parallel filtering operations makes selectivity a key determinant of leveraging the GPU's parallelism effectively.

5.6.3 Proposed Explanation

The distinction in architectural design between CPUs and GPUs significantly influences their respective sensitivities to different features in SQL query execution cost models. CPUs, characterized by a linear processing model, exhibit a pronounced sensitivity to data cardinality. This is primarily due to the sequential nature of CPU processing, where the volume of data directly impacts execution time. Features such as `c_tbl_ratio_length` and `c_len_possible_max` become critical in this context, as they provide insights into the volume and potential maximum size of data involved in operations, respectively.

CPU Sensitivity to Cardinality: For CPUs, the efficiency of data access and processing is paramount. The linear processing model means that operations on data sets with high cardinality inherently consume more time, as each piece of data is processed sequentially. This characteristic underscores the importance of length features, such as `c_tbl_ratio_length`, which indicates the relative sizes of tables in a join, and `c_len_possible_max`, representing the worst-case scenario in terms of data volume after a join. These features are pivotal in predicting execution costs on CPUs, as they directly relate to the volume of data being processed.

Conversely, GPUs leverage a parallel processing model, enabling them to handle multiple operations simultaneously. This capability allows GPUs to be less affected by high cardinality. However, the efficiency of this parallelism is contingent upon optimal memory and cache utilization. Given GPUs' typically lower memory capacities and fewer cache levels compared to CPUs, the organization and size of data become crucial. Features such as `t_row_size` and `c_tbl_min_row_size` are significant for GPUs, as they influence memory access patterns and the efficiency of cache utilization.

GPU Sensitivity to Memory and Cache Utilization: The parallel processing capability of GPUs necessitates data to be organized in a manner that maximizes memory access efficiency. Misaligned data or inefficient access patterns can lead to increased memory access times and underutilization of GPU cores. Therefore, row size features, specifically `t_row_size` and `c_tbl_min_row_size`, are indicative of performance on GPUs. These features directly impact how effectively the GPU's memory hierarchy is utilized, influencing the overall execution cost of SQL queries.

This analysis underscores the need to tailor cost models to the specific characteristics of CPUs and GPUs, taking into account their processing models, memory limitations, and data transfer efficiencies.

Chapter 6

System Evaluation

This chapter presents an evaluation of the benchmarking system designed in chapter 3, the accompanying methodology as described in chapter 4, and the experiment documented in chapter 5. This chapter aims to assess the effectiveness of measuring the performance of various cost models across different execution engines, as outlined in the research questions. The chapter begins by examining how well the system meets its design requirements, followed by an analysis of its use cases. Furthermore, this chapter discusses the limitations encountered during the system's development and evaluation, offering a critical perspective on the challenges faced. Finally, it outlines potential directions for future work, suggesting ways to extend and improve upon the current system.

6.1 Requirements Evaluation

This section assesses how well the developed benchmarking system meets the established functional and non-functional requirements. This evaluation is crucial for verifying the system's effectiveness in benchmarking cost models across various execution engines. This process aims to identify areas of success, and potential improvements, and ensure the system's alignment with the intended design goals and user needs.

Functional Requirements

FR 1 **SQL Parsing**: ✓

The system supports basic parsing of SQL schema and queries. The focus of parsing queries is the $WHERE$ clause, in which basic selection operations are supported ($=, \neq, <, \geq, >, \leq$, `IN`, `IS`, `BETWEEN`, `LIKE`) in conjunctive normal form, and equijoins. All group, order, limit, and reflection operations are left for future implementation, and sub-queries are also not implemented.

FR 2 **Cost Model Integration**: ✓

The system provides an interface for external cost models, facilitating easy integration and testing of new models.

FR 3 **Execution Engine Compatibility**: ✓

The system integrates with the DataFrame interface. It is capable of using any DataFrame framework as an operation simulator to run an SQL query.

This is achieved by developing an operation mapper. The supported operations are the same as the operations mentioned in the SQL parsing requirement evaluation.

FR 4 **Input Feature Support**: ✓

The system collects DB statistics, and performs histogram-based cardinality estimation. It then provides a wide set of data and hardware features that are available to the cost model.

FR 5 **Cost Prediction Collection**: ✓

The system provides an option of injecting a custom join order, calling the cost model on this order, yields the cost predictions.

FR 6 **Execution Profiling**: ✓

The system provides a profiler with point-cuts in many places to collect measurements of different operations.

FR 7 **Data Set Management**: ✓

The system can load any CSV dataset and queries. The supported datasets are SSB, JOB, and TPC-DS, as discussed in section 4.3

FR 8 **Result Reporting**: ✓

The system can collect cost predictions, and execution profiles, which are exported as plain data entries. Furthermore, the system also implements plotting scripts to summarize the data.

Non-Functional Requirements

NFR 1 **Portability**: ✓

The system is developed in Python, with detailed instructions on how to complete the initial configuration. Also, a Docker image with the necessary environment for CUDA is provided, when using cuDF.

NFR 2 **Usability**: ✓

The system provides a command line interface, documentation is written everywhere, and all options are documented.

NFR 3 **Plugability**: ✓

The system supports an interface for new cost models and can support any DataFrame framework as the execution engine.

NFR 4 **Extensibility**: ✓

The system is designed to support extensions to feature collection and other improvements to parsing or operation mapping.

NFR 5 **Consistency & Reproducibility**: ✓

The system guarantees to run deterministically, enumeration is ordered and random generation is seeded.

6.2 Evaluating Use Cases

This section evaluates the three use cases introduced in the chapter 3. It will present example data, describe their applications, highlight key findings, and explain the significance of each use case. If any limitations exist, they will also be discussed.

To ensure uniformity, the analysis in each section will primarily concentrate on two specific queries, namely JOB 2A and TPC-DS 50. Nonetheless, attention will be given to other queries in sections where the results are deemed exceptionally significant.

