UNIVERSITY OF CALIFORNIA, MERCED

Query Optimization using Sketches in Relational Database Systems

by

Yesdaulet Izenov

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

Committee in charge:
Professor Florin Rusu, Chair
Professor Sungjin Im
Professor Xiaoyi Lu

Fall 2023

The dissertation of Yesdaulet Izenov is approved:

Florin Rusu, Chair	Date
Sungjin Im	Date
Xiaoyi Lu	Date

University of California, Merced

To my family

Contents

List of Figures

List of Tables

Acknowledgments

Firstly, profound thanks to my advisor, Dr. Florin Rusu. His invaluable advice, guidance, and exemplary scientific spirit have been pivotal in shaping me as a researcher.

I would like to thank my dissertation committee, Dr. Sungjin Im and Dr. Xiaoyi Lu. Their feedback and direction enriched this dissertation.

My educational foundation owes much to Dr. Srikantha Tirthapura, Maksat Maratov, and Aliya Alshabayeva. Their knowledge and mentorship honed my skills and perspectives.

I would like to thank my coaches, Vyacheslav Pechersky, Vladimir Williams, Marcos Tome, Michael Quigley, and Bek-Ali Yerzhan for guiding me in both sports and life ambitions.

I also would like to express my appreciation to my current and previous labmates at UC Merced and Iowa State University for all the collaborations and help. Our time together remains in my memories.

My summer times at Meta, Oracle, and Kingland Systems were incredibly insightful. I am grateful to meet my colleagues and appreciate their helpful advice and knowledge.

I consider myself lucky to have many friends from Kazakhstan, with some accompanying me through memorable years in the USA. Since arriving in the USA, I significantly enlarged my circle of friends. I am thankful to meet great individuals and become friends from a diverse number of countries. My heart is full, acknowledging my friends from Kazakhstan and many other nations. Embracing diverse friendships in the USA has been a blessing.

Above all, my family remains my bedrock. Their sacrifices, love, and unwavering support have been the cornerstone of my journey. It is time to honor and give back to everyone who has been part of this incredible ride.

This dissertation is supported by the National Science Foundation (NSF), award number 2008815, by a U.S. Department of Energy Early Career Award (DOE Career), and by the University of California Merced (UCM).

Curriculum Vitae

Education

- Ph.D. in Electrical Engineering and Computer Science. University of California Merced. Merced, CA, USA. December 2023
- Master of Science in Computer Engineering. Iowa State University. Ames, Iowa, USA. May 2018
- Bachelor of Science in Information Systems. International Information Technology University. Almaty, Kazakhstan. May 2013

Work Experience

- Ph.D. Data Engineer Intern. Meta. Menlo Park, CA, USA.
  June 2022 - September 2022
- Research Assistant. Oracle MySQL Heatwave. Redwood City, CA, USA.
  January 2022 - June 2022
- Graduate Data Engineer Intern. Kingland Systems. Ames, IA, USA.
  April 2018 - July 2018

Publications

1. **Yesdaulet Izenov**, Asoke Datta, Brian Tsan, Florin Rusu. Sub-optimal Join Order Identification with L1-error. Accepted in SIGMOD 2024
2. **Yesdaulet Izenov**, Asoke Datta, Florin Rusu, Jun Hyung Shin. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. SIGMOD 2021
3. **Yesdaulet Izenov**, Asoke Datta, Florin Rusu, Jun Hyung Shin. Online Sketch-based Query Optimization. arXiv:2102.02440
4. **Yesdaulet Izenov**, Srikantha Tirthapura. Reducing Labeling Complexity in Streaming Data Mining. Thesis.

Abstract

Query Optimization using Sketches in Relational Database Systems

by

Yesdaulet Izenov

Doctor of Philosophy

in

Electrical Engineering and Computer Science

University of California Merced, 2023

Professor Florin Rusu, Chair

Query optimization remains a crucial element of relational database systems. With rapidly expanding data volumes and an increasing trend of machine-generated queries, the significance of query optimization is only increasing and requires continuous advancements. The objective of the query optimizer is to identify an optimal query execution plan from a vast number of semantically equivalent query plans. The success of this search process depends on the optimal operation of the internal inter-connected components of the query optimizer.

In this dissertation, we introduce COMPASS, a novel query optimization paradigm for in-memory databases based on a single type of statistics – Fast-AGMS sketches. While maintaining high accuracy, the highly parallelizable nature of Fast-AGMS empowers the query optimizer to accommodate more complex queries. Subsequently, we redefine the objective of the query optimizer to find a spanning tree with a low cost. Capitalizing on the polynomial time complexity of spanning tree algorithms, we present ESTE, an ensemble spanning tree-based enumeration strategy. ESTE systematically enumerates different parts of the search space, thereby enhancing the robustness of the query optimizer. We believe this perspective enables the application of well-studied spanning-tree algorithms to the field of query optimization. Finally, we address the impact of cardinality estimation errors on query optimizers. Given their inevitability, these errors can cause a domino effect, leading to additional mistakes in the subsequent components of the optimizer. We propose L1-error, a new indicator designed to identify sub-optimal plans. L1-error accounts for the fact that certain estimation errors may have more impact on selecting an optimal plan than others.

Chapter 1

Introduction

Despite years of research and development in both academia and industry, cost-based query optimization continues to be a vital aspect of relational database systems. The effectiveness of a database system is closely tied to the performance of its query optimizer. Given the growth of data volumes at a tremendously fast speed and the rising prevalence of machine-generated queries, the role of query optimization becomes ever more critical, needing continuous enhancements and innovations.

The query optimizer is a complex assemblage of inter-connected components, including cardinality estimator, cost function, plan search space, and plan enumeration. The objective of the query optimizer is to identify an optimal query execution plan that is expected to result in a fast runtime. During the search for an optimal query plan, the query optimizer compares a vast number of semantically equivalent query plans by assigning a cost to each, utilizing an analytical cost function.

The optimal operation of the query optimizer components is key to identifying an optimal query plan. Determining the cost of a query plan requires knowledge about the cardinalities of base and intermediate tables. Obtaining exact cardinalities is only possible by executing subqueries for every base and intermediate table. However, this contradicts the essence of query optimization, which aims to identify an optimal query plan without actually executing the subqueries. Thus, query optimizers estimate the cardinalities by utilizing a broad spectrum of statistical synopses, such as attribute-level histograms and table-level samples.

As the complexity of selection predicates and the number of join predicates amplify, three challenges emerge. Firstly, the statistics cannot be incrementally composed to effectively estimate the cost of higher-level join cardinalities needed in the search while maintaining estimation accuracy. Secondly, exhaustively enumerating a large set of query plans becomes a substantial computational bottleneck. Thirdly, cardinality estimation errors are exponentially propagated through join operators, potentially leading to suboptimal plans.

1.1 Challenges

To evaluate a query plan, query optimizers utilize a wide array of statistical synopses. Histograms, most frequent values, and the number of distinct values are used in estimating the selectivity of selection predicates on base tables. Estimating join cardinality requires correlated statistics on the join attributes. Although such statistics, such as correlated samples [49, 120, 64] are available, they require the creation of indexes for every possible combination of join attributes. This requirement considerably constrains their applicability in multi-way joins, leading to a reduced exploration of plan search space. Consequently, even advanced query optimizers often resort to data assumptions such as uniform distribution, inclusion, and independence [69]. However, these assumptions are often rare in practice and thus can lead to the selection of highly suboptimal query plans [63]. Given that the development and maintenance of a range of statistics entail substantial effort, it is predominantly the more established database systems that incorporate them.

Traditional query optimizer architectures employ either exhaustive or heuristic strategies to enumerate query plans. An exhaustive strategy enumerates every possible query plan to ensure plan optimality. However, an exhaustive enumeration becomes computationally expensive for queries with a large number of joins. Conversely, heuristic strategies are designed to enumerate a single query plan that is, ideally, near the optimal one. However, while more efficient in the plan search, heuristic strategies can result in significantly suboptimal query plans. In the case of large queries, shifting from an exhaustive to a heuristic strategy can result in a notable decline in the quality of query plans. An even worse scenario is to reimplement the optimizer to a different plan enumeration strategy causing significant expenses in terms of development. The suboptimality of the query plan can be so severe that the time saved during the plan search becomes negligible when compared to its large execution time. In other words, the saved time during the search by avoiding exhaustive enumeration is outweighed, due to the inefficiencies of executing the query with a suboptimal plan. This exhibits a critical trade-off in query optimization, where the choice of strategy can have significant implications on overall query performance. A strategy that incorporates elements of randomness to enumerate multiple query plans can be an intermediate strategy [104, 43, 106, 88, 107, 44]. This approach, while still more computationally efficient than exhaustive enumeration, diversifies the set of considered plans and thereby reduces the gap between plan optimality and optimization time. Randomized heuristics provide better chances to find superior query plans, compared to the plans selected by heuristics, by allocating additional time to plan enumeration. Thus, while randomized heuristic strategies require an extended optimization time, their practicality is more feasible in real-world applications. However, the random nature of plan search space exploration introduces an element of unpredictability and reduced interpretability in optimizer performance.

The most important data in query optimizer is cardinality estimations. The errors made in estimating cardinalities can drastically affect the effectiveness of the other components and hence compromise the robustness of the query optimizer. Despite several decades of advancement in cardinality estimation techniques, disastrously suboptimal plans, particularly for large queries, are still frequently selected, leading to substantial delays in executing queries. Q-error [81], the standard metric for quantifying the error of individual cardinality estimates, has been widely adopted as a surrogate for query plan optimality in recent work on synopses-based and learning-based cardinality estimation [111, 52, 16, 54, 26]. However, the only result connecting Q-error with plan optimality is an upper-bound on the cost of the worst possible plan computed from a set of cardinality estimates. There is no connection between Q-error and the real query plans generated by standard query optimizers. Thus, while Q-error is commonly used to evaluate individual cardinality estimates, it falls short as an indicator of suboptimal plans [32, 84]. Q-error, as an indicator, is solely based on estimation errors without taking into account their implications on the cost function and plan enumeration, which plays a crucial role in bridging the gap between the estimation error and the selection of an optimal plan.

1.2 Primary Contributions

In this section, we summarize the primary contributions of this dissertation as follows:

Online Sketch-based Query Optimization. We introduce COMPASS, a novel query optimization paradigm for in-memory databases based on a single type of statistics – Fast-AGMS sketches [19]. In COMPASS, query optimization and execution are intertwined. Selection predicates and sketch updates are pushed-down and evaluated online during query optimization. This allows Fast-AGMS sketches to be computed only over the relevant tuples – which enhances cardinality estimation accuracy. Fast-AGMS sketches, necessary for a given join cardinality estimation, are merged on demand, by addressing scalability by incrementally creating multi-way sketches from two-way sketches. Plan enumeration is performed over the query join graph by incrementally composing attribute-level sketches – not by building a separate sketch for every subplan. We prototype COMPASS in MapD [2] – an open-source parallel database – and perform extensive experiments over the complete JOB [65] benchmark. The results prove that COMPASS generates better execution plans – both in terms of cardinality and runtime – compared to four other database systems. Overall, COMPASS achieves a speedup ranging from 1.35X to 11.28X in cumulative query execution time over the considered competitors. We present this work in Chapter 3.

Spanning Tree-based Query Plan Enumeration. We propose a spanning tree-based plan enumeration strategy. This is an intermediate strategy between exhaustive and heuristic query plan enumerations, approaching the query optimization problem from a different perspective. We define the problem of finding an optimal query plan as finding spanning trees with low costs. This approach empowers the utilization of a series of spanning tree algorithms, thereby enabling systematic exploration of the plan search space over the join graph. Capitalizing on the polynomial time complexity of spanning tree algorithms, we present Ensemble Spanning Tree Enumeration (ESTE). This strategy utilizes two conventional spanning tree algorithms, Prim's and Kruskal's, together to enhance the robustness of the query optimizer. In ESTE, multiple query plans are enumerated exploring different areas of the search space. We show that ESTE is more robust in identifying efficient query execution plans for large queries. In the case of data and workload demand increase, we believe our approach can be a cheaper alternative to maintain optimizer robustness by integrating additional spanning tree algorithms – rather than completely changing and developing the optimizer to another plan enumeration algorithm. Experimental evaluations show relative effectiveness compared to exhaustive strategy in optimization time as well as demonstrate a notable improvement over one of the top-performing heuristics algorithms in terms of plan optimality. We present this approach in more detail in Chapter 4.

Sub-optimal Join Order Identification. In order to identify suboptimal query plans, we propose a learning-based method having as its main feature a novel measure called L1-error. Similar to Q-error [81], L1-error requires complete knowledge of true cardinalities and estimates for all the subplans of a query plan. Unlike Q-error, which considers the estimates independently, L1-error is designed as a permutation distance between true cardinalities and estimates for all the subplans having the same number of joins. This accounts for the fact that certain estimation errors may have more impact on the selection of an optimal plan than others, reflecting the reality that the plan search algorithms are more tolerant towards some estimation inaccuracies. Moreover, L1-error takes into account errors relative to the magnitude of their cardinalities and gives larger weight to small multi-way joins. This takes into account the reality that the accuracy of early-stage multi-way joins is more critical, and they are often more accurate than those joins at later stages. Our experimental results confirm that, when L1-error is integrated into a standard decision tree classifier, it leads to the accurate identification of suboptimal plans across four different benchmarks. This accuracy can be further improved by combining L1-error with Q-error into a composite feature that can be computed without overhead from the same data. L1-error is evaluated using cardinality estimations from one well-known and one recently proposed query optimizer. The technical details are presented in Chapter 5.

1.3 Outline

Chapter 2 describes the entire search procedure of the classical cost-based query optimization in relational database systems. Chapter 3 introduces COMPASS, a novel query optimization paradigm for in-memory databases based on a single type of statistics. Chapter 4 describes ESTE, an ensemble, spanning tree-based plan enumeration strategy. Chapter 5 introduces L1-error, a new indicator tailored to accurately identify suboptimal plans. Chapter 6 concludes the dissertation with a discussion of future work.

Chapter 2

Query Optimization

In this chapter, we describe the entire search procedure of the classical cost-based query optimization in relational database systems. Query optimizer consists of four elements, namely cost model, cardinality estimation, plan search space, and plan enumeration – which are depicted in Figure 2.1.

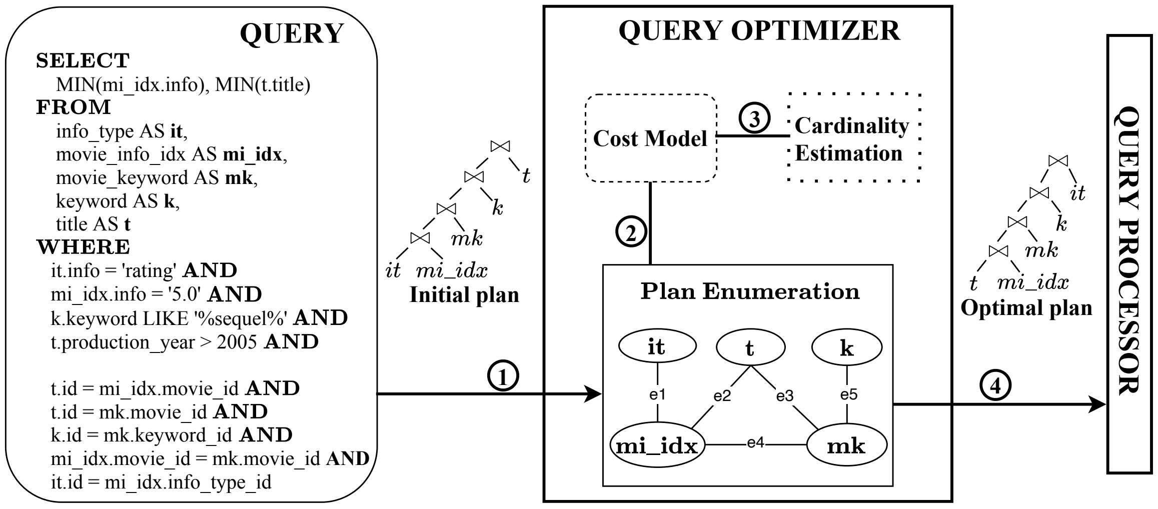


Figure 2.1: Cost-based query optimization architecture in relational database systems.

The goal of query optimization [69, 65, 63, 17] is to find an optimal query execution plan. Note that there can be more than one query plan with the minimum cost. The selected optimal query plan is expected to exhibit the fastest runtime for a given query. *Search space* is a set of all query plans – combinations of relational algebra operators – which are semantically equivalent. *Plan enumeration* is the procedure to enumerate the query plans in a given search space. The number of query plans is factorial in the number of relations. Thus, evaluating all of the query plans is not practical for a large number of relations. The execution time of a query plan cannot be determined without running the query thus alternative *cost functions* are defined. A common cost function is the sum of the intermediate results (cardinality) produced by the physical operators in the query execution plan. The function captures the correlation between the amount of accessed data and execution time – which is true in general. *Estimating the cardinality* of a relational algebra operator is itself a difficult problem. This knowledge is captured by incomplete statistics – or synopses – about the data [20]. For example, attribute histograms and the number of distinct values are optimal for selection predicates, while correlated samples are better for join predicates. With statistics, the cardinality can only be estimated – it is not exact. While accurate for simple predicates over a small number of attributes, cardinality estimation becomes harder for correlated predicates and multi-way joins. This is not necessarily a problem if all the plans are equally impacted. However, estimation errors vary widely across sub-plans and this can potentially lead to a highly sub-optimal plan. In the following sections, we describe each of these components in greater detail, which serves as essential background knowledge for this dissertation.

2.1 Query Join Graph and Query Plan

2.1.1 Join Graph

In Figure 2.2, the SQL statement has 4 selection predicates – point, subset, and range – σ on tables *it*, *mi_idx*, *k* and *t*, and 5 join predicates connecting 5 tables. Further, the query can be illustrated as an undirected join graph $G(V, E)$ in which every table is represented as a node $v \in V$ connected by edges $e \in E$ for every join predicate. Vertices V are connected as a single component with a list of weighted edges $E \subseteq \{(v1, v2, w) \mid v1, v2 \in V, w \in \mathbb{R}^+\}$. For example, the join predicate *it.id* = *mi_idx.info_type_id* is the edge $e1$ which connects the vertices *it* and *mi_idx* in the join graph. The edges $e2$, $e3$, and $e4$ form the cycle between *mk*, *t*, and *mi_idx*. In the case of no cycles in the join graph, the number of edges is $|E| = |V| - 1$. This is also the minimum number of edges required to have a connected graph without cross-joins – non-existent edges $e \notin E$ in the join graph. The topology of a join graph can be classified as either a chain, cycle, star, or clique. The number of edges increases from $|V| - 1$ for chain and star to $(|V| \cdot |V - 1|)/2$ for clique while the cyclic join graph falls between these two extremes.

```
SELECT
   MIN(mi_idx.info), MIN(t.title)
FROM
   info_type AS it,
   movie_info_idx AS mi_idx,
   movie_keyword AS mk,
   keyword AS k,
   title AS t
WHERE
   // selection predicates
   it.info = 'rating' AND
   mi_idx.info = '5.0' AND
   k.keyword LIKE '%sequel%' AND
   t.production_year > 2005 AND

   // join predicates
   t.id = mi_idx.movie_id AND
   t.id = mk.movie_id AND
   k.id = mk.keyword_id AND
   mi_idx.movie_id = mk.movie_id AND
   it.id = mi_idx.info_type_id
```

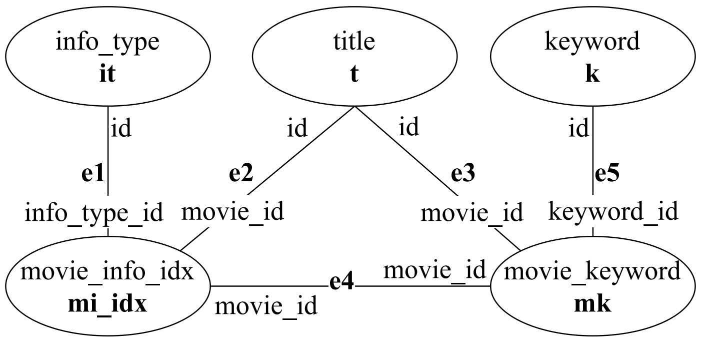


Figure 2.2: SQL statement and corresponding join graph.

2.1.2 Query Plan

A query plan $\mathcal{P}$ of a join graph G is an ordered sequence of $|V| - 1$ edges that cover all V vertices – tables to be joined, forming a single connected component. Four query plans are displayed in Figure 2.3 which are derived from the join graph in Figure 2.2. A plan tree is a connected, acyclic subgraph of the original graph that is built on existing edges $e \in E$ of the join graph. It also dictates the physical operators, such as join or scan, that will be deployed during the query execution. The order of edges in the plan tree corresponds to a path which is the join order of the tables in the query plan. For instance, in Figure 2.2, the edge sequence $(e1, e2, e3, e5)$ corresponds to the plan tree $\mathcal{P} = \{(it{-}\,mi\_idx), (mi\_idx{-}\,t), (t{-}\,mk), (mk{-}\,k)\}$ which results into the join order $(it \bowtie mi\_idx \bowtie t \bowtie mk \bowtie k)$. While the plan tree does not include cycles, this does not mean that the joins corresponding to the edges not included in the plan tree are dropped from the query plan – edge $e4$. Instead, these join predicates are kept and treated as filters. However, in the case of transitive join predicates, these joins can be completely eliminated from the query plan. Concretely, the example tree $\mathcal{P}$ does not include the edge $e4$ which is the join between ($mi\_idx - mk$) corresponding to the join predicate $mi\_idx.movie\_id = mk.movie\_id$. This predicate can be evaluated together with the join ($t - mk$) since mi_idx and mk are both part of the plan tree. However, this is not necessary for this query because the join cycle is based on transitive predicates – which already implies the ignored join condition. This simplification is not applicable to general cyclic queries such as those considered by worst-case optimal join algorithms [87].

2.1.2.1 Query Sub-Plan

A query subplan is defined as a query plan over a subgraph of the join graph. For instance, $\mathcal{P}_{it \bowtie mi\_idx \bowtie t} = \{(it - mi\_idx), (mi\_idx - t)\}$ corresponds to the query subplan over tables *it*, *mi_idx*, and *t*, and its associated subgraph. We refer to join order when we speak of the query plan and use the terms *query plan* and *join order* interchangeably.

2.1.2.2 Left-deep and Right-deep Query Plan

The sequence, in which edges are chosen, directly influences the resulting shape of the plan tree formed from the join graph. Given that an edge inherently involves a connection between two vertices, the shape of the corresponding plan tree invariably adheres to a binary shape. The left-deep tree, depicted in Figure 2.3 (a), – sets the right child to be a base table as a leaf node at every join. Right-deep tree, shown in Figure 2.3 (b), is the same tree shape but differs in the execution of internal by setting the left child to be a base table. The zig-zag tree establishes a balance by alternating the roles of the left and right children at each join level. In all cases, the invariant is to keep a single acyclic connected component and incrementally add adjacent edges until all the vertices are covered.

2.1.2.3 Bushy Query Plan

Bushy trees exhibit a distinctive structure that sets them apart from strictly left-deep or right-deep trees. A defining attribute of bushy trees is the possibility for an internal node to have both of its children as internal nodes. The bushy tree depicted in Figure 2.3 (c)

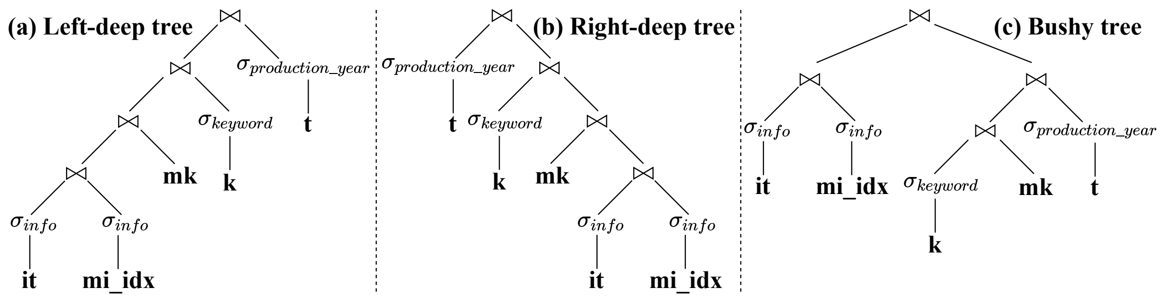


Figure 2.3: Plan tree structures.

has the root node which has two subtrees, $\mathcal{P}_{it \bowtie mi\_idx} = \{(it - mi\_idx)\}$ and $\mathcal{P}_{k \bowtie mk \bowtie t} = \{(k - mk), (t - mk)\}$, as its child nodes forming two separate acyclic connected components. Unlike their left or right-deep counterparts, bushy trees do not follow a strictly linear pattern in their order. Therefore, the bushy plan can be effectively capitalized on opportunities for parallelism and flexible execution ordering to optimize query performance.

2.2 Search Space

The search space is the set of all possible query plans in which all tables are joined [65]. Two query plans with the same shape but with different join orders are two different query plans. We first define the search space in terms of the number of binary trees using nodes V of the join graph. Further, we define the search space in terms of the number of spanning trees based on edges E of the join graph. We show that operating on edges incorporates connectivity information inherent in the join graph which naturally avoids query plans involving cross-joins.

2.2.1 Search Space in terms of Graph Nodes

In terms of the join graph nodes, the search space $t_b(V)$ is the set of distinct binary tree shapes that can be defined with $|V|$ nodes, where each node can be any of the V tables and hence, are permutable. The number of distinct binary tree shapes – denoted as $b(V)$ – extracted from the join graph G is calculated by the Catalan numbers [18]. Since each node can be any of the query tables, there are $V!$ ways to assign the tables to the nodes for each tree shape. Thus, the total size of the search space is:

$$t_b(V) = \frac{(2 \times |V|)!}{(|V|+1)!} \tag{2.1}$$

The number of binary tree shapes growths exponentially [18] with V including cross-joins – which is $\Omega\left((4^{|V|})/|V|^{3/2}\right)$. Figure 2.4 shows the search space coverage by different enumeration strategies. The x-axis presents enumeration strategies from heuristical strategy (faster search time) to exhaustive strategy (preserved plan optimality). The y-axis describes the search space coverage. We use the area of rectangles of different sizes to describe the correlation between the two axes. The largest solid rectangle covers all possible binary trees including cross-joins. The second solid rectangle covers bushy, left-deep, right-deep, and zig-zag trees.

2.2.2 Search Space in terms of Graph Edges

While the standard approach is defined over tables, we define the search space over joins – edges in the join graph – which directly excludes cross-joins. The size of the search space is given by the number of spanning trees that can be extracted from the join graph G – denoted as $t_s(V, E)$ – which is upper bounded by the number of ordered edge arrangements of size $(|V| - 1)$ selected from E:

$$t_s(V, E) \leq \frac{|E|!}{(|E| - |V| + 1)!} \tag{2.2}$$

The reason for having ordered arrangements instead of only combinations [10] is because the order in which edges are selected matters and different edge orders result in different join orders, thus, different query plans. This results in a considerable reduction in the number of all possible binary trees with $|V|$ nodes – illustrated in Figure 2.4. In the case of a query without cycles – a tree join graph with $(|V| - 1)$ edges – the bound is tight since $|E| = |V| - 1$ and the number of spanning trees is $t_s(V, E) = (|V| - 1)!$ where $0!$ to be equal to 1. The value of the bound increases with the number of joins, reaching its maximum value for a clique query with $|E| = |V| \times (|V| - 1)/2$ edges.

The number of spanning trees $t_s(V, E)$ increases at a slower rate than the bound because of the redundancy incurred by cycles. In this case, many arrangements result in invalid query

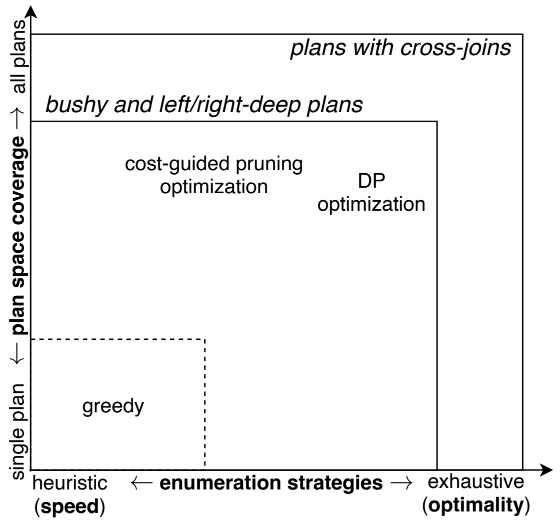


Figure 2.4: Plan search space.

plans that cover only a subset of the tables and include cycles. In Figure 2.2, an example of such arrangement is $\{(it - mi\_idx), (mi\_idx - t), (t - mk), (mi\_idx - mk)\}$. The exact value of $t_s(V, E)$ depends heavily on the topology of the join graph. For our example query, $t_s(V, E)$ is 72 while the bound is $5! = 120$. The rest is the number of invalid query plans – 48 invalid spanning trees. Thus, $t_s(V, E)$ is only 60% of the bound. Although this may seem small, as the number of tables in a query increases, so does $t_s(V, E)$, resulting in a huge search space that becomes even larger when taking into account the availability of indexes and the types of join algorithms. Thus, given the factorial size of both search spaces, finding an optimal query plan remains NP-hard [39].

2.2.2.1 Cross-join Search Space

Cross-join (Cartesian product) is to join two tables over a non-existent edge $e \notin E$ or join two components that do not share a connecting edge (join predicate) in the join graph. For example, in Figure 2.2, the join order $(it \bowtie mi\_idx \bowtie k \bowtie mk \bowtie t)$ includes the cross-join between subplan $(it \bowtie mi\_idx)$ and table k. Since this type of join does not require any condition to match data between the two tables, it combines every row of one table with every row of the other table. This is a costly join operation that results in a large number of rows in the outcome table. Thus query plans including cross-joins are highly resource-intensive due to the significant increase in the number of rows within the resulting intermediate tables. The binary trees that involve cross-joins can be considered a separate subspace – the largest rectangle in Figure 2.4. Despite the possibility that a plan involving a cross-join could potentially lead to an optimal plan, cross-joins are typically avoided due to their high costs. Nonetheless, even the resulting search subspace that excludes binary trees involving cross-joins still presents a computational challenge due to its vastness.

2.2.2.2 Left-deep Search Space

A standard approach for reducing the size of the search space is to constrain the shape of the query plans [80, 78]. This means only considering plan trees with a certain shape. Bushy trees and left-deep trees are two principal shapes of binary trees. Focusing on specific types of binary trees helps to streamline the enumeration process and manage the complexity of the search space, thereby simplifying the discovery of an optimal query plan. However, this reduction in search space may inadvertently omit optimal query plans. This balances the computational feasibility against the query plan optimality. The objective, therefore, is to find a query plan – optimal plan or near the globally optimal query plan – within the reduced search space. In order to limit the search space for the left-deep trees, a single connected component – a single subtree – without cycles must be maintained. Thus every node and edge has to be recursively considered at any step of the join order and ordered edge sequence, respectively. An edge that both nodes are not a member of the current single connected component is not considered. We repeat this procedure until we cover all the nodes, making sure that we always maintain a single connected component without cycles. The number of the spanning trees generated from the example query is 72 including 36 left-deep and 36 bushy query plans. Although the number of query plans considered in the left-deep search can be smaller, it does not mean the optimal plan cannot be selected. In the case of an optimal plan having a left-deep tree shape, reducing the initial search space to the left-deep search space does not affect the result but can improve the search time.

2.3 Cost Function

In order to evaluate and compare the valid query plan trees derived from the join graph, the query optimizer assigns a quantifiable cost to each candidate plan tree using a predefined analytical cost function. The optimal query plan is the plan tree with the minimum cost, as it is anticipated to have the fastest execution time. However, formulating a cost function that can accurately reflect execution time is challenging. While disk-based cost functions remain crucial, main memory cost functions can provide an accurate representation of the actual query runtime in modern database systems. Virtually all main memory cost functions are defined in terms of the number of tuples processed by the operators in the query plan, while disk-based cost functions consider block reads instead of tuples. In main memory setup, several cost functions have been employed in previous studies [83, 32, 112, 65, 31]. A simple cost model $\mathcal{C}_{out}$ defined in Equation 2.3 evaluates a query plan $\mathcal{P}$ based on the cumulative size, i.e., cardinality, of all intermediate results and it does not include the cardinality of the base tables:

$$\mathcal{C}_{out}(\mathcal{P}) = \begin{cases} 0 & \textbf{if } \mathcal{P} = R \\ |\mathcal{P}| + \mathcal{C}_{out}(\mathcal{P}_1) + \mathcal{C}_{out}(\mathcal{P}_2) & \textbf{if } \mathcal{P} = \mathcal{P}_1 \bowtie \mathcal{P}_2 \end{cases} \tag{2.3}$$

This cost function is solely determined by the join order, making it independent of the physical operators. In the left-deep search space, since the cardinality of the join at the root of the query tree is identical across all the query plans, the cost can be simplified to include only the cardinality of the intermediate joins. The order in which the tables are joined is pivotal as it can significantly influence the execution time. Inefficient ordering of table joins could result in unnecessary data movement, leading to notable resource wastage.

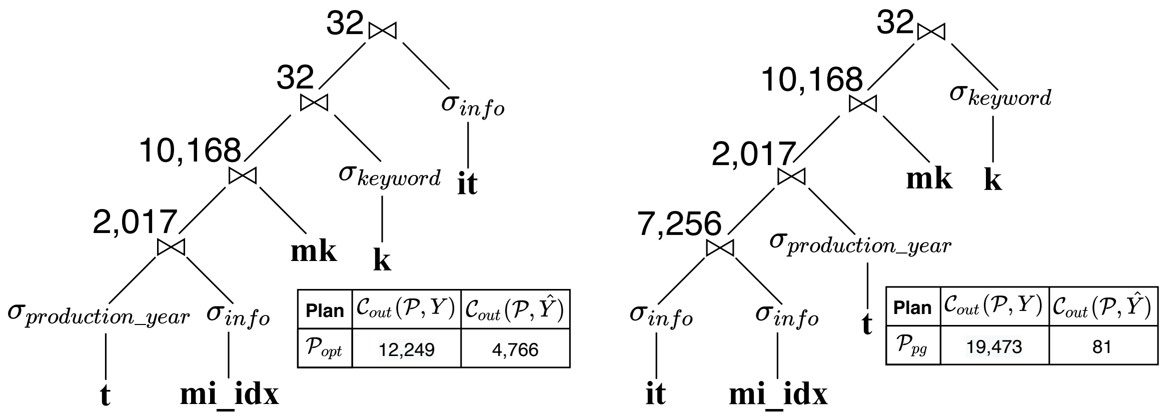
The costs of join orders $\mathcal{P}_{opt}$ and $\mathcal{P}_{pg}$ are displayed in Figures 2.5 and 2.6 – second columns in the tables. They are obtained by summing up the exact cardinalities Y of the three intermediate and one final join. For example, in Figure 2.5a, in the case of the optimal plan $\mathcal{P}_{opt}$, the cost $\mathcal{C}_{out}(\mathcal{P}_{opt}, Y) = 2,017{+}10,168{+}32{+}32 = 12,249$ is the sum of the cardinalities of subplans corresponding to the 2-way join $\mathcal{P}_{t \bowtie mi\_idx}$, the 3-way join $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk}$, the 4-way join $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk \bowtie k}$, and the 5-way join $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk \bowtie k \bowtie it}$, respectively. It is important to notice that the cardinality of $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk}$ is not the cardinality sum of $\mathcal{P}_{t \bowtie mi\_idx}$ and $\mathcal{P}_{mk \bowtie t}$ – edges ($mi\_idx - t$) and ($t - mk$) from the join graph – which are $2,017$ and $1.2M$, respectively. This is because only the tuples in the 2-way join $t \bowtie mi\_idx$ are subsequently joined with mk – not all the tuples in t.

$$\mathcal{C}(\mathcal{P}) = \begin{cases} \tau \times |R| & \textbf{if } \mathcal{P} = R \vee \mathcal{P} = \sigma(R) \\ |\mathcal{P}| + |\mathcal{P}_1| + & \textbf{if } \mathcal{P} = \mathcal{P}_1 \bowtie^{HJ} \mathcal{P}_2 \\ \quad \mathcal{C}(\mathcal{P}_1) + \mathcal{C}(\mathcal{P}_2) & \\ \mathcal{C}(\mathcal{P}_1) + \lambda \times |\mathcal{P}_1| & \textbf{if } \mathcal{P} = \mathcal{P}_1 \bowtie^{INL} \mathcal{P}_2 \wedge \\ \quad \times \max(\frac{|\mathcal{P}_1 \bowtie R|}{|\mathcal{P}_1|}, 1) & (\mathcal{P}_2 = R \vee \mathcal{P}_2 = \sigma(R)) \end{cases} \tag{2.4}$$

Leis et al. [65, 63] proposed another main memory cost function that incorporates physical operators shown in Equation 2.4. This cost function $\mathcal{C}$ recursively sums the cost of all the nodes in the query plan starting from the leaves – corresponding to scan operators – and following through the joins. In the leaf nodes, the size of a base table R is multiplied by a parameter $\tau = 0.2$ to differentiate between sequential and index scans. Notice a base table with selection predicates σ still requires a full table scan. For the intermediate join nodes, the cost function considers two different join operator implementations – hash join $\bowtie^{HJ}$ and index nested-loop join $\bowtie^{INL}$. The hash table is built on the child with the smallest cost – as in zig-zag trees. To differentiate between hash lookup and index lookup, parameter $\lambda = 2$ is used under the assumption that indexes are available on all the join attributes – otherwise, the hash join is utilized. Even though the cost function considers different physical operators, cardinality remains the main factor. This is in line with standard disk-based cost functions, which replace tuple cardinality with block cardinality. However, both types of cost functions may be significantly impacted by underlying poor cardinality estimates. This, in turn, can lead the query optimizer to choose suboptimal query plans.

2.4 Cardinality Estimation

The cost model is defined in terms of the cardinality Y of the join operators. In Equations 2.3 and 2.4, the cost function considers $\mathcal{P}$, $\mathcal{P}_1$, and R as input relations to compute the cardinality of the join operators. However, exact cardinality Y – the number of tuples that are expected to be produced – of $\mathcal{P}$, $\mathcal{P}_1$, and R can be computed exactly only by executing the join. This is a paradoxical situation in the process of query optimization. The query optimization is supposed to determine the optimal plan in which to perform the joins without actually performing them. This is where cardinality estimation comes into the picture. The role of cardinality estimation is to "guess" the exact cardinality Y without executing the join. In PostgreSQL [88], cardinality estimates are computed based on histograms, most frequent values, and distinct element statistics on the base tables. Any join estimate is computed by combining these statistics into simple arithmetic formulas that make general assumptions on uniformity, inclusion, and independence [69]. Consequently, these estimations $\hat{Y}$ are fed into the cost function in lieu of exact cardinalities, allowing the query optimizer to compute the cost of a plan. Henceforth, two plans can be ranked according to their estimated cost – instead of the exact cost. As long as the ranks of the plans based on the exact and estimated cost are identical, estimates can be a direct replacement for the exact cardinalities.



Plan	$\mathcal{C}_{out}(\mathcal{P}, Y)$	$\mathcal{C}_{out}(\mathcal{P}, \hat{Y})$
$\mathcal{P}_{opt}$	12,249	4,766

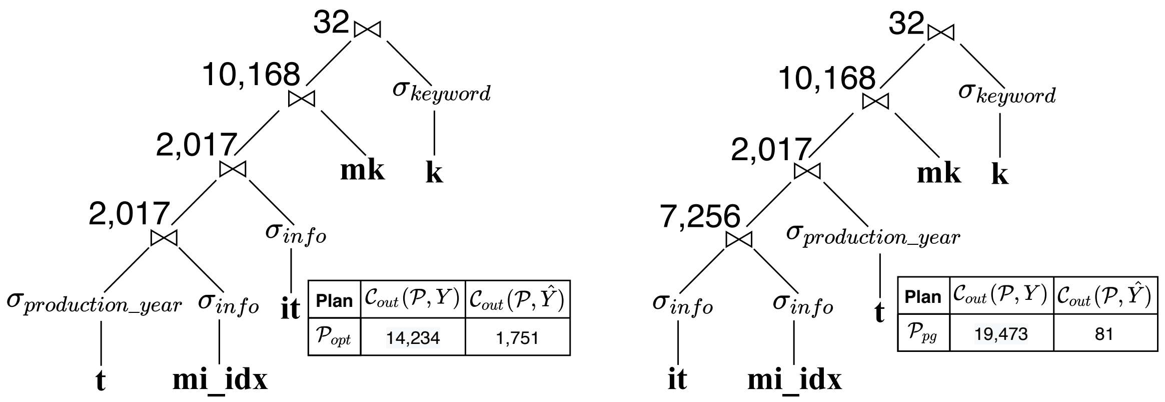
(a) Optimal plan

Plan	$\mathcal{C}_{out}(\mathcal{P}, Y)$	$\mathcal{C}_{out}(\mathcal{P}, \hat{Y})$
$\mathcal{P}_{pg}$	19,473	81

(b) PostgreSQL plan

Figure 2.5: Plans selected by exhaustive plan enumeration.

