Simpli-Squared: Optimizing Without Cardinality Estimates

Asoke Datta, Brian Tsan, Yesdaulet Izenov, Florin Rusu
University of California Merced
{adatta2,btsan,yizenov,frusu}@ucmerced.edu

ABSTRACT

Most query optimizers rely on cardinality estimates to optimize their execution plans. Traditional databases such as PostgreSQL, Oracle, and Db2 utilize synopses, such as histograms, samples, and sketches. Recent main-memory databases like DuckDB and Heavy.AI often operate with minimal or even without estimates, yet their performance does not necessarily suffer. To the best of our knowledge, no analytical comparison has been conducted between optimizers with and without cardinality estimates. In this paper, we present a comprehensive analysis of optimizers that use cardinality estimates and those that do not. To represent optimizers that don't use cardinality estimates, we design a simple graph-based optimizer that only utilizes join types and table sizes. Our evaluation on the Join Order Benchmark reveals that cardinality estimates have a marginal impact in non-indexed settings, whereas inaccuracies in estimates can be detrimental in indexed settings. Furthermore, the impact of cardinality estimates is negligible in highly parallel main-memory databases.

CCS CONCEPTS

• **Information systems** → **Query optimization**; **Query planning**.

KEYWORDS

query optimization, cardinality estimation, join ordering

ACM Reference Format:
Asoke Datta, Brian Tsan, Yesdaulet Izenov, Florin Rusu. 2024. Simpli-Squared: Optimizing Without Cardinality Estimates. In *2nd Workshop on Simplicity in Management of Data (SiMoD '24), June 14, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3663351.3663879

1 INTRODUCTION

Accurate cardinality estimation is well known to be vital for query optimization. However, Leis et al. [14, 15] observed that the impact of cardinality estimation is negligible when the database only has primary key indexes and hash joins — even with inaccurate cardinality estimates. This outcome arises due to lack of indexes on the fact tables necessitating costly full-table scans, leaving little margin for improvement between an optimal and a suboptimal join order. Nonetheless, it remains essential to avoid large joins (i.e., foreign key/foreign key, many-to-many, or between fact tables) for which even inaccurate cardinalities likely suffice.

This work is licensed under a Creative Commons Attribution International 4.0 License.
SiMoD '24, June 14, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0671-4/24/06.
https://doi.org/10.1145/3663351.3663879

In proposing the Join Order Benchmark [15], Leis et al. note that *"when the database has only primary key indexes, and once nested-loop joins have been disabled and rehashing has been enabled, the performance of most queries is close to the one obtained using the true cardinalities,"* which remains true even with inaccurate cardinality estimates. What does this imply for query optimizers without cardinality estimation, e.g., HEAVY.AI? Given that newer versions of PostgreSQL now include parallel processing, does this observation remain valid as more computational power is afforded to database systems? In this paper, we explore the answer to these questions using the Join Order Benchmark (JOB). To represent methods that don't use estimates, we designed a simple graph-based method - Simpli-Squared (Simpli^2) - which only utilizes join type and table size information.

In query optimization, the impact of cardinality estimation is well-studied. Leis et al. conducted an exhaustive study of how inaccurate cardinality estimates can lead to suboptimal plans, emphasizing the need for accurate cardinality estimates. Building upon this, Lee et al. [13] investigated the phenomenon within an industrial database system (Microsoft SQL Server) by progressively increasing cardinality estimation errors. However, analyzing the performance gap between query optimizers with and without cardinality estimation remains largely unexplored.

We compare the performance of three different optimization strategies: optimization with cardinality estimates, without cardinality estimates, and with the true cardinalities provided by an oracle. Our goal is to address the following questions:

- How do different optimization strategies impact query cost and execution time?
- What is the floor for performance — how badly can optimizers with inaccurate or even without estimates perform?
- Does parallel processing make the significance of cardinality estimates negligible — to what degree?

Our experiment utilizes the Internet Movie Database (IMDB) dataset and Join Order Benchmark (JOB). These experiments were carried out across several database systems: PostgreSQL [28], MonetDB [26], DuckDB [24], and HEAVY.AI [25]. This diverse setup allows us to investigate the performance of different query optimization strategies under varied conditions. The source code and data for our experiments are available on GitHub [5].