Use Case 1: Measuring Execution Times

This use case emphasizes the system's capacity to accurately measure query execution times over different stages, a capability that, while not unique to this system, is enhanced by its user-friendly approach to custom join order injection. This feature significantly elevates the system's utility beyond what is typically offered by traditional DBMS, where such flexibility might be more cumbersome or not readily accessible.

The system's standout feature is its convenience and adaptability in allowing users to specify custom join orders. This not only facilitates a straightforward comparison of execution times under varied scenarios but also enriches performance analysis and optimization efforts. The ability to experiment with different join orders without the constraints often imposed by other systems introduces a level of analysis depth that is both valuable and unique. By leveraging this capability, the system enables a detailed examination of how join order affects execution time.

The potential for further research utilizing this system is significant. For instance, by gathering data on various join orders, an analysis can be conducted on the average execution time of a join, irrespective of its sequence within the join order. Additionally, it is possible to investigate which join demonstrates the most substantial performance enhancement when positioned at the beginning of the execution sequence as opposed to the end.

Use Case 2: Generating Cost Predictions

This use case focuses on the system's ability to generate cost predictions for SQL queries across different join orders, highlighting the academic contribution of evaluating cost model accuracy in predicting computational costs. The core methodology involves integrating various cost models, generating predictions for predefined SQL queries, and comparing these predictions against actual execution times to assess accuracy.

The primary academic contribution of this use case lies in its systematic approach to evaluating the predictive performance of cost models within the system. By analyzing the correlation between predicted and actual costs in both CPU and GPU environments, this use case provides insights into how different cost models perform across various hardware configurations. This analysis not only contributes to the understanding of cost model effectiveness but also informs the development of more accurate models for SQL query optimization.

In conclusion, Use Case 2 underscores the importance of accurate cost predictions in optimizing SQL queries and highlights the system's capability to serve as a valuable tool for research in SQL query optimization and cost model evaluation.

Use Case 3: Getting Query Results

The last use case is about using the designed system as an SQL DB. This was not the focus of the system, and therefore the usage is not ideal. The search often takes longer than executing even the slowest join order, because generating cost predictions is not parallelized, and it was not optimized for fast execution but rather modularity, and wide support. Due to this reason, this use case is in its current form unusable.

6.3 Limitations

This section outlines the limitations encountered during the development and evaluation of the designed system, the methodology and the experiment conducted. These limitations are categorized into two main areas: system design limitations and experimental limitations.

6.3.1 Designed system & Benchamrking methodology

This section discusses the limitations of the designed system and the benchmarking methodology, pointing out its weaknesses and missing implementations. These limitations primarily stem from the deliberate decision to concentrate on a narrower aspect of the system's potential functionality, due to the limited scope of this thesis. Consequently, certain features were not implemented, reflecting a focused approach.

1. **CPU & GPU DataFrame Libraries Implement Different Join Algorithms:** cuDF (GPU) implements hash-join, while pandas (CPU) implements a combination of a sort-merge and hash joins. This inconsistency between execution engines could lead to variations in performance and optimization opportunities, limiting the system's ability to provide a uniform benchmarking and optimization framework across different hardware configurations. Modifying the DataFrame library, was out of the scope of this thesis.
2. **Simple Cardinality Estimation:** The system's cardinality estimation, crucial for optimizing query execution plans, follows a well-established, yet simple method of estimation. Inaccurate cardinality estimates can lead to suboptimal join orders and execution plans, adversely affecting the system's overall performance. This limitation restricts the system's effectiveness in achieving optimal query execution, highlighting a critical area for improvement in cost model accuracy.
3. **Data Features Only:** The system designed is currently collecting only data features, neglecting hardware characteristics that could impact query execution performance. This omission means that the system does not account for hardware-specific optimizations, such as parallel processing capabilities or memory hierarchies, which could influence the choice of the most efficient execution strategy. Ignoring hardware features limits the system's ability to tailor optimizations to specific hardware configurations, potentially leading to less-than-optimal performance on diverse platforms.
4. **Join Order Only:** The current system focuses solely on optimizing the join order without considering the optimization of the overall execution plan. This narrow focus overlooks other optimization opportunities, such as selection pushdown, projection pruning, or the use of indexes, which could significantly

enhance query performance. The system implements some of these techniques manually, however, the system limits its potential to fully optimize query execution, by ignoring full execution plan enumeration.

6.3.2 Experiment

This study, while providing valuable insights into the significance of various features in cost models for SQL query execution on CPU and GPU systems, is subject to several limitations.

1. **DataFrame Library Join Algorithms:** The choice of DataFrame libraries, Pandas for CPU and cuDF for GPU, introduces a potential confounding factor due to the inherent differences in their join algorithms. Pandas can perform both sort-merge join and hash-join, whereas cuDF consistently employs hash-join. This discrepancy means that observed differences in feature significance may not solely be attributable to hardware architecture but could also reflect the innate characteristics of the join algorithms used. The experiment operates under the assumption that the architectural differences between CPU and GPU are substantial enough to manifest in the cost model features, despite the variability in join algorithms.
2. **Limited Feature Selection:** The experiment is constrained by the selection of features, focusing exclusively on data features. This limited scope raises the possibility that other, untested features could significantly outperform those evaluated. The inherent limitation in the number of features considered, only a single feature for the baseline model and two for the comparison models, further restricts the study. It is conceivable that some features may only reveal their full predictive value when combined with others, suggesting that the inclusion of additional or relative features could uncover more complex relationships that were not captured in this analysis.
3. **Reliance on a Limited Set of Features:** The reliance on a limited set of features in the cost models might obscure the potential for discovering more intricate interactions that could enhance the models' accuracy. While the inclusion of relative features aimed to mitigate this issue, it remains possible that certain hidden relations between features are only discernible when a broader array of features is considered.

These limitations underscore the need for further research to explore the impact of different join algorithms, expand the feature set, and investigate the potential benefits of incorporating a wider variety of features into the cost models. Such efforts could lead to more refined and accurate cost models that better reflect the complexities of SQL query execution across diverse hardware environments.