From a statistical perspective, the goal of cardinality estimation is to minimize the difference between Y and $\hat{Y}$. However, from the cost model perspective, this is not necessary. Instead, what is required is that the costs of two different query plans, $\mathcal{C}_{out}(\mathcal{P}_{opt}, Y)$ and $\mathcal{C}_{out}(\mathcal{P}_{pg}, \hat{Y})$, satisfy the same query plan when computed based on estimates $\hat{Y}$ and when using the exact cardinalities Y. While accurate estimates imply the optimal query plan, accurate estimates are not required as long as they have similar errors. For example, estimates that are 100$\times$ larger than the exact value are extremely inaccurate. However, the cost model treats them identically and results in two costs that have the same query plan. Thus, estimation does not impact the cost model negatively. Obtaining accurate estimates is a challenging task. Despite a rich number of cardinality estimation techniques [64, 82, 46, 16, 42, 91, 52, 70, 68, 54, 55, 113, 89], cardinality estimation error is inevitable triggering a chain of more errors in subsequent optimizer components. These challenges underline the importance of ongoing research into improving the accuracy of cardinality estimates to enhance the efficiency of query optimization.



Plan	$\mathcal{C}_{out}(\mathcal{P}, Y)$	$\mathcal{C}_{out}(\mathcal{P}, \hat{Y})$
$\mathcal{P}_{opt}$	14,234	1,751

(a) Optimal plan

Plan	$\mathcal{C}_{out}(\mathcal{P}, Y)$	$\mathcal{C}_{out}(\mathcal{P}, \hat{Y})$
$\mathcal{P}_{pg}$	19,473	81

(b) PostgreSQL plan

Figure 2.6: Plans selected by greedy plan enumeration.

The estimated costs for the plans $\mathcal{P}_{opt}$ and $\mathcal{P}_{pg}$ are depicted in Figures 2.5 and 2.6 – third columns in the tables. The estimated costs are computed using the Postgres join cardinality estimates [88]. In Figure 2.5a, the estimated cost of the optimal plan $\mathcal{P}_{opt}$ is $\mathcal{C}_{out}(\mathcal{P}_{opt}, \hat{Y}) = 1,708 + 3,056 + 1 + 1 = 4,766$. It is obtained by adding the estimates for the same 2-way join $\mathcal{P}_{t \bowtie mi\_idx}$, 3-way join $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk}$, 4-way join $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk \bowtie k}$, and 5-way join $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk \bowtie k \bowtie it}$. The estimated cost is smaller than the exact cost by a factor of $2.6X$ – due to the underestimation of every join cardinality in the query plan. For the $\mathcal{P}_{pg}$ plan, the estimated cost is 81, compared to the exact cost of $19,473$. This is a factor of more than $240X$ smaller which is due to severe underestimations of every join cardinality. In this case, the relative order between $\mathcal{P}_{opt}$ and $\mathcal{P}_{pg}$ is not the same based on the exact and estimated cost. The reason is the inconsistency of the cardinality estimates.

2.5 Plan Enumeration

Within the scope of a given search space, the process of plan enumeration involves generating and evaluating a vast number of binary trees described in Section 2.2. Plan enumeration generates and evaluates various query plans, each is semantically equivalent but with varying costs. The challenge is finding an optimal plan, which entails the minimum cost, by exhaustively enumerating a massive number of candidate query plans. However, exhaustively enumerating all possible binary trees becomes infeasible for queries with a large number of tables. On one side, excluding cross-jons and limiting the search space to left-deep tree shapes reduces the total number of plan trees, thus expanding the feasibility limits of the exhaustive enumeration strategy for queries with a relatively broader range of complexity. On the other side, various heuristical strategies are used to partially explore a given search space. It further reduces the number of query plans to be generated and evaluated. Although both optimizations, space reduction and partial enumeration, reduce the search time, the plan enumeration process may potentially overlook optimal query plans since the optimal plans may be excluded from the radar of the plan enumeration in the resulting limited search space.

2.5.1 Exhaustive Enumeration

To exhaustively enumerate a given search space, each individual plan tree is produced and its associated cost is calculated. However, exhaustively enumerating all possible spanning trees becomes infeasible for complex queries with a large number of tables and joins. On one side, excluding cross-jons and limiting the search space to left-deep tree shapes reduces the total number of spanning trees, thus expanding the feasibility limits of the exhaustive enumeration strategy for queries with a relatively broader range of complexity. Figure 2.5 depicts the optimal plan $\mathcal{P}_{opt}$ and Postgres plan $\mathcal{P}_{pg}$ selected by exhaustive enumeration. The optimal plan $\mathcal{P}_{opt} = (t \bowtie mi\_idx \bowtie mk \bowtie k \bowtie it)$ is computed based on the exact cardinalities Y while the Postgres plan $\mathcal{P}_{pg} = (it \bowtie mi\_idx \bowtie t \bowtie mk \bowtie k)$ is computed based on the Postgres estimated cardinalities $\hat{Y}$. These plans are shown along with their exact intermediate costs. The reason they are different is that the Postgres estimates are inaccurate, thus resulting in the selection of a different join order. The cost of a plan with n joins is defined as the sum between the cardinality of the n-way join and the cost of the contained subplan with $n-1$ joins having minimum cost. For instance, as shown in Figure 2.5a, the cost of the 3-way join plan $\mathcal{P}_{t \bowtie mi\_idx \bowtie mk}$ based on the exact cardinalities Y adds the cardinality of the join to the minimum cost of the three contained 2-way join sub-plans $\mathcal{P}_{t \bowtie mi\_idx}$, $\mathcal{P}_{mi\_idx \bowtie mk}$ and $\mathcal{P}_{t \bowtie mk}$ to obtain $10,168 + min\{2,017;\ 47K;\ 1.2M\} = 12,185$. As the complexity of a query increases, exhaustive enumeration faces significant computational challenges and is mitigated through optimizations such as dynamic programming and cost-based pruning [100, 79, 21]. Although these optimization techniques can reduce the number of plans evaluated, they ensure the discovery of the plan tree with minimum cost.

2.5.2 Heuristical Enumeration

The exhaustive enumeration can be simplified by heuristical strategies by partially exploring the given search space. Thus it further reduces the number of query plans to be generated and evaluated. For example, greedy algorithms [106, 104, 27, 15, 85, 56] directly compute a single plan. Even though the decision at every step is locally optimal, there is no guarantee that the final plan has the minimum cost among all the plans. This is due to conditioning the available choices at a step on previous decisions. Consequently, greedily selected query plans tend to be less optimal compared to their exhaustive counterparts. Figures 2.5 and 2.6 exhibit this issue for the optimal plan $\mathcal{P}_{opt}$ and the Postgres plan $\mathcal{P}_{pg}$. The cost of the optimal plan $\mathcal{C}_{out}(\mathcal{P}_{opt}, Y) = 14,234$ selected by a greedy algorithm is larger than the optimal plan $\mathcal{C}_{out}(\mathcal{P}_{opt}, Y) = 12,249$ that is found exhaustively. In the case of Postgres, the greedy plan $\mathcal{C}_{out}(\mathcal{P}_{pg}, \hat{Y}) = 81$ has the same cost as the exhaustive plan $\mathcal{C}_{out}(\mathcal{P}_{pg}, \hat{Y}) = 81$. The lesser optimality of greedy query plans is expected given the smaller number of subplans considered by the greedy algorithm. At the same time, considering fewer subplans and building a plan bottom-up to find the optimal query plan means relying on estimates of smaller join size, which are – in principle – more accurate [65]. Thus, while exhaustive enumeration requires consistent estimation across all join sizes, the greedy algorithm is more sensitive to simpler estimates. Therefore, the reduction in the search space can be compensated by consistent estimation of small joins.

Chapter 3

Online Sketch-based Query Optimization

Consider query 6a from the JOB benchmark [65]:

```
SELECT MIN(k.keyword), MIN(n.name), MIN(t.title)
FROM cast_info ci, keyword k, movie_keyword mk, name n, title t
WHERE
  ▷ selection predicates
    k.keyword = 'marvel-cinematic-universe' AND
    n.name LIKE '%Downey%Robert%' AND t.production_year > 2010 AND
  ▷ join predicates
    k.id = mk.keyword_id AND t.id = mk.movie_id AND t.id = ci.movie_id AND
    ci.movie_id = mk.movie_id AND n.id = ci.person_id
```

The query has 3 selection predicates – point, subset, and range – and joins 5 tables with 5 join predicates – there is a triangle sub-query between tables *t*, *mk*, and *ci*. The corresponding join graph is depicted in Figure 3.1. For each join, the graph contains a named edge $e1$–$e5$ that connects the tables involved in the join predicate. For example, edge $e1$ represents the join predicate *k.id = mk.keyword_id*.

Figure 3.1 also includes the execution plans together with their cost – the total cardinality

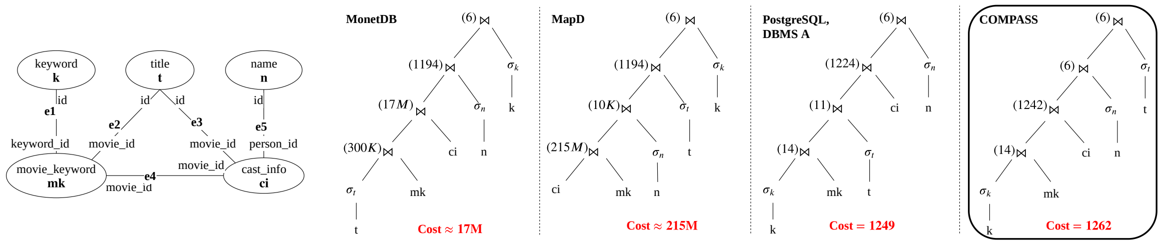


Figure 3.1: Join graph and corresponding execution plans for query JOB 6a. The numbers represent cardinality.

of the intermediate results defined in Equation 2.3 – for COMPASS and the four other databases considered in the work. Although all the plans are left-deep trees, their cost ranges from 1, 249 to 215 million tuples. This is entirely due to the statistics used for cardinality estimation. MapD [2] does not use any statistics, thus its cost is orders of magnitude higher. The plan is determined by sorting the tables in decreasing order of their size – the number of tuples. MonetDB [3] has a rule-based optimizer with minimum support for statistics [41] which generates a better plan. The reason why both of these systems have primitive optimizers is because they are relatively "young" and are targeted at modern architectures. They try to compensate for bad plans with highly optimized execution engines that make use of extensive in-memory processing supported by massive multithread parallelism and vectorized instructions. However, this approach is clearly limited.

PostgreSQL [88] and the industrial-grade DBMS A – name anonymized for legal reasons – are "mature" databases with advanced query optimizers. In order to find a much better plan, they use a large variety of statistics. Histograms, most frequent values, and number of distinct values are used to estimate the selectivity of the point predicate on attribute *k.keyword* and of the range predicate on *t.production_year*. The subset LIKE predicate on *n.name* is estimated with table-level samples. Estimating join cardinality requires correlated statistics on the join attributes. While such statistics exist, e.g., correlated samples [49, 120, 64], they require the existence of indexes on every join attribute combination, which severely limits their applicability in the case of multi-way joins. As a result, even advanced optimizers rely on crude formulas that assume uniformity, inclusion, and independence – which are likely to produce highly sub-optimal execution plans [63]. Since implementing and maintaining these many statistics requires considerable effort, it is completely understandable that only mature systems implement them.

Problem. We investigate how to design a lightweight – yet effective – query optimizer for modern in-memory databases. We have two design principles. First, we aim to capitalize on the highly parallel execution engine in the query optimization process. Since query execution is already fast, it is challenging to minimize the overhead incurred by the additional optimization. Second, the type and number of synopses included in the optimizer have to be minimal. Our goal is to employ a single type of synopsis built exclusively for single attributes and without the requirement of additional data structures such as indexes. The challenge is to design a composable – and consistent – synopsis that provides incremental cardinality estimates for the sub-plans generated in plan enumeration.

COMPASS Query Optimizer. We introduce the online sketch-based COMPASS query optimizer. Fast-AGMS sketches are the only statistics present in COMPASS [19]. These sketches are a type of correlated synopses for join cardinality estimation [97, 98] that use small space, can be computed efficiently in a single scan over the data, are linearly composable, and – more importantly – have statistically high accuracy. These properties allow for Fast-AGMS sketches to be computed online in COMPASS by leveraging the optimized parallel execution engine in modern databases. This is realized by decomposing query processing into two stages performed before and after the optimization. In the first stage, selection predicates are pushed-down and Fast-AGMS sketches are built concurrently only over the relevant tuples. Sketches are built for each two-way join independently – not for every combination of tables. In the query optimization stage, plan enumeration is performed over the join graph by incrementally composing the corresponding two-way join sketches in order to estimate the cardinality of multi-way joins. The optimal join ordering is finally passed to the execution engine to finalize the query. As shown in Figure 3.1, COMPASS identifies a plan as good as PostgreSQL and DBMS A, while relying exclusively on a single synopsis – Fast-AGMS sketches. In addition to the novel query optimization paradigm, we make the following technical contributions:

- We present a systematic approach of using sketches for join cardinality estimation in a query optimizer. This includes two-way and multi-way joins. We do this for two types of sketches – AGMS [8] and Fast-AGMS [19].
- We introduce two novel strategies to extend Fast-AGMS sketches to multi-way join cardinality estimation. The first strategy – sketch partitioning – is a theoretically sound

estimator for a given multi-way join. Since it does not support composition, sketch partitioning is not scalable for join order enumeration. The second strategy – sketch merging – addresses scalability by incrementally creating multi-way sketches from two-way sketches. Although this is done heuristically for a certain multi-way join taken separately, all the multi-way joins with a given size are equally impacted. This property guarantees estimation consistency in plan enumeration.

- We prototype COMPASS in MapD and perform extensive experiments over the complete JOB benchmark – 113 queries. The results prove the reduced overhead COMPASS incurs – below 500 milliseconds – while generating similar or better execution plans compared to the four database systems included in Figure 3.1. COMPASS outperforms the other databases both in terms of the number of queries it obtains the best result on, as well as on the cumulative workload execution time.

3.1 Preliminaries

3.1.1 Parallel In-Memory Databases

Database systems for modern computing architectures rely on extensive in-memory processing supported by massive multithread parallelism and vectorized instructions. GPUs represent the pinnacle of such architectures, harboring thousands of SMT threads that execute tens of vectorized SIMD instructions simultaneously. MapD, Ocelot [4], CoGaDB [5], Kinetica [6], and Brytlyt [7] are a few examples of modern in-memory databases with GPU support. They provide relational algebra operators and pipelines for GPU architectures [34, 14, 29] that optimize memory access and bandwidth. This results in considerable performance improvement for certain classes of queries. However, these databases provide only primitive rule-based query optimization – if at all. This limits drastically their applicability to general workloads. In COMPASS, we leverage the optimized execution engine of MapD to build a lightweight – yet accurate and general – query optimizer based on a single type of synopsis.

3.1.2 Sketches

Sketch synopses [20] summarize the tuples of a relation as a set of random values. This is accomplished by projecting the domain of the relation on a significantly smaller domain using random functions or seeds. In the case of join attributes, the correlation between attributes is maintained by using the same random function. While sketches compute only approximate results with probabilistic guarantees, they satisfy several major requirements of a query optimizer for in-memory databases – single-pass computation, small space, fast update and query time, and linearity:

- A sketch is built by streaming over the input data and considers each tuple at most once.
- A basic sketch is composed of a single counter and one or more random seeds – a few bytes. In order to improve accuracy, a standard method is to use multiple independent basic sketch instances. The number of instances is derived from the desired accuracy and confidence levels. In practice, very good accuracy can be achieved with sketches having size in kilobytes.
- The update of a sketch with a new tuple consists of generating one or more random numbers and adding them to the sketch counter. The answer to a query involves simple arithmetic operations on the sketch. In the case of multiple sketches, both the update and query are applied to all the instances. Overall, update and query time are linearly proportional to the sketch size.
- A sketch can be computed by partitioning the input relation into multiple parts, building a sketch for every part, and then merging the partial sketches. This mergeable property makes sketches amenable for parallel processing on modern hardware and can result in linear speedups in update and query time [93, 105].

While previous work addresses how to apply sketches to certain cardinality estimation problems that occur in query optimization, COMPASS is a complete query optimizer based exclusively on sketches. In addition to cardinality estimation, we show how to integrate the sketch estimations in plan enumeration. We are not aware of any work that integrates sketches effectively with plan enumeration. This is the main reason why sketches have not been integrated into a query optimizer before. COMPASS solves this problem.

3.2 Related Work

3.2.1 Cardinality Estimation

While exhaustive surveys on query optimization [17, 69] argue that each component is important in finding the optimal plan, Leis et al. [63, 65] show experimentally that cardinality estimation is the most dominant component in query optimization. However, consistency in estimations is more important than high accuracy only for a limited number of instances. There are four mainstream cardinality estimation approaches in the literature – histograms, sampling techniques, sketches, and, more recently, machine learning models. While histograms can provide accurate selectivity estimation for a single attribute in a relation [42], it is difficult for them to capture correlations between cross-join attributes [91], thus reducing their applicability to joins. Unlike histograms, sampling techniques [64, 82] can detect arbitrary correlations for common values. However, samples are sensitive to skewed and sparse data when few tuples are selected by a query [114]. As the query optimizer estimates a large number of joins, the cardinality drops quickly, causing wrong estimates for intermediate results. Estimating the cardinality of multi-way joins with AGMS sketches is introduced in [24, 25], while statistical analysis of two-way join sketch-based techniques is performed by Rusu and Dobra [97, 98]. Their results show that Fast-AGMS sketches are clearly superior to other sketches. In this work, we extend Fast-AGMS sketches to capture all the join attributes involved in a given query within a single sketch and efficiently estimate multi-way joins. Vengerov et al. [110] present an extension to AGMS sketches that captures selection predicates, while Cai et al. [16] introduce bound sketches that provide theoretical upper bounds for cardinality estimation. The problem with these approaches is that the online sketch-building process is not scalable. Hertzschuch et al. [37] maintain the pessimistic property for cardinality estimation, while replacing sketches with a simple formula based on statistics already available to the PostgreSQL query optimizer. This eliminates the sketch overhead while preserving the quality of the pessimistic plans – as long as the optimizer statistics estimate predicate selectivity accurately. Kernel density models for cardinality estimation (KDE) are introduced in [35, 52]. They are built on samples extracted either from the base tables or the join. While their accuracy is shown to be superior to any other method on JOB queries over at most five tables – the simplest in the benchmark – it is not clear how to generalize and fully integrate KDE models in plan enumeration. Specifically, the KDE implementation [51] builds a separate estimator for every query. No details are provided on how to apply the estimator to query sub-plans derived from the main query, which is the centerpiece of plan enumeration.

3.2.2 Query Re-Optimization

In order to overcome the inherent mis-estimations in the query optimizer, Adaptive Query Processing [22] allows the query processor to modify the optimal query plan computed by the optimizer in case of large deviations from the true cardinality values detected at runtime. The Mid-Query Re-Optimizer [48], ROX [49], and SkinnerDB [108] re-run the query optimizer at runtime in the case of large differences between estimations and the true cardinalities. Wu et al. [115] apply online sampling to correct the errors in the plans generated by the query optimizer. These approaches use the output of the query executor and sampling techniques to re-estimate the cardinalities based on already computed intermediate join outputs and change the query plan whenever the estimated values deviate significantly. In the self-adaptable LEO optimizer [74], the query engine monitors and uses the feedback from the execution engine in order to adjust the histogram-based synopses for better performance in subsequent queries. Eddies [11] process batches of tuples by following dynamic routing policies during query execution. Unlike these systems, COMPASS performs query optimization as a single stage, while query execution is partitioned into two phases – before and after the optimization. As in query re-optimization, COMPASS uses the intermediates – sketches – produced by the first phase of execution. However, this process is performed only once, thus its overhead is smaller compared to continuous re-optimization.

3.2.3 Machine Learning for Query Optimization

Using machine learning techniques and deep neural networks is a recent trend in query optimization. Join order enumeration [72, 59, 71], cardinality estimation [70, 68, 54, 55, 113, 89], selectivity estimation [119, 33, 26], and index structures [57] have been active research directions. Regarding the cardinality estimation problem, Malik et al. [70] propose to train neural network models based on cardinality distributions for a separate class of similar queries and estimate overall query cardinalities. Yang et al. [116] utilize neural networks to learn a function to estimate cardinalities of queries with range selection predicates. Kipf et al. [54] use multi-set convolutional neural networks in order to model join and selection predicates, and capture join correlations. Woltmann et al. [113] propose to train neural network models to estimate cardinalities in equi-joins. Marcus et al. [72] use reinforcement learning in order to efficiently explore the search space and find optimal join order plans. Different from these approaches, COMPASS uses traditional randomized algorithms to estimate cardinality. Moreover, COMPASS is fully integrated into a database engine, which is often not the case for these machine-learning solutions.

3.2.4 Plan Enumeration

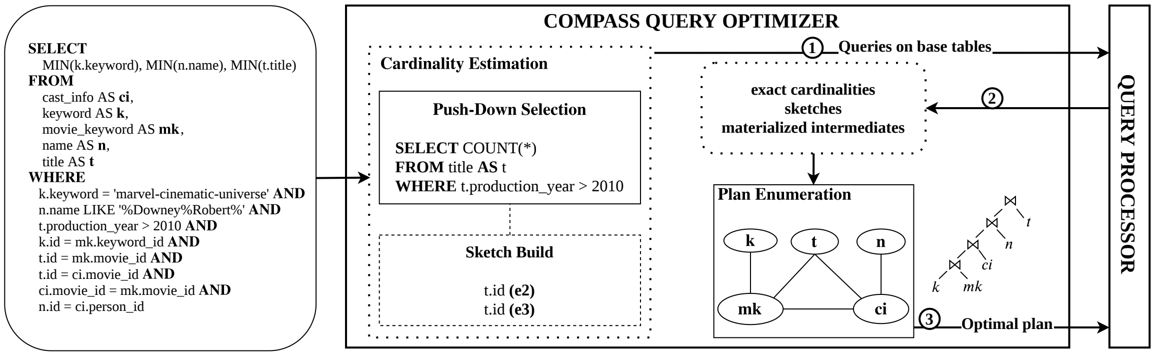
In plan enumeration, multiple semantically equivalent plans are explored in order to identify the optimal execution plan. Different exhaustive [109, 79] and heuristic-based [104] algorithms have been proposed. They consider different tree shapes – such as left-deep and bushy trees – in the search space. Leis et al. [63, 65] evaluate the influence of several plan enumeration algorithms and the impact of considering bushy trees. Several recent approaches have been proposed to optimize the plan enumeration phase by using GPUs [75, 76] and deep reinforcement learning models [72, 73]. In this work, we propose an adaptive graph traversal algorithm that efficiently explores the search space. This algorithm can be configured to cover plan enumeration from greedy to exhaustive search.

3.3 COMPASS: Online Sketch-based Query Optimization

In this section, we provide a high-level description of the COMPASS query optimization paradigm, while the technical details of cardinality estimation, join ordering, and plan enumeration are presented in Section 3.4, 3.5, and 3.6, respectively.

3.3.1 Workflow

The workflow performed by the COMPASS query optimizer is depicted in Figure 3.2. It consists of a two-step process that requires interaction with the query processor. First, the optimizer extracts the selection predicates and join attributes for every table. A sketch is built for every join attribute while performing the selection query on the base table, and only over the tuples that satisfy the predicate. Figure 3.2 shows the procedure for table *title* which has a range predicate and two join conditions – although both join predicates involve the same attribute *t.id*, two independent sketches have to be built. COMPASS leverages the high parallelism of in-memory databases and the mergeable property of sketches to execute this process with minimal overhead. Two additional optimizations can be applied to further reduce the overhead. Sketches for join attributes from tables without selection predicates can be built offline and plugged in directly. Sketches can be built only over a sample [99], which, however, incurs a decrease in accuracy. In the second step of the workflow, plan enumeration is performed by estimating the cardinality of all the sub-plans using the sketches built in the first step. This is possible only because the attribute-level sketches we design are incrementally composable. Otherwise, separate sketches have to be built for every enumerated sub-plan. In our example, there are two sketches on attribute *t.id*, one for join *e2* and one for join *e3* in the join graph (Figure 3.1). The sketch for *e2* is included in all the sub-plans that contain this join attribute – similar for *e3*. In a sub-plan that includes both *e2* and *e3*, these two sketches are first merged and then used in estimation as before. This process is performed incrementally during plan enumeration. Finally, the optimal plan is submitted for execution together with any materialized intermediates from step one.



```
SELECT
   MIN(k.keyword), MIN(n.name), MIN(t.title)
FROM
   cast_info AS ci,
   keyword AS k,
   movie_keyword AS mk,
   name AS n,
   title AS t
WHERE
  k.keyword = 'marvel-cinematic-universe' AND
  n.name LIKE '%Downey%Robert%' AND
  t.production_year > 2010 AND
  k.id = mk.keyword_id AND
  t.id = mk.movie_id AND
  t.id = ci.movie_id AND
  ci.movie_id = mk.movie_id AND
  n.id = ci.person_id
```

```
SELECT COUNT(*)
FROM title AS t
WHERE t.production_year > 2010
```

Figure 3.2: COMPASS workflow: online sketch-based query optimization for in-memory databases.

3.3.2 Partitioned Query Execution

As shown in Figure 3.2, COMPASS intertwines query optimization and evaluation by partitioning execution into push-down selection (step 1) and join computation (step 3). Query optimization, i.e., join ordering plan enumeration, is performed in between these two stages.

Since plan enumeration and join computation are standard, we focus on push-down selection, where online sketch building is performed. Push-down selection computes the exact selectivity cardinalities for all the base tables that have selections. This is similar to the ESC approach introduced in [101]. However, in addition to predicate evaluation, COMPASS also builds sketches for every join attribute in the table by piggybacking on the same traversal – sketch building is performed during the selection. Notice that this works both for sequential and index scans. It is important to emphasize that only the tuples that satisfy the predicate are included in the sketch, which increases their accuracy significantly. Moreover, the sketch update overhead is kept to the minimum necessary. While the exact cardinalities and sketches are always materialized due to their reduced size and role in optimization, the decision to materialize the selection output – the intermediate result – depends on its size. COMPASS follows the same approach as in [101]. If the intermediate size is smaller than a threshold, it is materialized. Otherwise, it is not, since the space reduction does not compensate for the access time reduction. Notice, though, that, even when intermediates are not materialized, sketches still contain only the relevant tuples for join cardinality estimation.

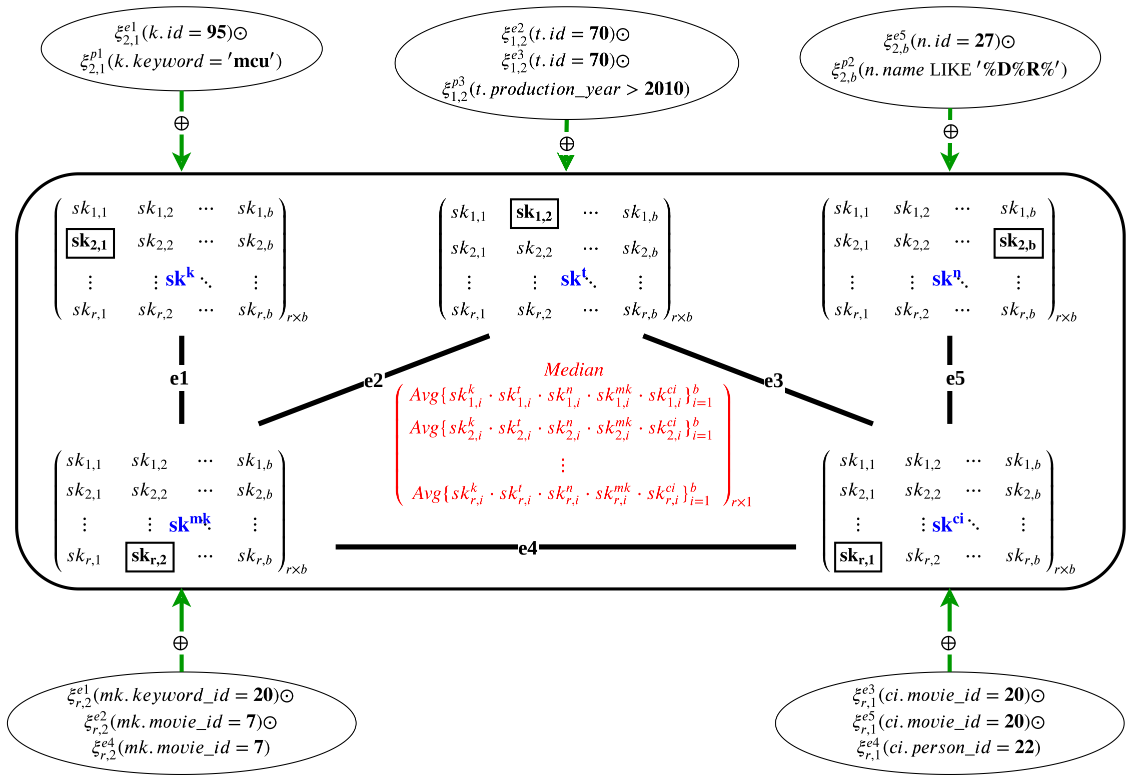
While the idea of partitioned query execution for XML processing is introduced in ROX [49], the COMPASS approach is different in several aspects. First, similar to adaptive query processing [22], COMPASS works for relational data and operators. However, COMPASS does not change the plan while the query is executing. This is not necessary because the sketch-based optimization strategy finds better plans in the first place. ROX can decompose a join graph into an arbitrary number of stages, each of which requires materialization. COMPASS, on the other hand, splits execution in exactly two stages and intermediate result materialization is only optional. The reason ROX requires materialization is that it uses chain sampling to estimate cardinalities. In order to provide acceptable accuracy, samples have to be extracted from the most recent intermediate results – not the base tables. Moreover, ROX chain sampling requires indexes on all the join attributes to guarantee a minimum sample size. This is a stringent constraint hardly satisfied in most real-world databases. Sketches, on the other hand, do not impose any constraints. Lastly, due to its incremental greedy exploration of the join order space, ROX considers only a limited number of plans – possibly sub-optimal. In COMPASS, plan enumeration is performed at once after push-down selection and can cover any portion of the join space. This can be achieved with the base table sketches which can be composed without the risk of becoming empty – the case for chain sampling.

3.3.3 Plan Enumeration

The join attribute-level sketches computed during push-down selection can be composed to estimate the cardinality of any valid join order – excluding cross products – generated during plan enumeration. In most cases, cross products are ignored by join enumeration algorithms anyway [64]. As shown in Section 3.5, sketch composition consists of two stages. First, the sketches of all the relevant join attributes in a table are merged together. An attribute is relevant for a partial join order if its join is part of the order. Second, the sketches across tables are combined to estimate the cardinality of the join order. Since the overall composition consists only of arithmetic operations, sketches can be integrated into any enumeration algorithm – exhaustive, bushy, or left-deep. Essentially, sketches can readily replace the standard join cardinality estimation formula based on table and join attribute distinct cardinality [30]. However, since sketches capture the correlation between join attributes and do not make the independence and containment assumptions, their accuracy is expected to be better.

3.3.4 Sketches vs. other Synopses

The decision to exclusively use sketches in COMPASS may seem questionable given that sketches are designed for specific stream processing tasks, while traditional databases support generic batch-oriented execution. To put it differently, there is a specific sketch for every streaming query, while synopses are for the entire database. To achieve generality, COMPASS has to build a set of sketches for every query – except base tables without predicates. However, this is done concurrently with push-down selection and is highly parallel, resulting in low overhead (Section 3.7). As a result, sketches do not require any maintenance under modification operations since they are built on the current data. This is not possible for any of the other database synopses. The benefit of having query-specific synopses is also exploited in [64], where index-based join sampling – a variation of ROX chain sampling [49] – is introduced. Index-based join sampling is performed during the plan enumeration of every query under the corresponding selection predicates. Since the sample size – both minimum and maximum – is carefully controlled, index-based join sampling has improved memory usage and accuracy because it avoids empty results. Compared to sketches, though, this sampling strategy has two serious shortcomings. First, it requires the existence of an index and complete frequency distribution on every join attribute. Sketches require nothing beyond the data. Second, the estimation of every join cardinality requires separate sampling from each of the involved tables. Since this process is time-consuming, plan enumeration is performed bottom-up – or breadth-first – in a limited time budget. Sketches can be composed incrementally in any order, without the need to access the data.



$\xi^{e1}_{2,1}(k.id = 95)\odot$ $\xi^{p1}_{2,1}(k.keyword = 'mcu')$
$\xi^{e2}_{1,2}(t.id = 70)\odot$ $\xi^{e3}_{1,2}(t.id = 70)\odot$ $\xi^{p3}_{1,2}(t.production\_year > 2010)$
$\xi^{e5}_{2,b}(n.id = 27)\odot$ $\xi^{p2}_{2,b}(n.name$ LIKE $'\%D\%R\%')$
sk^k
sk^t
sk^n
sk^{mk}
sk^{ci}
e1
e2
e3
e4
e5
Median
$Avg\{sk^k_{1,i} \cdot sk^t_{1,i} \cdot sk^n_{1,i} \cdot sk^{mk}_{1,i} \cdot sk^{ci}_{1,i}\}^b_{i=1}$
$Avg\{sk^k_{2,i} \cdot sk^t_{2,i} \cdot sk^n_{2,i} \cdot sk^{mk}_{2,i} \cdot sk^{ci}_{2,i}\}^b_{i=1}$
$Avg\{sk^k_{r,i} \cdot sk^t_{r,i} \cdot sk^n_{r,i} \cdot sk^{mk}_{r,i} \cdot sk^{ci}_{r,i}\}^b_{i=1}$
$\xi^{e1}_{r,2}(mk.keyword\_id = 20)\odot$ $\xi^{e2}_{r,2}(mk.movie\_id = 7)\odot$ $\xi^{e4}_{r,2}(mk.movie\_id = 7)$
$\xi^{e3}_{r,1}(ci.movie\_id = 20)\odot$ $\xi^{e5}_{r,1}(ci.movie\_id = 20)\odot$ $\xi^{e4}_{r,1}(ci.person\_id = 22)$

Figure 3.3: Cardinality estimation for query JOB 6a with AGMS sketches.

The other types of synopses – histograms and distinct cardinality – are not query-specific. Thus, they do not incur any creation overhead during optimization. To estimate join cardinality, the attribute-level instances of these synopses are composed of simple arithmetic operations [100, 30]. However, due to the strong assumptions – uniformity, independence, inclusion, and ad-hoc constants – made by these operations, the estimates can be highly inaccurate. Sketches do not make any of these assumptions because they capture correlations by design.

3.4 Sketch Cardinality Estimation

In this section, we present how the class of AGMS sketches is applied for estimating the cardinality of complex queries involving selection predicates and multi-way joins. We organize the presentation around the original AGMS sketches [8] which have known solutions to these problems. However, AGMS sketches are too inefficient to be accurate and cannot be integrated into query plan enumeration. This leads us to the Fast-AGMS sketches [19] which are asymptotically more efficient and have been shown to be statistically more accurate [97, 98]. However, Fast-AGMS sketches are limited to estimating two-way join cardinality. Our main contributions are to extend Fast-AGMS sketches to multi-way joins and to effectively integrate them in query plan enumeration.

3.4.1 AGMS Sketches

The basic AGMS sketch of an attribute from a relation consists of a single random value sk that summarizes the values of the attribute across all the tuples in the relation. For example, all the values of attribute *id* from table *keyword* can be summarized by a sketch $sk(k.id)$ computed as $sk(k.id) = \sum_{t \in k} \xi\,(t.id)$, where ξ is a family of $\{+1, -1\}$ random variables that are 4-wise independent. Essentially, a random value of either $+1$ or -1 is associated with each point in the domain of attribute *k.id*. Then, the corresponding random value is added to the sketch $sk(k.id)$ – initialized to 0 – for each tuple t in table *keyword*. Intuitively, the more frequent a value is, the more is "pulling" the sketch to its frequency. Since all the tuples are combined in the same sketch $sk(k.id)$, they are conflicting and the output can be far away from the frequency of each single tuple. This is where the 4-wise independence property of ξ is important. It guarantees that for any group of at most 4 different values of attribute *k.id*, the product of their corresponding ξ values is 0 on expectation – they cancel out. This, in turn, allows for each individual attribute value frequency to be unbiasedly estimated by multiplying the sketch with the corresponding ξ random value. For example, the frequency of $k.id = 5$ is estimated by the product $sk(k.id) \cdot \xi(5)$.

3.4.1.1 Two-Way Join Cardinality Estimation

Consider the join $e1$ between tables *keyword* and *movie_keyword* with predicate $k.id = mk.keyword\_id$ (Figure 3.1). The cardinality of this join operator can be estimated with two AGMS sketches $sk(k.id)$ and $sk(mk.keyword\_id)$ built on the join attributes. The requirement is that these sketches share the same family ξ of random variables – ξ^{e1} is associated with edge $e1$. ξ^{e1} guarantees that join keys with the same value are assigned the same $\{+1, -1\}$ random value – they are correlated. The basic AGMS estimator is the product of $sk(k.id)$ and $sk(mk.keyword\_id)$:

$$Est\left(|e1|\right) = sk(k.id) \cdot sk(mk.keyword\_id) = \sum_{x \in k} \sum_{y \in mk} \xi^{e1}\left(x.id\right) \cdot \xi^{e1}\left(y.keyword\_id\right) \tag{3.1}$$

Due to the 4-wise independence property of ξ^{e1}, this estimator is unbiased – its expectation equals the true $|e1|$ cardinality. However, its variance is high – it has poor accuracy. This is expected since a full table with any number of tuples is summarized as a single number. The standard technique to improve accuracy is to build multiple independent basic sketch estimators. This is achieved by using independent families of random variables ξ^{e1}. It is theoretically proven that, in order to obtain an estimator with relative error at most ϵ with confidence δ, $\mathcal{O}\left(1/\epsilon^2 \log\left(1/\delta\right)\right)$ basic sketches are necessary. As shown in Figure 3.3, they are grouped into a matrix of r rows and b columns. Then, the final AGMS estimator is obtained by averaging the b instances in each row and taking the median over the resulting r averages. In summary, an AGMS sketch has $\Omega(r \cdot b)$ update and query time, and its space usage is also $\Omega(r \cdot b)$. This assumes that the random number generators ξ have small seeds and produce their values fast – aspects that require careful implementation.

3.4.1.2 Multi-Way Join Cardinality Estimation

We show how to extend AGMS sketches to multi-way join cardinality estimation. For this, we add the join $e2$ between *movie_keyword* and *title* to $e1$ and aim to estimate the cardinality of this 3-table query. Following the approach for two-way joins, a family of sketches is built for edge $e2$ on attributes *mk.movie_id* and *t.id*, respectively. These sketches share their own family ξ^{e2} of random variables. Since two attributes from *mk* – *keyword_id* and *movie_id* – participate in join operators with other tables, we have to preserve their tuple connection. This is achieved by creating a single composed sketch $sk(mk.k\_id, mk.m\_id)$ instead of separate sketches for each attribute [25]. The value of $sk(mk.k\_id, mk.m\_id)$ is computed as:

$$sk(mk.k\_id,mk.m\_id) = \sum_{t \in mk} \xi^{e1}(t.k\_id) \cdot \xi^{e2}(t.m\_id) \tag{3.2}$$

where the product of the two random variables is added to the sketch. The cardinality estimator is defined as the product of three sketches in this case:

$$\begin{aligned} Est(|e1 \cup e2|) &= sk(k.id) \cdot sk(mk.k\_id,mk.m\_id) \cdot sk(t.id) \\ &= \sum_{x \in k} \sum_{y \in mk} \sum_{z \in t} \xi^{e1}(x.id) \cdot \xi^{e1}(y.k\_id) \cdot \xi^{e2}(y.m\_id) \cdot \xi^{e2}(z.id) \end{aligned} \tag{3.3}$$

As long as the families ξ^{e1} and ξ^{e2} are independent, this estimator is unbiased. However, its variance can be exponentially worse than that of the two-way join estimator – which makes sense, given the additional degree of randomness. Thus, to achieve the same accuracy, a considerably larger number of basic sketches are required.

This strategy can be generalized to complex queries involving any number of tables and join predicates. A sketch is built for every table. Independent random families ξ are used for every join predicate. The sketch corresponding to a table is updated with the product of all the ξ families incident to it, applied to the corresponding join attribute. In the case of our example query JOB 6a with 5 tables and 5 join predicates (Figure 3.3), there are 5 sketches and 5 families ξ. The sketch sk^{mk} for table mk is updated with the product $\xi^{e1}(k\_id) \cdot \xi^{e2}(m\_id) \cdot \xi^{e4}(m\_id)$ which includes a factor for each of the three join predicates. The unbiased cardinality estimator is the product of the 5 sketches $sk^{k} \cdot sk^{mk} \cdot sk^{t} \cdot sk^{ci} \cdot sk^{n}$. For the same number of basic sketches $r \cdot b$ as in the case of the $|e1|$ join, the accuracy of the $|e1 \cup e2 \cup e3 \cup e4 \cup e5|$ join can be exponentially worse.