2 CARDINALITY ESTIMATION (CE) BASED QUERY OPTIMIZATION

Cardinality Estimation(CE) based query optimization utilizes cardinality estimates to guide optimization decisions. This method employs various data synopses and statistics to identify the most efficient plan for executing a query. The optimizer leverages these cardinality estimates to make informed decisions regarding join order, physical operators, and indexes. For instance, if the optimizer

estimates that a specific join would generate a large number of rows, it may opt for a different join order to minimize the plan cardinality and improve performance. Likewise, during index selection, if the optimizer anticipates that a particular index will be highly selective and yield fewer rows, it may favor that index to enhance performance. Accurate cardinality estimation has the potential to boost query performance by allowing the optimizer to make better-informed decisions when executing a query. However, success depends on the precision of the statistics and data synopses.

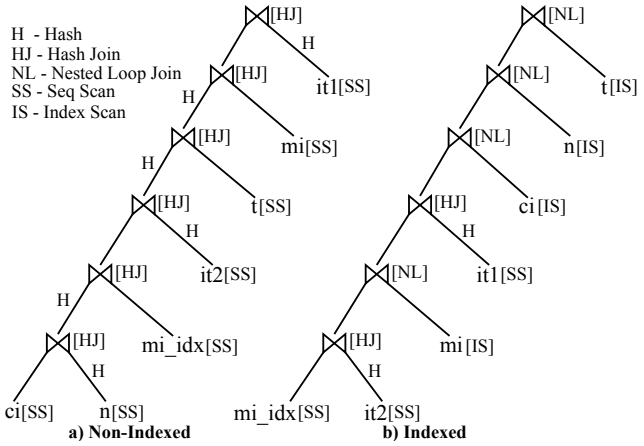


Figure 1: PostgreSQL execution plans for query 18a.

2.1 Non-Indexed

Many modern relational databases, such as **PostgreSQL**, employ a CE query optimizer [28]. The PostgreSQL optimizer leverages data and system statistics to estimate the cost of all potential join orders or plans, ultimately selecting the one with the lowest cost. Although the enumeration can benefit from join graphs by traversing only existing edges, the edge type — either primary/foreign key or foreign/foreign key — is not a primary consideration. Redundant and suboptimal orders are efficiently managed through dynamic programming and early pruning techniques.

A plan's cost is calculated by aggregating the cost of individual plan operators. The cost of an operator is contingent upon the number of accessed pages and processed tuples, with these quantities weighted by the configurable system parameters, `seq_page_cost` and `cpu_tuple_cost`, respectively. Exact costs are only known for base tables, necessitating estimations for all other operators without executing them. This constitutes the cardinality estimation problem in PostgreSQL.

Pre-computed data synopses or statistics are utilized for this purpose. PostgreSQL statistics are per attribute and include ranges, heavy hitters, the number of distinct values, and equi-depth histograms. Operator costs are estimated by integrating these statistics into formulas that assume uniformity, independence, and inclusion regarding the data [14, 15]. Inaccurate estimates result whenever these assumptions are violated, often in queries involving many-to-many joins or multiple predicates.

We execute JOB query 18a in PostgreSQL. To determine the optimal plan, PostgreSQL's enumeration algorithm exhaustively examines all potential join orders of the $ci, n, mi\_idx, it2, t, mi, it1$ tables and the available physical operators. For example, it evaluates permutations such as $(ci \bowtie n \bowtie mi\_idx \bowtie it2 \bowtie t \bowtie mi \bowtie it1)$, $((ci \bowtie n) \bowtie (mi\_idx \bowtie it2) \bowtie t \bowtie mi \bowtie it1)$, and others. In the non-indexed configuration, due to our main-memory setup, we constrain the join operators to hash join.

The enumeration algorithm also explores various options for building and probing hash tables. The hash table must fit in memory, which is why the smaller side of the join is typically selected for building the hash table. The optimal plan for query 18a is illustrated in Figure 1a, where the optimizer chooses to construct a hash table on the join result of $(ci \bowtie n)$ rather than on mi_idx. Subsequently, however, it builds the hash table on the base table $it2$ instead of the join result of $(ci \bowtie n \bowtie mi\_idx)$. This choice is made because, in certain cases, constructing a hash table on an intermediate join result can be more cost-effective than doing so on a base table.