6.4 Future Work

The list of future work presented below is not exhaustive but serves as an inspiration for potential directions that could further the research and development of this system. These suggestions aim to spark ideas for addressing current limitations and exploring new functionalities, with the understanding that each item represents a starting point for deeper investigation rather than a definitive solution.

1. **More Execution Engines:** Adding more execution engines to the system could enhance its versatility and applicability. This would allow for a broader comparison of SQL query performance across various platforms, potentially revealing unique optimization opportunities.
2. **Methodology Extension:** Refining the benchmarking methodology to include measures of prediction accuracy against absolute costs could offer insights into the effectiveness of different cost models. This could guide improvements in cost model selection and optimization approaches.
3. **Additional Metrics:** Introducing a wider range of evaluation metrics, such as memory, disk I/O, and CPU usage, could provide a more nuanced understanding of query performance and system efficiency. This expansion may help in identifying new optimization strategies and refining existing ones.
4. **Execution Tree Optimization:** Currently, the system focuses primarily on optimizing join orders. The proposed future work aims to extend this focus to include the full execution tree optimization. This enhancement involves enumerating all possible execution trees to identify the most efficient execution strategy. By considering not just the order of joins but also the selection, projection, and aggregation operations, this comprehensive approach could significantly broaden the scope of optimization. Such an expansion is expected to produce more widely applicable cost model benchmarking, opening the designed system to a broader audience.
5. **Optimization for Query Execution:** The system, as currently designed, prioritizes modularity and wide support over execution speed, particularly in generating cost predictions. This approach results in longer search times compared to even the slowest join orders, as seen in Use Case 3. Future work should focus on optimizing the system for faster query execution without sacrificing its modular design. This could involve parallelizing cost prediction generation, refining the system's architecture to reduce overhead, and implementing more efficient algorithms for cost prediction and query execution. Such optimizations would enhance the system's usability as an SQL DB, making it a more practical tool for real-world applications and research.

Chapter 7

Conclusion

This thesis embarked on an exploration of optimizing SQL query execution through the lens of cost model performance across different execution engines, with a particular focus on CPU and GPU systems. The research was motivated by the challenges inherent in join-order optimization and the need for a deeper understanding of cost models' behavior in varying computational environments. The primary research question (RQ1) sought to design a system capable of measuring any cost model's performance across different execution engines. This question was further complemented with subquestions focusing on the empirical measurement of cost model performance (RQ1.1) and identifying key features impacting cost models in CPU and GPU systems (RQ1.2).

7.1 Research Contributions

The contributions of this thesis are threefold:

1. **Design and Development of a Modular SQL Simulator:** A benchmarking system was developed, featuring a modular architecture that allows for the integration of various cost models and execution engines. This system facilitates the execution of SQL queries, optimization processes, and profiling of execution engines, thereby enabling a comprehensive evaluation of cost models across different scenarios.
2. **Methodology for Evaluating the Performance of Cost Models:** A structured methodology was established for empirically measuring the performance of cost models across different execution engines. This methodology ensures a consistent and reproducible approach to evaluating cost models, leveraging the capabilities of the developed benchmarking system.
3. **Analysis of Key Features:** Through an experiment designed to utilize the benchmarking system, key features influencing the performance of join-order optimization in CPU and GPU systems were identified and analyzed. This analysis provided insights into the critical factors affecting cost models' efficiency and execution of join orders, contributing to the development of more effective cost models and optimizers for both CPU and GPU systems.

7.2 Addressing the Research Questions

The research questions set forth at the beginning of this thesis have been addressed through the design and implementation of the modular SQL simulator, the development of a benchmarking methodology, and the analysis of key features impacting cost model performance. The system designed in response to RQ1 demonstrated its capability to measure the performance of various cost models across different execution engines, thereby providing a versatile tool for evaluating and understanding cost models in diverse computational environments. The methodology developed in response to RQ1.1 facilitated a structured and empirical approach to this evaluation, ensuring the reliability and consistency of the findings. Finally, the experiment conducted in response to RQ1.2 revealed significant insights into the key features that influence cost model performance, highlighting the importance of tailoring cost models to the specific characteristics of CPUs and GPUs.

7.3 Limitations & Future Work

The study, focusing on the developed benchmarking system, encounters limitations primarily due to the choice of DataFrame libraries (Pandas for CPU and cuDF for GPU) which introduces biases from their differing join algorithms. This affects the system's ability to uniformly benchmark across hardware configurations. Moreover, the system's emphasis on data features and join order optimization may miss out on other impactful features and optimization opportunities, such as selection pushdown or the use of indexes.

Future work should aim at expanding the system's capabilities by incorporating more execution engines and cost models, broadening the applicability and enhancing versatility. Refining the benchmarking methodology to include a wider range of evaluation metrics and integrating machine learning techniques into cost model development are promising directions. Additionally, exploring full execution tree optimization and the impact of hardware-specific optimizations could significantly improve the system's effectiveness in query optimization.

7.4 Concluding Remarks

In conclusion, this thesis presents a cohesive narrative built upon three foundational pillars: the development of a modular SQL simulator for simulating various cost models across different execution engines, the establishment of a structured benchmarking methodology that dictates the effective use of this system, and an applied study highlighting the significant features of cost models in CPU and GPU environments. Together, these contributions form a comprehensive framework that advances the understanding of cost model performance in SQL query optimization. The modular system provides a versatile platform for evaluation, the benchmarking methodology ensures consistent and empirical analysis, and the applied study offers practical insights into optimizing cost models for specific hardware architectures. The insights gained from this study not only contribute to the academic community but also offer practical guidance for the development of more efficient and effective database systems. As the field progresses, the methodologies and findings presented in this thesis have the potential to streamline the research process of query optimization and database management.