3.4.1.3 Selection Cardinality Estimation

Query JOB 6a contains 3 selection predicates – point on k, subset on n, and range on t. These have to be accounted for when estimating the overall query cardinality. AGMS sketches can handle selection predicates as long as the domain of the attribute is discrete – which is the case for the fixed-size data types in databases. The idea is to express the selection as a join predicate between the table and the domain of the selection attribute [95, 96]. Following the two-way join approach, a sketch is built on the selection attribute over all the tuples in the table. The sketch over the domain – which shares the same random family ξ – summarizes the values in the domain which satisfy the predicate by adding an entry for each of them to the sketch – for a point predicate, the sketch includes only the ξ value corresponding to the constant in the predicate; for a range, the ξ values for all points in the range; for a subset, the ξ values for the points in the subset. As long as the number of points is small, these sketches can be computed fast. Moreover, even for ranges, there is a specific fast range-summable random family ξ for which the sketch can be computed in constant time, independent of the range size [95, 96]. In the JOB 6a query depicted in Figure 3.3, the sketch update procedure for tables with predicates includes an additional factor corresponding to the selection attribute. For example, the sketch sk^k for table *keyword* is updated with the product $\xi^{e1}(id) \cdot \xi^{p1}(keyword)$. Overall, 8 families ξ and 8 sketches are required – the sketches over the domain of the selection attributes are not included in Figure 3.3. The final estimator is the product of these 8 sketches. Since this estimator is a multi-way join with a larger number of sketches, its accuracy becomes worse than that of the join sketches only.

3.4.1.4 Why AGMS Sketches Are Not Practical for Query Optimization?

As shown, AGMS sketches can be theoretically used to estimate the cardinality of arbitrarily complex queries with join and selection predicates. While all the sketches for a table can be built in a single scan, since the update time per AGMS sketch is linear in the sketch size, updating an exponential number of sketches becomes dominant. Moreover, the space requirement for all the sketches is also a problem. These scalability issues hinder the application of AGMS sketches to join order enumeration. However, AGMS sketches suffer from a more serious problem in query optimization – they cannot be incrementally composed. What this means is that a sketch used to estimate a two-way join between two relations cannot be used to estimate a three-way join that includes another relation. The addition of join $e2$ to $e1$ in our example illustrates this well. It is not possible to compute the sketch $sk(mk.k\_id, mk.m\_id)$ from sketch $sk(mk.keyword\_id)$. It is not even possible to compute $sk(mk.k\_id, mk.m\_id)$ from $sk(mk.keyword\_id)$ and $sk(mk.movie\_id)$. The reason is the order of multiplication and addition. The other direction – use $sk(mk.k\_id, mk.m\_id)$ instead of $sk(mk.keyword\_id)$ or $sk(mk.movie\_id)$ – is also not possible. Thus, in order to support plan enumeration, a separate sketch has to be built for every combination of the join attributes – which is an exponential number. For example, 7 sketches have to be built for both tables *mk* and *ci* which participate in 3 join predicates. If we include the attributes that can appear in selection predicates, the number of sketches that have to be built for a table can become exponential in the number of attributes in the table. While workload information can be used to reduce this number, there is little that can be done for tables that join with several other tables on different attributes. Practically, AGMS sketches cannot achieve the goal of having synopses only for single attributes.

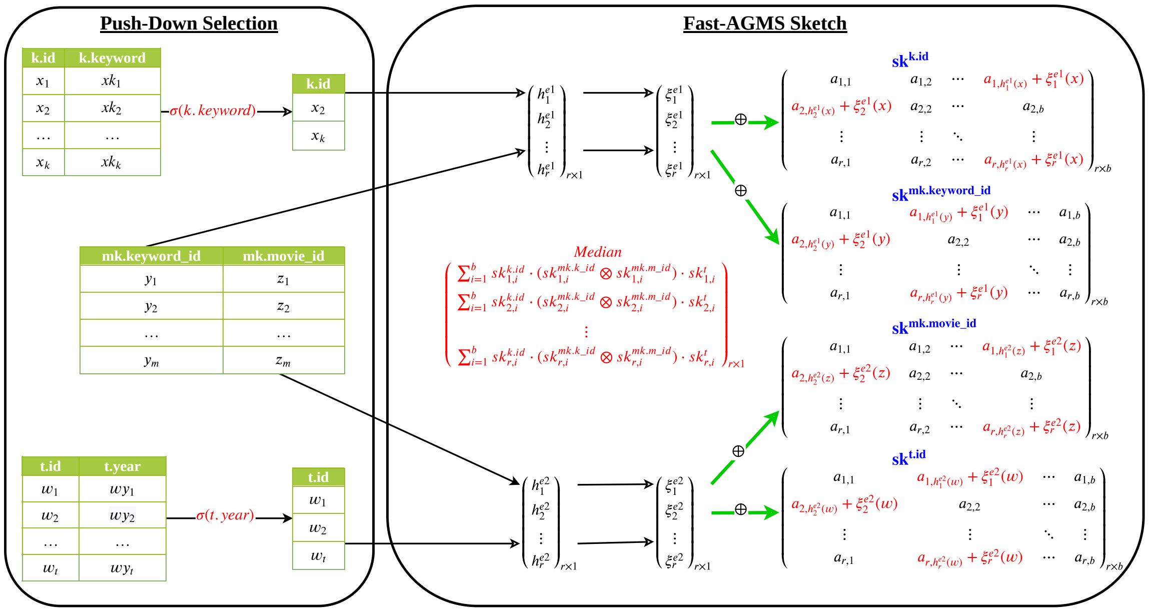


Figure 3.4: Cardinality estimation for query JOB 6a with Fast-AGMS sketches.

3.4.2 Fast-AGMS Sketches

Fast-AGMS sketches preserve the $(r \times b)$ matrix structure of AGMS sketches. However, they define a complete row of b counters as a basic sketch element (Figure 3.4). Only one of these counters is updated for every tuple, thus, a factor b reduction in update time is obtained. The updated counter is chosen by a random hash function h associated with the row. The purpose of h is to spread tuples with different values as evenly as possible – tuples with the same key still end up in the same bucket. On average, a factor b fewer tuples collide on the same counter, which preserves the frequency of each of them better. Since a full row is a sketch element, a single ξ family of random variables is associated with every row. Thus, a Fast-AGMS sketch with r rows requires only r hash and ξ random functions. The value of a counter j is $\mathit{sk(k.id)}_j = \sum_{t \in k, h(t.id)=j} \xi\,(t.id)$.

3.4.2.1 Two-Way Join Cardinality Estimation

In order to estimate join cardinality, the same principle applies – Fast-AGMS sketches are built over the join attributes using the same random functions h and ξ. The hash function h lands identical keys to the same bucket, while ξ gives the same sign. The unbiased estimator for a basic sketch sums up the product of the corresponding buckets:

$$Est\,(|e1|) = \sum_{j=1}^{b} \mathit{sk(k.id)}_j \cdot \mathit{sk(mk.keyword\_id)}_j \tag{3.4}$$

Summation is necessary because h partitions the tuples. As for AGMS sketches, the final estimate is obtained by taking the median of the r independent basic sketches. Although the accuracy of Fast-AGMS sketches is asymptotically equal to that of AGMS sketches [19] in the worst case, it has been shown statistically that Fast-AGMS sketches have considerably better accuracy than any other sketching technique on average [97]. The combined accuracy and fast update time make Fast-AGMS sketches suitable for query optimization.

3.4.2.2 Why Fast-AGMS Sketches Are Not Applicable to Query Optimization?

As far as we know, there is no work that extends Fast-AGMS sketches to multi-way join estimation. The main problem is the requirement to have independent hash functions h^{e1} and h^{e2} for the two join attributes. These functions allocate the attributes to different buckets, which means that the tuple is added to the sketch twice. Moreover, the relationship between attributes is lost. Since sketch-based selectivity estimation is also reduced to a join between the selection attribute and its domain, this implies that Fast-AGMS sketches cannot be used to estimate the cardinality of two-way joins with predicates. In fact, computing optimally the Fast-AGMS sketch of the domain of a range predicate does not have a solution. This is because there is no order relationship between the hash values of adjacent points in the domain. Due to these limitations, Fast-AGMS sketches have not been used in query optimization before. COMPASS introduces Fast-AGMS extensions for multi-way joins and solves the selectivity issue by pushing-down predicates during query optimization, and adding only the relevant tuples to the sketch.

3.5 Fast-AGMS Sketch Multi-way Join Estimation

We present two strategies to extend Fast-AGMS sketches to multi-way join cardinality estimation. The first strategy – sketch partitioning – is a theoretically sound estimator for a given multi-way join. Its limitation is that it cannot be composed/decomposed, thus, it is not scalable for plan enumeration. The second strategy – sketch merging – addresses the scalability issue by incrementally creating multi-way sketches from two-way sketches. Although this is done heuristically for a certain multi-way join taken separately, all the multi-way joins with a given size are equally impacted. We show empirically that this property is a good surrogate for accuracy – which is much harder to consistently achieve – in join order enumeration.

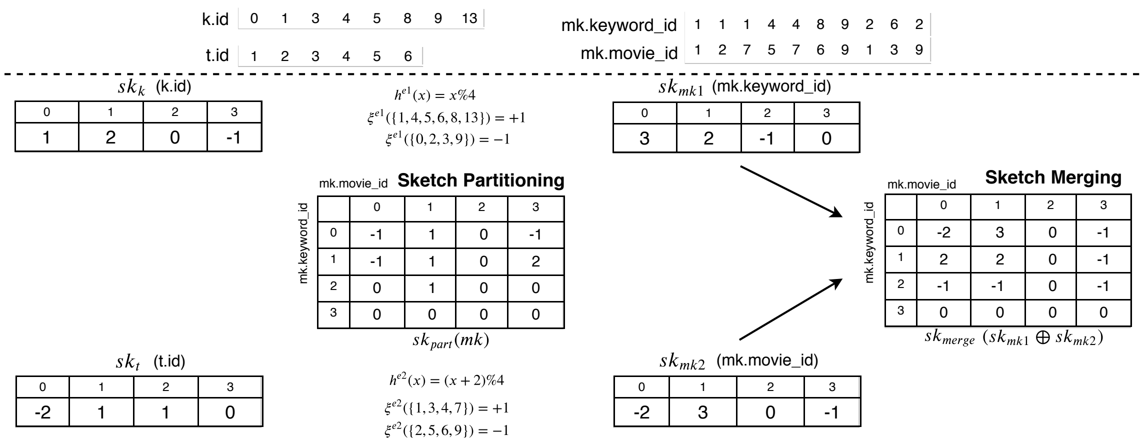


Figure 3.5: Fast-AGMS sketches for multi-way join cardinality estimation on query JOB 6a.

3.5.1 Sketch Partitioning

The idea of sketch partitioning is to reorganize the b buckets of the elementary sketch into a $(b_1 \times b_2)$ 2-D matrix – as done in [16] for frequency sketches. h^{e1} hashes a tuple $mk(k\_id, m\_id)$ to one of the b_1 rows, while h^{e2} hashes to one of the b_2 columns. Then, only the counter at indices $\left[h^{e1}(k\_id), h^{e2}(m\_id)\right]$ is updated with the product $\xi^{e1}(k\_id) \cdot$

$\xi^{e2}(m\_id)$. This process is depicted in Figure 3.5. For example, tuple (6,3) in mk adds 1 to the counter [2,1]. h^{e1} guarantees that all the tuples with $k\_id = 6$ are hashed to row 2, while h^{e2} sends tuples with $m\_id = 3$ to column 1. Conflicts happen only when the output of both hash functions is identical. Given the quadratic number of buckets compared to the sketch for a single attribute – while the number of tuples is the same – conflicts are less frequent. The cardinality estimate for the 3-table join $k \bowtie mk \bowtie t$ is obtained by summing up all the entries in the matrix resulting after the scalar multiplication between $sk(k.id)$ and every row in $sk_{part}(mk)$, followed by the scalar multiplication between the transpose of $sk(t.id)$ and every column in $sk_{part}(mk)$. This can be written as:

$$Est(|e1 \cup e2|) = \sum_{0 \leq i < b_1} \sum_{0 \leq j < b_2} sk_k[i] \cdot sk_{part}(mk)[i,j] \cdot sk_t[j] \tag{3.5}$$

It can be shown theoretically that this estimator is unbiased following the same proof as for AGMS sketches in [25]. Moreover, given the larger size of sketch $sk_{part}(mk)$, its accuracy is expected to be better. This procedure can be generalized to any number of join attributes by partitioning – or replicating – b into the corresponding number of dimensions. For example, a table with 3 joins has a 3-D tensor as its sketch, with one dimension for every join attribute. Thus, there is a polynomial factor increase in the size of the sketch and the estimate computation. This has to be carefully accounted for in the overall memory budget since the likelihood of conflicts varies with the dimensionality of the sketch tensor. The constraint to have the same number of buckets for a join predicate, e.g., $sk(k.id)$ has as many buckets as the number of rows in $sk_{part}(mk)$, makes memory allocation among sketches more complicated than for the 1-D AGMS sketch vectors.

Partitioned Fast-AGMS sketches are not scalable for join order enumeration. This is because separate sketches are required for every join. For example, in Figure 3.5, the 2-D sketch $sk_{part}(mk)$ is used for the 3-way join $k \bowtie mk \bowtie t$, while the 1-D sketches sk_{mk1} and sk_{mk2} are used for the 2-way joins $k \bowtie mk$ and $t \bowtie mk$, respectively. Building and storing these many sketches is impractical in query optimization. One alternative is to build only the sketches for up to k-way joins and use other methods to estimate higher-order join cardinality. This strategy is applied for run-time join samples in [64]. The drawback is that other statistics are required for the higher-order joins and the interaction between the estimates produced by these statistics and the sketch estimates has to be carefully controlled.

Our goal is to exclusively use sketches. Intuitively, we want to be able to either generate the 2-D sketch from the 1-D sketches or extract the 1-D sketches from the 2-D sketch. Unfortunately, none of these have a clear solution for Fast-AGMS sketches. The composition of sk_{mk1} and sk_{mk2} requires determining how to combine all the pairs of buckets in the 1-D sketches in order to compute the quadratic number of entries in the 2-D sketch. Since the identity of tuples is lost when they are inserted in the 1-D sketch, it is not possible to recreate the tuple and determine its corresponding 2-D bucket. Moreover, due to conflicts in the ξ random functions, we do not even know how many tuples belong to a 1-D bucket. For example, bucket 1 in sk_{mk1} is 2 even though 4 tuples are hashed to it. The extraction of a 1-D sketch from the 2-D sketch also does not work because of the ξ variables. Specifically, the update by the product $\xi^{e1} \cdot \xi^{e2}$ makes it impossible to retrieve the value of a 1-D bucket by summing up the corresponding 2-D buckets. For example, the value of bucket 0 in sk_{mk1} is not the sum of the buckets in row 0 of sketch $sk_{part}(mk)$. This property is true only for hash-based sketches [16].

3.5.2 Sketch Merging

We introduce the sketch merging heuristic as a lightweight method to compose two-way join Fast-AGMS sketches in order to estimate the cardinality of multi-way joins. The procedure works as follows. We build sketches for every two-way join predicate independently, as shown in Figure 3.5. The number of sketches corresponding to a table is equal to the number of joins it participates in. For example, tables k and t have one sketch, while mk has two sketches. We estimate any join combination generated during plan enumeration using only these sketches. The two-way joins $k \bowtie mk$ and $t \bowtie mk$ are estimated optimally with the sketch pairs (sk_k, sk_{mk1}) and (sk_t, sk_{mk2}), respectively. These are the most accurate sketch estimates we can get. For the 3-way join $k \bowtie mk \bowtie t$, we create a merged sketch $sk_{merge}(mk) = sk_{mk1} \oplus sk_{mk2}$ from the two 2-way join sketches on demand during plan enumeration. This merged sketch approximates the partitioned sketch $sk_{part}(mk)$ computed with the same random functions, without accessing the tuples. A bucket $[i, j]$ in sk_{merge} is set to the value having the minimum absolute magnitude among the corresponding $[i]$ and

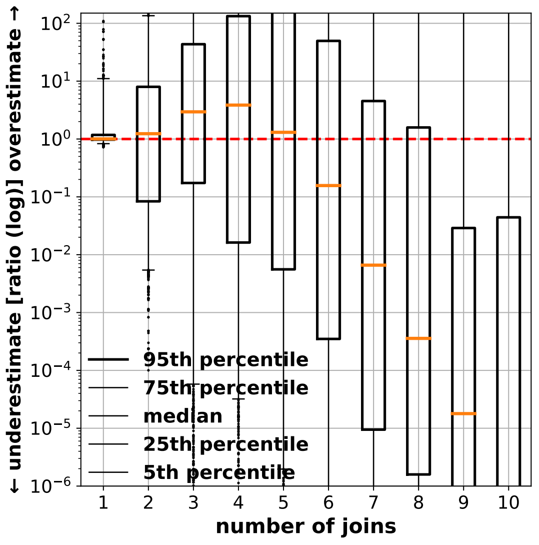
$[j]$ buckets in the two basic sketches:

$$sk_{merge}[i,j] = \begin{cases} sk_{mk1}[i], & \textit{if } |sk_{mk1}[i]| \leq |sk_{mk2}[j]| \\ sk_{mk2}[j], & \textit{if } |sk_{mk1}[i]| > |sk_{mk2}[j]| \end{cases} \tag{3.6}$$

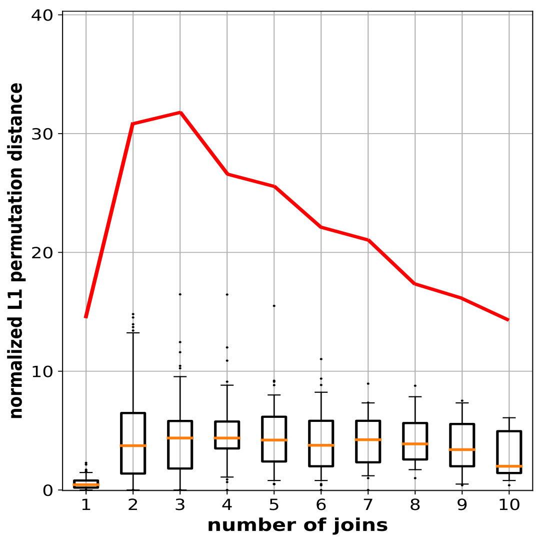
For the example in Figure 3.5, bucket $[0, 0]$ is set to -2 because $|3| > |-2|$, while bucket $[0, 2]$ to 0 because $|0| < |3|$. The reason for this merge procedure is multi-folded. The interaction between the random functions ξ is considered – albeit not through a direct multiplication – by preserving the sign of the value in the basic sketch. The absolute magnitude corresponds to the maximum number of tuples with a given join key that is hashed to the bucket – assuming no conflicts. These tuples are partitioned across the buckets of the other join key. The minimum is chosen because this is the maximum number of tuples that can have identical values for both join keys when considered together. However, this is an overestimate because the exact tuple pairing is lost. This can be seen when comparing the magnitude of the values in the two 2-D sketches in Figure 3.5. In fact, sketch merging is likely to always overestimate join cardinality. The only caveat is the interaction between the ξ functions.

Sketch merging can be generalized to any number of joins by applying the procedure iteratively. Moreover, (n+1)-D sketches can be derived incrementally from n-D sketches in a single step – without the need to start from the basic sketches. This property can be exploited to speed up the computation and reduce memory usage in bottom-up plan enumeration since only the highest dimensional sketches have to be maintained. An even more important property of sketch merging is that it is consistent in how it handles the multi-way joins with the same number of predicates. Specifically, all these joins rely on the same basic sketches and the same assumptions for merging. Thus, it is likely that these estimates exhibit similar accuracy behavior – the same type of errors for equal join size.

In order to verify this claim, we depict the accuracy of sketch merging for the JOB queries in Figure 3.6. We use two measures to quantify accuracy. The first is the ratio between the sketch estimate and the true cardinality for all the enumerated sub-plans having at most ten joins (Figure 3.6a). We observe that the median ratio is within a factor of 10 for up to six joins, which is better than any previous practical results [64]. For a larger number of



(a) estimate/true



(b) L1-distance

Figure 3.6: Accuracy ratio (a) and L1-distance between the estimated sketch permutation and the correct join order (b).

joins, sketch merging generates underestimates systematically. In the previous results [65], this behavior occurs starting from 3-way joins.

2-join	$mk \bowtie k$	$ci \bowtie n$	$mk \bowtie t$	$ci \bowtie t$	$mk \bowtie ci$
True cardinality	14	486	0.3M	6M	215M
Sketch merging	3K	7.5K	6.4M	14M	397M

3-join	$mk \bowtie k \bowtie t$	$t \bowtie ci \bowtie n$	$mk \bowtie ci \bowtie k$	$mk \bowtie ci \bowtie n$	$mk \bowtie ci \bowtie t$
True cardinality	11	61	1242	10K	17M
Sketch merging	6.7K	35K	0.7M	1.5M	8.7B

4-join	$mk \bowtie ci \bowtie n \bowtie k$	$mk \bowtie ci \bowtie n \bowtie t$	$mk \bowtie ci \bowtie k \bowtie t$
True cardinality	6	<u>1194</u>	<u>1224</u>
Sketch merging	198	**<u>18M</u>**	**<u>6.5M</u>**

Table 3.1: 2-, 3-, and 4-way join L1 permutation distance for JOB 6a.

The second measure is the normalized L1 distance [103] between the permutation generated by sketch merging and the correct join order. Given n sub-plans of the same size, the correct order $\mathcal{C}$ is obtained by sorting them in increasing order of their cardinality. The permutation corresponding to sketch merging $\mathcal{S}$ is obtained by sorting the sub-plans based on the sketch estimates. The L1 distance is defined as $\sum_{i=1}^{n} |\mathcal{S}_i - \mathcal{C}_i|$, the sum of the differences between the position in the permutation and the correct order. For example, the L1 distance for the 4-way joins in query JOB 6a (Table 3.1) is $0 + 1 + 1 = 2$. The normalized L1 distance – we divide the distance by the number of sub-plans in the query – is depicted in Figure 3.6b. The closer the distance is to zero, the more similar is the permutation to the correct order. For reference, we plot the line corresponding to the maximum L1 distance. The join orders generated by sketch merging have an L1 distance that is significantly below the maximum. In particular, for 2-way join sub-plans, the distance is almost zero, while for sub-plans with more joins, the distance is constantly below 10. This confirms that sketch merging selects orders that are close to optimal most of the time.

3.6 Fast-AGMS Sketch Join Order Enumeration

In this section, we present how merged Fast-AGMS sketches are integrated into a novel plan enumeration algorithm we introduce in this work. It is important to emphasize that – due to the proposed generalization to multi-way joins – Fast-AGMS sketches can be embedded into any enumeration algorithm. According to the investigation of different join order enumeration algorithms performed in [65, 63], plan enumeration is not the most critical component of a query optimizer – it has a relatively small impact on plan quality compared to cardinality estimation accuracy. This confirms the approach taken by many query optimizers [88, 3] that consider only a plan subset – left- or right-deep plans, which are a permutation of the query tables. We follow a similar approach and introduce a heuristic plan enumeration algorithm that allows us to explore the overall impact of merged Fast-AGMS sketches. Although this algorithm borrows ideas from previous work [104], we argue that it is original in the presented form.

COMPASS uses the join graph (Figure 3.1) in plan enumeration. This guarantees that only valid join order plans are considered and cross products are ignored. Plan enumeration becomes a graph traversal problem. We design a depth-first search (DFS) traversal algorithm (Algorithm 1) that enumerates left-deep plans following the edges in the join graph. This is achieved by considering all the vertices as the source of DFS and backtracking whenever a complete plan is reached (line 26). The number of plans explored from a source vertex is controlled by a user-defined parameter *max_plans* (line 25) that plays a similar role to the timeout used to confine the plan search space in [64]. However, our algorithm does not require bottom-up plan enumeration since sketches can be merged and combined in any order. The *max_plans* parameter also allows the plan search to restart from other sources in the presence of high-degree vertices and cycles – instead of getting locked on the initial selections. On a continuum spectrum that has left-deep greedy search at one extreme and exhaustive enumeration at the other [30], the proposed algorithm can be configured anywhere in-between by controlling the value of parameter *max_plans*. When a single plan is enumerated from every vertex, we obtain left-deep ordering. This is achieved by selecting the vertex v that has the smallest cardinality when appended to the current plan in line 35. When all the plans are enumerated from every vertex – set $max\_plans = \infty$ – we obtain exhaustive enumeration. The tradeoff between these two alternatives is evident – the number of explored plans vs. enumeration overhead. Since sketch merging and estimation are fast operations amenable to parallelization, the overhead is small – see the experiments in Section 3.7. Thus, *max_plans* values larger than 1 are amenable – *max_plans* is set by default to 10. This allows for a more comprehensive exploration of the join order space compared to alternative synopses that incur higher overhead [49, 64].

We design two heuristics that control the order in which the plan space is explored and the depth of exploring sub-optimal plans. First, we sort the vertices according to a normalized cost function $f(v)$ that combines vertex cardinality and the number of join predicates the vertex participates in (line 7). The configurable weights α and β control the relative importance of these factors. They are set by default to 0.5, which assigns equal weight to each factor. *DFS Traversal* is invoked from the source vertices in increasing order of cost $f(v)$. The intuition is to generate sub-plans with small cardinalities and limited orders as early as possible in the enumeration – get the left-deep plan corresponding to $max\_plans = 1$ first. The second heuristic is the early pruning of sub-optimal plans (line 17). Whenever the cost of a sub-plan exceeds the minimum cost of a complete plan, the algorithm backtracks to a shorter sub-plan that can still become optimal.

Fast-AGMS sketch cardinality estimation is invoked for every enumerated sub-plan (line 14) and to decide the order in which vertices are explored (line 31) – while the call on line 14 can be eliminated, we keep it for clarity. The *Fast-AGMS Estimate* function performs sketch merging and estimation only for the joins included in the sub-plan. In order to avoid recomputation, the estimates are cached. Alternatively, the merged sketches corresponding

Algorithm 1 Fast-AGMS Sketch Join Order Enumeration

```
 1: Let $G = (V, E)$ be the join graph
 2: Let $C$ be the join order set, initially $\emptyset$
 3: Let $p$ be the number of complete plans, initially 0
 4: Let $min\_cost$ be the minimum cost, initially $\infty$
 5: Let $max\_plans$ be the maximum threshold of plans enumerated from a source vertex
 6: procedure PLAN ENUMERATION($G$)
 7:     Let $S$ be the set of vertices $V$ sorted in increasing order of function $f$ that combines table cardinality with the degree of $v$ in $G$: $f(v) = \alpha \cdot \frac{cardinality(v)}{\max\{cardinality(u), \forall u \in V\}} + \beta \cdot \frac{deg(v)}{max\{deg(u), \forall u \in V\}}$, where $\alpha$ and $\beta$ are user-defined constants
 8:     for each vertex $v \in S$ do
 9:         $p \leftarrow 0$
10:         DFS TRAVERSAL($G$, $\{v\}$, 0)
11: end procedure
12: procedure DFS TRAVERSAL($G$, $C$, $cost$)
13:     $curr\_est \leftarrow$ FAST-AGMS ESTIMATE($C$)
14:     $cost \leftarrow cost + curr\_est$
15:     ▷ Early pruning
16:     if $cost > min\_cost$ then return
17:     ▷ Evaluate complete plan
18:     if $|C| = |V|$ then
19:         if $cost < min\_cost$ then
20:             $min\_cost \leftarrow cost$
21:             $opt\_path \leftarrow C$
22:         $p \leftarrow p + 1$
23:         if $p = max\_plans$ then abort
24:         return
25:     ▷ Recursive enumeration in increasing order of the join cardinality estimates
26:     Let $L$ be the set of vertices $v \notin C$ that are adjacent to a vertex $u \in C$
27:     for each vertex $v \in L$ do
28:         $e[v] \leftarrow$ FAST-AGMS ESTIMATE($C \cup \{v\}$)
29:     Let $L'$ be set $L$ sorted in increasing order of $e[v]$
30:     for each vertex $v \in L'$ do
31:         DFS TRAVERSAL($G$, $C \cup \{v\}$, $cost$)
32: end procedure
```

to join subsets a table is involved in and the product of the basic sketches corresponding to a sub-plan can also be cached. They provide different levels of reuse and computation to generate the estimate. Caching the estimate provides the least reuse while caching merged sketches and products allows for incremental estimate evaluation. In our implementation, we settle for a combined solution in which all the estimates, and merged sketches and products of up to three sketches are cached. This ensures that the estimate corresponding to any sub-plan is computed only once and allows for incremental extension of the core sub-plans. It is important to emphasize that the input to *Fast-AGMS Estimate* is always represented only by the sketches corresponding to the two-way joins.

We illustrate how the plan enumeration algorithm works for query JOB 6a based on the join graph in Figure 3.1 and the join cardinality estimates in Table 3.1. The five vertices are sorted in the order $n - k - t - mk - ci$ based on function $f(v)$ applied on the tables resulting after selection push-down. mk and ci are the last two because they have the largest degree and cardinality – and have no selection predicates. k and n come before t because they have a smaller degree. Although n has an order of magnitude more tuples than k, the predicate on n is very selective and outputs a smaller cardinality. The degree being the same, n is first in the order. Thus, *DFS Traversal* is performed with n as the first source. The first enumerated plan is $n - ci - t - mk - k$ and its estimated cost is $\approx 18M$. While a left-deep plan search finishes the enumeration at this point, our algorithm backtracks and explores alternative plans – we set $max\_plans = 2$ in this case. The other plan enumerated from n is $n - ci - mk - k - t$ which has an estimated cost of $\approx 1.5M$. This is the optimal plan from n. Since the number p of explored plans from n reaches the value of max_plans, in the next step, *DFS Traversal* is performed from k. The first plan enumerated from k is $k - mk - t - ci$ which is pruned early because its partial cost of $\approx 6.5M$ is larger than the complete minimum cost of $\approx 1.5M$. The next plan $k - mk - ci - n - t$ has an estimated cost of only $\approx 0.7M$ and becomes the optimal plan up to this point – in fact, it is the optimal plan identified by COMPASS (Figure 3.1). The third – and final – plan considered from k is $k - mk - ci - t$. This plan is pruned early. Notice that the enumeration terminates without reaching the maximum number of allowed plans. The enumeration starting from t does not proceed beyond its immediate neighbors because of the large cardinality estimates. The only complete plan enumerated from mk is $mk - k - ci - n - t$ which has the same cost as the minimal cost plan – they are equivalent. All the plans starting from ci are pruned early.

The left-deep plan identified by PostgreSQL and DBMS A is $k - mk - t - ci - n$ (Figure 3.1). Although this plan has a slightly lower cost, it is not enumerated by our algorithm. The reason is the large sketch estimate for the join $k \bowtie mk \bowtie t \bowtie ci$ which stops the enumeration early. While this estimate is inaccurate, the more thorough plan space exploration allows our algorithm to identify an alternative plan with a cost almost identical. We point out that if we apply the same greedy strategy as in the left-deep plan search using the sketch estimates in Table 3.1, we would get the same optimal plan as PostgreSQL and DBMS A. This is because, once we reach the sub-plan $k - mk - t$, the only alternative is to choose ci – there is no backtracking. Thus, left-deep search identifies the plan $k - mk - t - ci - n$ by chance rather than by considering estimates for four-way joins – known to be unreliable for any type of synopses, not only sketches.

3.7 Empirical Evaluation

We perform an extensive experimental study over the complete JOB benchmark [65] in order to evaluate the performance of COMPASS and compare it against four other database query optimizers (Figure 3.1). While our main goal is to determine whether COMPASS is a complete optimizer – which requires effective integration of cardinality estimation in plan enumeration – we also perform a detailed comparison between Fast-AGMS sketches and several state-of-the-art methods for multi-way join cardinality estimation. Moreover, we assess the impact of the proposed plan enumeration algorithm. To this end, our evaluation investigates the following questions:

- What is the quality of the query execution plans generated by COMPASS? We measure plan quality as the total cardinality of the intermediate results since this is independent of specific execution engine optimizations. Moreover, logical optimizers use cardinality information as the main criterion to rank plans.
- What is the execution time – or runtime – for the COMPASS plans? Since this is highly dependent on the underlying query processing engine, we execute the plans in MapD, PostgreSQL and DBMS A. This allows us to identify the correlation – if there is one – between plan quality and execution time.
- What is the overall JOB workload runtime? While individual queries allow for localized analysis, the workload execution time measures the reliability of COMPASS. However,

due to the high variance in JOB query complexity, this measure alone is not an absolute indicator of the quality of an optimizer.

- How does COMPASS compare against the pessimistic optimizers that minimize upper bound cardinality, i.e., over-estimates? Since the highly-optimized pessimistic plans are shown to be considerably faster than the default PostgreSQL plans [16, 37], we are interested in where COMPASS stands on this scale.
- What is the optimization overhead incurred by sketch merging in plan enumeration? While significantly improving upon sketch partitioning, it is not clear if online sketch merging during push-down selection is fast enough to be practical. We deem COMPASS to be a practical optimizer if it manages to consistently outperform the other databases and also incurs a reduced overhead.
- How does the accuracy of Fast-AGMS sketch merging compare against state-of-the-art methods for multi-way join cardinality estimation? Previous studies [110, 52] include only AGMS sketches, which are known to be considerably worse than Fast-AGMS for two-way join estimation [97, 98].
- How does the plan enumeration algorithm driven by sketch merging compare against standard algorithms? Does the larger search space improve the plan quality compared to the greedy left-deep enumeration? Alternatively, how close (far) is the proposed algorithm to exhaustive enumeration?

3.7.1 Experimental Setup

3.7.1.1 Implementation

We implement COMPASS in MapD (version 3.6.1) [2]. The source code is publicly available in Github [45]. MapD has a highly parallel GPU-accelerated query execution engine. Relational operators are compiled into CUDA kernels that are executed concurrently across the SIMD GPU architecture. In order to reduce data movement, MapD coalesces multiple relational operators into a single CUDA kernel. For joins, this corresponds to a worst-case optimal join algorithm [87]. The MapD query optimizer, however, is not as sophisticated as its execution engine. It relies on the Calcite SQL compiler [1] to get a lexicographic – in the order in which the query is written – query execution plan. The join order is computed based on a primitive heuristic that sorts the tables in decreasing order of their cardinality.

Moreover, selection predicates are not considered in the optimization. COMPASS brings a principled cost-based optimization procedure to the MapD query optimizer.

The COMPASS implementation consists of two modules – a scan operator that integrates Fast-AGMS sketch construction with push-down selection and a lightweight join order enumeration algorithm. For sketch construction, we adapt a publicly available two-way join Fast-AGMS sketch implementation [94] to the MapD CUDA kernel API. This requires parallelizing both the update and the estimation functions. The scan operator filters only the relevant tuples to be passed to the sketch update. Since separate sketch instances are created for every GPU block warp, this requires an additional merge stage – currently performed on the CPU. The sketches used throughout the experiments have 11 rows of 1023 buckets, for a total of roughly $\approx$ 11K integers. Assuming 4-byte integers, the memory usage of a sketch is $\approx$ 45KB. The largest query has 28 joins. With two sketches per join, the maximum memory usage for a query is $\approx$ 2.5MB ($2 \cdot 28 \cdot 45$), which is quite small. Depending on the parallelization approach, there can be a sketch instance on every GPU block warp. In our case, the NVIDIA Tesla K80 has 26 block warps, resulting in a total of $\approx$ 65MB memory usage for the most complex JOB query. The COMPASS plan enumeration algorithm depicted in Algorithm 1 replaces the primitive sorting heuristic from MapD. It determines the optimal plan based on the sketches computed by the scan operator and the configurable enumeration logic. Greedy join enumeration is the default algorithm used throughout experiments.

3.7.1.2 Database Systems & Hardware

The other three databases we use in addition to MapD are PostgreSQL (v.11.5), MonetDB (v.11.33.11), and the commercial DBMS A. PostgreSQL and DBMS A are used as the common ground in all the experiments because of their extensibility. Both of them allow us to inject and execute the join orders computed by the other databases – the `CROSS JOIN` statement in PostgreSQL and the hints in DBMS A, respectively. We configure PostgreSQL with 2GB memory per operator, 32GB buffer cache size, and we force the optimizer to use dynamic programming in plan enumeration for queries with no more than 18 join predicates. These settings follow prior art [65]. We use an optimized docker image publicly available for DBMS A, while for MonetDB we keep the default configuration. All the systems run on a Ubuntu 16.04 LTS machine with 56 CPU cores (Intel Xeon E5-2660), 256GB RAM, HDD storage, and an NVIDIA Tesla K80 GPU.

3.7.1.3 Dataset and Query Workload

We perform the experiments on the IMDB dataset [12] which has been used extensively to evaluate query optimizers [63] and has become a de-facto standard. The JOB benchmark [92] defines 113 queries – grouped into 33 families – over the IMDB dataset. These queries vary significantly in their complexity, with the simplest one having 4 joins and the most complex one having 28 joins. This variability manifests itself in execution times that are highly different. To compensate for this, we split the queries into three groups and examined each group separately. These groups are based on the number of joins in the query: group1 contains queries with 4-9 joins; group2, 10-19 joins; and group3, 20-28 joins. We organize the results according to these groups.

3.7.1.4 Methodology

To quantitatively assess the quality of a join order plan, we use two metrics – intermediate result cardinality and query execution time. The total cardinality of the intermediate results quantifies how many tuples are produced by all the joins in the plan – defined in Equation 2.3. The lower this number is, the better the plan. This is the primary metric used in logical query optimization to estimate the cost of a plan. However, the actual execution time depends on specific query processing optimizations. Thus, the execution time is not entirely correlated with the cardinality.

In order to fairly evaluate the join orders produced by every database, we use both PostgreSQL and DBMS A as common ground. First, we run the queries in each database and collect their join plans. Then, we inject these plans into PostgreSQL and DBMS A, respectively, and measure their runtime. Moreover, we execute all the subqueries in the plans to compute the intermediate cardinality. Notice that every system generates its plan independently based on its own algorithm and statistics. PostgreSQL and DBMS A serve as common execution engines for all the plans. This procedure allows for a holistic comparison of the query optimizers – independent of the execution engine.

3.7.2 Results

We present the results of our extensive experimental evaluation, organized based on the investigation questions defined at the beginning of this section. The answers to the questions are summarized after the presentation of the results.

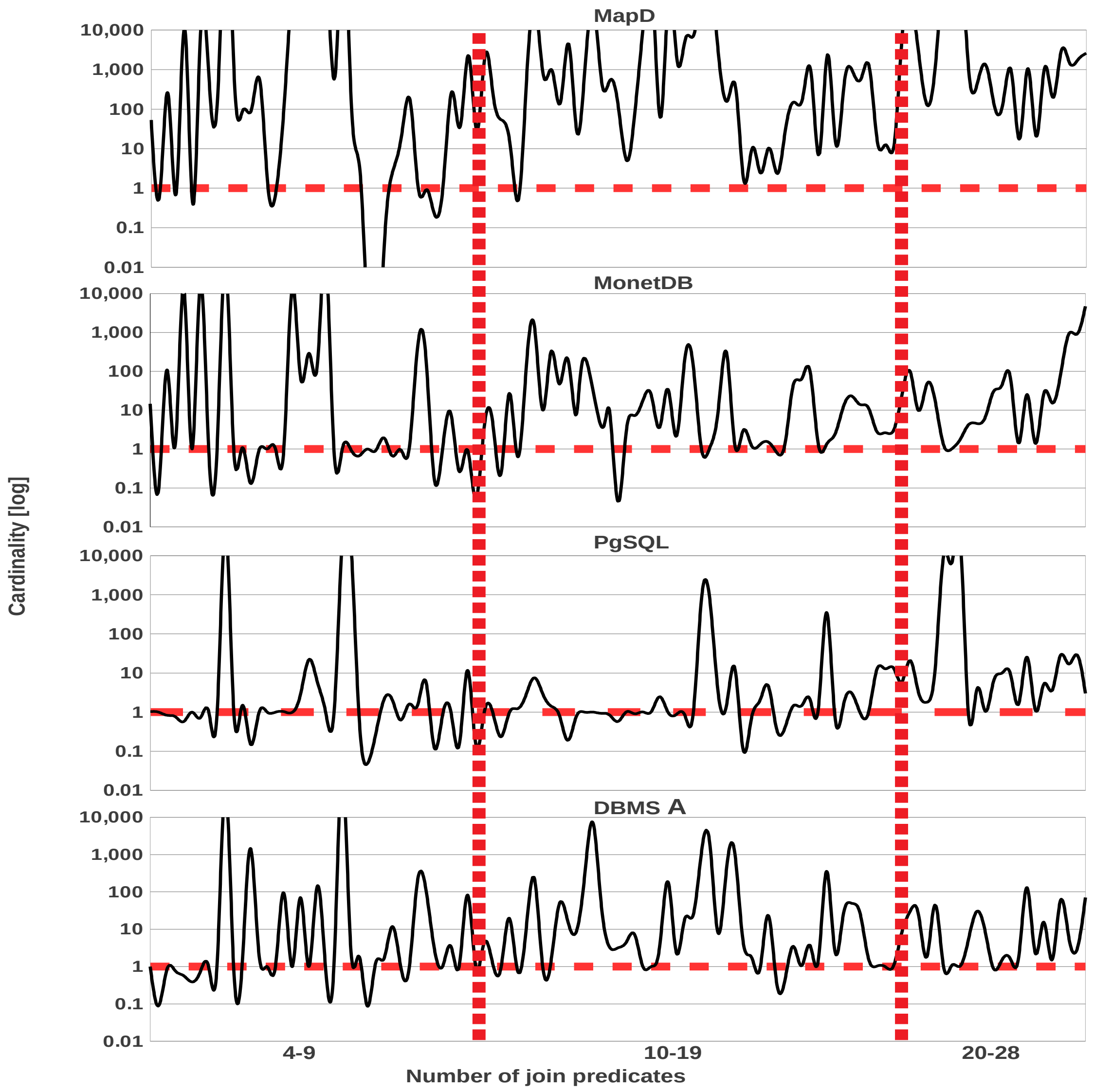


Figure 3.7: Cardinality (in PostgreSQL) as a normalized ratio to COMPASS.