2.2 Indexed

In the presence of indexes, the PostgreSQL optimizer follows a process akin to the one employed in non-indexed settings. The integration of indexes within a database system broadens the choices of available physical operators, as it offers alternative methods for data access beyond a full table scan. Instead of examining all records in a table to locate the desired information, the index facilitates direct access to pertinent data, thereby improving efficiency and overall performance.

Consider JOB query 18a with the optimal indexed join order: $mi\_idx \bowtie it2 \bowtie mi \bowtie it1 \bowtie ci \bowtie n \bowtie t$ which differs from the non-indexed setting(Figure 1b). In the presence of indexes, the optimizer determines if using an index will be beneficial, given that the cost of an index scan differs from that of a sequential scan. For example, if an index is available on the mi_idx table's join column ($info\_type\_id$) with $it2$ tables join column (id), PostgreSQL's cost function evaluates the cost of the join with the index (20K) and without the index (16K) and chooses the more cost-effective option, in this case, Sequential Scan followed by Hash Join. Conversely, for the $mi \bowtie mi\_idx$ join, the optimizer opts for an index scan on mi_idx in conjunction with a nested loop join, given the cost of the join using an index is 8K and without is 234K.

3 SIMPLI-SQUARED (Simpli^2)

Simpli^2, is designed to completely eschew the use of synopses, statistics, or cardinality estimations. Instead, it employs the query's join graph and key/foreign key constraints to guide its decision-making. While the incorporation of key/foreign key constraints in join optimization has precedents [3, 11], those prior approaches have typically combined such constraints with cardinality estimates, which is not the case here. Additionally, Simpli^2 takes table sizes as auxiliary input parameters. These inputs are used to annotate the vertices (tables) and edges (joins) of the join graph. Table vertices are annotated with their respective sizes, while join edges are classified as either one-to-many (1:n) or many-to-many (n:m) based on their key/foreign key constraints.

These heuristics can be intuitively understood in the context of the JOB schema. Many-to-many joins typically involve two large fact tables, while one-to-many joins occur between a fact table

Method	Statistics	Estimates	Join Graph Use	PostgreSQL Integration
PostgreSQL	distincts, heavy hitters, histograms	all	minimal	native
Pessimistic	Count-Min sketches, PostgreSQL	many-to-many joins	join type	estimates in optimizer
Simplicity	table sizes, heavy hitters, PostgreSQL	all	join type	query rewriting
COMPASS	Fast-AGMS sketches	joins	graph	query rewriting
Simpli^2	**table sizes**	**none**	**join type**	**query rewriting**

Table 1: Analytical comparison between few cardinality estimates approaches and Simpli^2.

and a significantly smaller table. By prioritizing one-to-many joins, the number of rows participating in the join from the fact tables is reduced. Furthermore, prioritizing smaller tables decreases the likelihood of generating large join results early in the join order.

Simpli^2 has two optimization objectives: minimizing the number of accessed tuples, and maximizing the utilization of available indexes. The cost of joining a foreign key table is determined by its cardinality when indexes are unavailable. When there are indexes, its cost is inversely proportional to the number of adjacent primary key tables. We express these objectives in the following formula.

$$C_{S2}(FK_m) = \begin{cases} |FK_m|, & \text{non-indexed} \\ \frac{|FK_m|}{2^{T_{FK_m}}}, & \text{indexed} \end{cases}$$

Where FK_m is a foreign key table and T_{FK_m} is the number of primary key tables adjacent to FK_m. The cost of FK_m is calculated differently depending on the presence of indexes. The join order of a query is determined by sorting it's foreign key tables in ascending order of their cost.

3.1 Non-Indexed

The Simpli^2 algorithm is outlined in Algorithm 1. It starts by identifying the foreign key tables and their primary key join candidates, splitting the join graph into multiple – possibly overlapping – components. The join order is determined by sorting the foreign key tables using C_{S2}, their cardinalities, in ascending order. Each foreign key table is appended to the join order, followed by its component. The primary key tables in each component is appended to the join order in ascending order of their cardinality. If the same primary key table is present in multiple components, it will only be appended to the join order with the component where it initially appears. Any remaining tables not yet part of the join order are subsequently appended. Each component in the join graph is treated as a subquery, with the exception of the first one, resulting in a bushy join order.