Bibliography

[1] Alaa Aljanaby, Emad Abuelrub, and Mohammed Hosni Odeh. "A Survey of Distributed Query Optimization". In: *Int. Arab J. Inf. Technol.* 2 (2005), pp. 48–57. URL: https://api.semanticscholar.org/CorpusID:15317825.

[2] Shivnath Babu and P. Bizarro. "Adaptive Query Processing in the Looking Glass". In: *Conference on Innovative Data Systems Research*. 2005. URL: https://api.semanticscholar.org/CorpusID:6583157.

[3] Peter Bakkum and Kevin Skadron. "Accelerating SQL Database Operations on a GPU with CUDA". In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. GPGPU-3. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, pp. 94–103. ISBN: 9781605589350. DOI: 10.1145/1735688.1735706. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1735688.1735706.

[4] *BlazingSQL: A lightweight, GPU accelerated, SQL engine built on the RAPIDS.ai ecosystem*. https://github.com/BlazingDB/blazingsql. Accessed: 2024-02-28.

[5] Sebastian Breß and Gunter Saake. "Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in DBMS". In: *Proc. VLDB Endow.* 6.12 (2013), pp. 1398–1403. ISSN: 2150-8097. DOI: 10.14778/2536274.2536325. URL: https://doi.org/10.14778/2536274.2536325.

[6] Surajit Chaudhuri. "An Overview of Query Optimization in Relational Systems". In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '98. Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 34–43. ISBN: 0897919963. DOI: 10.1145/275487.275492. URL: https://doi.org/10.1145/275487.275492.

[7] *cuDF - Python GPU DataFrame library*. https://docs.rapids.ai/api/cudf/stable/. Accessed: 2024-02-28.

[8] *DuckDB: a Fast in-process Analytical Database*. https://duckdb.org. Accessed: 2024-02-27.

[9] Charlie Garrod. *Lecture 14: Query Planning and Optimization*. 15-445/645 Database Systems (Spring 2023), Carnegie Mellon University. Available online: https://15445.courses.cs.cmu.edu/spring2023/. 2023.

[10] Yuxing Han et al. *Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation*. 2021. arXiv: 2109.05877 [cs.DB].

[11] *HeavyDB: Open Source Analytical Database & SQL Engine*. https://www.heavy.ai/product/heavydb. Accessed: 2024-02-27.

[12] Joseph M. Hellerstein. "Optimization techniques for queries with expensive methods". In: *ACM Trans. Database Syst.* 23.2 (1998), pp. 113–157. ISSN: 0362-5915. DOI: 10.1145/292481.277627. URL: https://doi.org/10.1145/292481.277627.

[13] Benjamin Hilprecht et al. *DeepDB: Learn from Data, not from Queries!* 2019. arXiv: 1909.00607 [cs.DB].

[14] Konstantinos Karanasos et al. *Extending Relational Query Processing with ML Inference*. 2019. arXiv: 1911.00231 [cs.DB].

[15] Hai Lan, Zhifeng Bao, and Yuwei Peng. "A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration". In: *Data Science and Engineering* 6.1 (2021), pp. 86–101. DOI: 10.1007/s41019-020-00149-7. URL: https://doi.org/10.1007%2Fs41019-020-00149-7.

[16] Rubao Lee et al. "The Art of Balance: A RateupDB™ Experience of Building a CPU/GPU Hybrid Database Product". In: *Proc. VLDB Endow.* 14.12 (July 2021), pp. 2999–3013. ISSN: 2150-8097. DOI: 10.14778/3476311.3476378. URL: https://doi-org.tudelft.idm.oclc.org/10.14778/3476311.3476378.

[17] Viktor Leis et al. "Query optimization through the looking glass, and what we found running the Join Order Benchmark". In: *The VLDB Journal* 27 (2017), pp. 643 –668.

[18] Riccardo Mancini et al. "Efficient Massively Parallel Join Optimization for Large Queries". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 122–135. ISBN: 9781450392495. DOI: 10.1145/3514221.3517871. URL: https://doi.org/10.1145/3514221.3517871.

[19] Ryan Marcus et al. "Bao: Making Learned Query Optimization Practical". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 1275–1288. ISBN: 9781450383431. DOI: 10.1145/3448016.3452838. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3448016.3452838.

[20] Pat O'Neil, Betty O'Neil, and Xuedong Chen. *Star Schema Benchmark*. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF. Revision 3, June 5, 2009. 2009.

[21] *pandas - Python Data Analysis Library*. https://pandas.pydata.org/. Accessed: 2024-02-28.

[22] Matthew Perron et al. *How I Learned to Stop Worrying and Love Re-optimization*. 2019. DOI: 10.48550/ARXIV.1902.08291. URL: https://arxiv.org/abs/1902.08291.

[23] *PostgreSQL: The World's Most Advanced Open Source Relational Database*. https://www.postgresql.org. Accessed: 2024-02-27.

[24] *PyGAD - Python Genetic Algorithm!* https://pygad.readthedocs.io/en/latest/. Accessed: 2024-03-20.

[25] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. "Query Processing on Heterogeneous CPU/GPU Systems". In: *ACM Comput. Surv.* 55.1 (2022). ISSN: 0360-0300. DOI: 10.1145/3485126. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3485126.

[26] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. "A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1617–1632. ISBN: 9781450367356. DOI: 10.1145/3318464.3380595. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3318464.3380595.

[27] Tarique Siddiqui et al. "Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 99–113. ISBN: 9781450367356. DOI:

10.1145/3318464.3380584. URL: https://doi.org/10.1145/3318464.3380584.

[28] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill Education, 2020. ISBN: 9781260084504. URL: https://books.google.nl/books?id=5xbQuwEACAAJ.

[29] John Miles Smith and Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". In: *Commun. ACM* 18.10 (1975), 568–579. ISSN: 0001-0782. DOI: 10.1145/361020.361025. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/361020.361025.

[30] Ji Sun and Guoliang Li. "An end-to-end learning-based cost estimator". In: *Proc. VLDB Endow.* 13.3 (2019), pp. 307–319. ISSN: 2150-8097. DOI: 10.14778/3368289.3368296. URL: https://doi.org/10.14778/3368289.3368296.