3.7.2.1 Query-level Analysis

In this experiment, we compare COMPASS against every other system for each query in the JOB benchmark. We measure both the intermediate result cardinality, as well as the execution time – taken as the median value over 9 runs. The execution plans are obtained by performing the query in each system. These are subsequently injected in PostgreSQL and DBMS A, and executed on the same execution engine. The cardinalities are generated by executing all the subqueries in the plan in the corresponding order – which is done in PostgreSQL. This information is extracted from the individual plans. Figure 3.7 and 3.8 depict the results normalized to COMPASS. All the values are divided by the COMPASS results – represented as a horizontal dotted line at position 1 on the y-axis. A point below this line means that the other system has a better result, otherwise, COMPASS performs better. The results are grouped by the number of joins in the JOB queries (x-axis) and separated by two dotted vertical lines.

MapD. MapD consistently produces execution plans that have cardinality two orders of magnitude or larger than COMPASS. With a few exceptions, all MapD plans are worse. There is one such query – the discontinuity going to zero in the figure – that indeed has cardinality zero and MapD correctly detects it. However, this is only a matter of chance because the first join in the plan – between the largest tables in the query – does not produce any results. The reason for this poor plan quality is the lack of statistics in the MapD query optimizer. Decisions are taken solely based on the full table cardinality – the number of tuples before any selection predicate. Therefore, the resulting plans are highly sub-optimal. While runtime follows cardinality – with many results 100X slower than COMPASS – the correlation between the two is not complete. There are several queries for which the MapD cardinality is considerably worse, while the execution time is similar to or better than COMPASS. This is the case for some of the complex queries with 20 or more joins executed in PostgreSQL. In this situation, MapD chooses a large well-connected table early in the plan. This allows it to check many join predicates at the beginning and prune a large number of tuples. On the other hand, COMPASS – and the other systems – start from small tables on the periphery of the join graph and make their way to the highly connected tables in the center. This strategy produces many staged intermediate results that increase the runtime. While the runtime trend across PgSQL and DBMS A is similar, we observe that queries with 20 or more joins are handled better by DBMS A, while queries with less than 20 joins are faster in PgSQL. This is an indication that DBMS A is better optimized for complex queries.

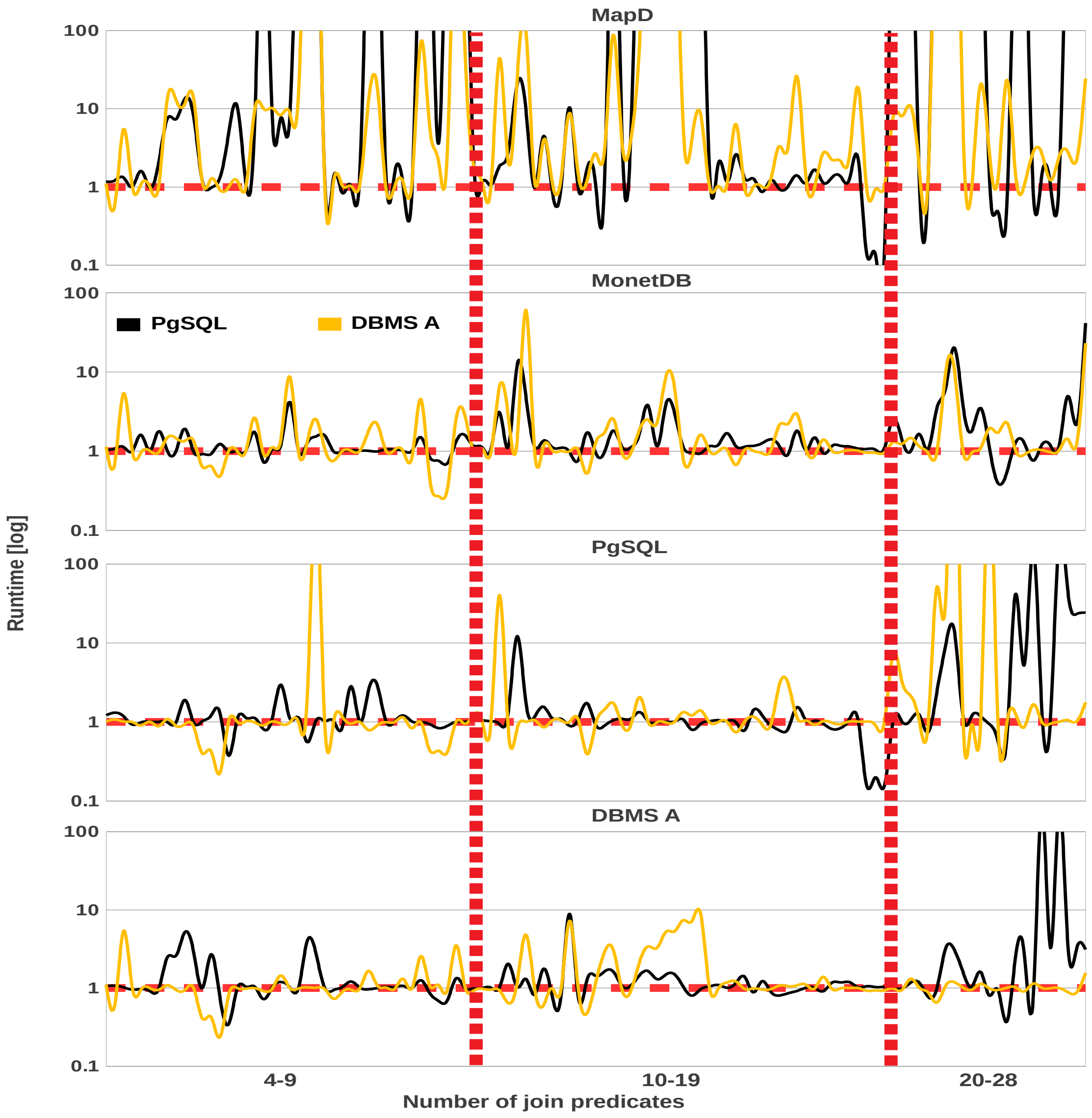
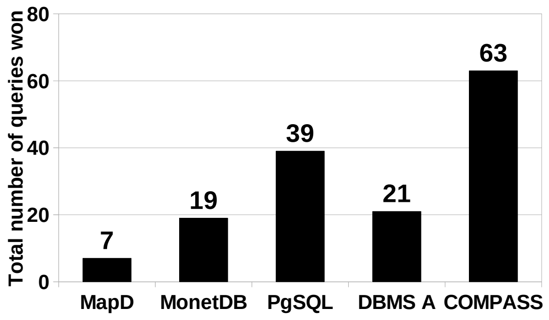


Figure 3.8: Runtime (in PostgreSQL and DBMS A) as a normalized ratio to COMPASS.

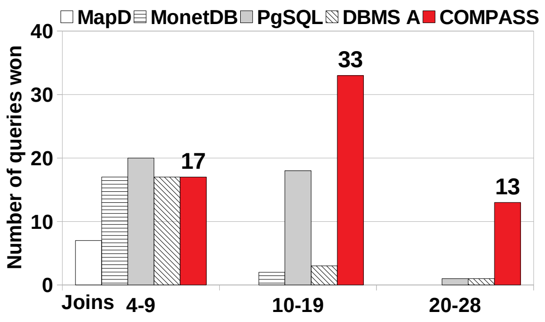
MonetDB. The trend of the cardinality results in MonetDB follows the one in MapD. While the majority of the results are worse than COMPASS, the ratio is smaller than for MapD. This improvement is due to the more advanced rule-based MonetDB query optimizer with limited statistics support. However, compared to the full sketch-based COMPASS, the MonetDB cardinalities are considerably worse – many times an order of magnitude or more. Interestingly enough, though, the corresponding query runtimes fare much better than predicted by the cardinality. With few exceptions, they are always within a factor of 10 – more often less – off of COMPASS. Moreover, they are independent of query complexity and do not exhibit spikes. Overall, the MonetDB runtimes are the most consistent with COMPASS across both PgSQL and DBMS A. This is because the MonetDB query optimizer finds plans that are executed similarly to COMPASS – albeit they have higher cardinality.

PostgreSQL. The cardinality results for PostgreSQL – PgSQL in the figure – are the closest to COMPASS among all the systems. This is entirely due to the advanced statistics the PostgreSQL optimizer employs. While mildly better than COMPASS for several queries, PostgreSQL still exhibits spikes that go beyond a factor of 1000X. The reason is the failure to detect correlations between join attributes. Since the plans are optimized for the PostgreSQL execution engine in this case, we expect the runtimes to be optimal. This is indeed the case for queries with less than 20 joins. However, for 20 or more joins, the PostgreSQL runtime is considerably worse compared to COMPASS. This is where the PostgreSQL optimizer drops dynamic programming in plan search. With a few exceptions where there are dramatic spikes that go beyond 100X, the PgSQL plans executed in DBMS A perform as well as or better than in PostgreSQL itself. This is especially true for complex queries having 20 or more joins. Overall, COMPASS generates more stable plans than PostgreSQL. Although not specifically optimized for it, PostgreSQL executes them as fast – or faster – than its own plans.

DBMS A. The commercial DBMS A produces plans that have consistently higher cardinality than COMPASS across all the JOB queries. This clearly shows that the employed statistics do a poor job of estimating the join cardinality. However, when executed in PostgreSQL, these plans have unexpectedly good runtimes – except for queries with more than 20 joins. This is likely due to the more complex cost function that considers other parameters beyond cardinality in determining the optimal plan such as in Equation 2.4. Interestingly enough, when executing its own plans, DBMS A does not fare better than PostgreSQL, except for the complex queries with more than 20 joins. In fact, DBMS A has a worse runtime for queries with 10 to 20 joins. The runtimes of DBMS A and COMPASS are close to each other and always within a factor of 10X. This confirms that the COMPASS plans are also optimal for DBMS A.



(a) Cardinality comparison



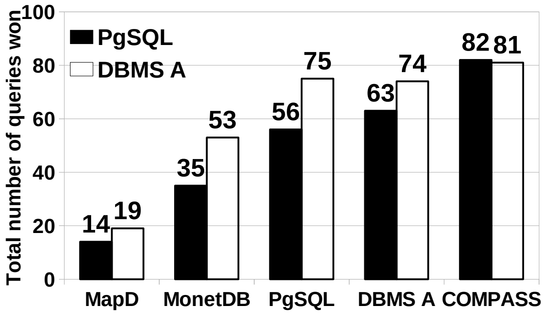
(b) Cardinality vs. number of joins

Figure 3.9: Distribution of winning queries across the databases in terms of intermediate cardinality. The databases with the lowest cardinality are the winners. The total (149) is larger than 113 – the number of queries in JOB – because there are multiple queries for which more than one database is the winner.

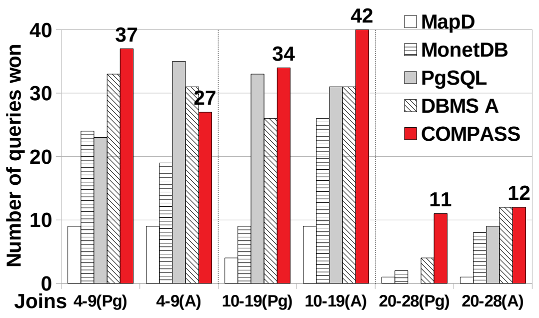
3.7.2.2 Aggregated Workload Statistics

We aggregate the query-level results (Figure 3.7 and 3.8) in order to obtain an overall view of the relative performance of the compared systems. These aggregated results are depicted in Figure 3.9 and 3.10, respectively. They give the total number of queries for which a database performs the best, as well as the distribution as a function of the number of joins in the query. In the case of cardinality, a database is counted if it achieves the minimum cardinality among all the databases. For runtime, a database is counted if it comes within 10% of the fastest runtime – computed as the median of 9 runs. This bound compensates for variations in the environment.

Based on Figure 3.9a, COMPASS achieves the plan with the minimum cardinality for 63 out of the 113 JOB queries. This represents approximately 56% of the workload. PostgreSQL (PgSQL) comes in second place with 39 queries. The other three databases obtain the best cardinality in less than 20% of the queries each, with MapD winning only 7 queries. The careful reader notices that the sum of the winning queries is larger than 113. This is because there are queries for which two or more systems achieve the same best cardinality – case in which we count each of them. The distribution of the winning queries in terms of the



(a) Execution time comparison



(b) Execution time vs number of joins

Figure 3.10: Distribution of winning queries across the databases in terms of execution time when the optimal plans are plugged-in and executed in PostgreSQL and DBMS A. We obtain the optimal plan for every database from its query optimizer and execute it in PostgreSQL and DBMS A. All the databases within 10% of the fastest execution time are considered winners. Thus, the total (250) is larger than 113 – the number of queries in JOB.

number of joins is depicted in Figure 3.9b. While for the simpler queries with less than 10 joins all the systems perform similarly, COMPASS clearly dominates the others when the complexity increases. PostgreSQL is the only other database that performs sufficiently well, however, only for queries with a moderate number of joins. These results prove the benefit of using statistics in query optimization, especially for complicated queries. While the PostgreSQL statistics perform well for simple to moderate queries, COMPASS sketches are less sensitive to the number of joins in the query – they provide more consistent estimates. Moreover, COMPASS is not heavily impacted by the greedy join enumeration algorithm. When PostgreSQL switches from dynamic programming – more than 18 joins – it fails to find any best plan.

The aggregated runtime results in PgSQL and DBMS A are depicted in Figure 3.10a and 3.10b. They follow closely the corresponding cardinality results – with one exception. The runtime for the commercial DBMS A is much better than its cardinalities anticipate – DBMS A has the best runtime for 63 and 74 queries, while its cardinality is best only for 21 queries. The reasons are outlined when the individual query results are discussed. Additionally, DBMS A benefits from the bound on runtime since it often comes within the fastest system. Overall, COMPASS achieves the fastest runtime for 82 (PgSQL) and 81 (DBMS A) out of the 113 JOB queries – 72% of the workload – which is more than any other database. This proves the superiority of the identified plans and confirms the correlation between cardinality and runtime. The correlation manifests more clearly for queries with a larger number of joins because of the higher runtime, which makes ties more unlikely. Moreover, the correlation is stronger for PgSQL than for DBMS A since the number of winning queries is higher in DBMS A for all systems except COMPASS. A careful reader observes that the runtime results are higher than the cardinality results for all the systems – and larger than 113 when summed up. This is because it is more common to have close-enough runtimes than it is to have the same cardinality – multiple counting is more frequent. Based on these results, we conclude that COMPASS is the optimizer with the most consistent and resilient plans on the JOB benchmark.

3.7.2.3 Total Workload Runtime

The runtimes for the complete JOB workload execution in PostgreSQL and DBMS A using the plans generated by each database are included in Table 3.2. Given the high variance among queries, these numbers have to be taken with a grain of salt since they may be dominated by a few complex queries with a large number of joins. Nonetheless, we follow prior art [16, 108] and include them together with the aggregated workload statistics. As expected, COMPASS has the overall fastest runtime. Somewhat unexpectedly, MonetDB comes in second for the PgSQL execution with a runtime that is almost twice as large as that of COMPASS. The reason is that MonetDB does not fail dramatically for any of the JOB queries. While it performs consistently slower, it never derails on heavily sub-optimal plans. The runtime for PgSQL and DBMS A in PgSQL is dominated by the long-running queries with 20 or more joins, which pull the total time to more than 8X and 5X that of COMPASS. These outliers are sufficient to skew the overall runtime. In the case of MapD, there are 30 queries that do not finish execution even after a timeout of 20 minutes per query. Thus, the very large runtime. When the workload is executed in DBMS A, all systems except DBMS A incur an increase in runtime. The increase is most significant for COMPASS as it stands at 50% more than in PgSQL. On the other hand, DBMS A has a reduction of more than 50% of its PgSQL runtime. Nonetheless, COMPASS still has the overall fastest runtime, which is 35% faster than DBMS A.

Database	**Runtime (minutes)**		**Ratio to COMPASS**	
	PgSQL	DBMS A	PgSQL	DBMS A
MapD	>300	>300	**>23**	**>13**
MonetDB	27.52	35.71	**2.19**	**1.65**
PgSQL	103.00	244.31	**8.20**	**11.28**
DBMS A	70.72	29.22	**5.63**	**1.35**
COMPASS	**12.56**	**21.66**	1.00	1.00

Table 3.2: JOB benchmark runtime in PgSQL and DBMS A.

3.7.2.4 Comparison with Pessimistic Plans

We compare the plans produced by COMPASS against the pessimistic plans generated in [37]. The pessimistic plans are determined by minimizing the worst-case cardinality estimates. Thus, they always produce overestimates of the true cardinality. This is in contrast to COMPASS, which generates both over- and under-estimates. The pessimistic plans for all the JOB queries in PostgreSQL are available at [36]. They are generated by rewriting the SQL statements such that selection predicates and one-to-many – key/foreign-key – joins are evaluated before the many-to-many joins. Moreover, ordering is performed separately for one-to-many and many-to-many joins. This partitioning of the search space results in a massive reduction of the number of considered join orders. A similar idea is employed in [16], where only at most 2-D partitioned sketches are built.

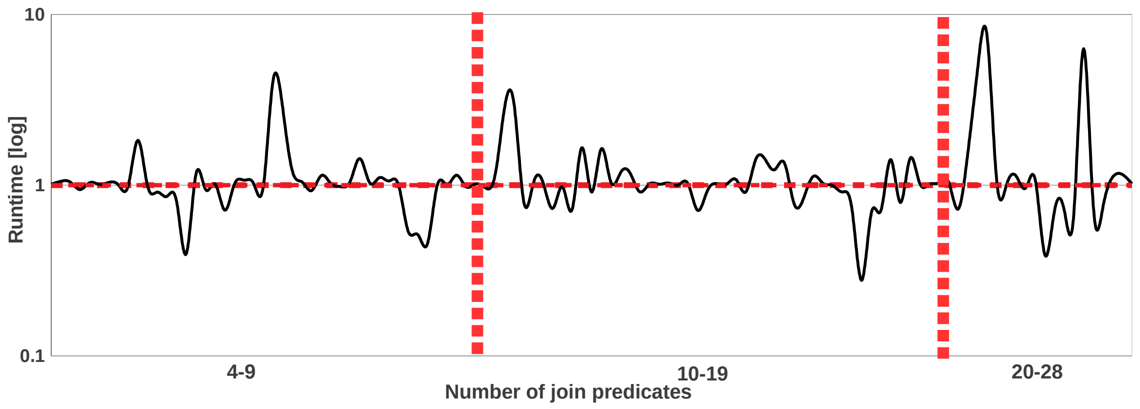


Figure 3.11: Runtime of pessimistic plans (in PostgreSQL) as a normalized ratio to COMPASS.

The comparison between the runtime of the COMPASS plans and that of the pessimistic plans executed in PostgreSQL is depicted in Figure 3.11. The results are normalized to the runtime of the COMPASS plans. We observe that the difference between these plans is smaller than for the other systems – an indication that the plans have closer runtime. There are queries for which COMPASS generates faster plans and queries for which the pessimistic plans are better. Overall, there is a slight advantage for COMPASS since the curve is above the horizontal 1-axis more often. Moreover, the gap is higher for COMPASS, reaching a factor of almost 10X for certain queries. In terms of number of joins, the best plans are almost evenly distributed among the two methods. Table 3.3 summarizes the individual query results from Figure 3.11. The timing results are higher than previously published [37] because no indexes are defined over the key attributes. COMPASS achieves a slightly better performance both in the number of queries won – 88 vs. 83 – as well as in the cumulative runtime – COMPASS is faster by approximately 25 seconds. The main reason for the better performance of pessimistic plans – compared to other systems – is the separate optimization of the joins. The partition of the join graph based on the many-to-many joins and their independent ordering reduces the multi-way join estimation error significantly. The separate evaluation of the one-to-many joins in every partition reduces the error further. Moreover, the estimation for these simpler joins is more accurate. While partitioning the join order space reduces estimation complexity, it also ignores orderings that can result in better plans. For example, the cardinality of a many-to-many join can be smaller than that of a one-to-many join. Pessimistic plans ignore these interleaved orders. Given the holistic approach that considers the complete join graph, the COMPASS results are quite impressive given the size of the multi-way joins. This is possible because of the good accuracy – and consistency – of the merged Fast-AGMS sketches. We conjecture that COMPASS can be improved by adopting a similar tiered approach to join ordering.

Query plans	**Queries won**	**Runtime (seconds)**	**Ratio to COMPASS**
Pessimistic	83	777.10	**1.03**
COMPASS	**88**	**753.86**	1.00

Table 3.3: JOB benchmark execution for pessimistic [16, 37] and COMPASS plans in PostgreSQL.

3.7.2.5 Runtime in MapD

In this experiment, we evaluate the impact COMPASS has on the MapD database. For this, we replace the default MapD query optimizer with COMPASS and execute the JOB

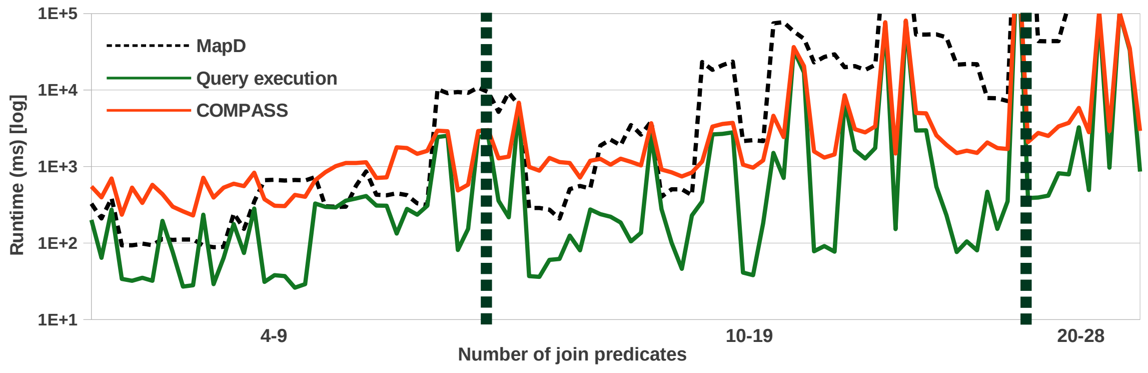
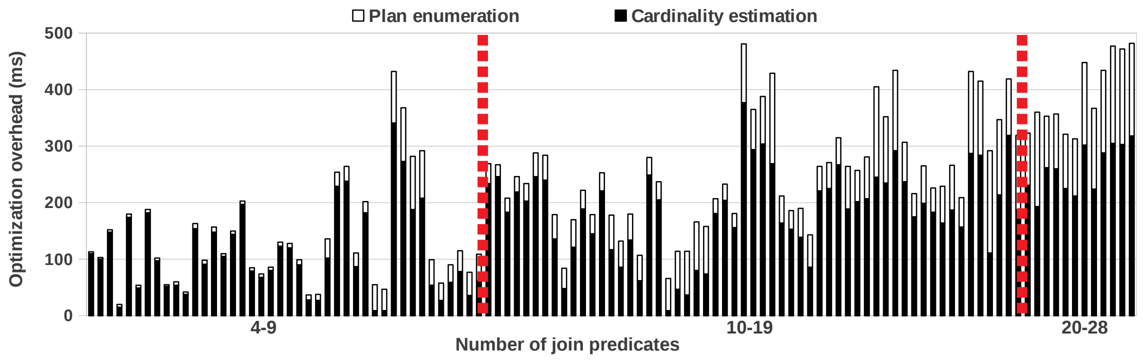


Figure 3.12: Query runtime in MapD with the COMPASS query optimizer.

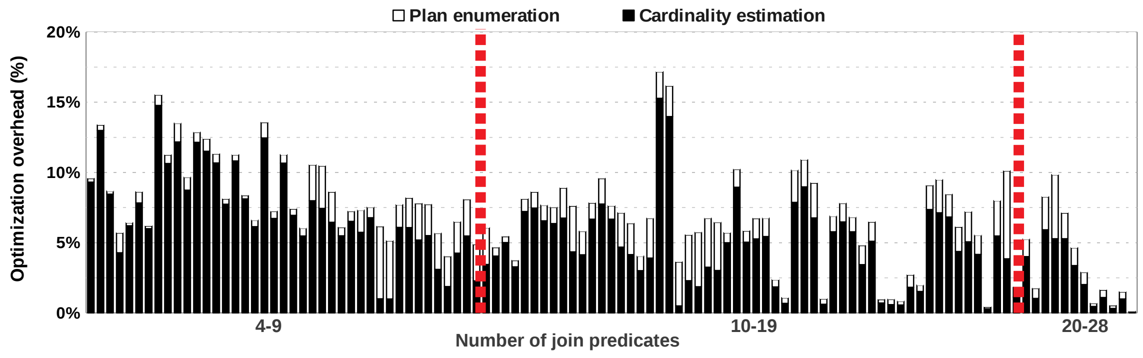
benchmark in both scenarios. We measure the end-to-end query runtime, as well as only the query execution time without optimization – these are the same in MapD. We report the median over 9 runs. Figure 3.12 depicts the results for every query. We observe that MapD outperforms COMPASS for simple and some moderate queries. However, the differences are not significant, as opposed to the difference for more complex queries. This may be surprising given the primitive MapD query optimizer. However, its execution engine is quite different from PostgreSQL. It is highly optimized for parallel in-memory processing. This alleviates the need for careful optimization of simple queries. For more complicated queries, though, sketch-based optimization pays off as COMPASS finds considerably better plans. In fact, MapD fails on 8 queries and times out after 30 minutes on 8 other queries. COMPASS finishes all the queries and is faster than MapD for 74 of them, which represents 65% of the workload. The total runtime for the 97 queries MapD successfully runs is included in Table 3.4. COMPASS has a runtime of 6.21 minutes to MapD's 47.64 – which is a net speedup of 7.67X. This proves both that sketches can be effectively computed at runtime, as well as their benefit to generate better query plans, which result in faster execution. The last point is clear when we compare only the execution time, without optimization overhead – less than ten COMPASS plans have execution times larger than MapD.

3.7.2.6 COMPASS Overhead

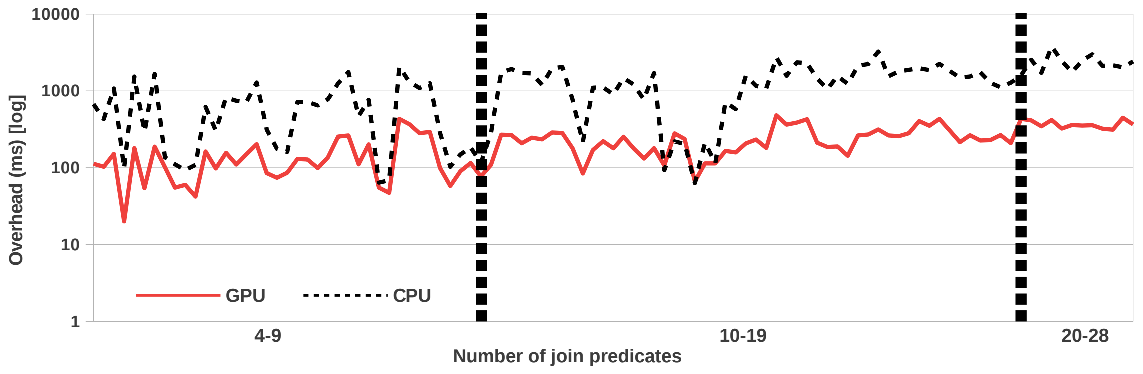
We measure the optimization overhead of building Fast-AGMS sketches, as well as that of sketch-based plan enumeration, for the COMPASS MapD implementation. Sketch building can be performed either on GPU or CPU while merging and plan enumeration are performed



(a) Optimization overhead on GPU.



(b) Optimization overhead on GPU as a percentage from the total runtime.



(c) Optimization overhead on GPU and CPU.

Figure 3.13: The overhead of the COMPASS optimizer implemented in MapD.

on CPU. The results are depicted in Figure 3.13. Figure 3.13a and 3.13b show the absolute and relative overhead, respectively, for the GPU execution. The overhead in queries with up to 9 joins is at most 15% – or at most 420 ms. At first glance, the overhead may seem significant. However, this time is not spent in vain since the plans selected by COMPASS are quite fast even with the overhead included. For these simpler queries, there is no significant difference between plans. Thus, a primitive optimizer as in MapD is sufficient. The overhead for the rest of the workload is at most 17% – or 500 ms. This overhead becomes negligible in the overall execution time. As a result, COMPASS largely outperforms the other four databases. Figure 3.13c shows that – as expected – sketch building is more efficient on GPU than on CPU due to the higher degree of parallelism. In both cases, the optimization overhead increases with the number of joins in the query. For GPU, the overhead is in the order of hundreds of milliseconds (ms), with a maximum of around 500 ms for certain complex queries. For the CPU, the overhead is always below 5 seconds, which is relatively small for queries that take minutes to run. Given that this is only a prototype, we believe that the sketch overhead can be further reduced with more optimized code.

Database	**Queries won**	**Runtime (minutes)**	**Ratio to COMPASS**
MapD	42	47.64	**7.67**
COMPASS	**74**	**6.21**	1.00

Table 3.4: JOB benchmark execution in MapD.

3.7.2.7 Comparison with State-of-the-art Synopses for Join Cardinality Estimation

We compare COMPASS against seven methods for join cardinality estimation following the study presented in [52]. These methods are PostgreSQL, AGMS sketches [9, 8], random table samples (TS), correlated samples [110], join samples (JS), KDE with table samples (TS+KDE), and KDE with join samples (JS+KDE) [52]. We perform all the experiments for join cardinality estimation on three types of JOB queries over at most five tables – the simplest in the benchmark – as presented in [52]. We use the publicly available code, workload, and data from [51]. The results are depicted in Figure 3.14. We observe that the COMPASS accuracy matches that of the best estimators closely for all the queries. The only two estimators that always outperform COMPASS are based on join samples (Join Sample and JS+KDE). These types of estimators require indexes on all possible join attribute combinations, thus, they have a high set-up and maintenance cost. Moreover, the KDE models are trained on query samples with the same set of selection and join predicates, i.e., the same type of training queries with different constant values. In addition, JS+KDE requires training for every join size. Thus, it is not clear what the behavior of the KDE estimators on different types of queries – the sub-queries enumerated by the optimizer, in particular. Notice that the results also include sketches (AGMS). However, these are the AGMS sketches, not the Fast-AGMS sketches on which COMPASS is built. Given the detailed sketch comparison in [97, 98], the difference between the two is expected. Since we perform a thorough query plan evaluation with PostgreSQL (Postgres), the comparison in terms of accuracy is interesting. While COMPASS and Postgres have very similar accuracy – with an advantage for COMPASS – our holistic results prove that COMPASS generates better query plans both in terms of quality and execution time. This proves that other factors beyond single query accuracy – such as sub-plan enumeration and estimator composition – have to be considered by the optimizer. Overall, the comparison in terms of accuracy is limited to a series of relatively simple hand-picked queries with at most four joins – nothing close to the full JOB benchmark. While useful, it fails to confirm the practicality of the considered approaches in a complete query optimizer, which COMPASS does.

3.7.2.8 Plan Enumeration Analysis

We assess the performance of the join order enumeration algorithm by varying the search space. We set four different dimensions in the left-deep search space–*greedy*, *full-greedy*, *limit-10*, and *exhaustive*. The *greedy* solution traverses the join graph from a single source and greedily adds nodes relying on the sketch estimations. The source node is chosen as the smallest table from the two-way join that has the smallest cardinality estimation. *full-greedy* executes the *greedy* traversal from every node in the join graph. The join order with the smallest overall estimation is selected. *limit-10* enhances *full-greedy* with backtracking. Rather than stopping after the first greedy plan is generated, *limit-10* traverses additional paths in the join graph until the first 10 join orders are generated, i.e., max_plans is set to 10 in Algorithm 1. Lastly, *exhaustive* traverses the entire search space to find the optimal plan, i.e., max_plans is set to ∞ in Algorithm 1. Sketch merging and estimation are arithmetic operations amenable to parallelization. Thus, the traversal algorithm quickly explores even the large spaces enumerated by *exhaustive*. To further speed up the enumeration phase, two practical heuristics are applied in Algorithm 1 (see Section 3.6).

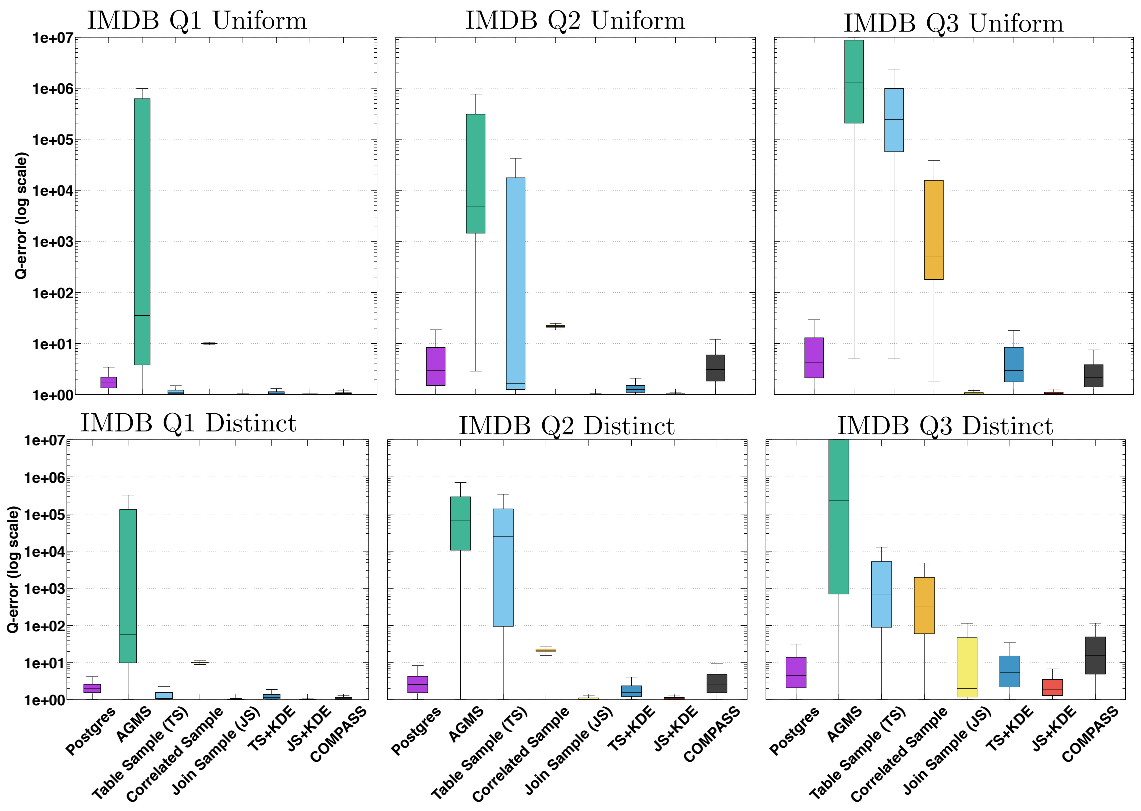


Figure 3.14: Comparison with state-of-the-art techniques for cardinality estimation.

We collect the join order corresponding to every approach for all the JOB queries and compare their intermediate cardinality and runtime. The results are depicted in Figure 3.15. Intuitively, we expect that the larger the search space is, the higher the odds of finding a better join order plan. However, as discussed in [65, 63], the plan enumeration algorithm has a relatively small impact on plan quality. Our results confirm this hypothesis. We observe that no particular approach is significantly better for the proposed sketch-driven traversal algorithm. In larger search spaces, sub-optimal plans are pruned by the early stopping criteria since the overall cardinality estimates exceed the current minimum cost. The plan search space is not only limited by the join graph. The search space may shrink because of the early stopping criteria that compare the overall cardinality estimates for the sub-queries. For example, it is possible the execution plans for *greedy* and *full-greedy* are the same, although *full-greedy* searches from each node separately. The reason is that all the other execution plans selected by *full-greedy* may have a larger overall cost. In fact, even the sub-queries may have already a larger cost than the plan selected by *greedy* solution – thus the enumeration algorithm backtracks. As a result, there is no particular preference for the search space size since the cardinality differences are not significant in Figure 3.15 (upper part of the figure). Although there are some outliers – spikes in the figure – they are mostly selected by *full-greedy*, *limit-10*, and *exhaustive*. This behavior is caused by the over- and under-estimates. Also, there is no particular trend when comparing the runtime corresponding to the different plans – except for a few outliers in *exhaustive*. Moreover, there is no correlation between cardinality and runtime results for the outliers. We believe the plan differences chosen by these four different solutions are not significantly different in order to notice changes in terms of the runtime. The results confirm that the plan chosen by *greedy* is not necessarily slow and, thus, we conclude that COMPASSS is not heavily impacted by the join enumeration algorithm. In fact, COMPASS outperforms the other systems, especially for complex queries with a large number of joins.

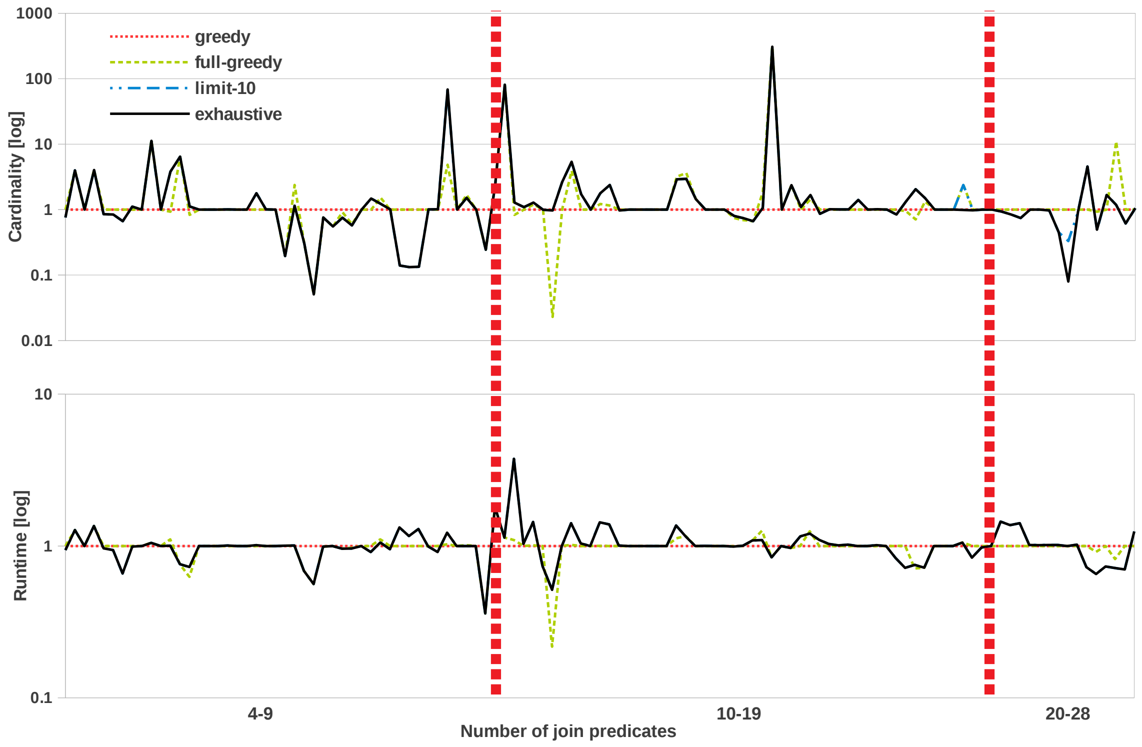


Figure 3.15: The effect of plan enumeration algorithms on the cardinality and runtime of the selected plan. Values are normalized with respect to the cardinality and runtime of the plan determined by the greedy algorithm. Values above the "1" horizontal line are worse while values below are better.

3.7.3 Summary

Based on the presented results, we can answer the questions raised at the beginning of the experimental section:

- COMPASS generates query plans with the lowest cardinality among all the considered systems for 56% of the queries in the workload. This percentage increases to 65% for complicated queries with 10 or more joins.
- The better plans identified by COMPASS translate into faster query runtimes in PostgreSQL, DBMS A, and MapD. Out of the 113 JOB queries, COMPASS achieves the fastest runtime for more than 80 in PostgreSQL and DBMS A, and 74 in MapD. This confirms the correlation between cardinality and runtime. DBMS A is the only database that does not satisfy this correlation, which can be problematic for a user.
- COMPASS and MonetDB are the only databases that perform all the JOB queries without serious hiccups both in PostgreSQL and DBMS A. The other systems have several queries for which the runtime "explodes". This results in a significantly higher workload runtime. On the PostgreSQL engine, COMPASS outperforms MonetDB by a factor of 2.19X, while on DBMS A by 1.65X. DBMS A optimizes queries specifically for its engine, resulting in a significant reduction in runtime compared to PostgreSQL. However, COMPASS is faster by a factor of 1.35X. Moreover, COMPASS is at least 7.67X faster than MapD.
- Even though COMPASS treats the complete join graph in order – which results in a much larger search space – it manages to outperform the pessimistic optimizers – which perform join ordering in a tiered approach. While only a 3% reduction in the workload runtime, this improvement is significant because COMPASS achieves a faster runtime and higher speedup for more queries.
- The overhead incurred by the COMPASS optimizer in MapD is less than 500 milliseconds on GPU and less than 5 seconds on CPU. While this may be too large for simple queries, it results in faster execution for more than 91% of the queries.
- The accuracy achieved by COMPASS matches – and often surpasses – that of most of the state-of-the-art methods for join cardinality estimation. The only methods that have better accuracy are based on join samples, which have never been fully integrated into a query optimizer because of their complexity. If we consider only methods implemented in existent query optimizers, COMPASS has better overall accuracy.

- The COMPASS plan enumeration algorithm can be customized to perform the search for the optimal join order from a limited greedy to an exhaustive left-deep tree. However, the difference between these alternatives is not significant enough to compensate for their gap in overhead. Thus, the fast greedy join order enumeration is sufficient to achieve the optimal COMPASS plans.

3.8 Conclusions

We introduce the online sketch-based COMPASS query optimizer, which uses exclusively Fast-AGMS sketches for cardinality estimation and plan enumeration. Fast-AGMS sketches are computed online by leveraging the optimized parallel execution engine in modern databases. Selection predicates and sketch updates are pushed-down and evaluated online during query optimization. Plan enumeration is performed over the query join graph by incrementally composing the corresponding sketches. We prototype COMPASS in MapD and perform extensive experiments over the complete JOB benchmark. The results prove the reduced overhead COMPASS incurs while generating better execution plans than four other database systems – COMPASS outperforms four other databases on all the considered metrics over the JOB benchmark.

Chapter 4

Spanning Tree-based Query Plan Enumeration

Plan enumeration is a critical component of query optimization. To find an optimal query plan, conventional query optimizer architectures employ either exhaustive or heuristic plan enumeration strategies. In the case of large queries, some architectures offer flexibility by switching from exhaustive to heuristic strategy [88, 86]. This implies a trade-off between plan optimality and optimization time. Exhaustive strategy requires enumerating every possible query plan to ensure the selection of the optimal one, yet this thoroughness results in considerable optimization time. On the other hand, heuristic strategies enumerate a single plan which may not be optimal but offers faster optimization time. Particularly, deterministic heuristic strategies, such as greedy algorithms, tend to get trapped in local optima – consistently yielding suboptimal plans. Shifting between these two extremes, exhaustive and heuristic strategies, can lead to a significant change – increase or decrease in optimizer performance. As an intermediate strategy, plan enumeration can enumerate more than a single query plan by introducing randomness in selecting query plans to diversify the set of query plans [104, 43, 106, 88, 107, 44]. This increases the chances to locate superior query plans thus reducing the gap between plan optimality and optimization time. While randomized heuristic strategy requires an additional optimization time, by maintaining a polynomial time complexity, this strategy ensures its feasibility for practical use. However, the optimizer performance becomes less predictable and interpretable due to the random behavior of the search space exploration.

In this work, we define query optimization as the problem of finding spanning trees with low costs. The entire search for optimal query plans is defined in terms of join graph edges, thus naturally avoiding cross-joins. Our objective is to identify an ordered sequence of edges that span all vertices in the join graph while minimizing the total sum of edge weights. This formulation leverages the extensive application of spanning tree algorithms in query optimization. Building on this foundation, we introduce Ensemble Spanning Tree Enumeration (ESTE) as a novel, intermediate strategy for plan enumeration. ESTE leverages a set of fast, spanning tree-based algorithms to enumerate multiple spanning trees with low costs, each exploring different areas of the search space. Within ESTE, we implement two classic spanning tree algorithms, Prim's and Kruskal's, to enhance the robustness of the query optimizer. ESTE enables a systematic and extensive exploration of distinct areas within the search space, thereby improving the quality of query plan generation. Furthermore, the flexibility of ESTE allows for the incorporation of additional spanning tree algorithms in response to changes in data and workload, thus maintaining robustness and even enhancing the performance of the query optimizer. We believe that our approach offers a more cost-effective way to sustain optimizer robustness, favoring the integration of new algorithms over a complete change and redevelopment of the optimizer for different plan enumeration strategies. We outline the key technical contributions of our work and highlight how it diverges from and builds upon prior work as follows:

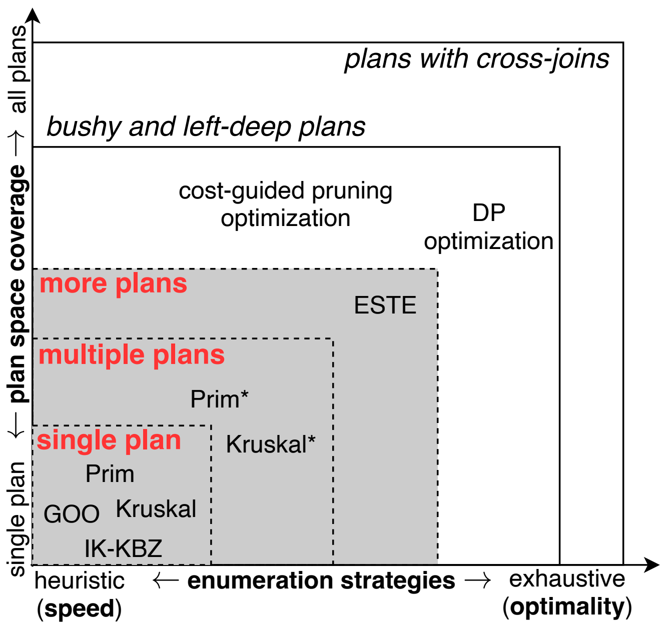
- We present a novel perspective on query optimization by framing it as a problem of finding spanning trees with low costs. Unlike in the MVP and GreedyJoinOrdering-3 [62, 77], unnecessary joins and cross-joins are naturally avoided by representing tables as vertices and operating over existing edges in the join graph. Unlike in directed plan graph representations [83, 31], the search space does not increase due to the simplicity of the join graph representation.
- In this work, we adapt both Prim's and Kruskal's algorithms to account for changes in edge weights as spanning trees are built. Unlike in IK-KBZ and GreedyJoinOrdering-3 [58, 77], the resulting spanning tree is not restricted to left-deep trees.
- Unlike in previous work [58, 27, 77], we utilize a cost function that considers physical operators such as multiple join algorithms and scan operators [65, 63]. Thereby the

resulting spanning tree is a physical plan rather than a logical plan.

- We introduce the Ensemble Spanning Tree Enumeration (ESTE) strategy, a novel approach that systematically harnesses the unique plan search space exploration methods inherent to Prim's and Kruskal's algorithms. Unlike in the GOO [27], multiple query plans are enumerated. Unlike in the GreedyJoinOrdering-3 [77], multiple enumeration algorithms pass over the join graph. ESTE explores the search space by employing a stack of spanning tree algorithms to provide robustness to query optimization.
- We evaluate query plans produced by the ESTE strategy and compare to an exhaustive enumeration and GOO [27], which is recognized as one of the top-performing heuristic methods in recent research [65, 63, 86, 31].
- We examine the performance behavior of ESTE in the presence of cardinality estimation errors. As expected, we observe a decline in efficiency across all enumeration methods, including exhaustive enumeration. The results exhibit a high performance of ESTE, comparable to those achieved through exhaustive enumeration, while also maintaining a low optimization time.

4.1 Enumeration Strategies

In this section, we outline existing common strategies and describe the trade-off between discovering the optimal plan and preserving computational efficacy [61, 104]. These strategies are discussed in terms of their plan search space coverage, the quality of the selected plan, and computational overhead. Figure 4.1 shows the search space coverage by different enumeration strategies. The x-axis presents enumeration strategies starting from heuristical strategy (faster search time) to exhaustive strategy (preserved plan optimality). The y-axis describes the search space coverage. We use the area of rectangles of different sizes to describe the correlation between the two axes. Note that there is no position difference on the y-axis for the enumeration algorithms within the same rectangle. The largest solid rectangle covers all possible query plans including the plans containing cross-joins. The second solid rectangle covers the rest of the tree shapes including bushy, left-deep, right-deep, and zig-zag trees.



plans with cross-joins
bushy and left-deep plans
cost-guided pruning optimization
DP optimization
more plans
ESTE
multiple plans
Prim*
Kruskal*
single plan
Prim
GOO
Kruskal
IK-KBZ
single plan ← plan space coverage → all plans
heuristic (speed) ← enumeration strategies → exhaustive (optimality)

Figure 4.1: Enumeration strategies and their ability to cover the plan search space.

4.1.1 Query Plan Optimality

A straightforward and most expensive approach in plan enumeration is to generate all possible edge arrangements and evaluate them based on a cost model. This strategy is shown as the largest rectangle in Figure 4.1. This approach guarantees the optimality of the selected plan. However, due to the combinatorial explosion of possible query plans, this naive enumeration approach becomes computationally prohibitive for queries involving a large number of relations. A solution to apply dynamic programming is initially put forth in the context of plan enumeration in [100, 109]. Dynamic programming offers enumerating of possible trees in a bottom-up fashion – from smaller trees to larger trees. At each step, it evaluates the cost of the different plans using a cost model and stores the least costly plan for each subquery. Selinger et al. [100] proposed a solution that restricts the search space to left-deep trees. Vance and Maier [109] presented an algorithm to cover bushy trees as well as cross-join search space. These optimizations target the largest rectangle in Figure 4.1. However, these solutions do not fully utilize the information available in the join graph. The issue of not fully utilizing the information contained in the join graph is addressed by introducing graph-based dynamic programming [79, 80]. Alternatively, top-down variations are proposed while maintaining similar efficiency and proposing cost-based pruning mechanisms [21, 28]. Illustrated within the second largest rectangle in Figure 4.1, these enumeration methods, based on graph-based dynamic programming, provide optimizations to circumvent unnecessary enumeration of subplans which are deemed suboptimal, thus further ensuring effectiveness. While these methods enhance the search overhead and guarantee polynomial-time performance on chain queries, their efficacy may be constrained when applied to queries with a large number of relations.

4.1.2 Optimization Time

Deterministic heuristics. In large queries, even with integrated graph-based dynamic programming and pruning optimizations, exhaustive enumeration can still be computationally demanding. Therefore, enumeration strategies based on deterministic heuristics, such as greedy and rule-based, are employed. These strategies utilize general principles to potentially select optimal plans, though they do not assure query plan optimality [106, 104, 27, 15, 85, 56]. In exchange for sacrificing the certainty of achieving an optimal solution, deterministic heuristics provide predictability, simplicity, and fast optimization time. These strategies are depicted in the smallest rectangle in Figure 4.1. Deterministic heuristics invariably generate a single plan for a given query.

```
SELECT COUNT(*)
FROM keyword k, movie_keyword mk, title t,
   company_name cn, movie_companies mc
WHERE k.keyword = 'character-name-in-title' AND cn.country_code = '[de]'
   AND mk.keyword_id = k.id AND t.id = mk.movie_id AND t.id = mc.movie_id
   AND mk.movie_id = mc.movie_id AND mc.company_id = cn.id
```

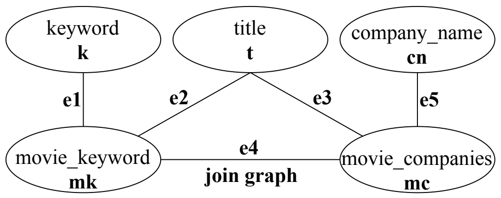


Figure 4.2: SQL statement for JOB query 2a and its join graph. The SQL statement has 2 point selection predicates and 5 join predicates connecting 5 tables. Further, the query is illustrated as an undirected join graph $G(V, E)$ in which every table is represented as a vertex v connected by edges e for every join predicate.

The greedy plan enumeration strategy operates by progressively constructing a binary tree. This method selects the next joining relation with the minimum cost at each join level with the hope that these locally optimal decisions can lead to the global optimal plan. In

Figure 4.3, the middle and rightmost trees are built greedily for query 2a illustrated in Figure 4.2. For example, in the case of the rightmost tree, the edge that results in the subplan with the minimum cost is selected at each join. We start with the 2-way join $\mathcal{P}_{mc \bowtie cn}$ having the minimum cost $\mathcal{C}(\mathcal{P}_{mc \bowtie cn}, Y) = 1M$ out of all 2-way joins. Next, we select the 3-way join $\mathcal{P}_{mc \bowtie cn \bowtie t}$ having the minimum cost $\mathcal{C}(\mathcal{P}_{mc \bowtie cn \bowtie t}, Y) = 0.3M$ out of all 3-way joins that can be constructed from $\mathcal{P}_{mc \bowtie cn}$. We repeat this procedure until we cover all the vertices in the join graph, making sure that we always maintain a single connected component without cycles. The pursuit of global optimality is compromised as decisions are made based on local conditions – choosing the next join operation solely based on its standalone cost, thus disregarding past decisions. Conditioning the available relations at each join level is based on prior decisions thus further diminishing the probability of finding an optimal query plan. Figure 4.3 illustrates this issue. The exhaustive search finds the optimal plan $\mathcal{P}_{opt}$ based on reaching all distinct 14 subplans, computing all distinct 32 join costs and comparing all possible 72 query plans – excluding the cross-joins. However, for the plans $\mathcal{P}_{kru}$ and $\mathcal{P}_{pri}$, 3-way joins $\mathcal{P}_{mk \bowtie k \bowtie mc}$, $\mathcal{P}_{mk \bowtie k \bowtie t}$, $\mathcal{P}_{mk \bowtie t \bowtie mc}$ are never considered. In 4-way joins, $\mathcal{P}_{mk \bowtie k \bowtie mc \bowtie t}$ and $\mathcal{P}_{mk \bowtie k \bowtie mc \bowtie cn}$ are also never reached. The former 4-way join has significantly less cost and intermediate table cardinality. As a result, the greedy query plans are suboptimal compared to their exhaustive counterparts. Their overall plan costs are $\mathcal{C}(\mathcal{P}_{opt}, Y) = 1.6M$, $\mathcal{C}(\mathcal{P}_{kru}, Y) = 2.4M$ and $\mathcal{C}(\mathcal{P}_{pri}, Y) = 4.6M$, respectively. These cost differences are expected given the smaller number of subplans, join costs and full plans considered by the greedy algorithms – 9, 10 and 2, respectively.

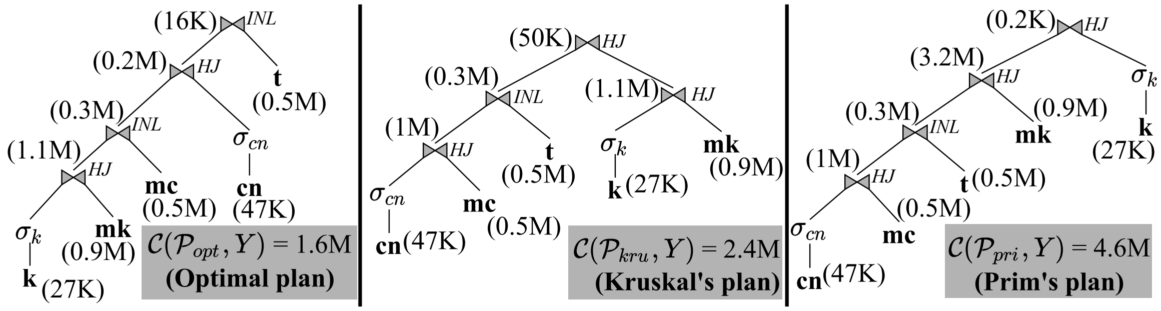


Figure 4.3: Three query plans ($\mathcal{P}_{opt}$, $\mathcal{P}_{kru}$ and $\mathcal{P}_{pri}$) for query 2a selected using 3 different query plan enumeration algorithms (Exhaustive, Kruskal, and Prim) along with their costs $\mathcal{C}$ computed using true cardinalities Y.

Randomized heuristics. Deterministic behavior of the heuristics can be trapped in local optima and may not always find the optimal query plan. Randomized heuristics, on the other hand, introduce an element of randomness into the decision-making process [104, 43, 106, 88, 107, 44]. Randomized heuristics such as genetic algorithms [88], simulated annealing [44], iterative improvement [107] and learning-based methods [72, 108, 121], offer ability to randomly explore different areas of a given search space thus increasing the chances to find better query plans. By initiating from diverse starting points or considering a larger number of plans, these methods strive to identify the optimal plan. Although this might necessitate a longer search time, the process remains within polynomial time complexity, thereby ensuring its feasibility in practical applications. It is important to note that while randomized heuristics may seem attractive due to their potential for finding better query plans through a broad search space exploration, their effectiveness in optimization time may be undermined if suboptimal query plans are selected. Therefore, the impact of suboptimal choices could offset the benefits of the diversified exploration of the search space.

Aside from providing faster search time, greedy algorithms inherently prioritize early-stage joins and consider fewer subplans of large join size. This property becomes important when using estimated cardinalities. In principle, cardinality estimates for smaller joins are more accurate due to the reduced correlation between tables. Hence, by focusing on smaller joins, the likelihood of substantial estimation errors that could potentially lead to suboptimal plan selection is diminished [65, 63]. Furthermore, as joins occurring later in the query plan often process fewer data, owing to a higher number of applied predicates, suboptimal decisions made at a later stage are likely to have a less pronounced effect on the overall execution performance. Therefore, while exhaustive enumeration strategies demand consistent and accurate cardinality estimations across all join sizes, the reduced dependency on such estimates intrinsic to greedy algorithms can mitigate suboptimal query plans, enhancing the robustness of the selected query plans [42].

4.2 Spanning Tree-based Enumeration Strategies

A Minimum Spanning Tree (MST) is a subgraph of a graph that encompasses all the vertices while ensuring that the cumulative cost of its edges is the lowest possible. This problem is a fundamental topic in graph theory and is applied in various fields, including network design, clustering, and query optimization in databases. To find MST, traditional minimum spanning tree algorithms, such as Prim's and Kruskal's [18], operate under the assumption that edge weights are fixed – otherwise, optimality is not guaranteed. These algorithms maintain an invariant which ensures that the partial MST being constructed at any given time is always a subset of a full MST, upholding the integrity and optimality of the solution throughout the MST construction. In the context of query optimization, the problem statement is:

Optimal query plan is the minimum spanning tree, where the objective is to identify a set of edges spanning all vertices in the join graph while minimizing the total edge costs that are pre-defined and static during the spanning tree construction.

It is important to note that there can be multiple optimal query plans – several minimum spanning trees – within a specified search space. This case occurs in graphs that contain cycles or edges with the same weights.

Krishnamurthy et al. [58, 40] propose to utilize a minimum spanning tree algorithm to transform a cyclic graph into a tree. The join graph is formed of nodes and edges with pre-computed selectivities as weights. The cost of a spanning tree is computed as the product of its edge weights as opposed to the common approach which is the sum of the edge weights. Consequently, the selected minimum spanning tree is used in the IK-KBZ algorithm to generate a left-deep join order. Note that the optimality of the IK-KBZ algorithm is not guaranteed in the case of cyclic queries and is restricted to left-deep trees. Fegaras [27] proposed the GOO algorithm to find a query plan that uses Kruskal's algorithm on a join graph, assigning weights to the nodes as the size of the tables, and the edges as selectivities between two tables. As tables are joined, the corresponding nodes are merged, and the weight of the merged node corresponds to the size of the intermediate table. If the two nodes share a common edge, the new weight is adjusted as the product of the weights of the two edges. Notice that both approaches avoid cross-joins – the second largest rectangle in Figure 4.1. However, they do not ensure an optimal query plan due to their greedy decisions in constructing a spanning tree over a cyclic join graph.

A solution to assign static weights within the join graph fails to encapsulate the dynamic effects of prior subplan joins – multi-table correlation – thereby potentially leading to less optimal query plans. For example, in Figure 4.3, the edge $(mk - k)$ appears in all three spanning trees. Its weight is $1.1M$ in $\mathcal{P}_{opt}$ and $\mathcal{P}_{kru}$, and $0.2K$ in $\mathcal{P}_{pri}$. This is due to the different edge subsequences that precede $(mk - k)$ in the spanning trees. Consequently, the weight of an edge is derived from the edge subsequence – or query subplan – it is appended to. We cannot assign a static weight to an edge in the join graph and then sum up the weights for all the edges in the spanning tree to obtain an accurate cost of the query plan. This precludes the direct application of well-known spanning tree algorithms such as Prim and Kruskal to identify the query plan with minimum cost as they fall short in their capacity to efficiently accommodate dynamic changes in edge weights. Therefore, given the crucial roles of both the sequence in which edges are arranged and the variable characteristics of edge costs, we refine the problem statement as:

Optimal query plan is a spanning tree, where the objective is to identify an "ordered sequence" of edges spanning all vertices in the join graph while minimizing the total edge costs that are "dynamically changing as tables are successively joined".

In Section 4.2.1, we initially demonstrate how a single plan can be derived from a sequence of edges. In Section 4.2.2, we detail the exhaustive enumeration of the entire spanning tree search space which identifies an optimal query plan. Section 4.2.3 presents two classical spanning tree algorithms, Prim's and Kruskal's each adapted to cater to dynamic edge weight changes and enumerating a single query plan. Finally, in Section 4.2.4, we introduce the Ensemble Spanning Tree Enumeration (ESTE) strategy designed to amplify the performance of Prim's and Kruskal's by capturing a broader search space and enumerating more than a single query plan.

4.2.1 Spanning Tree-based Plan Generation

In Figure 4.3, we illustrate three spanning trees $\mathcal{P}_{opt}$, $\mathcal{P}_{kru}$ and $\mathcal{P}_{pri}$. At the leaf nodes, we show the size of the base tables. For each internal node, we compute join cost $\mathcal{C}$ as in Equaiton 2.4 computed based on true cardinalities Y of base and intermediate tables. For the join operator selection, we compute the cost of hash join HJ, and index nested-loop join INL if one of the child nodes is a base table and assume the presence of indexes on leaf nodes. The join operator with the minimum cost is selected at each intermediate node. Lastly, we compute the total cost $\mathcal{C}$ of all three spanning trees. Each spanning tree is generated from bottom to top following the ordered edge sequence from left to right: $\mathcal{P}_{opt} = \{(mk{-}k), (mk{-}mc), (mc{-}cn), (t{-}mk)\}$, $\mathcal{P}_{kru} = \{(mc{-}cn), (t{-}mc), (mk{-}k), (t{-}mk)\}$

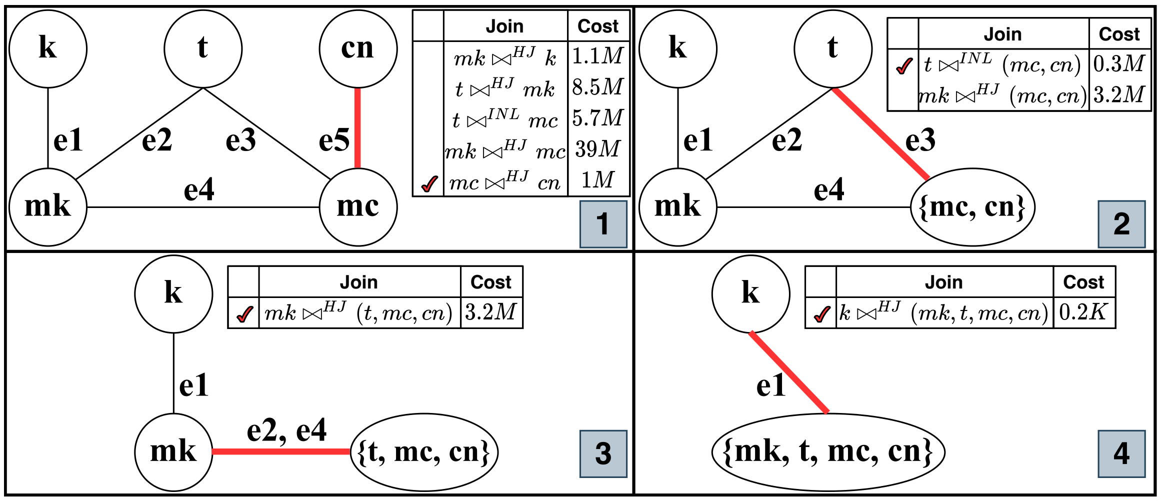
and $\mathcal{P}_{pri} = \{(mc - cn), (t - mc), (mk - mc), (mk - k)\}$. To illustrate generation and the cost computation of a spanning tree, we take $\mathcal{P}_{opt}$ as an example. The shape of the $\mathcal{P}_{opt}$ is a left-deep tree. The subplan $\mathcal{P}_{mk \bowtie k} = \{(mk - k)\}$ is a 2-way join having two children as base tables k and mk. The cost of hash and index nested-loop joins are $1.1M$ and $10M$, respectively. Thus, the hash join is selected as the join operator type for $\mathcal{P}_{mk \bowtie k}$. The size of the output table for $\mathcal{P}_{mk \bowtie k}$ is $42K$. The next larger subplan $\mathcal{P}_{mk \bowtie k \bowtie mc} = \{(mk - k), (mk - mc)\}$ is a 3-way join having two children as subtree $\mathcal{P}_{mk \bowtie k}$ and base table mc. The index nested-loop join is selected with the cost of $0.3M$ while the cost of hash join is $0.7M$. It is important to note the cost of the 3-way join plan $\mathcal{P}_{mk \bowtie k \bowtie mc}$ takes into account the edge subsequence – the cardinality of the contained 2-way join subplan $\mathcal{P}_{mk \bowtie k}$. The 4-way join $\mathcal{P}_{mk \bowtie k \bowtie mc \bowtie cn} = \{(mk - k), (mk - mc), (mc - cn)\}$ has two children as subtree $\mathcal{P}_{mk \bowtie k \bowtie mc}$ and base table cn. Their HJ and INL join costs are $0.2M$ and $0.3M$, respectively. The hash join is selected and the edge on the join graph is changed accordingly. Lastly, the final 5-way join $\mathcal{P}_{mk \bowtie k \bowtie mc \bowtie cn \bowtie t} = \{(mk - k), (mk - mc), (mc - cn), (t - mk)\}$ has two children as subtree $\mathcal{P}_{mk \bowtie k \bowtie mc \bowtie cn}$ and base table t. The indexed nested-loop join is selected as the final join operator – the costs are $HJ = 0.5M$ and $INL = 16K$, respectively. The remaining edge $(t - mc)$ creates a cycle. Thus the corresponding join predicate $t.id = mc.movie\_id$ can be evaluated with the join $(t - mk)$ – join predicate $t.id = mk.movie\_id$.

4.2.2 Exhaustive Spanning Tree-based Enumeration

Due to the dynamic alteration of edge weights resulting from ongoing joins, standard minimum spanning tree algorithms are not guaranteed to yield optimal query plans. Hence, to uncover an optimal query plan, it becomes imperative to enumerate all possible spanning trees over the join graph and select the one with the minimum cost. Unlike enumerating over nodes, adopting an edge-centric enumeration method results in query plans that do not include cross-joins. This is because the edge enumeration leverages the inherent structure of the preset connections within the pre-defined join graph, thereby naturally excluding the possibility of cross-joins – the second largest rectangle in Figure 4.1. Edges that would generate a cycle within spanning trees are instead incorporated as supplementary filter predicates in the query plan. This approach ensures that the acyclic property of spanning trees is maintained throughout the process. Rather than employing an exhaustive arrangement-based enumeration strategy with subsequent cost assessments, it is more beneficial to use a recursive algorithm that leverages backtracking. This approach allows for more efficient exploration techniques of the search space, such as cost-based pruning and graph-based dynamic programming, reducing the computational burden and improving practical feasibility. Every feasible edge has to be recursively considered at any step of the traversal and if the cost of the current subplan surpasses the current minimal cost, it allows for a backtrack. This process aids in circumventing parts of the search space that are not necessary to explore, thus enhancing efficiency by preventing wasteful computations. Employing graph-based dynamic programming systematically builds upon subplans to avoid redundant computations and adapt to the dynamic weight changes in edge costs. Even though these may reduce the number of plans considered, it is guaranteed that the spanning tree with minimum cost is found.

4.2.3 Heuristic Spanning Tree-based Enumeration

The intrinsic ability of edge-based enumeration to preclude cross-joins in the search space of the spanning tree presents a substantial advantage. However, even with the above optimizations in place, exhaustive edge enumeration can still be computationally prohibitive, especially when dealing with large queries. Hence, the natural step forward is to modify the traditional minimum spanning tree algorithms, such as Prim's and Kruskal's, to accommodate the unique characteristics of a join graph, particularly the dynamic nature of edge weight changes. In addition to their computational efficiency due to polynomial time complexity, these greedy algorithms provide tangible advantages when dealing with cardinality estimates. Their bottom-up approach in building the query plan, which emphasizes smaller join size estimates, is an immensely beneficial feature. Given that later stages of query processing involve fewer data, and during query optimization, large join size estimates are generally far less accurate than the smaller joins [65]. These aspects make a greedy approach pragmatic, balancing the need for accuracy and practical efficiency. We show Prim's and Kruskal's algorithms that take into account the order of the edge sequence and the dynamic change of edge costs. While they may not provide the optimal query plan due to their inherent greedy nature, these algorithms deliver spanning trees with low costs and have the advantage of fast search times.



	Join	Cost
	$mk \bowtie^{HJ} k$	$1.1M$
	$t \bowtie^{HJ} mk$	$8.5M$
	$t \bowtie^{INL} mc$	$5.7M$
	$mk \bowtie^{HJ} mc$	$39M$
✓	$mc \bowtie^{HJ} cn$	$1M$

	Join	Cost
✓	$t \bowtie^{INL} (mc, cn)$	$0.3M$
	$mk \bowtie^{HJ} (mc, cn)$	$3.2M$

	Join	Cost
✓	$mk \bowtie^{HJ} (t, mc, cn)$	$3.2M$

	Join	Cost
✓	$k \bowtie^{HJ} (mk, t, mc, cn)$	$0.2K$

Figure 4.4: Step-by-step illustration of Prim's plan enumeration algorithm on the join graph of query 2a.

4.2.3.1 Prim's Plan Enumeration

Within the search space of spanning trees, where cross-joins are naturally eliminated, Prim's algorithm functions within the left-deep subspace which further reduces the search space. Even further, due to its greedy decision-making nature, its exploration of this space is only partial, resulting in the enumeration of just one complete query plan in the form of a left-deep tree. At each iteration, Prim's algorithm maintains a single main component, which progressively builds up the final query plan. In Figure 4.4, we demonstrate an adaptation of Prim's algorithm that takes into account changes in edge weights to select an ordered sequence of edges and subsequently generate the query plan. First, an edge of a 2-way join with the minimum cost is selected from the list of all edges corresponding to 2-way joins shown in the table in step 1. The list of edges is not necessarily sorted as the composition of this list is heavily reliant on previous join selection and their weights are updated at each iteration. For each join, the optimal join operator is selected as a hash join HJ or indexed nested-loop INL based on the cost computed as in Equation 2.4. Edge $e5$, subplan $\mathcal{P}_{mc \bowtie cn} = \{(mc - cn)\}$, is selected as 2-way hash join with the minimum cost $\mathcal{C}(\mathcal{P}_{mc \bowtie cn}, Y) = 1M$. The nodes mc and cn form a combined component $\{mc, cn\}$ shown in step 2. The weights of the edges adjacent to nodes mc and cn are updated – edges $e3$ and $e4$. Next, an edge of a 3-way join with the minimum cost is selected from the list of all edges corresponding 3-way joins and adjacent to the main component $\{mc, cn\}$. Edge

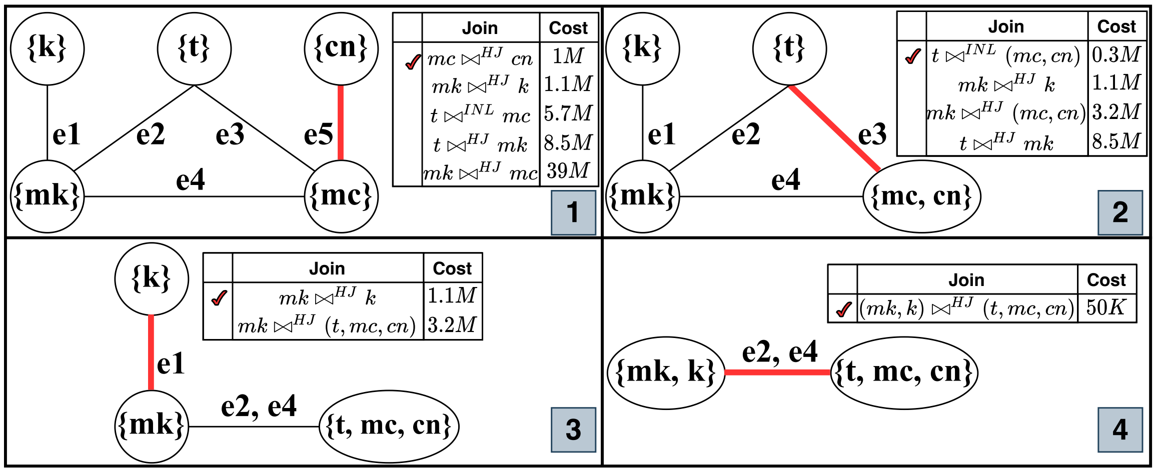
$e3$, subplan $\mathcal{P}_{mc \ltimes cn \ltimes t} = \{(mc - cn), (t - mc)\}$, is selected as 3-way indexed nested-loop join with the minimum cost $\mathcal{C}(\mathcal{P}_{mc \ltimes cn \ltimes t}, Y) = 0.3M$. The component $\{mc, cn\}$ and node t form a combined component $\{mc, cn, t\}$ shown at step 3. In step 3, the weights of the edges $e2$ and $e4$ adjacent to $\{mc, cn, t\}$ are updated. Notice that both edges of a 4-way join have the same cost of $3.2M$ as a hash join, and either of the edges becomes a cyclic edge. Thus, we randomly select an edge, $e2$ or $e4$, and transform the other one into a filter predicate. After building the subplan $\mathcal{P}_{mc \ltimes cn \ltimes t \ltimes mk} = \{(mc - cn), (t - mc), (mk - mc)\}$, the cost of the remaining edge $e1$ adjacent to the main component $\{mc, cn, t, mk\}$ is updated and selected as a 5-way hash join. The resulting plan $\mathcal{P}_{mc \ltimes cn \ltimes t \ltimes mk \ltimes k} = \{(mc - cn), (t - mc), (mk - mc), (mk - k)\}$ spans all the nodes in the original join graph. In Figure 4.3, the rightmost tree $\mathcal{P}_{pri}$ represents the spanning tree generated based on Prim's plan enumeration.

Algorithm 2 Prim's Algorithm for Plan Enumeration

```
Input: $\mathcal{S}_{pri} \leftarrow \emptyset$ // main component
       $\mathcal{S} \leftarrow \{v \mid v \in V\}$ // complement component
       $\mathcal{E} \leftarrow \{e : \mathcal{C}(\mathcal{P}_e) \mid e \in E\}$ // subplans and costs
Output: $\mathcal{P}_{pri} \leftarrow []$ // query plan
 1: while $\mathcal{E} \neq \emptyset$ do
 2:     // find a join with the minimum cost
 3:     find $e \mid \mathcal{C}(\mathcal{P}_e) < \mathcal{C}(\mathcal{P}_{e^*}), \forall e^* \in \mathcal{E}$
 4:     $\mathcal{E} \leftarrow \mathcal{E} \,/\, e$
 5:     add $e$ to $\mathcal{P}_{pri}$
 6:     // either of $v1_e$ and $v2_e$ is a base table
 7:     $\mathcal{S}_{pri} \leftarrow \mathcal{S}_{pri} \cup \{v1_e, v2_e : |v_e| = 1\}$
 8:     $\mathcal{S} \leftarrow \mathcal{S} \,/\, \{v1_e, v2_e : |v_e| = 1\}$
 9:     for each $e' \in \mathcal{E}$ do
10:         if $v1_{e'} \notin \mathcal{S}_{pri} \wedge v2_{e'} \notin \mathcal{S}_{pri}$ then
11:             // remove edges nonadjacent to $\mathcal{S}_{pri}$
12:             $\mathcal{E} \leftarrow \mathcal{E} \,/\, e'$
13:         else if $v1_{e'} \in \mathcal{S}_{pri} \wedge v2_{e'} \in \mathcal{S}_{pri}$ then // cyclic edge
14:             add $e'$ to $\mathcal{P}_{pri}$ as a filter predicate
15:         else // update edges adjacent to $\mathcal{S}_{pri}$ and costs
16:             $\mathcal{E} \leftarrow e' \cup \mathcal{S}_{pri} : \mathcal{C}(\mathcal{P}_{e' \cup \mathcal{S}_{pri}})$
```

The pseudo-code of Prim's plan enumeration algorithm is given in Algorithm 2. It starts with an empty set $\mathcal{S}_{pri}$ and its complement $\mathcal{S}$ of size $|V|$ containing all the nodes. The list of 2-way join edges along with their pre-computed costs are given in $\mathcal{E}$. Until the set $\mathcal{E}$ becomes an empty set, at each iteration (lines 1-18), an edge e of the current join size is selected that is adjacent to the main component $\mathcal{S}_{pri}$ and has the minimum cost (lines

2-5). Initially, all 2-way join edges are considered as adjacent to $\mathcal{S}_{pri}$. It is not necessary to build a min-heap data structure since the list of edges must be adjacent to $\mathcal{S}_{pri}$ and their weights are updated at each iteration. In lines 7-9, the components $\mathcal{S}_{pri}$ and $\mathcal{S}$ are updated with respect to the selected edge e. In lines 11-18, it prepares the edges adjacent to $\mathcal{S}_{pri}$ with updated costs (lines 17-18) for the next iteration. In the case of cyclic edge, it transforms the edge to a filter predicate (lines 15-16). In lines 12-14, the edges that are no longer adjacent to $\mathcal{S}_{pri}$ are dropped. The algorithm executes in polynomial time running in $\mathcal{O}\left(|E|^2\right)$ and allocating $\mathcal{O}\left(|V|+|E|\right)$ memory. Notice that lines 2-5 can be moved into lines 11-18. For demonstration purposes, we split them into two separate code blocks.



	Join	Cost
✓	$mc \bowtie^{HJ} cn$	$1M$
	$mk \bowtie^{HJ} k$	$1.1M$
	$t \bowtie^{INL} mc$	$5.7M$
	$t \bowtie^{HJ} mk$	$8.5M$
	$mk \bowtie^{HJ} mc$	$39M$

	Join	Cost
✓	$t \bowtie^{INL} (mc, cn)$	$0.3M$
	$mk \bowtie^{HJ} k$	$1.1M$
	$mk \bowtie^{HJ} (mc, cn)$	$3.2M$
	$t \bowtie^{HJ} mk$	$8.5M$

	Join	Cost
✓	$mk \bowtie^{HJ} k$	$1.1M$
	$mk \bowtie^{HJ} (t, mc, cn)$	$3.2M$

	Join	Cost
✓	$(mk, k) \bowtie^{HJ} (t, mc, cn)$	$50K$

Figure 4.5: Step-by-step illustration of Kruskal's plan enumeration algorithm on the join graph of query 2a.

4.2.3.2 Kruskal's Plan Enumeration

Kruskal's algorithm is an enhancement of Prim's approach, operating over the bushy and left-deep subspaces. Due to its inherently greedy character, this method also only partially explores the subspace, resulting in the enumeration of a single plan. This plan may exhibit either a bushy or left-deep tree structure. During each iteration, Kruskal's algorithm combines two components – each of potentially differing sizes – in a bottom-up fashion. The merging of two subcomponents each larger than one is indicative of a bushy tree structure. The inclusion of bushy tree shapes in the consideration set extends the opportunity to identify query plans that potentially exhibit a lower cost or even optimal plans. In Figure 4.5, we show the updated variation of Kruskal's algorithm that takes into account changes in edge weights to select an ordered sequence of edges and subsequently generate the query plan. As in Prim's, first, an edge of a 2-way join with the minimum cost is selected from the list of all edges corresponding to 2-way joins shown in the table at step 1. Each node is a separate component, and for each join, the optimal join operator is selected as a hash join HJ or indexed nested-loop INL based on the cost computed as in Equation 2.4. The list of edges is sorted based on their costs. The first edge, $e5$ corresponding to subplan $\mathcal{P}_{mc \bowtie cn} = \{(mc - cn)\}$, is selected as 2-way hash join with the minimum cost $\mathcal{C}(\mathcal{P}_{mc \bowtie cn}, Y) = 1M$. The node components $\{mc\}$ and $\{cn\}$ form a combined component $\{mc, cn\}$ shown at step 2. As in Prim's, the weights of the edges $e3$ and $e4$ adjacent to $\{mc\}$ and $\{cn\}$ are updated. Additionally, we keep the edges $e1$ and $e2$ that are not adjacent to $\{mc\}$ and $\{cn\}$ with the same costs. The selection of these edges leads to bushy trees. In the table at step 2, we list all four edges that potentially can be selected in the next join. Next, an edge of a 3-way or 2-way join with the minimum cost is selected from the list of all edges corresponding 3-way joins and adjacent to the component $\{mc, cn\}$, as well as nonadjacent edges corresponding 2-way joins. Edge $e3$, subplan $\mathcal{P}_{mc \bowtie cn \bowtie t} = \{(mc - cn), (t - mc)\}$, is selected as 3-way indexed nested-loop join with the minimum cost $\mathcal{C}(\mathcal{P}_{mc \bowtie cn \bowtie t}, Y) = 0.3M$. The components $\{mc, cn\}$ and $\{t\}$ form a combined component $\{mc, cn, t\}$ shown at step 3. In step 3, the weights of the edges $e2$ and $e4$ adjacent to $\{mc, cn, t\}$ are updated. As in Prim's, both edges of a 4-way join have the same cost of $3.2M$ as a hash join, and either of the edges becomes a cyclic edge. The 2-way join $\mathcal{P}_{t \bowtie mk}$ is dropped from the list since it became adjacent to the component $\{mc, cn, t\}$ and cannot form a separate subtree. However, the edge $e1$ is still kept as it can be selected to form a bushy structure. Since the $e1$ has a smaller cost of $1.1M$ than edge $e2$ and $e4$ with the cost of $3.2M$, it is selected as a 2-way hash join forming a separate subtree – subplan $\mathcal{P}_{mk \bowtie k} = \{(mk - k)\}$. The node components $\{mk\}$ and $\{k\}$ form a combined component $\{mk, k\}$ shown at step 4. Lastly, the cost of the remaining edges $e2$ and $e4$ adjacent to both components $\{mc, cn, t, mk\}$ and $\{mk, k\}$ are updated. Both edges have the same cost of $50K$ as a hash join, and either of the edges becomes a cyclic edge. We randomly select an edge, $e2$ or $e4$, and transform the other one into a filter predicate. The selected 5-way hash join is a subplan $\mathcal{P}_{mc \bowtie cn \bowtie t \bowtie mk \bowtie k} = \{(mc - cn), (t - mc), (mk - mk), (t - mk)\}$ which has a bushy structure. The resulting plan spans all the nodes in the original join graph. In Figure 4.3, the middle tree $\mathcal{P}_{kru}$ represents the spanning tree generated based on Kruskal's enumeration algorithm.