[31] *TPC-DS*. https://www.tpc.org/tpcds/. Accessed: 2024-03-20.

[32] Florian Waas and Arjan Pellenkoft. "Join Order Selection (Good Enough Is Easy)". In: *Advances in Databases*. Ed. by Brian Lings and Keith Jeffery. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 51–67. ISBN: 978-3-540-45033-7.

[33] Xiaoying Wang et al. "Are we ready for learned cardinality estimation?" In: *Proceedings of the VLDB Endowment* 14.9 (2021), pp. 1640–1654. DOI: 10.14778/3461535.3461552. URL: https://doi.org/10.14778%2F3461535.3461552.

[34] Wentao Wu. "A Note On Operator-Level Query Execution Cost Modeling". In: *CoRR* abs/2003.04410 (2020). arXiv: 2003.04410. URL: https://arxiv.org/abs/2003.04410.

[35] Wentao Wu et al. "Predicting query execution time: Are optimizer cost models really unusable?" In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 1081–1092. DOI: 10.1109/ICDE.2013.6544899.

[36] Xiang Yu et al. "Reinforcement Learning with Tree-LSTM for Join Order Selection". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 2020, pp. 1297–1308. DOI: 10.1109/ICDE48307.2020.00116.

[37] Yuan Yuan et al. "Spark-GPU: An accelerated in-memory data processing engine on clusters". In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016, pp. 273–283. DOI: 10.1109/BigData.2016.7840613.

Appendix A

Selected Queries

SSB

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, ddate, part, supplier
where lo_orderdate = d_datekey       # JOIN ID: 0
and lo_partkey = p_partkey           # JOIN ID: 1
and lo_suppkey = s_suppkey           # JOIN ID: 2
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1
```

LISTING A.1: Query SSB 21

```
select c_nation, s_nation, d_year, sum(lo_revenue)
as revenue from customer, lineorder, supplier, ddate
where lo_custkey = c_custkey         # JOIN ID: 0
and lo_suppkey = s_suppkey           # JOIN ID: 1
and lo_orderdate = d_datekey         # JOIN ID: 2
and c_region = 'ASIA' and s_region = 'ASIA'
and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc
```

LISTING A.2: Query SSB 31

```
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as
   profit
from ddate, customer, supplier, part, lineorder
where lo_custkey = c_custkey          # JOIN ID: 0
and lo_suppkey = s_suppkey            # JOIN ID: 1
and lo_partkey = p_partkey            # JOIN ID: 2
and lo_orderdate = d_datekey          # JOIN ID: 3
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation
```

LISTING A.3: Query SSB 41

```
select d_year, s_nation, p_category, sum(lo_revenue -
   lo_supplycost) as profit
```

```
from ddate, customer, supplier, part, lineorder
where lo_custkey = c_custkey           # JOIN ID: 0
and lo_suppkey = s_suppkey             # JOIN ID: 1
and lo_partkey = p_partkey             # JOIN ID: 2
and lo_orderdate = d_datekey           # JOIN ID: 3
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (d_year = 1997 or d_year = 1998)
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category
```

LISTING A.4: Query SSB 42

```
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost)
    as profit
from ddate, customer, supplier, part, lineorder
where lo_custkey = c_custkey            # JOIN ID: 0
and lo_suppkey = s_suppkey              # JOIN ID: 1
and lo_partkey = p_partkey              # JOIN ID: 2
and lo_orderdate = d_datekey            # JOIN ID: 3
and c_region = 'AMERICA'
and s_nation = 'UNITED␣STATES'
and (d_year = 1997 or d_year = 1998)
and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1
```

LISTING A.5: Query SSB 43

JOB

```
SELECT MIN(t.title) AS movie_title
FROM company_name AS cn,
     keyword AS k,
     movie_companies AS mc,
     movie_keyword AS mk,
     title AS t
WHERE cn.country_code = '[de]'
  AND k.keyword = 'character-name-in-title'
  AND cn.id = mc.company_id             # JOIN ID: 0
  AND mc.movie_id = t.id                # JOIN ID: 1
  AND t.id = mk.movie_id                # JOIN ID: 2
  AND mk.keyword_id = k.id              # JOIN ID: 3
  AND mc.movie_id = mk.movie_id;        # JOIN ID: 4
```

LISTING A.6: Query JOB 2A

```
SELECT MIN(k.keyword) AS movie_keyword,
       MIN(n.name) AS actor_name,
       MIN(t.title) AS hero_movie
FROM cast_info AS ci,
     keyword AS k,
     movie_keyword AS mk,
     name AS n,
     title AS t
```

```
WHERE k.keyword IN ('superhero',
                    'sequel',
                    'second-part',
                    'marvel-comics',
                    'based-on-comic',
                    'tv-special',
                    'fight',
                    'violence')
  AND t.production_year > 2000
  AND k.id = mk.keyword_id              # JOIN ID: 0
  AND t.id = mk.movie_id                # JOIN ID: 1
  AND t.id = ci.movie_id                # JOIN ID: 2
  AND ci.movie_id = mk.movie_id         # JOIN ID: 3
  AND n.id = ci.person_id;              # JOIN ID: 4
```

LISTING A.7: Query JOB 6F

```
SELECT MIN(an.name) AS alternative_name,
       MIN(chn.name) AS voiced_char_name,
       MIN(n.name) AS voicing_actress,
       MIN(t.title) AS american_movie
FROM aka_name AS an,
     char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     movie_companies AS mc,
     name AS n,
     role_type AS rt,
     title AS t
WHERE ci.note IN ('(voice)',
                  '(voice: Japanese version)',
                  '(voice) (uncredited)',
                  '(voice: English version)')
  AND cn.country_code = '[us]'
  AND n.gender = 'f'
  AND rt.role = 'actress'
  AND ci.movie_id = t.id                # JOIN ID: 0
  AND t.id = mc.movie_id                # JOIN ID: 1
  AND ci.movie_id = mc.movie_id         # JOIN ID: 2
  AND mc.company_id = cn.id             # JOIN ID: 3
  AND ci.role_id = rt.id                # JOIN ID: 4
  AND n.id = ci.person_id               # JOIN ID: 5
  AND chn.id = ci.person_role_id        # JOIN ID: 6
  AND an.person_id = n.id               # JOIN ID: 7
  AND an.person_id = ci.person_id;      # JOIN ID: 8
```

LISTING A.8: Query JOB 9D