The pseudo-code of Kruskal's plan enumeration algorithm is given in Algorithm 3. It starts with the set $\mathcal{S}$ of size $|V|$ containing all the nodes as separate components. The list of 2-way join edges as values along with their pre-computed costs as keys stored in a min-heap data structure $\mathcal{E}$. Until the set $\mathcal{E}$ becomes an empty set, at each iteration (lines 1-24), the edge e which has the minimum cost is selected – the first element extracted from the min-heap (lines 2-3). In Kruskal's algorithm, it is efficient to keep a sorted list of edges since some of the joins of lower size are kept which can be selected to build a separate subtree. Hence, we maintain the min-heap data structure to keep edges sorted by cost. In lines 5-7, the components to which the two nodes $v1$ and $v2$ of the current edge e belong are found. Every edge extracted from the min-heap either forms a cycle (lines 9-10) or merges two components (lines 11-24). In the case of cyclic edge, it transforms the edge to a filter predicate. Otherwise, the two components are merged (lines 12-14). In lines 16-24, the min-heap is updated with respect to the selected edge e. For demonstration purposes, in lines 17-18, we show the case where the edges that potentially can create a separate subtree are kept with the same costs. Otherwise, an edge that no longer can form a separate subtree is dropped (lines 20-22). In the case of adjacent and valid join, the costs of the corresponding joins are updated (lines 23-24). The algorithm also executes in polynomial time running in $\mathcal{O}\left(|E|^2 \times log(|E|)\right)$ and allocating $\mathcal{O}\left(|V| + |E|\right)$ memory.

4.2.4 Ensemble Spanning Tree-based Enumeration

A significant drawback of greedy algorithms lies in their enumeration of a single full plan, which may result in being stuck in local optima and missing globally optimal plans. This characteristic can be observed in Figure 4.6. In Figure 4.6, we plot all spanning trees generated from the join graph, sorted by their cost excluding the cross-joins. Two optimal plans are marked in red triangles located in the bottom left corner. The left-deep and bushy spanning trees are indicated as blue square and green diamond, respectively. The lowest rectangle shows near-optimal plans which have cost differences of at most 1.31. In the top right corner, the worst query plans are shown inside the upper rectangle – cost differences of at least 27.9. Notice these three clusters of spanning trees are characterized by their highest join costs. In the figure, both the plans generated by Prim's and Kruskal's algorithms are situated in their respective clusters of spanning trees. Each of these clusters is defined by its highest join cost, signifying their confinement to local optima. From the figure, we

Algorithm 3 Kruskal's Algorithm for Plan Enumeration

```
Input: $\mathcal{S} \leftarrow \{\{v\} \mid v \in V\}$ // $V$ components
       $\mathcal{E} \leftarrow \{\mathcal{C}(\mathcal{P}_e) : e \mid e \in E\}$ // min-heap
Output: $\mathcal{P}_{kru} \leftarrow []$ // query plan
 1: while $\mathcal{E} \neq \emptyset$ do
 2:     // extract the first join from the min-heap
 3:     extract $e$ from $\mathcal{E}$
 4:     // find components in $\mathcal{S}$
 5:     $l_e \leftarrow component(v1_e)$
 6:     $r_e \leftarrow component(v2_e)$
 7:     if $l_e = r_e$ then // cyclic edge
 8:         add $e$ to $\mathcal{P}_{kru}$ as a filter predicate
 9:     else// new component joining $l_e$ and $r_e$
10:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{l_e \cup r_e\}$
11:         $\mathcal{S} \leftarrow \mathcal{S} \;/\; \{l_e, r_e\}$
12:         add $e$ to $\mathcal{P}_{kru}$
13:         for each $e' \in \mathcal{E}$ do
14:             if $v1_{e'} \notin \{l_e \cup r_e\} \wedge v2_{e'} \notin \{l_e \cup r_e\}$ then
15:                 // keep nonadjacent edges with the same costs
16:             else
17:                 if invalid edge then
18:                     // edges that cannot form a subtree
19:                     extract $e'$ from $\mathcal{E}$ // remove adjacent edges
20:                 else // update adjacent edges and costs
21:                     $\mathcal{E} \leftarrow \mathcal{C}(\mathcal{P}_{\{e' \cup l_e \cup r_e\}}) : e'$
```

can see limiting the enumeration to a single full query plan is not effective when exploring the search space, as it bears the risk of getting trapped in local optima. Hence, given the limitations of enumerating a single plan and the prohibitive computational expense of exhaustive enumeration, the logical approach is to enumerate more than one query plan. This intermediate strategy can help balance the search for optimality with computational efficiency. Enumerating more than one query plan increases the robustness of the search process and uncovers more parts of the search space.

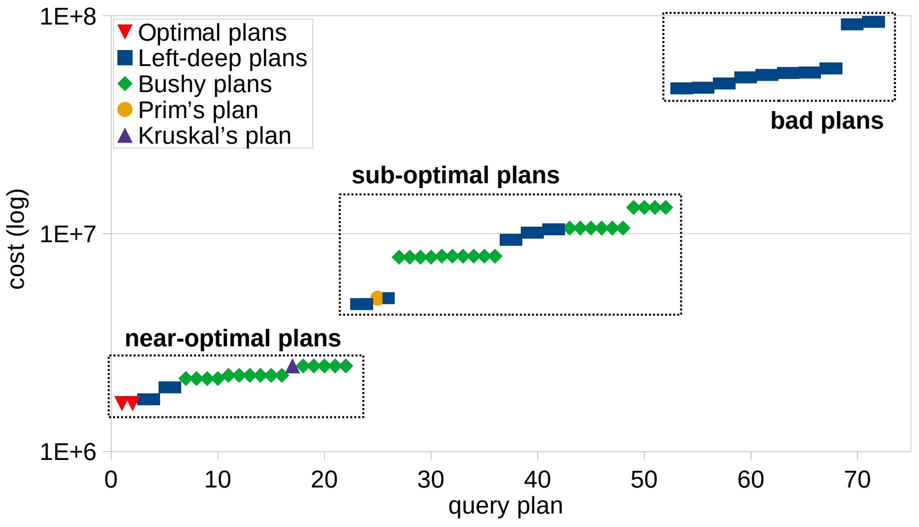


Figure 4.6: Query plans of query 2a sorted by cost and colored by tree shape. Query plans selected by Prim's and Kruskal's algorithms are shown in orange circles and pink triangles pointed up, respectively.

A direct method to facilitate the exploration of more query plans is to employ randomized approaches[104, 43, 106, 88, 107, 44]. In these approaches, the enumeration could randomly begin from a particular edge in the join graph. This kind of stochastic approach could increase the diversity of the examined query plans and help escape local optima. However, the optimizer performance becomes less predictable and interpretable due to the random behavior of the search space exploration. In this work, we take advantage of minimum spanning tree algorithms which also offer polynomial time complexity. Rather than starting from random edges, each spanning tree algorithm is initiated from every edge in the join graph [58, 77]. Not only would this approach preserve the interpretability of the process, but it would also enhance the robustness of the search, ensuring a more comprehensive coverage of the search space. Unlike prior work [58, 77], we take one step further, we propose the Ensemble Spanning Tree Enumeration (ESTE) strategy – the incorporation of a wider variety of spanning tree algorithms to further diversify the set of query plans generated. This strategy maintains polynomial-time complexity while enhancing the robustness of the plan enumeration.

The complexity of the Prim's and Kruskal's discussed in Section 4.2.3 becomes $\mathcal{O}\left(|E|^3\right)$ and $\mathcal{O}\left(|E|^3 \times log(|E|)\right)$, respectively. This intermediate enumeration strategy creates an ensemble of spanning trees broadening the search coverage without escalating the computational costs. In terms of space coverage based on our example *query 2a*, together Prim's and Kruskal's single plan enumerations reach 10 or 71.4% out of all 14 distinct subplans, computed 9 or 28.1% out of all 32 distinct join costs, and 2 or 2.8% different plans enumerated out of all possible 72 query plans excluding the cross joins. In ESTE, 14 or 100% subplans, 26 or 81.2% join costs, and 8 or 11.1% different plans are covered. By making minor trade-offs in execution time, ESTE is able to explore larger portions of the search space. In our experimental results, we demonstrate that this approach yields benefits in terms of discovering more efficient query plans thus reducing the plan costs.

4.3 Empirical Evaluatiion

We perform the experimental study over the comprehensive JOB benchmark [65, 63] to evaluate the performance of the spanning tree-based enumeration algorithms. We set the spanning tree-based enumeration algorithms against one of the most outstanding, if not the best, heuristic enumeration algorithms – GOO [27]. Our decision is based on previous evaluations which have consistently demonstrated GOO's superior performance across various configurations, graph topologies, and benchmarks among other proposed heuristic enumeration algorithms [65, 63, 86, 31]. Employing GOO as a comparative baseline is expected to provide a solid starting point for our performance assessment. Throughout the section, we provided in-depth micro analysis over the example query *2a*. In Section 4.3.2, we perform a macro analysis over all JOB queries and summarize the overall workload performance in Section 4.3.3. Our evaluation addresses the following questions:

- Assess the performance (plan cost, optimization and execution times) of the spanning tree-based strategy, and compare it against exhaustive and heuristic strategies.

- Assess whether the ESTE strategy enhances the performance by devoting additional time to the examination of a broader portion of the search space.
- Assess the response of the spanning tree-based enumeration algorithms to the errors in cardinality estimates. This will help us understand how these algorithms adapt to and perform in the presence of cardinality estimation errors.

4.3.1 Experimental Setup

4.3.1.1 Dataset & Query Workload

We perform the experiments on the IMDB dataset [12], which has been used extensively to evaluate query optimizers [63, 65, 86, 16, 90, 31] and has become a de-facto standard. The JOB benchmark [92] defines 113 queries – grouped into 33 families – over the IMDB dataset. These queries vary significantly in their complexity. The simplest queries have 4 join predicates with the largest join size of 4, while the most complex queries have 28 join predicates with the largest join size of 17. The workload encompasses a diverse range of graph topologies which are incorporated within individual join graphs. This variability manifests itself in execution times that are highly different. To compensate for this, we split the queries into three complexity groups and examine each group separately. These complexity groups are based on the number of join predicates: simple queries with 4-9 joins, moderate queries with 10-19 joins, and complex queries with 20-28 joins, respectively.

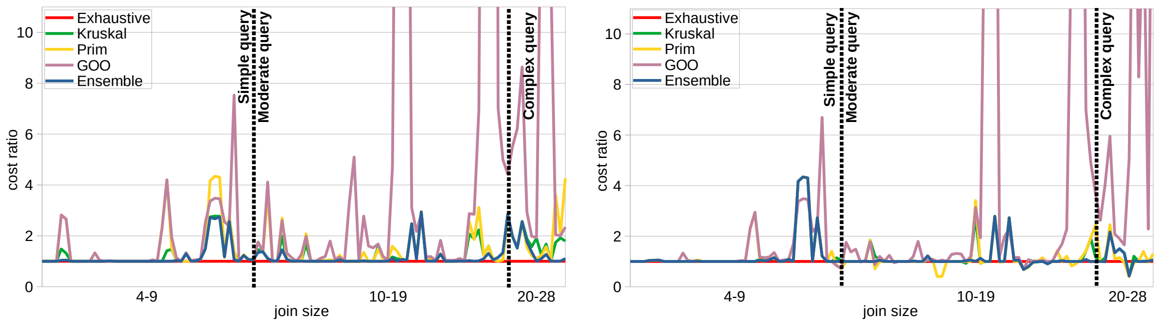
4.3.1.2 Database System & Hardware

For cardinality estimations, we chose a well-known database system PostgreSQL (version 15.1) [88]. PostgreSQL uses a large variety of statistics on base tables. Any join estimate is computed by combining the statistics into simple arithmetic formulas that make general assumptions on uniformity, inclusion, and independence. We run the subplans of the JOB queries in PostgreSQL to collect their join cardinality estimations as well as the true cardinalities. We run PostgreSQL with default configurations, and set operator and buffer size to 16GB and 64GB, respectively. For measuring execution times, we utilize *pg_hint* to force the cost function from Equation 2.4 along with physical operators and force the join order. All the experiments run on an Ubuntu 22.04 LTS machine with 56 CPU cores (Intel Xeon E5-2660), 256GB RAM, and an NVIDIA Tesla K80 GPU.

4.3.1.3 Scope, Methodology & Implementation

To quantitatively assess the quality of a query plan, we use the cost function $\mathcal{C}$ from Equation 2.4 in all the algorithms evaluated in this section including GOO. This cost function is designed to suit an in-memory setting, and it has been demonstrated that it effectively correlates with query runtime [63, 65]. Despite this, the cost function can be extended to cater to a disk-based setting, and to account for other types of join algorithms such as merge-join.

In this study, we did not include heuristics with a randomized flavor, as the ESTE initiates from every edge, potentially resulting in superior query plans while still preserving interpretability. While randomized algorithms might potentially achieve faster runtime, if they locate better query plans among query plans collected in $\mathcal{O}\left(|E|^3 \times log(|E|)\right)$, maintaining interpretability and unpredictable performance may become challenging.



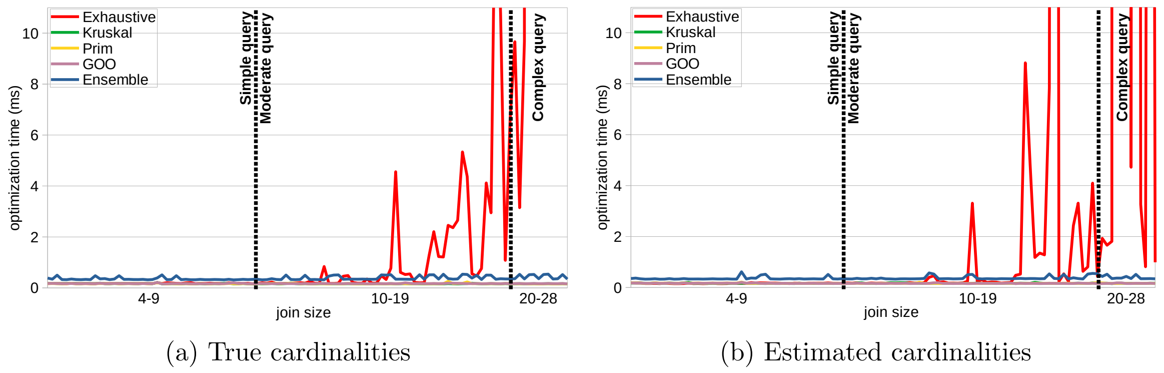
(a) True cardinalities (b) Estimated cardinalities

Figure 4.7: Cost ratio of selected JOB query plans against the optimal query plan costs using true join costs.

4.3.2 Macro Evaluation

4.3.2.1 Evaluation based on True Cardinality

In Figures 4.7a and 4.7b, we evaluate plan quality, in terms of their costs, selected by four heuristic enumeration algorithms – Prim's, Kruskal's, ESTE and GOO – compared against the exhaustive enumeration. The y-axis shows the cost ratio of the heuristic plan and the optimal plan selected by the exhaustive enumeration. The x-axis shows the 113 workload queries grouped – by dotted vertical lines – according to their join complexity as described in Section 4.3.1. The enumeration algorithms are shown as colored solid lines: exhaustive as red, Kruskal's as green, Prim's as yellow, ESTE as blue, and GOO as light purple. In Figure 4.7a, we compare the costs of the query plans selected by these enumeration algorithms computed using true cardinalities. From the figure, we observe GOO selects relatively worse query plans. Interestingly, in GOO, we observe that the join cardinality costs can reach saturation rapidly leading to zero tuples which makes it difficult to compare different query plans. Prim's is the worst among spanning tree-based enumeration algorithms due to the search in the limited left-deep search space. However, in certain instances, the algorithm chooses superior plans compared to Kruskal's. Prim's unique path of search space exploration contributes to enhancing the performance of ESTE. Regardless, ESTE, in certain cases, chooses other plans that have higher costs due to join options with equal costs. On the other hand, Kruskal's tends to perform noticeably better and ESTE enhances the performance of both Prim's and Kruskal's chosen query plans. As anticipated, all the enumeration algorithms begin to experience a drop in efficiency as the complexity of the queries increases. In Section 4.3.3, we show their absolute performance numbers based on the entire workload.

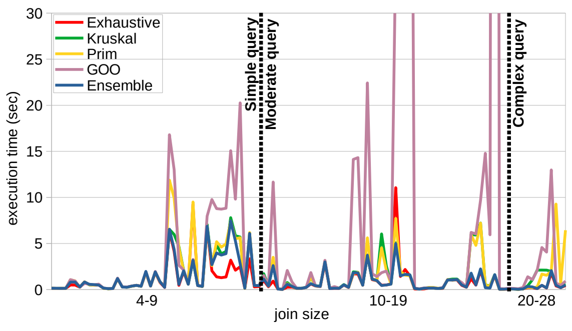


(a) True cardinalities

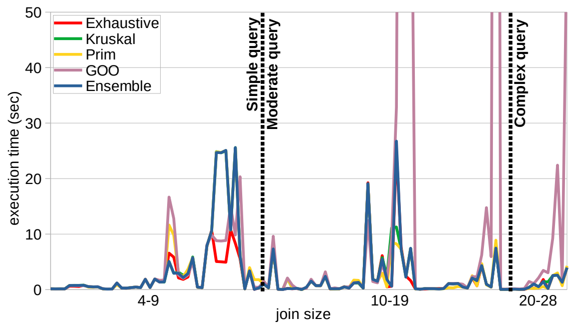
(b) Estimated cardinalities

Figure 4.8: Optimization time (ms) of enumeration algorithms for all JOB query plans.

In Figures 4.8a and 4.8b, we evaluate the optimization times of all five enumeration algorithms. In Figure 4.8a, the optimization time of the exhaustive enumeration becomes significantly high as the query complexity increases. Because of this, we were not able to get query plans for family 29 in JOB and further applying exhaustive enumeration for complex queries is not suitable. This confirms, once again, that exhaustive enumeration is not an option in dealing with complex queries. ESTE pays minimal extra time – maximum 0.54ms – in optimization for more robust behavior in selecting high-quality query plans. It is considered GOO to be one of the fastest heuristic enumeration algorithms. Interestingly, Prim's and Kruskal's exhibit very close optimization time to GOO – all three show optimization time around 0.2ms. All spanning tree-based and GOO enumerations maintain a stable optimization time across different query complexities.



(a) True cardinalities



(b) Estimated cardinalities

Figure 4.9: Execution time (sec) of all JOB query plans.

Figures 4.9a and 4.9b present the execution times for all five enumeration algorithms. In Figure 4.9a, it is clear that GOO's suboptimal plans indeed result in longer execution time. Expectedly, the optimal plans selected by the exhaustive enumeration exhibit the best performance except for a small number of exceptions. This is due to inaccuracies in the cost function. Upon examining these cases, we observe that indexed-loop join operators are favored along with large tables because of their presence of indexes. This, in turn, generates a large size intermediate table causing large latencies in the later stages of execution. In contrast, Prim's and Kurskal's plans perform significantly better compared to GOO's plans, delivering efficient execution times. In addition, ESTE brings more consistency and better performance by paying minimal extra time in optimization time.

4.3.2.2 Evaluation based on Estimated Cardinality

In Figure 4.7b, we compare the costs of the plans computed using PostgreSQL estimated cardinalities, across different enumeration algorithms. Expectedly, the enumeration algorithms perform relatively worse in the presence of cardinality estimation errors. Although their cost ratio decreased (lower ratio on the y-axis), these small ratios indicate the selection of more similar plans which all can be potentially suboptimal. As the join size increases, the error propagates to the larger join size estimations resulting in less reliable estimates for larger joins. This can lead to unpredictable behavior, particularly because of the existence of severe cardinality underestimations of suboptimal plans or overestimations in optimal plans misguide enumeration algorithms. This figure exhibits such behavior. We observe the cost ratios that are below one which indicate plans that are better than the estimated optimal plan selected by the exhaustive enumeration. Another noteworthy observation is that these estimation errors can cause cardinalities to reach saturation resulting in similar plan costs. This complicates the comparison between different plans and increases the risk of selecting suboptimal join order and physical operators. As with true cardinalities, GOO tends to select plans that have comparatively higher costs, especially in complex queries. In contrast, Prim's and Kruskal's algorithms reaffirm their dominance over GOO, even in the face of cardinality estimation errors. Meanwhile, ESTE enhances the performance by utilizing Prim's and Kruskal's, demonstrating more consistency in query plan quality.

In Figure 4.8b, we compare the optimization time of the enumeration algorithms when estimated cardinalities are utilized. The behavior of all algorithms remains similar to true cardinalities except for exhaustive enumeration. As previously mentioned, estimation errors can cause cardinalities to reach saturation resulting in similar plan costs, thus the benefit of the cost-based pruning technique within the exhaustive search diminishes. Consequently, the algorithm is forced to explore a significantly larger part of the search space, which results in a significant increase in optimization time.

Figure 4.9b presents the execution times of query plans selected by the enumeration algorithms when using estimated cardinalities. We first observe all four enumeration algorithms demonstrate a decline in performance compared to the plans selected using true cardinalities – larger scale on the y-axis. Just as with true cardinalities, GOO's inaccurate decisions lead to high execution time. All three spanning tree-based plans exhibit execution times close to the exhaustive enumeration. In several cases, ESTE chooses a different plan than Prim's and Kruskal's due to multiple join options with the same costs, leading to slightly slower execution times. To summarize, these results highlight the potential benefits of leveraging a combination of fast, spanning tree-based heuristics to find the balance between plan quality and computational constraints, especially in complex queries.

Query Complexity	Enumeration Strategy														
	Exhaustive			**Kruskal**			**Prim**			**ESTE**			**GOO**		
	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)
Simple	1	7.91	61	1.44	7.58	86	2.09	7.47	103	1.36	15.75	75	2.33	7.65	150
Moderate	1	83.19	46	1.16	8.63	70	1.45	8.89	71	1.05	20.83	44	6.86	8.94	848
Complex	1	4,171	4	1.62	1.94	11	2.48	1.92	22	1.09	5.28	5	5.66	2.0	29
TOTAL	1	4,262	111	1.34	18.15	167	1.86	18.28	196	1.24	41.85	124	4.1	18.59	1,026

Table 4.1: Overall cost ratio of true query plan costs, optimization (ms) and execution (ms) times on JOB.

4.3.3 Overall Evaluation

Tables 4.1 and 4.2 represent the overall performance of all five enumeration algorithms. The queries, grouped by complexity as described in Section 4.3.1, are shown in the rows while the performance of each enumeration algorithm is represented in columns. The last row provides the total workload performance for each enumeration algorithm, including its overall cost ratio, and optimization and execution times. Table 4.1, based on true cardinalities, GOO shows relatively worse results across all levels of query complexity. While Prim's and Kruskal's are significantly better at finding more efficient query plans, ESTE enhances this performance even more by probing a larger portion of the search space. While the optimization time of exhaustive enumeration is significantly high, ESTE spends at most $2.5X$ extra optimization time than Prim's, Kruskal's and GOO. As shown in Figures 4.7 and 4.9, this consistent extra time investment trades off for consistently better plans, thus paying off during the query execution. Consequently, in this table, we observe the execution time of ESTE is superior to Prim's and Kruskal's, and considerably better than GOO's. Interestingly, across all query complexities, the overall execution time of ESTE is noticeably close to the execution time of optimal plans obtained by the exhaustive enumeration.

Query Complexity	Enumeration Strategy														
	Exhaustive			**Kruskal**			**Prim**			**ESTE**			**GOO**		
	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)	cost ratio	opt (ms)	exec (sec)
Simple	1	7.41	102	1.38	7.38	178	1.72	7.55	195	1.38	15.99	177	1.91	7.45	152
Moderate	1	185	107	1.05	8.88	114	1.04	8.71	102	1.05	19.95	115	4.25	8.55	805
Complex	1	24,327	13	1.02	1.97	15	1.16	1.96	12	1.02	4.72	14	14.06	1.95	110
TOTAL	1	24,520	222	1.24	18.24	307	1.44	18.21	309	1.24	40.65	306	3.21	17.95	1,067

Table 4.2: Overall cost ratio of estimated query plan costs, optimization (ms) and execution (ms) times on JOB.

In Table 4.2, we compare the overall workload performances of the enumeration algorithms when operated using PostgreSQL estimated cardinalities. As depicted in Figures 4.7 and 4.9, the effects of cardinality estimation errors on plan costs and execution time are detrimental. In all enumeration algorithms, including the exhaustive, we observe a $2 - 2.5X$ times increase in runtime, while GOO's execution time remains high, similar to the results with true cardinalities shown in Table 4.1. Although cost ratios seem to decrease compared to true cardinalities, meaning more similarity to the plans obtained by the exhaustive enumeration, their absolute values significantly increased, leading to longer execution times. In optimization time, we observe a prohibitively large increase in the case of exhaustive enumeration due to the closeness of query plan costs affected by estimation errors – thus the cost-based pruning technique becomes less effective, forcing the enumeration algorithm to explore a much larger plan space. However, similar to true cardinalities, the optimization time of spanning tree-based and GOO algorithms exhibit consistent behavior across all query complexities. To summarize, spanning tree-based enumeration algorithms offer a superior strategy for finding efficient plans, outperforming GOO, one of the existing efficient heuristic enumeration algorithms. By spending minimal extra optimization time, ESTE further improves the performance over Prim's and Kruskal's enumeration algorithms. This indicates that despite the challenges brought by cardinality estimation errors, it is still possible to achieve comparatively efficient query plans.

4.4 Related Work

A simple and common way to represent a query is through an undirected join graph. Once constructing the join graph, the objective is to evaluate different join orders, each can be illustrated by a join tree. It is straightforward to see that a join order is a spanning tree connecting all the vertices. To our knowledge, the concept of finding a join order by finding a spanning tree was first introduced in [40, 58]. The IK-KBZ algorithm, proposed by Krishnamurthy et al. [58], is designed to find an optimal left-deep tree in an acyclic graph. The algorithm is well-suited for star queries, without resorting to cross-joins. As a heuristic extension to handle cyclic queries, the authors suggest finding a spanning tree with the minimum cost, where the cost is calculated as the multiplication of pre-defined selectivities of the edges. Then, the spanning tree is used as an input tree for the IK-KBZ algorithm [58]. Fegaras introduces the Greedy Operator Ordering (GOO) algorithm to greedily enumerate a single plan, which can either be a bushy or left-deep tree, on a join graph [27]. In the join graph, the weights on vertices and edges are cardinalities and selectivities, respectively. At each step, GOO joins two vertices with the minimum cost into a new vertex with updated cardinality weight. The cost is the cardinalities of both vertices multiplied by their selectivity without considering the cost of physical operators. Additionally, if the two vertices share a common adjacent vertex, both edges are merged into one and the weight is updated as the multiplication of both selectivities. The selectivity weights on the other edges remain the same.

The concept of initiating the plan search from each vertex in the join graph, as mentioned in [58, 77], assesses more than a single join order, leading to potentially more efficient query plans. Moerkotte proposes the GreedyJoinOrdering-3 to greedily select the next relation to join, while considering cross-joins [77]. The cost of joining tables is calculated with respect to the sequence of already joined tables thus maintaining a sorted list of weights does not offer an advantage. This is because the cost of a join operation is dynamic and depends on the previously formed joins, making static weights less relevant.

There are other ways of representing queries. Lee et al. [62] defined the join graph where each vertex is a join operator and an edge is present between two vertices if both join operators share a common base relation. The authors propose the Maximum Value Precedence (MVP) algorithm to find a directed spanning tree. However, a notable aspect of their join graph definition is that it increases the search space due to the inclusion of spanning trees that feature extra, unnecessary joins. These are referred to as ineffective spanning trees. Finding the shortest path on a directed plan graph was introduced in [83, 31]. The plan graph is a directed acyclic graph where a vertex represents a set of tables (subgraph) and an edge is established if one vertex is a subgraph of another. Due to different combinations of table sets, a plan graph typically has a larger number of vertices and edges compared to a join graph.

4.5 Conclusions

In this work, we frame query optimization as finding spanning trees with low costs. We calibrated our objective to find an ordered sequence of edges that spans all nodes in the join graph while minimizing total edge costs that dynamically change as tables are joined.

We adapted Prim's and Kruskal's algorithms to this new objective and leveraged them in ESTE. Owing to its polynomial-time complexity, ESTE can systematically explore wider and distinct areas of the search space. This rapid and comprehensive exploration of the search space results in superior query plan quality. While ESTE operates significantly faster than exhaustive strategy, it pushes the boundaries of heuristic strategy by significantly surpassing one of the most accomplished heuristic algorithms.

Chapter 5

Sub-optimal Join Order Identification with L1-error

Q-error is the standard metric for quantifying the error of individual cardinality estimates [81]. It is defined formally as the maximum quotient between the estimated and true cardinalities, thereby equally penalizing both overestimations and underestimations. Q-error is widely adopted in recent work on learning-based cardinality estimation methods [111, 52, 16, 54, 26] as a surrogate for the quality of query execution plans, which is measured by P-error –the ratio between the cost of the selected and optimal query plans [32]. However, the only theoretical result connecting the two is a worst-case upper-bound [81] stating that the cost of a query plan computed with estimates having a maximum Q-error of Q is at most Q^4 times larger than the cost of the plan computed with true cardinalities – which is assumed to be optimal. Given only estimated and true cardinalities, the bound provides a rough idea of how bad the worst possible query plan could be. Then, it is obvious that for small values of the Q-error, the gap between the worst and optimal plan is relatively small.

Since real query optimizers aim to identify the optimal plan – not the worst – we investigate how useful is the Q-error in assessing the optimality of a query plan based solely on cardinality estimates. For this, we compute the optimal plan using true cardinalities while the database plan is derived using PostgreSQL estimates. These two sets of values are fed into an exhaustive plan enumeration algorithm over a search space consisting of plans with arbitrary structure – including left-deep and bushy. The cost of a plan is computed using the cost function defined in Equation 2.4, which is introduced in [63]. This cost function considers both hash and index nested loop joins.

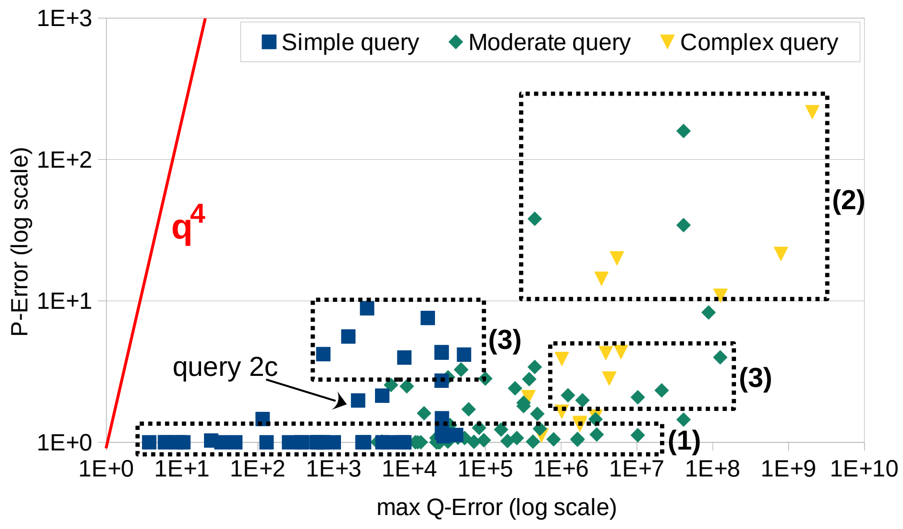


Figure 5.1: P-error and Q-error are computed for all 113 JOB queries. Queries are grouped by complexity: 45 Simple with 4-9 join predicates, 53 Moderate with 10-19 join predicates, and 15 Complex with 20-28 join predicates.

As shown in Figure 5.1, which displays P-error as a function of the Q-error for the queries in the JOB benchmark [65], there is no observable relationship between the two beyond the worst-case upper bound. First, a large number of simple and moderate queries have optimal plans – P-error close to 1 – even though they exhibit large variation on the Q-error – more than six orders of magnitude **(1)**. No matter if the Q-error is 10 or 10^7, an optimal plan can be selected using the same cardinality estimates. Second, we intuitively expect that Q-error is somewhat correlated with P-error – as the Q-error increases, so does the P-error. This happens only for a limited number of complex queries **(2)**. These cases imply the selection of sub-optimal plans with different join orders. Third, the results include queries for which the relationship between Q-error and P-error is reversed **(3)**. The P-error of a query is larger than the P-error of another query even though its Q-error is smaller. This type of inversion shows that the relationship between the two errors is not even monotonic. Finally, since the plans selected by the optimizer are far away from the worst-case, we argue that Q-error falls short as an indicator for the sub-optimality of query plans. Therefore, we need to consider alternative metrics that focus on the real query plans generated by an optimizer rather than the worst-case plan.

Problem. Given a query plan generated by an optimizer based on a set of cardinality estimates, our goal is to determine if it is sub-optimal. Even though we consider general cost functions that include physical operator details, our understanding of sub-optimality is mostly with respect to the join order. We define sub-optimality in terms of P-error: a plan with a cost at least c times larger than the cost of the plan computed with true cardinalities, where c is a user-defined parameter, is considered sub-optimal. Therefore, we make the implicit assumption that true cardinalities generate the optimal plan. Identical to Q-error computation, sub-optimal plans are identified outside of the runtime query optimization process and they require complete knowledge of true cardinalities and estimates.

High-level approach. We propose a learning-based method for the sub-optimal plan identification problem. We treat identification as binary classification, where plans with a cost at least c times larger than the optimal are considered sub-optimal – the other plans are optimal. The focus of our work is on finding the best features for the classifier – not on designing a new classifier. To this end, we employ standard decision trees. The classifier is trained on a workload of query plans correctly labeled and is expected to accurately predict sub-optimal plans from a testing dataset.

L1-error. The main contribution of this work is the design of the L1-error feature for sub-optimal plan classification. Similar to Q-error, L1-error requires complete knowledge of true cardinalities and estimates for all the sub-plans of a query plan. Unlike Q-error, which considers the estimates independently, L1-error is defined as the distance between the permutations – orders – corresponding to true cardinalities and estimates for all the sub-plans having the same size, i.e., the number of joins. Intuitively, the more different the two permutations are, the higher the chance the plan computed using estimates is significantly different than the optimal plan, thus, likely sub-optimal. The same observation is made in [81], where a plan is known to be optimal if the two permutations are identical. We move beyond this limited case and define a quantitative measure for the difference between permutations. Moreover, L1-error takes into account errors relative to the magnitude of cardinalities since larger cardinalities have a greater influence on plan optimality and, hence, their errors should incur higher penalties. L1-error also considers that small multi-way joins are more critical, with their cardinality estimates likely to be more accurate than larger joins [65, 90].

We summarize our main technical contributions as follows:

- We conduct an in-depth data-driven analysis of the impact of Q-error on cardinality estimation and query plan optimality. We study how Q-errors are distributed across different join sizes and what their impact on finding optimal query plans is. Moreover, we identify practical limitations of Q-error as a feature for sub-optimal plan classification.
- We introduce the L1-error feature for identifying sub-optimal query plans. L1-error is specifically tailored to assess how cardinality estimation errors impact plan enumeration algorithms. It is designed to bridge the cardinality estimation errors and enumeration algorithms, ultimately enhancing the interpretability of query optimizer performance.
- We apply L1-error as the single feature of a standard decision tree classifier and evaluate its accuracy in identifying sub-optimal query execution plans over four different benchmarks, including JOB [65, 63], JOB-light [54], JCCH [13], and DSB [23] benchmarks. Our experimental results confirm that L1-error is an accurate feature for identifying sub-optimal plans. This accuracy can be further improved by combining L1-error with Q-error into a composite feature that can be computed without overhead from the same data.

5.1 Q-Error

Query optimizers often fail to find an optimal plan because of errors and sub-optimal decisions made during the stages of cardinality estimation, cost function, and plan enumeration. In the cost function, inaccuracies in measuring the exact cost of each physical operator in the plan can lead the query optimizer to select a sub-optimal plan. Similarly, attempts to counter computational constraints in plan enumeration through heuristic enumeration and pruning the search space can mislead the query optimizer by excluding optimal plans from its scope of consideration. However, even under ideal conditions for the cost function and plan enumeration, unavoidable errors in cardinality estimations can jeopardize these stages [63, 65]. In this section, we delve into the widely used Q-error metric [81] that quantifies the errors in cardinality estimation. We discuss how this metric can be employed to evaluate the sub-optimality of a plan by providing a greater understanding of the extent of the estimation errors and their impacts on selecting the optimal plan.

5.1.1 Q-error for Cardinality Estimation

The Q-error metric was proposed as a means to quantify the degree of error in individual cardinality estimations [81]. It has since become a preferred choice for quantifying the accuracy of synopses [16, 90] and learning-based models [52, 54]. Furthermore, it is extensively employed in empirical studies to understand and improve the accuracy of these estimations, thus playing a significant role in query optimization [65, 111, 61].

The Q-error of an individual cardinality estimation is defined as:

$$q_i = \max\left(\frac{\hat{Y}_i}{Y_i}, \frac{Y_i}{\hat{Y}_i}\right) \tag{5.1}$$

where Y_i and $\hat{Y}_i$ are true and estimated cardinality of a single sub-plan. The Q-error value is in the range of $[1, +\infty)$. In the case of zero values in the denominator, the zeroes can be replaced by a small number – in this work, we use 10^{-4} for this purpose. The Q-error quantifies the deviation of the estimated cardinality $\hat{Y}_i$ from the true cardinality Y_i treating under- and over-estimation equally.

```
SELECT COUNT(*)
FROM company_name cn, keyword k, movie_keyword mk, movie_companies mc, title t
WHERE k.keyword = 'character-name-in-title' and cn.country_code = '[sm]'
  and mc.movie_id = mk.movie_id and cn.id = mc.company_id and k.id = mk.keyword_id and t.id = mk.movie_id and t.id = mc.movie_id
```

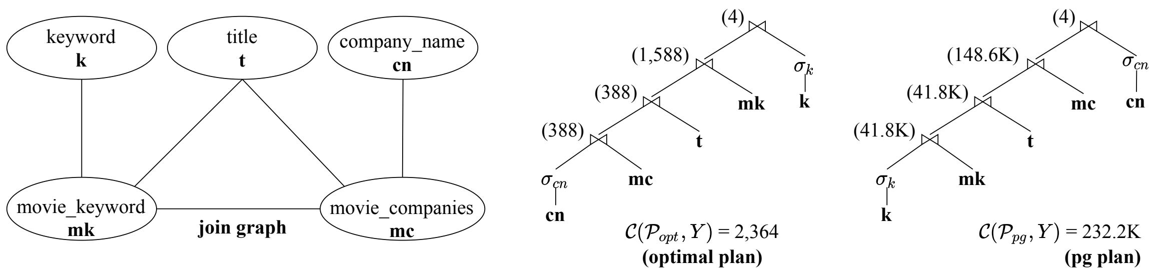
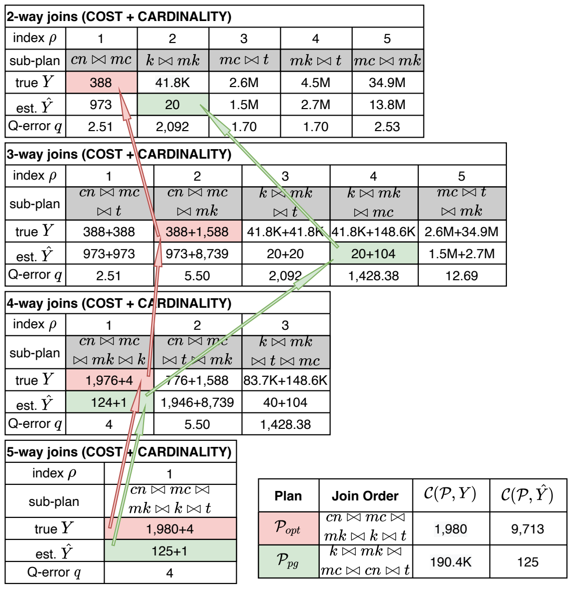


Figure 5.2: SQL statement for JOB query 2c, its corresponding join graph, and the query plans selected using true cardinalities Y (for the optimal plan $\mathcal{P}_{opt}$) and cardinality estimations $\hat{Y}$ (for the PostgreSQL plan $\mathcal{P}_{pg}$). The join sizes shown in parentheses are exact – not estimates – while the plan costs $\mathcal{C}(\mathcal{P}_{opt}, Y)$ and $\mathcal{C}(\mathcal{P}_{pg}, Y)$ are also computed using true cardinalities.

Consider query 2c from the JOB benchmark, which is depicted in Figure 5.2. This query joins five tables with five join predicates – including a triangle cycle among tables t, mk, and mc – and has two point selection predicates σ on tables cn and k. The figure also includes the join graph $G(V, E)$, in which every table is represented as a vertex v connected by edges e for the corresponding join predicates. For example, the join predicate $cn.id = mc.company\_id$ is represented as the edge between the cn and mc vertices of the join graph.



2-way joins (COST + CARDINALITY)					
index ρ	1	2	3	4	5
sub-plan	$cn \bowtie mc$	$k \bowtie mk$	$mc \bowtie t$	$mk \bowtie t$	$mc \bowtie mk$
true Y	388	41.8K	2.6M	4.5M	34.9M
est. $\hat{Y}$	973	20	1.5M	2.7M	13.8M
Q-error q	2.51	2,092	1.70	1.70	2.53

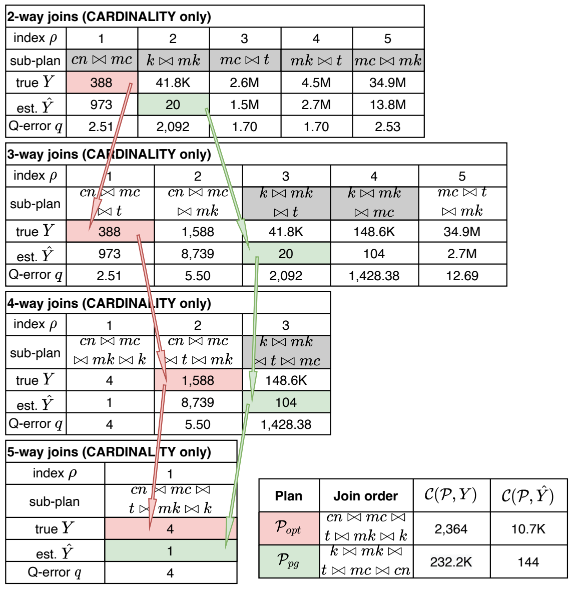
3-way joins (COST + CARDINALITY)					
index ρ	1	2	3	4	5
sub-plan	$cn \bowtie mc \bowtie t$	$cn \bowtie mc \bowtie mk$	$k \bowtie mk \bowtie t$	$k \bowtie mk \bowtie mc$	$mc \bowtie t \bowtie mk$
true Y	388+388	388+1,588	41.8K+41.8K	41.8K+148.6K	2.6M+34.9M
est. $\hat{Y}$	973+973	973+8,739	20+20	20+104	1.5M+2.7M
Q-error q	2.51	5.50	2,092	1,428.38	12.69

4-way joins (COST + CARDINALITY)			
index ρ	1	2	3
sub-plan	$cn \bowtie mc \bowtie mk \bowtie k$	$cn \bowtie mc \bowtie t \bowtie mk$	$k \bowtie mk \bowtie t \bowtie mc$
true Y	1,976+4	776+1,588	83.7K+148.6K
est. $\hat{Y}$	124+1	1,946+8,739	40+104
Q-error q	4	5.50	1,428.38

5-way joins (COST + CARDINALITY)	
index ρ	1
sub-plan	$cn \bowtie mc \bowtie mk \bowtie k \bowtie t$
true Y	1,980+4
est. $\hat{Y}$	125+1
Q-error q	4

Plan	Join Order	$\mathcal{C}(\mathcal{P}, Y)$	$\mathcal{C}(\mathcal{P}, \hat{Y})$
$\mathcal{P}_{opt}$	$cn \bowtie mc \bowtie mk \bowtie k \bowtie t$	1,980	9,713
$\mathcal{P}_{pg}$	$k \bowtie mk \bowtie mc \bowtie cn \bowtie t$	190.4K	125

(a) Exhaustive plan search



2-way joins (CARDINALITY only)					
index ρ	1	2	3	4	5
sub-plan	$cn \bowtie mc$	$k \bowtie mk$	$mc \bowtie t$	$mk \bowtie t$	$mc \bowtie mk$
true Y	388	41.8K	2.6M	4.5M	34.9M
est. $\hat{Y}$	973	20	1.5M	2.7M	13.8M
Q-error q	2.51	2,092	1.70	1.70	2.53

3-way joins (CARDINALITY only)					
index ρ	1	2	3	4	5
sub-plan	$cn \bowtie mc \bowtie t$	$cn \bowtie mc \bowtie mk$	$k \bowtie mk \bowtie t$	$k \bowtie mk \bowtie mc$	$mc \bowtie t \bowtie mk$
true Y	388	1,588	41.8K	148.6K	34.9M
est. $\hat{Y}$	973	8,739	20	104	2.7M
Q-error q	2.51	5.50	2,092	1,428.38	12.69

4-way joins (CARDINALITY only)			
index ρ	1	2	3
sub-plan	$cn \bowtie mc \bowtie mk \bowtie k$	$cn \bowtie mc \bowtie t \bowtie mk$	$k \bowtie mk \bowtie t \bowtie mc$
true Y	4	1,588	148.6K
est. $\hat{Y}$	1	8,739	104
Q-error q	4	5.50	1,428.38

5-way joins (CARDINALITY only)	
index ρ	1
sub-plan	$cn \bowtie mc \bowtie t \bowtie mk \bowtie k$
true Y	4
est. $\hat{Y}$	1
Q-error q	4

Plan	Join order	$\mathcal{C}(\mathcal{P}, Y)$	$\mathcal{C}(\mathcal{P}, \hat{Y})$
$\mathcal{P}_{opt}$	$cn \bowtie mc \bowtie t \bowtie mk \bowtie k$	2,364	10.7K
$\mathcal{P}_{pg}$	$k \bowtie mk \bowtie t \bowtie mc \bowtie cn$	232.2K	144

(b) Greedy plan search

Figure 5.3: Query plans over the left-deep tree search space corresponding to JOB query 2c.

In the last row of every table from Figure 5.3, we show the individual Q-error for the corresponding sub-plan. Q-errors for sub-plans in both exhaustive and greedy plan enumeration are the same. This is because Q-error solely measures the error between true and estimated join cardinalities and does not take into account the sub-plan costs. In the figure, across all join sizes, the Q-error of $\mathcal{P}_{mc \bowtie t}$ and $\mathcal{P}_{mk \bowtie t}$ are the smallest – both equal to 1.70 – which are underestimates of the true cardinalities in this case. The least accurate estimations are for the 2-way join $\mathcal{P}_{k \bowtie mk}$ and the 3-way join $\mathcal{P}_{k \bowtie mk \bowtie t}$. These Q-errors are equal to 2,092, which also underestimates the true cardinalities. Alternatively, the cardinality of $\mathcal{P}_{cn \bowtie mc \bowtie t}$ is overestimated when the Q-error is 2.51.

In Figure 5.4a, at each join level, we depict the Q-error measured for the 70,407 sub-plans generated from all 113 queries in the JOB benchmark – excluding cross-joins [63, 65]. The sub-plans are grouped by the number of joins – ranging from 2 to 17 – shown on the x-axis. At each join size, Q-errors are shown via boxplots including 95, 75, 50, 25, and 5 percentiles. Additionally, Figure 5.4b provides a breakdown of the number of estimates grouped by join size, which illustrates the effect of estimation errors. By examining these estimates, we can gain a clearer understanding of how estimation errors are distributed across different join sizes and how these errors can impact the overall performance of finding optimal query plans. The results show that cardinality estimations for 2-way joins based on $1,336$ estimations – which are 1.9% of all sub-plans – are the most accurate. Estimation accuracy from 3-way to 6-way joins starts decreasing – median Q-errors are between 10^1 and 10^3 based on $21,690$ estimations, which are 30.8% of all the sub-plans. Starting from 7-way to 13-way joins, median Q-error significantly increases – over 10^3 – which includes $46,544$ estimations, or 66.1% of all sub-plans. The rest of the queries form 1.2% of all sub-plans, which is 837 queries. Interestingly, although the number of sub-plans is small, we observe relatively smaller errors starting from 14-way to 17-way joins. These observations indicate that errors increase exponentially with the increase in join size [42]. Inaccuracies in cardinality estimation can have a cumulative detrimental effect on finding optimal query plans. Specifically, significant errors in cardinality estimations at higher-level joins can outweigh and misdirect the query optimizer, causing it to essentially select a query plan at random. Such errors reduce the likelihood of finding the optimal join order and selecting efficient physical operators, thus compromising the overall effectiveness of the query optimizer.

In Figure 5.5, we illustrate the effect of errors in cardinality estimation on plan enumeration. This figure provides a visual understanding of how inaccuracies in cardinality estimation can influence the performance of different plan enumeration algorithms, highlighting the importance of accurate estimations in finding an optimal query plan. For every JOB query – shown on the x-axis grouped by join size – we compute the cost $\mathcal{C}$ of four different plans (shown on the y-axis) selected by exhaustive and greedy enumeration when utilizing both true cardinalities Y and the PostgreSQL estimations $\hat{Y}$. These costs are plotted relative to the cost of the exhaustive plan with true cardinalities $\mathcal{C}(\mathcal{P}_{opt}, Y)$ – the horizontal solid red line at 1. The plans $\mathcal{C}(\mathcal{P}_{opt}, Y)$ are optimal within the search space. As expected, the costs of these optimal plans are lower than the costs of the other plans – equal to or above the red horizontal line. We also observe that several plans selected by the greedy enumeration based on true cardinalities have a higher cost than the greedy plans computed with estimates – green spikes above the blue line. These cases occur when misestimated cardinalities $\hat{Y}$ accidentally lead the greedy enumeration to more efficient plans than those selected based on true cardinalities Y.

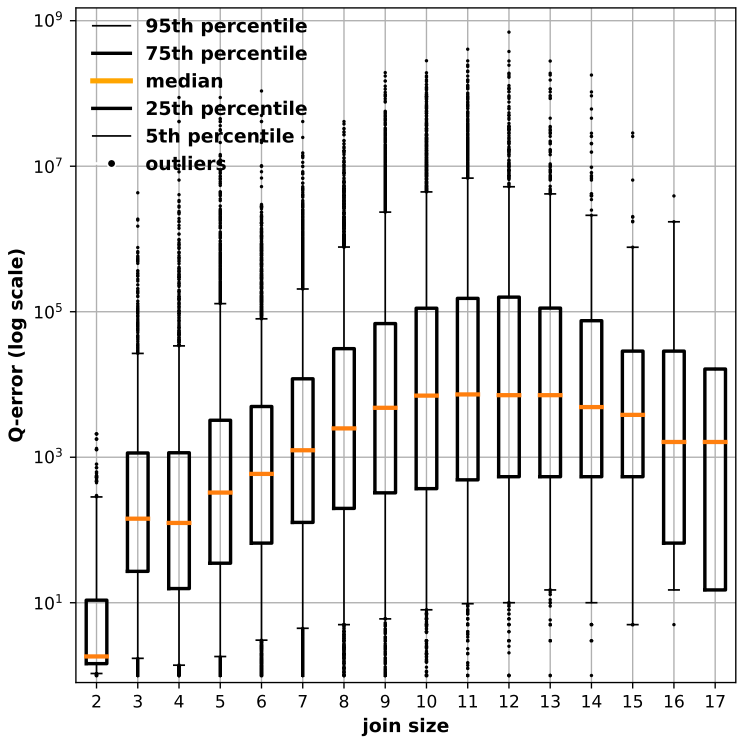
The figure also demonstrates the growing need for accurate estimations as the query complexity increases – the gaps among the red, black, and blue lines around the second red vertical line. This behavior is expected as we begin to observe significant misestimations starting from 7-way joins and beyond – as shown in Figure 5.4a, the Q-error exceeds 10^3 in these cases. Moreover, these significant misestimations adversely impact the exhaustive plan enumeration, as the cumulative effect of estimation errors misguides the cost and enumeration components. For moderate and complex queries, plans selected by exhaustive enumeration have higher costs than plans greedily selected based on estimations – black spikes above the blue line. This means the greedy search algorithm appears to make better decisions based on relatively accurate early-stage join estimations while the exhaustive enumeration is misguided by large misestimations of larger joins. Consequently, the supposedly optimal plans are underestimated compared to the actual optimal plans. Hence, they are selected by the exhaustive enumeration algorithm. These observations suggest that the advantage of exhaustive search diminishes when operating on significant misestimations of large joins. Such errors hinder the enumeration algorithm from finding the optimal query plan.

5.1.2 Q-error for Plan Optimality

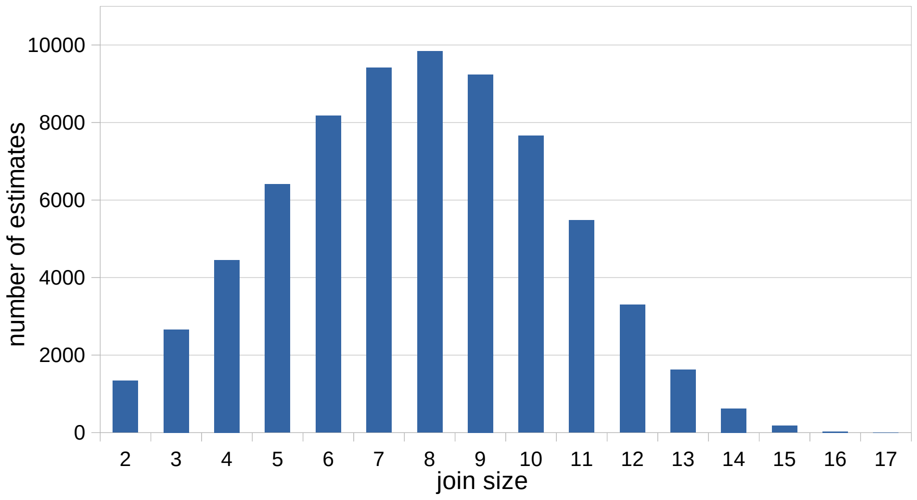
In addition to measuring the error of an individual estimate, Moerkotte et al. [81] introduce a theoretical upper-bound on the ratio of the cost of the selected $\mathcal{P}_{pg}$ and optimal $\mathcal{P}_{opt}$ query plans using Q-error:

$$\frac{\mathcal{C}(\mathcal{P}_{pg}, Y)}{\mathcal{C}(\mathcal{P}_{opt}, Y)} \leq q^4 \tag{5.2}$$

where $q = \max_{i \subseteq X} \{q_i\}$ and X is the set of all sub-plans. The cost ratio between the selected and optimal plan has recently been named P-error [83, 66, 32]. In Figure 5.3, for query 2c, the maximum Q-error is $q = 2,092$. The bound states that, given the estimated



(a) Q-error computed using PostgreSQL estimates.



(b) Number of estimates in exhaustive enumeration.

Figure 5.4: The distribution of the Q-error and the number of sub-plans as a function of the number of joins for all the 70,407 sub-plans generated from the 113 JOB queries.

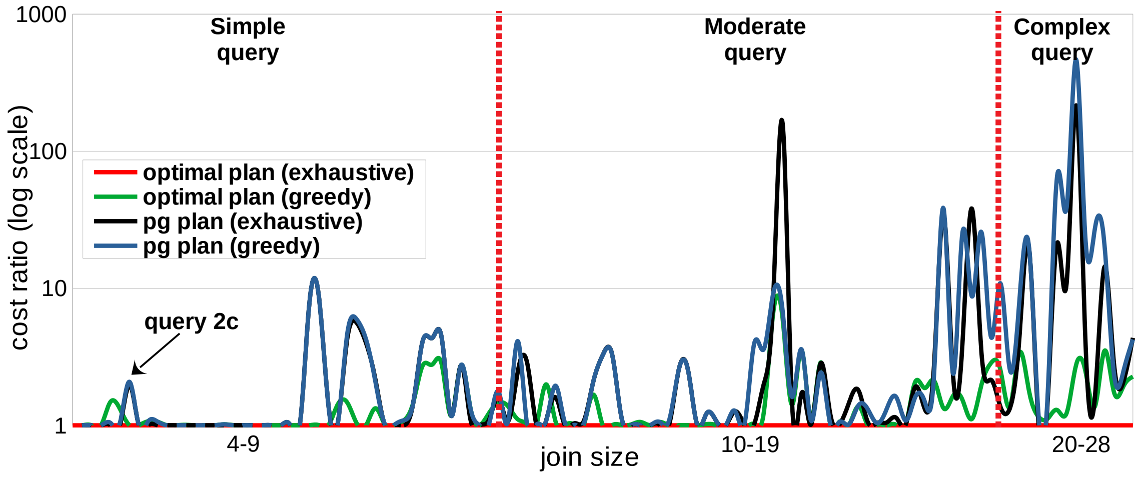


Figure 5.5: Impact of cardinality estimation on plan enumeration. The costs are normalized to the cost of the optimal plan obtained by exhaustive enumeration using true cardinalities.

and true cardinalities of a query, we can determine whether the selected plan is equivalent to the optimal plan without having to enumerate and select expected and optimal query plans to compute P-error. In other words, if the P-error of the selected and optimal plan is at most q^4, then the selected plan is identical – or close – to the optimal plan. This approach provides a theoretical method to evaluate the optimality of a query plan based on cardinality estimates, reducing the computational burden associated with exhaustive enumeration.

To evaluate the Q-error bound on a larger and more complex workload operating on real-world data, we compute both the Q-error and P-error for the entire JOB benchmark consisting of 113 queries [63, 65]. The results are presented in Figure 5.1 from the Introduction 5. We categorize the queries based on the number of joins: 45 simple queries with 4-9 join predicates, 53 moderate queries with 10-19 join predicates, and 15 complex queries with 20-28 join predicates, respectively. To compute the P-error for a query, the join orders $\mathcal{P}_{opt}$ and $\mathcal{P}_{pg}$ are determined by the exhaustive enumeration algorithm with both true and PostgreSQL estimated cardinalities. The figure also includes the upper-bound from Equation 5.2 as an exponential function – represented by the red line.

The results show a large number of simple and moderate queries with optimal plans – P-error equal to 1 – despite having a high Q-error. The P-error for these queries indeed falls within the bound, hence their plans are optimal. However, many other queries that comply with the bound exhibit significantly larger P-error and even higher Q-error. These queries are primarily moderate or complex queries having plans that are not optimal, thus, resulting in different join orders despite satisfying the bound. For query 2c from Figure 5.3a, the optimal plan $\mathcal{P}_{opt}$ is $(cn \bowtie mc \bowtie mk \bowtie k \bowtie t)$, which has the true cost $\mathcal{C}(\mathcal{P}_{opt}, Y) = 1,980$. The PostgreSQL selected plan $\mathcal{P}_{pg}$ is $(k \bowtie mk \bowtie mc \bowtie cn \bowtie t)$, which has the true cost $\mathcal{C}(\mathcal{P}_{pg}, Y) = 190.4K$. This results in a P-error of 96 and a Q-error of $q = 2,092$. According to the bound [32], $96 \leq 2,092^4 \approx$ 2E+13 is correct. However, the gap between the two values is immense – more than 10 orders of magnitude. For moderate and complex queries, we observe larger cost deviations and extremely high Q-error values. This indicates that the selected and optimal query plans are very different despite satisfying the Q-error bound. Therefore, we argue that Q-error is too loose as a bound and, as an indicator, fails to identify sub-optimal query plans. Intuitively, we expect a small P-error to correspond with a small Q-error and a large P-error with a large Q-error. However, our observations show that queries with a large P-error can have a small Q-error and vice versa. Consequently, these observations show that the maximum Q-error bound falls short in accurately determining the optimality of query plans [32, 84].

5.2 L1-error

Finding an optimal join order highly depends on the accuracy of cardinality estimations and how the misestimation errors "impact" the plan enumeration algorithm. In Figure 5.3, we show the sub-plans enumerated by exhaustive and greedy plan search algorithms sorted by the true cardinality Y. The sorted sub-plans are shown for every join size k of query 2c. For 2-way joins, if the sub-plans are sorted by the estimated cardinality $\hat{Y}$, then the relative order becomes different from the order of the sub-plans sorted by true cardinality Y. The difference occurs because of the impact of cardinality misestimation errors of sub-plans $cn \bowtie mc$ and $k \bowtie mk$. However, this difference is minimal in this case – only on one position – henceforth, the plan enumeration algorithm is likely to make accurate decisions in choosing optimal sub-plans. In this section, we introduce the L1-error to quantify the permutation distance between the order of the true cardinalities and that of the estimates for a given sub-plan size. While the importance of the relative ordering of the estimates has been pointed out in previous work [81], L1-error is the first measure to analytically quantify and employ it in determining if a particular join order is optimal.

5.2.1 Relative Sub-Plan Arrangement

By sorting the sub-plans for each join size k by true and estimated cardinality, we have two sub-plan arrangements – position vectors of the same length. For instance, in 2-way joins, the two position vectors are defined as $\rho = (1, 2, 3, 4, 5)$ sorted by Y and $\widehat{\rho} = (2, 1, 3, 4, 5)$ sorted by $\hat{Y}$, respectively. To differentiate between ρ and $\widehat{\rho}$, we name ρ as the identity permutation in the rest of the work. Similarly, for each join size k, both position vectors are denoted as ρ and $\widehat{\rho} \in \mathbb{N}^d$ by $(1, 2, \ldots, d)$ where d is the number of sub-plans of size k. There is an extensive range of metrics available, such as Spearman's footrule [102] and Kendall's tau [50], to measure the distance between two arrangements (or ranked lists) [67]. This field has been extensively researched and is a well-studied area. Among the various metrics available, we find Spearman's footrule distance (also known as L1 distance) is particularly effective in quantifying the impact of cardinality estimation errors on plan search algorithms. It provides a strong measure of the distance between two sub-plan arrangements. While it is certainly possible to substitute Spearman's footrule distance with other metrics to compare precision performances, we show, through our observations, that Spearman's footrule distance intuitively fits our problem well and delivers accurate results. In the remainder of this work, for the sake of convenience, we refer to Spearman's footrule distance as L1-error. L1-error measures the element-wise absolute differences between two position vectors ρ and $\widehat{\rho}$:

$$\mathrm{L1}^k\left(\rho, \widehat{\rho}\right) = \sum_{i}^{d} \left|\rho(i) - \widehat{\rho}(i)\right| \tag{5.3}$$

where k is join size, and $\rho(i)$ and $\widehat{\rho}(i)$ are the positions of i-th sub-plan in the position vectors ρ and $\widehat{\rho}$. In the case of identical sub-plan arrangements, $\mathrm{L1}^k(\rho, \widehat{\rho}) = 0$. In the case of discrepancies, L1-error captures the cardinality estimation errors that affect the relative arrangement of sub-plans. In Figure 5.6, we demonstrate the position vectors ρ and $\widehat{\rho}$ for each join size in query 2c – first two rows of the tables located on the left side of the white vertical bars. The position vectors are generated based on the cardinalities shown in Figure 5.3b. L1-error measures, $\mathrm{L1}^k(\rho, \widehat{\rho})$, for 2, 3 and 4-way joins are 2, 8 and 2, respectively.

5.2.2 Weighted Relative Sub-Plan Arrangement

From Equation 5.3, we observe that the L1-error does not take into account the following criteria which are important to capture for plan search algorithms:

- Significantly over and underestimating cardinality should be associated with greater penalties. Conversely, mispositioning sub-plans with similar cardinalities in the position vectors should carry fewer and relatively similar penalties.
- Cardinality misestimations that occur early in the position vector should attract greater penalties. This is particularly beneficial for plan search algorithms, which are more likely to choose a sub-plan from the first half of the position vector.

Given the limitations identified in the original L1-error, we propose an enhanced L1-error. This new design seeks to improve upon the original by more effectively capturing the impacts of cardinality misestimations on plan search algorithms in query optimization. Kumar et al. [60] propose a method for measuring the distance between two ranked lists, with extensions to factor in the weights of elements and positions, as well as similarities between elements. In our study, we adapt and employ variations of these proposed extensions that align intuitively with our problem context.

5.2.2.1 Sub-plan Impact Weights

First, we define a misestimation impact of sub-plans within a weight symmetric matrix W for each join size k. Each i-th row of length d in this matrix represents a sub-plan, and misestimation impact weights are defined as:

$$W_{i,j}^k = \max\left(\frac{Y_j}{Y_i}, \frac{Y_i}{Y_j}\right) \tag{5.4}$$

where $W_{i,j} \geq 1$, and Y_j and Y_i are true cardinality values. In the case of zero cardinality, denominators can be replaced with a small number. The weight scales are comparable across different query complexities because $W_{i,j}$ only captures the relative differences in cardinality.

2-way join (CARDINALITY only)					
ρ	1	2	3	4	5
$\hat{\rho}$	2	1	3	4	5
Swap cost	1.0	107.84	62.36	1.73	7.71
Monotonic weight	1.0	108.84	171.2	172.93	180.64

3-way join (CARDINALITY only)					
ρ	1	2	3	4	5
$\hat{\rho}$	3	4	1	2	5
Swap cost	1.0	4.09	26.35	3.55	234.69
Monotonic weight	1.0	5.09	31.44	34.99	269.68

4-way join (CARDINALITY only)			
ρ	1	2	3
$\hat{\rho}$	1	3	2
Swap cost	1.0	397.0	93.55
Monotonic weight	1.0	398.0	491.55

Impact weight, W (2-way join)					
	1	2	3	4	5
1	1.0	107.84	6,724.56	11,659.61	89,854.74
2		1.0	62.36	108.12	833.26
3			1.0	1.73	13.36
4				1.0	7.71
5					1.0

Impact weight, W (3-way join)					
	1	2	3	4	5
1	1.0	4.09	107.84	382.87	89,854.74
2		1.0	26.35	93.55	21,954.43
3			1.0	3.55	833.26
4				1.0	234.69
5					1.0

Impact weight, W (4-way join)			
	1	2	3
1	1.0	397.0	37,138.0
2		1.0	93.55
3			1.0

Figure 5.6: Sub-plan weights used by the plan enumeration algorithms for JOB query 2c.

Thus, the specific scales of cardinality in simple and complex queries, and join sizes are not consequential. In Figure 5.6, for each join size k, we illustrate the computed weight matrix as a separate table on the right side of the white vertical bars. Row and column indexes in each weight matrix represent sub-plan positions in the identity position vector ρ. The weight matrix signifies the relative differences in cardinality between any two sub-plans of the same join size. A larger weight implies a greater disparity between the two sub-plans in terms of their cardinalities. For example, in Figure 5.3b, the cardinality of 2-way sub-plan $\mathcal{P}_{cn \bowtie mc}$ is 388 and it is significantly less than the cardinalities of the other four 2-way sub-plans with cardinalities of $41.8K$, $2.6M$, $4.5M$, and $34.9M$. Therefore, in row 1 of the weight matrix for $k = 2$, we observe much larger weights. The same weight matrix also exhibits sub-plan weights of similar cardinality, a scenario frequently encountered when estimating sub-plan cardinalities. Cardinalities of sub-plans $\mathcal{P}_{mc \bowtie t}$ and $\mathcal{P}_{mk \bowtie t}$ of join size 2 are relatively close – $2.6M$ and $4.5M$, respectively. Hence, the impact of mispositioning these two sub-plans has less penalty – weight value is 1.73. We define the L1-error that assigns impact penalties for sub-plan misestimations proportionate to their cardinality magnitudes:

$$\mathrm{L1}_W^k = \sum_i^d \sum_{\substack{j:\rho(j)<\rho(i) \\ \wedge\ \widehat{\rho}(j)>\widehat{\rho}(i)}} W_{i,j}^k \quad + \sum_{\substack{j:\rho(j)>\rho(i) \\ \wedge\ \widehat{\rho}(j)<\widehat{\rho}(i)}} W_{i,j}^k \tag{5.5}$$

To measure the difference between ρ and $\widehat{\rho}$ with sub-plan misestimation impacts, the outer term sums weights for each sub-plan i when it is mispositioned with other sub-plans. The left inner term aggregates the weights of sub-plans when their true cardinalities are overestimated more than the estimated cardinality of the sub-plan i in $\widehat{\rho}$, despite those sub-plans having true cardinalities smaller than the true cardinality of sub-plan i in ρ. Similarly, the right inner term sums the weights of sub-plans with underestimated true cardinality that is more than the estimated cardinality of the sub-plan i in $\widehat{\rho}$, despite those sub-plans having larger true cardinalities than the true cardinality of sub-plan i in ρ.

For example, in Figure 5.6, sub-plan $\mathcal{P}_{k\bowtie mk\bowtie t}$ of join size 3 is located at position 3 in ρ. Its true cardinality of $41.8K$ is underestimated as 20 which is less than the estimated cardinalities 973 and $8,739$ for sub-plans $\mathcal{P}_{cn\bowtie mc\bowtie t}$ and $\mathcal{P}_{cn\bowtie mc\bowtie mk}$ which have true cardinalities as 388 and $1,588$, respectively. Hence, in the weight matrix W^3, the left inner term sums the penalty weights $W_{3,1} = 107.84$ and $W_{3,2} = 26.35$ of $\mathcal{P}_{k\bowtie mk\bowtie t}$ for being misplaced in $\widehat{\rho}$ to the left of sub-plans $\mathcal{P}_{cn\bowtie mc\bowtie t}$ and $\mathcal{P}_{cn\bowtie mc\bowtie mk}$. Similarly, the right term sums the penalty weights for being misplaced to the right side of sub-plans locations in $\widehat{\rho}$ despite their larger true cardinality and higher locations in ρ. However, sub-plan $\mathcal{P}_{k\bowtie mk\bowtie t}$ is not overestimated than sub-plans $\mathcal{P}_{k\bowtie mk\bowtie mc}$ and $\mathcal{P}_{mc\bowtie t\bowtie mk}$. Sub-plan $\mathcal{P}_{k\bowtie mk\bowtie t}$ has true cardinality of $41.8K$ and estimate as 20, while sub-plans $\mathcal{P}_{k\bowtie mk\bowtie mc}$ and $\mathcal{P}_{mc\bowtie t\bowtie mk}$ have true cardinality of $148.6K$ and $34.9M$, and estimates as 104 and $2.7M$, respectively. Thus, the right term is equal to 0 and the overall misposition penalty weight for sub-plan $\mathcal{P}_{k\bowtie mk\bowtie t}$ is $107.84 + 26.35 + 0 = 134.19$. For sub-plans of join size $k = 3$, total misestimation impact is $\mathrm{L1}_W^3 = 1,221.22$.

5.2.2.2 Sub-plan Position Weights

Intuitively, imposing larger penalties for the misestimation of smaller cardinalities is desirable. This is because plan search algorithms are more likely to select sub-plans with smaller cardinalities. The position vectors represent the locations of sub-plans sorted by their true cardinality, thus sub-plans with smaller cardinalities are expected to be positioned early or left half in ρ. In Figure 5.3, search algorithms, when guided by true cardinality, tend to select sub-plans from the left half of sub-plan lists on every join step. Consequently, preserving the relative order of sub-plans with smaller cardinalities is generally of significant importance. In other words, we impose a higher penalty for the early position differences between position vectors ρ and $\widehat{\rho}$ than the differences at the tail. To achieve desired position-based penalty weights, we need monotonically increasing weights similar to in [60]. We define the cost of a swap between two adjacent sub-plans i and $i-1$ in the position vectors as the ratio between their true cardinalities $Y_i/Y_{i-1} \geq 1$. In Figure 5.6, we show the swap costs in the third row named as *Swap cost* in each join table. Then, the monotonically increasing swap weights are defined as:

$$\mu_i = \mu_{i-1} + \frac{Y_i}{Y_{i-1}} \tag{5.6}$$

where $\mu_1 = 1$, and $\mu_i < \mu_j < \mu_k$ such that $i < j$ and $j < k$. In Figure 5.6, we show the monotonically increasing weights in the fourth row named as *Monotonic weight* in each join table. This monotonic property offers weights by considering both the distance in position and the closeness in cardinality values of the sub-plans. For example, in Figure 5.6, sub-plan $\mathcal{P}_{k \bowtie mk \bowtie t}$ of join size 3 is at distance 1 from sub-plan $\mathcal{P}_{k \bowtie mk \bowtie mc}$ while $\mathcal{P}_{cn \bowtie mc \bowtie t}$ is away from $\mathcal{P}_{k \bowtie mk \bowtie mc}$ for 3 positions. Similarly, true cardinality 388 of $\mathcal{P}_{cn \bowtie mc \bowtie t}$ is much less than true cardinality $148.6K$ of $\mathcal{P}_{k \bowtie mk \bowtie mc}$ while sub-plan $\mathcal{P}_{k \bowtie mk \bowtie t}$ has true cardinality of $41.8K$. Hence, the respective monotonic weights for these three sub-plans are 1.0, 31.44, and 34.99. Integrating position-based monotonic weights μ into $\mathrm{L1}_W^k$ assigns greater penalties to the misestimation of smaller true cardinalities that are positioned early in the position vector ρ. By assigning these estimations more weight, L1-error more accurately reflects the significance of their errors, improving its ability to predict sub-optimal query plans:

$$\mathrm{L1}^k = \sum_{i}^{d} \mu_i^{-1} \quad \times \quad \left[\sum_{\substack{j:\rho(j)<\rho(i) \\ \wedge\ \widehat{\rho}(j)>\widehat{\rho}(i)}} W_{i,j}^k \quad + \sum_{\substack{j:\rho(j)>\rho(i) \\ \wedge\ \widehat{\rho}(j)<\widehat{\rho}(i)}} W_{i,j}^k \right] \tag{5.7}$$

In Figure 5.6, in 3-way joins, penalty weights assigned to the five sub-plans are 490.71, 23.55, 4.27, 13.62, and 0, respectively. L1-error assigns a higher penalty to sub-plans $\mathcal{P}_{cn \bowtie mc \bowtie t}$

and $\mathcal{P}_{cn \bowtie mc \bowtie mk}$, which are 490.71 and 23.55 respectively, because of their earlier positions in ρ. Despite being placed at the fourth position, sub-plan $\mathcal{P}_{k \bowtie mk \bowtie mc}$ has a higher penalty weight of 13.62 than $\mathcal{P}_{k \bowtie mk \bowtie t}$ with 4.27 penalty weight at the third position. This is because $\mathcal{P}_{k \bowtie mk \bowtie mc}$ has a larger cardinality, making it riskier to misplace. Consequently, overall L1-error for 3-way join is $\mathrm{L1}^3 = 532.15$. This illustration exemplifies how L1-error effectively penalizes misestimations based on their impact on plan search algorithms.

5.3 L1-error for Plan Optimality

In this section, we explore the potential of the L1 error as a complement or perhaps even an alternative metric to Q-error to evaluate query plans. In addition, we examine how the L1 error, as an independent feature, can be utilized in classifying sub-optimal query plans, thereby demonstrating its efficacy as a reliable indicator.

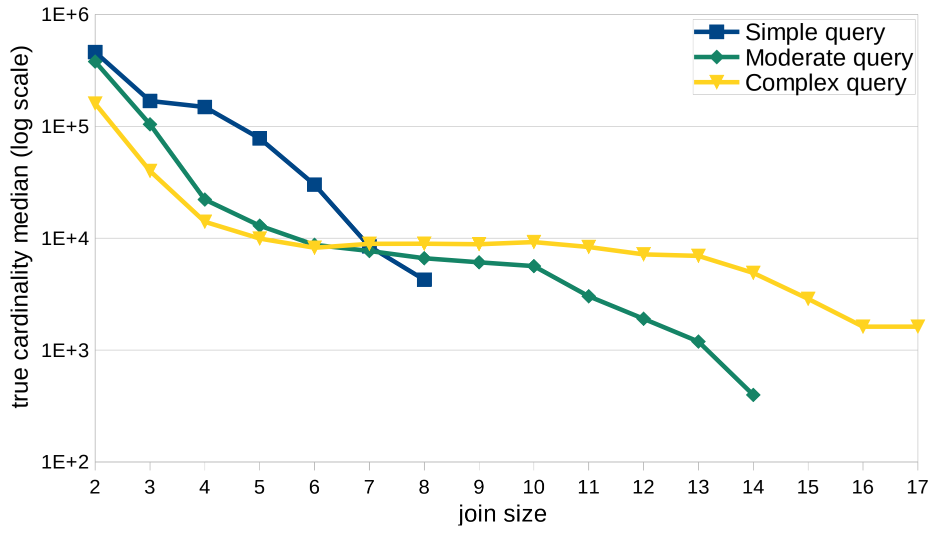


Figure 5.7: Importance of small joins at the beginning and decreasing impact of large joins at the end of query plan enumeration algorithms.

5.3.1 Query-level L1-error

In Section 5.2, we show how $\mathrm{L1}^k$ can independently assess cardinality estimation errors at each join size k. It is essential to note that the number of sub-plans d at each join size k may vary depending on the complexity of the query – different lengths of the position vectors ρ

and $\widehat{\rho}$. By aggregating individual L1-error at each join size into $\mathrm{L1}^k$, we can form a feature vector of size K, starting from 2-way joins $k = 2$. Given that queries can have different join sizes, the dimensions of these feature vectors can also vary. We define a query-level L1-error as:

$$\mathrm{L1}_Q = \sum_{k=2}^{K} \mathrm{L1}^k \tag{5.8}$$

The magnitude of this query-level L1 error can be influenced by the complexity of the query – various numbers of join sizes involved. Consequently, the aggregation approach, which sums up all join-level L1 errors, can be particularly sensitive to the complexity of the query. This implies that more complex queries with a larger number of join sizes may naturally result in higher aggregated L1 errors, emphasizing the influence of query complexity on the overall L1-error calculation. To alleviate this issue, we can take into account our insights from Figure 5.4a. Cardinality estimations for higher-level joins tend to be severely inaccurate. There is a visible decline in accuracy following 3-way joins, and a considerable drop after 6-way joins. Given this, it is evident that higher-level join cardinality estimations are unreliable and should, therefore, be excluded from consideration or given less weight in comparison to lower-level join cardinality estimations [112]. This approach would counterbalance the tendency for larger errors in complex queries. To further support this decision, we present Figure 5.7 that illustrates how intermediate data decreases as the join size grows. For each query complexity, we group all JOB sub-plans – including $PK + FK$ and $FK + FK$ joins – by join size and plot the median cardinality value. The results show a consistent trend across all query complexity groups: as join size increases and more filter predicates come into play, intermediate data reduces. Therefore, selecting optimal sub-plans in the early stages of join sizes is crucial to prevent the propagation of large intermediate data. Decisions made at later stages, influenced by inaccuracies, are less impactful due to the smaller volume of intermediate data processed at higher-level joins. In the figure, we observe a notable drop in data after 6-way join sizes in simple queries, 10-way join in moderate queries, and 15-way join in complex queries. Interestingly, the volume of intermediate data for complex queries between 6 to 13 joins remains stable. This implies the importance of choosing optimal sub-plans up to 13-way joins. While these results are primarily derived from the analysis of JOB sub-plans, we believe this observed trend applies to a wide range of workloads. Taking into account the discussed facts and observed trends, we assign weights to already computed join-level L1-errors L1^k at each join size using an inverse logistic function. For notational purposes, let

$$w_k = \frac{e^{-t \times k}}{1 + e^{-t \times k}} \tag{5.9}$$

where t represents the logistic growth rate or the steepness of the curve. With this in mind, we can define the weighted query-level L1-error as follows:

$$\text{L1}_Q = \sum_{k=2}^{K} w_k \times \text{L1}^k = w^T \times \text{L1} \tag{5.10}$$

L1-errors across different join sizes are aggregated into a unified L1-error while assigning lower weights to L1-errors at high-level joins. The logistic growth rate t can be tuned based on the performance of the cardinality estimator in use. For example, we set $t = 1.5$, thus $w\_7 = 0.000028$, based on the Q-error values generated by PostgreSQL, depicted in Figure 5.4a. It begins to reduce the impact of join-level L1-errors starting from 7-way joins as we observe significant estimation errors at higher-level joins.

5.3.1.1 Algorithm Overview

Algorithm 4 shows an overview of computing L1-error for a given query Q. The inputs are the maximum join size K and all sub-plans S along with their true Y and estimated $\hat{Y}$ cardinalities. Algorithm 4 consists of two functions Query-L1-error (lines 1-4) and Join-L1-error (lines 5-14). For the sake of clarity, we separate these two functions, although these two functions can be combined. At the query level, in function Query-L1-error, join weights w for each join size k are generated as in Equation 5.9 (line 2). The impact of joins decreases as their sizes increase following the 'S'-shaped sigmoid curve. Join-level $\text{L1} \in \mathbb{R}^K$ are aggregated into the final query-level L1_Q (line 4). Join-L1-error computes join-level L1^k for each join size k. For the sub-plans S_k of size k, two position vectors ρ and $\hat{\rho}$ are created and sorted in increasing order based on the true Y and estimated $\hat{Y}$ cardinalities, respectively (lines 7-9). Simultaneously, impact weight matrix W^k and position weights δ^k are generated from Equations 5.4 and 5.6, respectively (lines 10-12). Lastly, a join-level L1^k

for join size k is computed from Equation 5.7 (line 13). We repeat the process for each join size (lines 6-13) yielding a feature vector, L1 of length K, of join-level L1^k (line 14).