```
SELECT MIN(cn.name) AS movie_company,
       MIN(mi_idx.info) AS rating,
       MIN(t.title) AS drama_horror_movie
FROM company_name AS cn,
  company_type AS ct,
  info_type AS it1,
  info_type AS it2,
  movie_companies AS mc,
  movie_info AS mi,
  movie_info_idx AS mi_idx,
```

```
  title AS t
WHERE cn.country_code = '[us]'
  AND ct.kind = 'production␣companies'
  AND it1.info = 'genres'
  AND it2.info = 'rating'
  AND mi.info IN ('Drama',
                  'Horror')
  AND mi_idx.info > '8.0'
  AND t.production_year BETWEEN 2005 AND 2008
  AND t.id = mi.movie_id                  # JOIN ID: 0
  AND t.id = mi_idx.movie_id              # JOIN ID: 1
  AND mi.info_type_id = it1.id            # JOIN ID: 2
  AND mi_idx.info_type_id = it2.id        # JOIN ID: 3
  AND t.id = mc.movie_id                  # JOIN ID: 4
  AND ct.id = mc.company_type_id          # JOIN ID: 5
  AND cn.id = mc.company_id               # JOIN ID: 6
  AND mc.movie_id = mi.movie_id           # JOIN ID: 7
  AND mc.movie_id = mi_idx.movie_id       # JOIN ID: 8
  AND mi.movie_id = mi_idx.movie_id;      # JOIN ID: 9
```

LISTING A.9: Query JOB 12A

```
SELECT MIN(mi_idx.info) AS rating,
       MIN(t.title) AS northern_dark_movie
FROM info_type AS it1,
  info_type AS it2,
  keyword AS k,
  kind_type AS kt,
  movie_info AS mi,
  movie_info_idx AS mi_idx,
  movie_keyword AS mk,
  title AS t
WHERE it1.info = 'countries'
  AND it2.info = 'rating'
  AND k.keyword IN ('murder',
                    'murder-in-title',
                    'blood',
                    'violence')
  AND kt.kind = 'movie'
  AND mi.info IN ('Sweden',
                  'Norway',
                  'Germany',
                  'Denmark',
                  'Swedish',
                  'Denish',
                  'Norwegian',
                  'German',
                  'USA',
                  'American')
  AND mi_idx.info < '8.5'
  AND t.production_year > 2010
  AND kt.id = t.kind_id                   # JOIN ID: 0
  AND t.id = mi.movie_id                  # JOIN ID: 1
  AND t.id = mk.movie_id                  # JOIN ID: 2
  AND t.id = mi_idx.movie_id              # JOIN ID: 3
  AND mk.movie_id = mi.movie_id           # JOIN ID: 4
  AND mk.movie_id = mi_idx.movie_id       # JOIN ID: 5
  AND mi.movie_id = mi_idx.movie_id       # JOIN ID: 6
```

```
  AND k.id = mk.keyword_id                # JOIN ID: 7
  AND it1.id = mi.info_type_id            # JOIN ID: 8
  AND it2.id = mi_idx.info_type_id;       # JOIN ID: 9
```

LISTING A.10: Query JOB 14A

TPC-DS

```
select  i_item_id,
        s_state, grouping(s_state) g_state,
        avg(ss_quantity) agg1,
        avg(ss_list_price) agg2,
        avg(ss_coupon_amt) agg3,
        avg(ss_sales_price) agg4
from store_sales, customer_demographics, date_dim, store, item
where ss_sold_date_sk = d_date_sk and     # JOIN ID: 0
      ss_item_sk = i_item_sk and          # JOIN ID: 1
      ss_store_sk = s_store_sk and        # JOIN ID: 2
      ss_cdemo_sk = cd_demo_sk and        # JOIN ID: 3
      cd_gender = 'F' and
      cd_marital_status = 'D' and
      cd_education_status = '2 yr Degree' and
      d_year = 2002 and
      s_state in ('TN', 'TN', 'TN', 'TN', 'TN', 'TN')
group by rollup (i_item_id, s_state)
order by i_item_id
        ,s_state
limit 100;
```

LISTING A.11: Query TPC-DS 27

```
select
   s_store_name
  ,s_company_id
  ,s_street_number
  ,s_street_name
  ,s_street_type
  ,s_suite_number
  ,s_city
  ,s_county
  ,s_state
  ,s_zip
  ,sum(case when (sr_returned_date_sk - ss_sold_date_sk <= 30 )
      then 1 else 0 end)  as "30 days"
  ,sum(case when (sr_returned_date_sk - ss_sold_date_sk > 30)
      and
                (sr_returned_date_sk - ss_sold_date_sk <= 60)
                   then 1 else 0 end )  as "31-60 days"
  ,sum(case when (sr_returned_date_sk - ss_sold_date_sk > 60)
      and
                (sr_returned_date_sk - ss_sold_date_sk <= 90)
                   then 1 else 0 end)  as "61-90 days"
  ,sum(case when (sr_returned_date_sk - ss_sold_date_sk > 90)
      and
                (sr_returned_date_sk - ss_sold_date_sk <= 120)
                   then 1 else 0 end)  as "91-120 days"
```