Algorithm 4 L1-error

Input: largest join size K of input query Q, set of query sub-plans S, true Y and estimated $\hat{Y}$ cardinalities
Output: L1_Q weighted query-level L1-error

```
1: function QUERY-L1-ERROR($K$, $S$, $Y$, $\hat{Y}$)
2:     $w \in \mathbb{R}^{K-1} \leftarrow$ join size weights from Equation 5.9
3:     $\text{L1} \leftarrow$ JOIN-L1-ERROR($K, S, Y, \hat{Y}$)
4:     return $\text{L1}_Q = w^T \times \text{L1}$
5: end function
6: function JOIN-L1-ERROR($K$, $S$, $Y$, $\hat{Y}$)
7:     for each join size $k \in \{2 \dots K\}$ do
8:         $S_k \subset S \leftarrow$ subset of sub-plans of size $k$ in $S$
9:         $\rho \leftarrow$ positions of $S_k$ increasingly sorted by $Y$
10:        $\hat{\rho} \leftarrow$ positions of $S_k$ increasingly sorted by $\hat{Y}$
11:        // compute weights from Equations 5.4 and 5.6
12:        $W^k \leftarrow$ impact weights of sub-plans in $S_k$
13:        $\delta^k \leftarrow$ position weights of sub-plans in $S_k$
14:        $\text{L1}^k \leftarrow$ L1-error for join size $k$ from Equation 5.7
15:    return $\text{L1} \leftarrow$ join-level L1 feature vector of size $K-1$
16: end function
```

5.3.2 Applications of L1-error

While minimizing individual Q-errors can enhance the overall efficiency of a query optimizer, the metric often falls short of accurately indicating query plan optimality [32, 111]. Q-error is conventionally used to assess cardinality estimation techniques and synopses, as a separate sub-task that influences the likelihood of finding an optimal query plan [65, 52]. In recent years, Q-error has been widely adopted in learning-based approaches [111], where it is utilized in the post-training evaluation phase [38, 118, 117, 122] and during training [54, 83]. In Section 5.1, we observe that being solely an error measurement, cannot reliably identify sub-optimal query plans. This limitation arises from the fact that other crucial factors, such as cost function and plan enumeration, that bridge the gap between estimation error and the selection of an optimal plan, are not taken into account by Q-error.

In this work, we present L1-error as a metric to characterize plan sub-optimality, taking as an input only cardinalities and without requiring plan enumeration to compute P-error.

Contrary to Q-error which primarily concentrates on estimation precision, L1-error prioritizes the relative order of sub-plans – a critical aspect for cost function and plan enumeration algorithms. It accounts for the impact weights of sub-plans and their relative displacement in the presence of estimation errors. Therefore, unlike Q-error, L1-error is capable of accurately identifying queries with sub-optimal plans. This suggests that L1-error can serve as a complementary metric to Q-error to evaluate query plans and can be employed in future research to evaluate the sub-optimality of query plans produced by synopses and learning-based models. In the current work, we evaluate L1-error as a standalone measure. For this purpose, we frame the identification of queries with sub-optimal query plans as a binary classification task.

5.4 Empirical Evaluation

In the current study, we assess L1-error as a separate measure and frame the identification of queries with sub-optimal query plans as a binary classification task. This allows us to evaluate the standalone efficacy of L1-error. Our evaluation of L1-error spans three different facets – varying sources of cardinality estimates, plan search algorithms, and workloads and data.

5.4.1 Experimental Setup

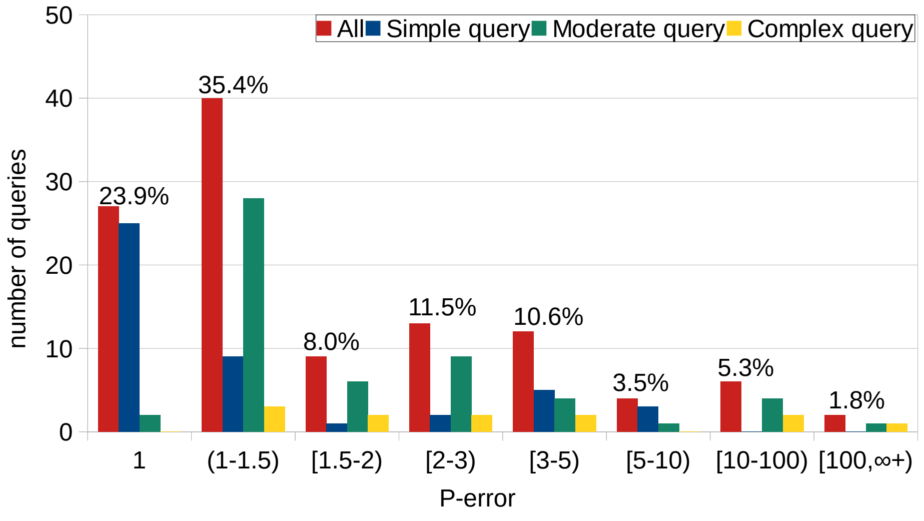
5.4.1.1 Dataset & Query Workloads

We perform the experiments on the JOB benchmark [92] over IMDB dataset [12], which has seen extensive use in evaluating query optimizers and has thereby established itself as a standard benchmark [63, 65]. The JOB benchmark defines 113 queries grouped into 33 families. These queries vary significantly in their complexity, with the simplest having 4 join predicates and the largest join size of 4, and the most complex having 28 join predicates with the largest join size of 17. This variability manifests itself in execution times that are highly different. To compensate for this, we split the queries into three complexity groups – simple (4-9 join predicates), moderate (10-19 join predicates), and complex (20-28 join predicates) – and examine each group separately. We also perform experiments using the JOB-light benchmark [53], a simpler version of the JOB benchmark that includes 67 simple queries that can be represented with a star join graph and feature 2 to 4 join predicates. To show how L1-error generalizes to different workloads and data, we evaluate L1-error on JCCH [13] and DSB [23] benchmarks. We use scale factors of 1 and 10 and generate 511 and 1440 queries, respectively.

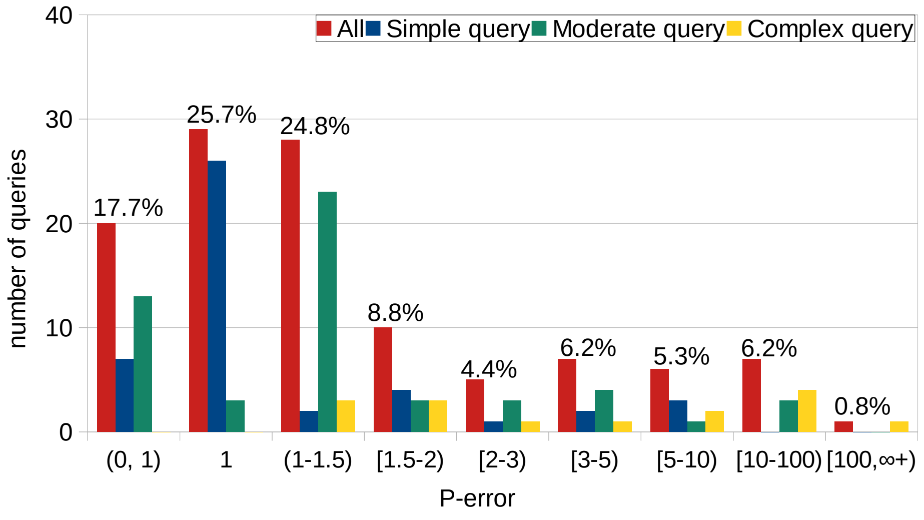
5.4.1.2 Methodology & Implementation

For cardinality estimations, we select two distinct cardinality estimators – PostgreSQL 15.1 [88], a widely recognized database system, and COMPASS [46, 47], a more recent system. We run the sub-plans of the four workloads in PostgreSQL to collect their estimated and true join cardinalities. Additionally, we collect estimated join cardinalities for the sub-plans of JOB and JOB-light produced by COMPASS. We compute the P-Error for each query and use it as a true label in our binary classification task. We acquire query plans utilizing exhaustive and greedy plan search algorithms based on estimated and true cardinalities. The cost of the query plans is calculated using the cost function $\mathcal{C}$ defined in Equation 2.4.

We show the performance of the L1-error-based binary classifier via confusion matrices depicted in Tables 5.1 and 5.2. We label queries with sub-optimal plans as 'positive' (P) and 'negative' (N) otherwise. These classifications are shown in the fifth to eighth columns of the confusion tables. In the tables, we report four measures: 'true positive' (TP), 'true negative' (TN), 'false positive' (FP), and 'false negative' (FN) shown on the ninth to twelfth columns. In addition, we report overall accuracies on test data of the classifiers based on L1-error, Q-error and both in Figures 5.9 and 5.10. In the binary classification task, we use a CART decision tree model of a tree depth of 5 from the Scikit-learn library (version 1.1.2). For the logistic growth rate in Equation 5.9, we set $t = 1.5$, which begins weighting 4-way joins at ≈ 0.002. We partition the queries into training and testing, using a 70% to 30% split, respectively, to have a large enough test data for the classification task. In JCCH and DSB, we split the data into 80% to 20%. The resulting data sizes are shown in the third and fourth columns in Tables 5.1 and 5.2.



(a) Exhaustive enumeration



(b) Greedy enumeration

Figure 5.8: Distribution of JOB queries based on P-error. Plans are selected using true and PostgreSQL estimated cardinalities.

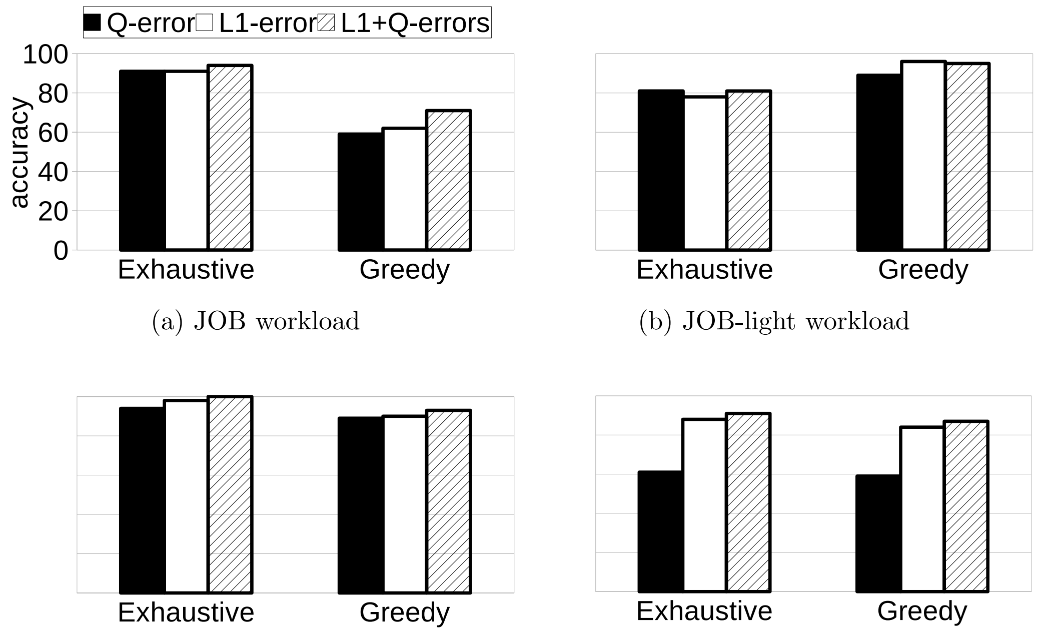
Benchmark	**Enumerator**	**Train queries**	**Test queries**	**Actual**		**Predicted**		**TP**	**TN**	**FP**	**FN**
				Sub-optimal (Positive)	Optimal (Negative)	Sub-optimal (Positive)	Optimal (Negative)				
JOB 113 queries	Exhaustive	79	34	87 61 train *26 test*	26 18 train *8 test*	94 67 train *27 test*	19 12 train *7 test*	85 60 train *25 test*	17 11 train *6 test*	9 7 train *2 test*	2 1 train *1 test*
	Greedy			46 32 train *14 test*	67 47 train *20 test*	51 34 train *17 test*	62 45 train *17 test*	33 24 train *9 test*	49 37 train *12 test*	18 10 train *8 test*	13 8 train *5 test*
JOB-light 67 queries	Exhaustive	40	27	9 5 train *4 test*	58 35 train *23 test*	3 2 train *1 test*	64 39 train *25 test*	1 1 train *0 test*	56 35 train *21 test*	2 0 train *2 test*	8 4 train *4 test*
	Greedy			2 1 train *1 test*	65 39 train *26 test*	0 0 train *0 test*	67 40 train *27 test*	0 0 train *0 test*	65 39 train *26 test*	0 0 train *0 test*	2 1 train *1 test*
JCCH 511 queries	Exhaustive	408	103	150 120 train *30 test*	361 288 train *73 test*	127 99 train *28 test*	384 309 train *75 test*	127 99 train *28 test*	361 288 train *73 test*	0 0 train *0 test*	23 21 train *2 test*
	Greedy			120 96 train *24 test*	391 312 train *79 test*	59 45 train *14 test*	452 363 train *89 test*	59 45 train *14 test*	391 312 train *79 test*	0 0 train *0 test*	61 51 train *10 test*
DSB 1440 queries	Exhaustive	1152	288	727 582 train *145 test*	713 570 train *143 test*	668 529 train *139 test*	772 623 train *149 test*	586 462 train *124 test*	631 503 train *128 test*	82 67 train *15 test*	141 120 train *21 test*
	Greedy			668 534 train *134 test*	772 618 train *154 test*	626 503 train *123 test*	814 649 train *165 test*	529 424 train *105 test*	675 539 train *136 test*	97 79 train *18 test*	139 110 train *29 test*

Table 5.1: Evaluation of L1-error on query plans selected using PostgreSQL cardinality estimates.

5.4.2 Results

5.4.2.1 L1-error Performance on PostgreSQL.

In this section, we first examine L1-error performance identifying the sub-optimality of query plans selected based on PostgreSQL's cardinality estimation. Subsequently, we analyze L1-error performance on different sets of cardinality estimates collected from the COMPASS estimator used to select query plans. We start the evaluation with the JOB workload, composed of a mix of 113 simple, moderate and complex queries. Out of the four workloads, JOB is the most challenging due to its relatively complex graph topology and large number of joins. Figure 5.8 displays the distribution of JOB queries, grouped by their P-error shown on the x-axis. In Figure 5.8a, when employing exhaustive enumeration, we observe that 23.9% of the selected plans are equivalent to optimal plans (P-error = 1), while 35.4% of 113 queries are near-optimal (P-error < 1.5), thus may be considered successful. The remaining 40.7% of the overall number of queries exhibits larger cost differences, some of which are severely sub-optimal. In the case of greedy enumeration, Figure 5.8b, we observe that 25.7% and 24.8% of the selected plans are equivalent to optimal plans (P-error = 1) and near-optimal (P-error < 1.5), respectively. Interestingly, 17.7% of the queries have even better plans than the plans selected using true cardinalities (P-error < 1). This is due to greedy decisions made during the enumeration. The remaining 31.8% of the plans demonstrate higher P-errors, and some are severely sub-optimal. These sub-optimal plans mainly include moderate and complex queries, and their sub-optimal plans should be accurately identified.



(a) JOB workload

(b) JOB-light workload

(c) JCCH workload

(d) DSB workload

Figure 5.9: L1-error classifier accuracy on test data. Query plans are selected using PostgreSQL cardinality estimates.

The objective is to directly classify sub-optimal query plans using only L1-error, not by P-error. The first two rows in Table 5.1 depict the performance of L1-error evaluated on JOB. In the case of exhaustive enumeration, the number of positive queries (sub-optimal) is higher than negative queries (optimal) – columns 5 and 6. Based on the results, we observe a similar trend but now query plans are classified by L1-error as an indicator of queries with sub-optimal plans – columns 7 and 8. The difference between predicted positive and negative queries is also high. Out of 87 true sub-optimal plans (Positive), 85 sub-optimal query plans are correctly classified (TP), resulting in 2 FN. On the other hand, the number of misclassifying optimal plans as sub-optimal (FP) is higher which is not as critical as FN. In the case of greedy enumeration, we observe an opposite trend – the number of positive queries (sub-optimal) is lower than negative queries (optimal). While the classifier results in 18 FP and 13 FN, the predicted Positive and Negative results follow the pattern of the actual Positive and Negative results. Analyzing the misclassified optimal plans, in both enumerations, we observe that the classifier primarily misclassifies queries with P-error < 1.5.

We now evaluate L1-error on simple queries in JOB-light – third and fourth rows in Table 5.1. Unlike in JOB, this particular workload presents a relatively high unbalanced class ratio for the classification task. From the results, we notice a significant difference between the number of true sub-optimal plans (Positive of 9 and 2) and true optimal plans (Negative of 58 and 65) in exhaustive and greedy enumerations, respectively. This is expected because, in simple queries, the join sizes are smaller, thus cardinality estimations are relatively accurate. Therefore, this results in a high number of queries with optimal plans in both search algorithms. The results exhibit a similar trend but now queries are classified by L1-error as an indicator. The difference between predicted positive and negative queries is also high. Even though a large P-error may have less impact on the execution time of simple queries, the classifier is still efficient in identifying plan sub-optimality. As in JOB, the misclassified query plans show P-error < 1.5.

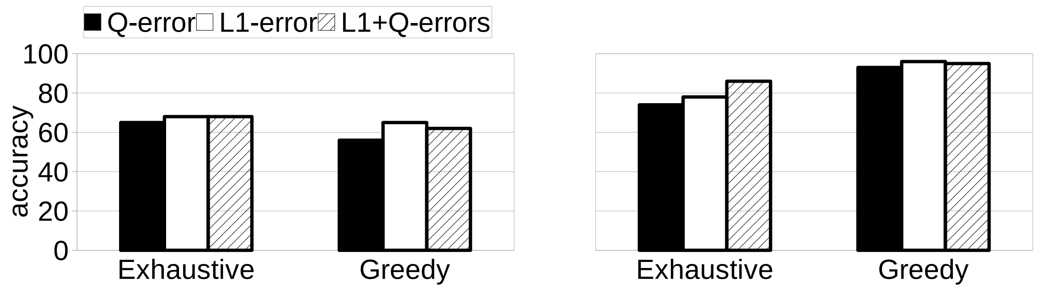
To conduct a comprehensive assessment of L1-error across a more expansive dataset and query spectrum, we present evaluations performed on JCCH (rows 5 and 6) and DSB (rows 7 and 8) workloads. Unlike the previous two workloads, JCCH and DSB show a relatively more balanced class ratio, albeit with the predomination of optimal plans. This is due to the workload complexity standing between JOB and JOB-light. Based on the results, the predicted Positive and Negative results are in a similar trend as in actual Positive and Negative results. The predicted class ratio also follows a similar trend. Interestingly, the classifier avoids misclassifying optimal plans as sub-optimal, with 0 FN in both enumeration algorithms. Looking into the misclassified queries reveals query plans exhibiting P-error values centered around 1.78.

Figure 5.9 illustrates the accuracy of the classifier on the test data. We compare the classifier based on L1-error and the classifier based on Q-error as well as the classifier that utilizes both L1-error and Q-error to identify sub-optimal query plans. Overall, we observe an improvement over the Q-error classifier except in exhaustive enumeration on JOB-light attributed to the small number of test data. The results above suggest L1-error is a viable indicator for identifying sub-optimal query plans and can be used in tandem with Q-error to assess query optimizers to identify query sub-optimality. The classifier based on both

Benchmark	**Enumerator**	**Train queries**	**Test queries**	**Actual**		**Predicted**		**TP**	**TN**	**FP**	**FN**
				Sub-optimal (Positive)	Optimal (Negative)	Sub-optimal (Positive)	Optimal (Negative)				
JOB 113 queries	Exhaustive	79	34	78 55 train *23 test*	35 24 train *11 test*	100 68 train *32 test*	13 11 train *2 test*	76 54 train *22 test*	11 10 train *1 test*	24 14 train *10 test*	2 1 train *1 test*
	Greedy			57 40 train *17 test*	56 39 train *17 test*	80 55 train *25 test*	33 24 train *9 test*	55 40 train *15 test*	31 24 train *7 test*	25 15 train *10 test*	2 0 train *2 test*
JOB-light 67 queries	Exhaustive	40	27	14 8 train *6 test*	53 32 train *21 test*	10 4 train *6 test*	57 36 train *21 test*	7 4 train *3 test*	50 32 train *18 test*	3 0 train *3 test*	7 4 train *3 test*
	Greedy			3 2 train *1 test*	64 38 train *26 test*	4 2 train *2 test*	63 38 train *25 test*	3 2 train *1 test*	63 38 train *25 test*	1 0 train *1 test*	0 0 train *0 test*

Table 5.2: Evaluation of L1-error on query plans selected using COMPASS cardinality estimates.

L1-error and Q-error exhibits overall improvement in identifying sub-optimal plans. The combined approach can provide a more comprehensive evaluation, considering both the absolute accuracy of individual estimates and their impact on query plan optimality.



(a) JOB workload. (b) JOB-light workload.

Figure 5.10: L1-error classifier accuracy on test data. Query plans are selected using COMPASS cardinality estimates.

5.4.2.2 L1-error Performance on COMPASS.

In order to evaluate L1-error on a different dimension, we collect true and estimated cardinalities from the COMPASS estimation for JOB and JOB-light workloads. In Table 5.2, we present the classifier performance identifying sub-optimal query plans selected by exhaustive and greedy enumeration algorithms using COMPASS cardinality estimates. As with PostgreSQL estimates, we observe similar class ratios – actual Positives and Negatives on columns 5 and 6 – in JOB and JOB-light. The predicted Positives and Negatives once again follow a similar pattern except for greedy enumeration in JOB. The number of estimated Positives is noticeably higher than the actual Positives. Thus, the classifier results in a higher FP while still maintaining a low FN. A closer scrutiny of the misclassified queries reveals a central tendency of query plans exhibiting P-error values around 1.0. In Figure 5.10, we compare the classifiers using L1-error, Q-error, and both performed, on JOB and JOB-light. As in PostgreSQL estimates, we observe a positive trend over Q-error. The combined classifier on L1-error and Q-error shows improved accuracy on the test data.

5.4.3 Summary

The experimental results can be summarized as follows:

- L1-error correctly classifies the optimality of query plans by following the trend and ratio between true sub-optimal and optimal query plans. The results contain only a small number of false negatives – the case when true sub-optimal plans are misclassified.
- The classifier based exclusively on L1-error identifies sub-optimal plans more accurately than the classifier that has Q-error as a feature. When having a combined feature consisting of both L1-error and Q-error, the best accuracy is achieved. These results prove that L1-error acts as an important feature both alone as well as in conjunction with Q-error.
- L1-error maintains its accuracy across multiple sets of cardinality estimates and workloads. This proves its generality both for different cardinality estimation synopses as well as across various datasets and queries.

5.5 Related Work

5.5.1 Cardinality Estimation Errors

The importance of each component within a query optimizer is widely acknowledged in comprehensive studies on query optimization [17, 69]. However, Leis et al. empirically prove that cardinality estimates hold paramount importance [63, 65]. They observe instances where cardinality miscalculations do not inevitably lead to sub-optimal query plans. This is because, provided the errors in misestimation are evenly distributed across a large portion of the estimates, sub-optimal query plans can be avoided. Consequently, estimation consistency can be more important than high accuracy, since consistency does not disrupt the ranking of query plans. Perron et al. [90] also confirm the importance of cardinality estimations as well as the importance of cardinality estimation of smaller joins than higher-level joins. Based on the JOB benchmark, their findings reveal that obtaining the true cardinalities up to four-way joins suffices to achieve near-optimal runtime performance. This highlights the need for accurate and consistent cardinality estimations, particularly for lower-level joins, in optimizing query performance.

5.5.2 Indicators of Sub-optimal Plans

Moerkotte et al. [81] introduce Q-error as a measure of individual join cardinality misestimation. At the query level, the authors propose leveraging the maximum Q-error across all sub-query cardinality estimates as a theoretical upper-bound to identify sub-optimal query plans. This serves as the upper-bound for P-error, which is the cost ratio between the selected and optimal plans, to denote the optimality of query plans. However, Han et al. highlight the limitations of the Q-error bound as an indicator of query plan optimality [32]. They observe that Q-error treats all cardinalities with equal weights, which is not always reflective of the realities of query optimizers. As a result, even in the presence of significantly large Q-error values, a query optimizer can still choose an optimal plan, and, conversely, a low Q-error is not always an indicator of an optimal plan.

5.6 Conclusions

We introduce L1-error, a novel indicator designed to identify sub-optimal join orders. L1-error emphasizes cardinality estimation errors that influence plan search algorithms, specifically those errors that disrupt the cardinality-based sorted order of sub-plans. Importantly, L1-error disregards estimation errors that do not bear any impact on the plan search algorithms. L1-error also takes into account that the cardinality estimates of earlier multi-way joins tend to be more accurate and critical than those of later joins. Our empirical results, across four different benchmarks, prove that as a standalone metric, L1-error can efficiently identify sub-optimal join orders in both moderate and complex queries.

Chapter 6

Conclusions and Future Work

In this chapter, we conclude the dissertation by reflecting on query optimization in relational database systems, and opportunities for future work on query optimization.

Online Sketch-based Query Optimization. We present COMPASS, an online sketch-based query optimizer that utilizes a single type of statistics, Fast-AGMS sketches, for both cardinality estimation and plan enumeration. The Fast-AGMS sketches are built per join attribute, pushing down the selection predicates and capitalizing on the advanced parallel execution engine inherent to modern database systems. This approach facilitates the building of sketches tailored to each specific query, while simultaneously capturing the current state of the data. During the query optimization phase, the Fast-AGMS sketches are incrementally composed on demand in the plan enumeration. We implement a prototype within the MapD database and conduct comprehensive experiments utilizing the JOB benchmark. The experiments demonstrate superior execution plan generation by COMPASS with reduced overhead. Specifically, COMPASS shows improved performance, surpassing the other four well-known databases across all the considered metrics within the JOB benchmark.

In this work, we show an efficient way of composing Fast-AGMS sketches to support multi-way joins. An interesting research direction can be to investigate alternative merging strategies for Fast-AGMS sketches. Another research direction is to explore SIMD-optimized sketch algorithms – for CPU and GPU – to gain lower overhead.

Spanning Tree-based Query Plan Enumeration. We approach query optimization through the lens of spanning trees, adjusting our aim towards identifying an ordered set of edges that encompasses all nodes within the join graph. We have tailored Prim's and Kruskal's spanning tree algorithms to our objective, incorporating them into ESTE. Capitalizing on the polynomial-time complexity of spanning tree algorithms, ESTE is proficient at systematically evaluating extensive and diverse segments of the search space. The experiments show that the advantage of ESTE translates into a significant enhancement in the quality of query plans. ESTE outperforms one of the most efficient heuristic algorithms currently available in terms of plan optimality while posing low overhead.

Randomized algorithms also have the potential to identify efficient plans out of query plans enumerated in the same amount of time required for ESTE. However, preserving the robustness and interpretability of the query optimizer becomes challenging. Nevertheless, analysis of plan quality by randomized and ESTE can be interesting to study. A research direction can be applying other spanning tree algorithms which enable exploring even more distinct areas of the search space. We believe this direction can leverage the application of spanning tree algorithms in the domain of query optimization.

Sub-optimal Join Order Identification with L1-error. We introduce L1-error, an indicator tailored to identify suboptimal join orders in query optimization. L1-error prioritizes cardinality estimation errors that affect plan search algorithms, particularly those that change the cardinality-based sorted order of subplans. L1-error ignores estimation errors that do not have any consequential impact on the plan enumeration algorithms. Moreover, L1-error utilizes the fact that the cardinality estimates of earlier multi-way joins are typically more precise and vital than those of later joins. Our empirical evaluation, utilizing four widely used benchmarks, has conclusively shown that L1-error operates effectively in identifying suboptimal join orders across a spectrum of query complexities.

Our findings indicate that L1-error is an efficient indicator for classifying suboptimal join order and can be employed in conjunction with Q-error. Utilizing both L1-error and Q-error in the post-evaluation of query optimizers, including learning-based optimizers to assess the efficacy of their trained models can be an interesting research direction.

Bibliography

[1] Apache Calcite. https://calcite.apache.org.

[2] MapD. https://www.heavy.ai.

[3] MonetDB. www.monetdb.org.

[4] MonetDB Ocelot. https://bitbucket.org/msaecker/monetdb-opencl/src/simple_mem_manager/.

[5] MonetDB Ocelot. https://bitbucket.org/msaecker/monetdb-opencl/src/simple_mem_manager/.

[6] MonetDB Ocelot. https://bitbucket.org/msaecker/monetdb-opencl/src/simple_mem_manager/.

[7] MonetDB Ocelot. https://bitbucket.org/msaecker/monetdb-opencl/src/simple_mem_manager/.

[8] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. In *PODS*, pages 10–20, 1999.

[9] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *STOC*, pages 20–29, 1996.

[10] J. Arndt. *Matters Computational: Ideas, Algorithms, Source Code.* Springer Science and Business Media, 2010.

[11] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, pages 261–272, 2000.

[12] P. Boncz. The IMDB Dataset. `http://homepages.cwi.nl/~boncz/job/imdb.tgz`.

[13] P. Boncz, A.-C. Anatiotis, and S. Klabe. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. *TPCTC*, pages 103—-119, 2017.

[14] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. *GPU-accelerated Database Systems: Survey and Open Challenges*, pages 1–35. 2014.

[15] N. Bruno, C. Galindo-Legaria, and M. Joshi. Polynomial Heuristics for Query Optimization. In *ICDE*, pages 589–600, 2010.

[16] W. Cai, M. Balazinska, and D. Suciu. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *SIGMOD*, pages 18–35, 2019.

[17] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, pages 34–43, 1998.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2022.

[19] G. Cormode and M. Garofalakis. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *PVLDB*, pages 13–24, 2005.

[20] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundation and Trends in Databases*, 4(1-3):1–294, 2012.

[21] D. DeHaan and F. W. Tompa. Optimal Top-Down Join Enumeration. In *SIGMOD*, pages 785–796, 2007.

[22] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[23] B. Ding, S. Chaudhuri, J. Gehrke, and V. Narasayya. DSB: A Decision Sup-

port Benchmark for Workload-Driven and Traditional Database Systems. *PVLDB*, 14(13):3376—-3388, 2021.

[24] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *SIGMOD*, pages 61–72, 2002.

[25] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-Based Multi-query Processing over Data Streams. In *EDBT*, pages 551–568, 2004.

[26] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. Narasayya, and S. Chaudhuri. Selectivity Estimation for Range Predicates Using Lightweight Models. *PVLDB*, 12(9):1044–1057, 2019.

[27] L. Fegaras. A New Heuristic for Optimizing Large Queries. In *DEXA*, pages 726–735, 1998.

[28] P. Fender and G. Moerkotte. A New, Highly Efficient, and Easy to Implement Top-Down Join Enumeration Algorithm. In *ICDE*, pages 864–875, 2011.

[29] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*, pages 1603–1618, 2018.

[30] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[31] I. Haffner and J. Dittrich. Efficiently Computing Join Orders with Heuristic Search. *SIGMOD*, 1(1), 2023.

[32] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *PVLDB*, 15(4):752–765, 2022.

[33] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Multi-Attribute Selectivity Estimation Using Deep Learning. *CoRR*, arXiv:1903.09999v2, 2019.

[34] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *TODS*, 34(4):1–39, 2009.

[35] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD*, pages 1477–1492, 2015.

[36] A. Hertzschuch. Simplicity Done Right. `https://github.com/axhertz/SimplicityDoneRight`.

[37] A. Hertzschuch, C. Hartmann, D. Habich, and W. Lehner. Simplicity Done Right for Join Ordering. In *CIDR*, 2021.

[38] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. DeepDB: Learn from Data, not from Queries! *PVLDB*, 13(7):992–1005, 2020.

[39] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[40] T. Ibaraki and T. Kameda. On the Optimal Nesting Order for Computing N-relational Joins. *TODS*, 9(3):482–502, 1984.

[41] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[42] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. *SIGMOD Record*, 20(2):268–277, 1991.

[43] Y. E. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. *SIGMOD*, 19(2):312–321, 1990.

[44] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. In *SIGMOD*, pages 9–22, 1987.

[45] Y. Izenov. The COMPASS Query Optimizer. `https://github.com/yizenov/compass_query_optimizer`.

[46] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. COMPASS: Online Sketch-based Query Optimization for In-memory Databases. In *SIGMOD*, pages 106–117, 2021.

[47] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. Online Sketch-based Query Optimization. *CoRR*, arXiv:2102.02440v1, 2021.

[48] N. Kabra and D. J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *SIGMOD*, pages 106–117, 1998.

[49] A. R. Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: Run-time Optimization of XQueries. In *SIGMOD*, pages 615–626, 2009.

[50] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[51] M. Kiefer. join-kde. `https://github.com/martinkiefer/join-kde`.

[52] M. Kiefer, M. Heimel, S. Breß, and V. Markl. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *PVLDB*, 10(13):2085–2096, 2017.

[53] A. Kipf. JOB-light Benchmark. `https://github.com/andreaskipf/learnedcardinalities/blob/master/workloads/job-light.sql`.

[54] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities:Estimating Correlated Joins with Deep Learning. In *CIDR*, 2019.

[55] A. Kipf, D. Vorona, J. Muller, T. Kipf, B. Radke, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Estimating Cardinalities with Deep Sketches. *CoRR*, arXiv:1904.08223v1, 2019.

[56] D. Kossmann and K. Stocker. Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. *TODS*, 25(1):43–82, 2000.

[57] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, pages 489–504, 2018.

[58] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *VLDB*, volume 86, pages 128–137, 1986.

[59] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR*, arXiv:1808.03196v2, 2018.

[60] R. Kumar and S. Vassilvitskii. Generalized Distances Between Rankings. In *Proceedings of the 19th International Conference on World Wide Web*, pages 571–580, 2010.

[61] H. Lan, Z. Bao, and Y. Peng. Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering*, 6:86–101, 2021.

[62] C. Lee, C.-S. Shih, and Y.-H. Chen. Optimizing large join queries using a graph-based approach. *IEEE Transactions on Knowledge and Data Engineering*, 13(2):298–315, 2001.

[63] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[64] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*, 2017.

[65] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDBJ*, 27:643–668, 2018.

[66] B. Li, Y. Lu, C. Wang, and S. Kandula. Q-error Bounds of Random Uniform Sampling for Cardinality Estimation. *CoRR*, arXiv:2108.02715v2, 2021.

[67] C. Lioma and N. D. Hansen. A Study of Metrics of Distance and Correlation between Ranked Lists for Compositionality Detection. *Cognitive Systems Research*, 44:40–49, 2017.

[68] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *CASCON*, pages 53–59, 2015.

[69] G. Lohman. Is Query Optimization a Solved Problem? `https://wp.sigmod.org/?p=1075`, 2014.

[70] T. Malik, R. C. Burns, and N. V. Chawla. A Black-Box Approach to Query Cardinality Estimation. In *CIDR*, 2007.

[71] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *VLDBJ*, 12(11), 2019.

[72] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM*, pages 1–4, 2018.

[73] R. Marcus and O. Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. *CoRR*, arXiv:1809.10212v2, 2018.

[74] V. Markl, G. M. Lohman, and V. Raman. LEO: An Autonomic Query Optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[75] A. Meister. GPU-Accelerated Join-Order Optimization. In *PhD Workshop at VLDB*, volume 176, page 1, 2015.

[76] A. Meister and G. Saake. Challenges for a GPU-Accelerated Dynamic Programming Approach for Join-Order Optimization. In *GI-Workshop on Foundations of Databases*, 2016.

[77] G. Moerkotte. Building Query Compilers. `https://pi3.informatik.unimannheim.de/~moer/querycompiler.pdf`.

[78] G. Moerkotte and P. Fender. Counter Strike: Generic Top-Down Join Enumeration for Hypergraphs. *PVLDB*, 6(14):1822–1833, 2013.

[79] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products. In *PVLDB*, pages 930–941, 2006.

[80] G. Moerkotte and T. Neumann. Dynamic Programming Strikes Back. In *SIGMOD*, pages 539–552, 2008.

[81] G. Moerkotte, T. Neumann, and G. Steidl. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB*, 2(1):982–993, 2009.

[82] M. Muller, G. Moerkotte, and O. Kolb. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *PVLDB*, 9(11):1016–1028, 2018.

[83] P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Flow-Loss: Learning Cardinality Estimates That Matter. *PVLDB*, 14(11):2019–2032, 2021.

[84] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *ICDE Workshop*, pages 154–157, 2020.

[85] T. Neumann. Query Simplification: Graceful Degradation for Join-Order Optimization. In *SIGMOD*, pages 403–414, 2009.

[86] T. Neumann and B. Radke. Adaptive Optimization of Very Large Join Queries. In *SIGMOD*, pages 677–692, 2018.

[87] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-Case Optimal Join Algorithms: [Extended Abstract]. In *PODS*, pages 37–48, 2012.

[88] Postgresql. `www.postgresql.org`.

[89] J. Ortiz, M. Balazinska, J. Gehrke, and S. Sathiya Keerthi. An Empirical Analysis of Deep Learning for Cardinality Estimation. *CoRR*, arXiv:1905.06425v2, 2019.

[90] M. Perron, Z. Shang, T. Kraska, and M. Stonebraker. How I Learned to Stop Worrying and Love Re-optimization. In *ICDE*, pages 1758–1761, 2019.

[91] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *PVLDB*, pages 486–495, 1997.

[92] G. Rahn. Join Order Benchmark (JOB). `https://github.com/gregrahn/join-order-benchmark`.

[93] P. Roy, A. Khan, and G. Alonso. Augmented Sketch: Faster and More Accurate Stream Processing. In *SIGMOD*, pages 1449–1463, 2016.

[94] F. Rusu. Sketches for Size of Join Estimation. `https://faculty.ucmerced.edu/frusu/Projects/Sketches`.

[95] F. Rusu and A. Dobra. Fast Range-Summable Random Variables for Efficient Aggregate Estimation. In *SIGMOD*, pages 193–204, 2006.

[96] F. Rusu and A. Dobra. Pseudo-Random Number Generation for Sketch-Based Estimations. *TODS*, 32(2), 2007.

[97] F. Rusu and A. Dobra. Statistical Analysis of Sketch Estimators. In *SIGMOD*, pages 187–198, 2007.

[98] F. Rusu and A. Dobra. Sketches for Size of Join Estimation. *TODS*, 33(15), 2008.

[99] F. Rusu and A. Dobra. Sketching Sampled Data Streams. In *ICDE*, pages 381–392, 2009.

[100] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[101] J. H. Shin, F. Rusu, and A. Suhan. Exact Selectivity Computation for Modern In-Memory Database Query Optimization. *CoRR*, arXiv:1901.01488, 2019.

[102] C. Spearman. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[103] StackExchange. Distance Between Two Permutations? `https://math.stackexchange.com/questions/2492954/distance-between-two-permutations`.

[104] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDBJ*, 6(3):191–208, 1997.

[105] C. Stylianopoulos, I. Walulya, M. Almgren, O. Landsiedel, and M. Papatriantafilou. Delegation Sketch: A Parallel Design with Support for Fast and Accurate Concurrent Operations. In *EuroSys*, 2020.

[106] A. Swami. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. In *SIGMOD*, pages 367–376, 1989.

[107] A. Swami and A. Gupta. Optimization of Large Join Queries. In *SIGMOD*, pages 8–17, 1988.

[108] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. SkinnerDB:

Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*, pages 1153–1170, 2019.

[109] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. *SIGMOD Record*, 25(2):35–46, 1996.

[110] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen. Join Size Estimation Subject to Filter Condition. *PVLDB*, 8(12):1530–1541, 2015.

[111] X. Wang, C. Qu, W. Wu, J. Wang, and Q. Zhou. Are We Ready For Learned Cardinality Estimation? *PVLDB*, 14(9):1640–1654, 2021.

[112] F. Wolf, M. Brendle, N. May, P. R. Willems, K.-U. Sattler, and M. Grossniklaus. Robustness Metrics for Relational Query Execution Plans. *PVLDB*, 11(11):1360–1372, 2018.

[113] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality Estimation with Local Deep Learning Models. In *aiDM*, pages 1–8, 2019.

[114] W. Wu. Sampling-Based Cardinality Estimation Algorithms: A Survey and An Empirical Evaluation. 2012.

[115] W. Wu, J. F. Naughton, and H. Singh. Sampling-Based Query Re-Optimization. In *SIGMOD*, pages 1721–1736, 2016.

[116] T. Yang, L. Liu, Y. Yan, M. Shahzad, Y. Shen, X. Li, B. Cui, and G. Xie. SF-sketch: A Two-stage Sketch for Data Streams. *CoRR*, arXiv:1701.04148v3, 2017.

[117] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB*, 14(1):61–73, 2021.

[118] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep Unsupervised Cardinality Estimation. *PVLDB*, 13(3):279–292, 2019.

[119] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Selectivity Estimation with Deep Likelihood Models. *CoRR*, arXiv:1905.04278v2, 2019.

[120] F. Yu, W. Hou, C. Luo, D. Che, and M. Zhu. CS2: A New Database Synopsis for Query Estimation. In *SIGMOD*, pages 469–480, 2013.

[121] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*, pages 1297–1308, 2020.

[122] R. Zhu, Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *PVLDB*, 14(9):1489–1502, 2021.