```
  ,sum(case when (sr_returned_date_sk - ss_sold_date_sk  > 120)
      then 1 else 0 end)  as ">120 days"
from
    store_sales
    ,store_returns
    ,store
    ,date_dim d1
    ,date_dim d2
where
    d2.d_year = 2000
and d2.d_moy  = 8
and ss_ticket_number = sr_ticket_number          # JOIN ID: 0
and ss_item_sk = sr_item_sk                      # JOIN ID: 1
and ss_sold_date_sk   = d1.d_date_sk             # JOIN ID: 2
and sr_returned_date_sk   = d2.d_date_sk         # JOIN ID: 3
and ss_customer_sk = sr_customer_sk              # JOIN ID: 4
and ss_store_sk = s_store_sk                     # JOIN ID: 5
group by
    s_store_name
    ,s_company_id
    ,s_street_number
    ,s_street_name
    ,s_street_type
    ,s_suite_number
    ,s_city
    ,s_county
    ,s_state
    ,s_zip
order by s_store_name
     ,s_company_id
     ,s_street_number
     ,s_street_name
     ,s_street_type
     ,s_suite_number
     ,s_city
     ,s_county
     ,s_state
     ,s_zip
limit 100;
```

LISTING A.12: Query TPC-DS 50

Appendix B

Detailed Correlation Results

This appendix shows intermediate correlation results for each query and all cost models designed in chapter 5. These intermediate results are combined as described in chapter 4.

	FEATURE	MEAN CORRELATION	IMPROV. OVER BASE	job/12a	job/14a	job/2a	job/6f	job/9d	ssb/q21	ssb/q31	ssb/q41	ssb/q42	ssb/q43	tpcds/ query_27	tpcds/ query_50
	baseline	0,28713868	0,00%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,75181
Table-level	t_unique	0,28749774	0,13%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28126	0,15106	0,63508	0,80637	0,42824	-0,12640	0,75410
	t_id_size	0,28847492	0,47%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28018	0,14282	0,63541	0,80689	0,43796	-0,12641	0,75786
	t_row_size	0,28748704	0,12%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,27802	0,14540	0,63496	0,80624	0,42695	-0,12631	0,75668
	t_cache_age	0,28788000	0,26%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,27925	0,16145	0,63511	0,80605	0,42493	-0,12643	0,75663
	t_cluster_size	0,28743254	0,10%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,75384
	t_bounds_low	0,29636769	3,21%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,35951	0,15732	0,63078	0,80679	0,52072	-0,12639	0,75408
	t_bounds_high	0,29206621	1,72%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,35951	0,15732	0,63078	0,80679	0,52072	-0,12647	0,72435
	t_bounds_range	0,28902698	0,66%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,29303	0,15434	0,63520	0,80678	0,42708	-0,12649	0,76172
Join-level	c_len_res	0,30049587	4,65%	-0,06328	-0,10800	0,43909	0,74459	-0,03458	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,74883
	c_len_possible_max	0,31492544	**9,68%**	-0,03325	-0,05736	0,23684	0,54850	0,35609	0,27988	0,14858	0,63646	0,80647	0,42716	-0,12621	0,72474
	c_len_unique_max	0,28784896	0,25%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28068	0,15131	0,63510	0,80635	0,42799	-0,12640	0,75672
	c_selectivity	0,30085582	4,78%	-0,19817	-0,29772	0,44280	0,84199	-0,03070	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,94999
	c_cluster_size	0,28738645	0,09%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,75353
	c_cluster_overlap	0,28956873	0,85%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,76864
Relative	**c_tbl_ratio_length**	0,33839827	**17,85%**	0,29115	-0,15109	0,38747	0,84195	0,02424	0,28046	0,15135	0,63543	0,80630	0,43151	-0,12638	0,74697
	c_tbl_ratio_unique	0,29637753	3,22%	-0,19822	-0,15110	0,44284	0,84199	-0,03070	0,35958	0,37189	0,63592	0,80996	0,48695	-0,12621	0,71861
	c_tbl_ratio_row_size	0,29549365	2,91%	-0,19820	-0,15103	0,44284	0,84199	-0,03070	0,38653	0,41655	0,64154	0,81136	0,47733	-0,13056	0,70098
	c_tbl_ratio_cache_age	0,28847279	0,46%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28327	0,17780	0,63543	0,80664	0,42907	-0,12643	0,75379
	c_tbl_ratio_bounds_range	0,28787822	0,26%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28327	0,17780	0,63522	0,80631	0,42907	-0,12644	0,74994
	c_tbl_min_length	0,30460677	**6,08%**	-0,03813	-0,04027	0,43276	0,72874	-0,03364	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,72016
	c_tbl_min_unique	0,28720952	0,02%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28075	0,15073	0,63513	0,80632	0,42827	-0,12641	0,75229
	c_tbl_min_row_size	0,29365939	2,27%	-0,19822	-0,15110	0,44281	0,84199	-0,03070	0,81690	0,92602	0,66947	0,80109	0,61674	-0,13267	0,40544
	c_tbl_min_cache_age	0,28722941	0,03%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28068	0,15617	0,63514	0,80611	0,43108	-0,12642	0,74998
	c_tbl_min_bounds_range	0,28757255	0,15%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28009	0,15100	0,63514	0,80636	0,42904	-0,12642	0,75450
	c_tbl_max_length	0,29986150	4,43%	-0,06327	-0,10800	0,43910	0,74459	-0,03458	0,28055	0,15130	0,63511	0,80636	0,42806	-0,12642	0,74443
	c_tbl_max_unique	0,28714763	0,00%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28055	0,15130	0,63511	0,80636	0,42807	-0,12642	0,75187
	c_tbl_max_row_size	0,28720675	0,02%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,27827	0,14600	0,63500	0,80629	0,42696	-0,12635	0,75453
	c_tbl_max_cache_age	0,28722941	0,03%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28068	0,15617	0,63514	0,80611	0,43108	-0,12642	0,74998
	c_tbl_max_bounds_range	0,28757255	0,15%	-0,19820	-0,15109	0,44284	0,84199	-0,03070	0,28009	0,15100	0,63514	0,80636	0,42904	-0,12642	0,75450

FIGURE B.1: Feature Comparison, with Detailed Correlation for Each Query, for CPU

	FEATURE	MEAN CORRELATION	IMPROV. OVER BASE	job/12a	job/14a	job/2a	job/6f	job/9d	ssb/q21	ssb/q31	ssb/q41	ssb/q42	ssb/q43	tpcds/ query_27	tpcds/ query_50
	baseline	0,33065836	0,00%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,62525
Table-level	t_unique	0,33368100	0,91%	-0,04867	0,08158	0,69451	0,77520	-0,13740	0,33740	0,24924	0,56191	0,80548	0,33754	0,15025	0,62481
	t_id_size	0,33367849	0,91%	-0,04862	0,08156	0,69450	0,77519	-0,13740	0,31382	0,36732	0,55925	0,80612	0,29328	0,15096	0,62531
	t_row_size	0,36911600	**11,63%**	-0,04867	0,08157	0,69451	0,77519	-0,13740	0,70024	0,93849	0,59159	0,83871	0,48324	0,12881	0,60425
	t_cache_age	0,33845344	2,36%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,44527	0,29776	0,55049	0,81583	0,34469	0,15095	0,62542
	t_cluster_size	0,34128688	3,21%	-0,04882	0,08162	0,69428	0,77516	-0,13740	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,69671
	t_bounds_low	0,33648421	1,76%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,36755	0,22101	0,55298	0,79893	0,39714	0,15099	0,62530
	t_bounds_high	0,33648777	1,76%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,36755	0,22101	0,55298	0,79893	0,39714	0,15103	0,62530
	t_bounds_range	0,33418237	1,07%	-0,04862	0,08155	0,69450	0,77519	-0,13740	0,19531	0,25299	0,56054	0,80290	0,41163	0,15048	0,62552
Join-level	c_len_res	0,33116647	0,15%	-0,04524	0,07957	0,69563	0,79753	-0,14126	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,62231
	c_len_possible_max	0,33394749	0,99%	-0,08359	0,08605	0,45679	0,61130	0,32211	0,26128	0,12038	0,61756	0,80659	0,25382	0,20241	0,53951
	c_len_unique_max	0,34399719	4,03%	-0,04847	0,08169	0,69445	0,77506	-0,13740	0,45315	0,26327	0,56931	0,81449	0,43641	0,15371	0,62222
	c_selectivity	0,35712950	**8,01%**	-0,04856	0,31552	0,69450	0,77522	-0,13740	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,62522
	c_cluster_size	0,33964979	2,72%	-0,04924	0,08224	0,69432	0,77514	-0,13740	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,68558
	c_cluster_overlap	0,33199039	0,40%	-0,04845	0,08157	0,69451	0,77519	-0,13740	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,63402
Relative	**c_tbl_ratio_length**	0,36722063	**11,06%**	0,06659	0,08156	0,55698	0,77514	0,19788	0,29900	0,25409	0,55927	0,80456	0,30559	0,15110	0,62580
	c_tbl_ratio_unique	0,33839162	2,34%	-0,04865	0,08156	0,69450	0,77519	-0,13740	0,38064	0,29776	0,56058	0,80100	0,37902	0,15073	0,62528
	c_tbl_ratio_row_size	0,36894394	**11,58%**	-0,04867	0,08166	0,69456	0,77520	-0,13740	0,64544	0,90507	0,58643	0,83330	0,50031	0,13483	0,61208
	c_tbl_ratio_cache_age	0,33192574	0,38%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,30189	0,27956	0,55875	0,80478	0,31144	0,15099	0,62526
	c_tbl_ratio_bounds_range	0,34547797	4,48%	-0,04862	0,08157	0,69450	0,77519	-0,13740	0,38064	0,47080	0,56112	0,80890	0,41163	0,15087	0,62498
	c_tbl_min_length	0,33316296	0,76%	-0,06713	0,23605	0,58922	0,53638	0,04839	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,55518
	c_tbl_min_unique	0,33079942	0,04%	-0,04863	0,08157	0,69450	0,77519	-0,13740	0,27939	0,25857	0,55800	0,80299	0,31578	0,15070	0,62512
	c_tbl_min_row_size	0,38780454	**17,28%**	-0,04860	0,08155	0,69448	0,77519	-0,13740	0,79125	0,95064	0,59039	0,81679	0,64248	0,14487	0,64533
	c_tbl_min_cache_age	0,33287062	0,67%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,31823	0,29297	0,55692	0,80564	0,31477	0,15097	0,62526
	c_tbl_min_bounds_range	0,33988195	2,79%	-0,04864	0,08155	0,69449	0,77519	-0,13740	0,38064	0,26734	0,56124	0,80873	0,41163	0,15022	0,62502
	c_tbl_max_length	0,33116626	0,15%	-0,04524	0,07957	0,69563	0,79753	-0,14126	0,29905	0,25406	0,55914	0,80458	0,30286	0,15095	0,62231
	c_tbl_max_unique	0,33107320	0,13%	-0,04861	0,08156	0,69449	0,77519	-0,13740	0,27939	0,27306	0,55800	0,80299	0,31578	0,14958	0,62512
	c_tbl_max_row_size	0,37869506	**14,53%**	-0,04866	0,08157	0,69450	0,77519	-0,13740	0,81247	0,98370	0,60150	0,84194	0,54074	0,12762	0,61281
	c_tbl_max_cache_age	0,33287062	0,67%	-0,04863	0,08156	0,69450	0,77519	-0,13740	0,31823	0,29297	0,55692	0,80564	0,31477	0,15097	0,62526
	c_tbl_max_bounds_range	0,33988195	2,79%	-0,04864	0,08155	0,69449	0,77519	-0,13740	0,38064	0,26734	0,56124	0,80873	0,41163	0,15022	0,62502

FIGURE B.2: Feature Comparison, with Detailed Correlation for Each Query, for GPU