We execute JOB query 18a with the Simpli^2 algorithm. We compute the components for each foreign key table: *mi_idx*, *mi*, and *ci*. The components are then combined, resulting in the join order depicted in Figure 2 as a bushy tree.

3.2 Indexed

In the presence of indexes, assuming each table has at least one index, the Simpli^2 algorithm has the additional objective of maximizing the utilization of those indexes. While the algorithm itself remains the same, the cost determined by C_{S2} differs from the non-indexed setting. To calculate the cost of a foreign key table with indexes, C_{S2} divides the table's cardinality $|FK_m|$ by $2^{T_{FK_m}}$ where T_{FK_m} represents the number of adjacent primary key tables. Thus,

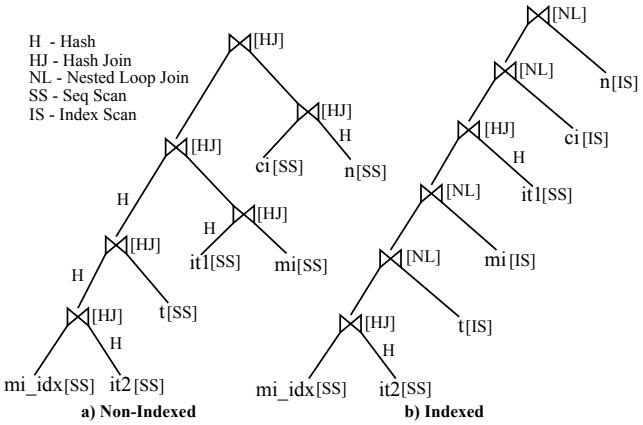


Figure 2: *Simpli^2* execution plans for query 18a.

foreign key tables with more adjacent primary key tables are prioritized in the join order. Bushy trees, which minimize access to base tables later in the join order and subsequently reduce index usage, are counterproductive to our objective. To address this, we default to left-deep trees when there are indexes. The impact of these modifications on query 18a's join order is depicted in Figure 2. Additionally, the choice of physical operators is adapted to benefit from indexes. For instance, index scans are preferred over sequential scans, and indexed nested loop joins are preferred over hash joins whenever possible.

Following Hertzschuch et al. [11], we rewrite queries using the explicit join order determined by Simpli^2 algorithm. In a non-indexed setting, we construct a subquery for each join component – except the first. When executing a rewritten query with a database system, e.g., PostgreSQL, the optimizer must be configured to follow the given join order and disable subquery unnesting. We leave the selection of physical operators up to the optimizer's default, which may use statistics when indexes are available. Simpli^2 operates independently of cardinality estimates, making it a viable standalone option for databases with limited or no reliance on cardinality estimates. Additionally, Simpli^2 can be used to create query plans on refined search space produced using methods like LIP [22].

4 EXPERIMENTAL EVALUATION

We experiment on both indexed and non-indexed configurations in PostgreSQL [28]. In the indexed setting, we disable sort-merge join, while in the non-indexed setting, sort-merge and nested-loop joins are both disabled. Hash joins are preferred in main-memory and non-indexed setups to ensure optimal performance [14, 15].

Algorithm 1: Simpli-Squared (Simpli^2)

Data: Query join-graph $G(V, E)$ where edges are key/foreign key constraints
Result: Join-order JO, initialized to an empty list

$FK \leftarrow$ list of all foreign key tables from V
Sort FK in ascending order by C_{S2}
for $f \in FK$ **do**
  $FK_c \leftarrow$ list of primary key tables adjacent to f
  Sort FK_c in ascending order by cardinality
  Append f to JO
  for $c \in FK_c$ **do**
    if $c \notin JO$ **then**
      Append c to JO
    end
  end
  note : In non-indexed setting, each $f \in FK$ produces separate sub-query after it joins one or more primary key table
end
while $\exists$ *table* $c \in V \ni c \notin JO$ **do**
  if $\exists c' \in JO \ni (c, c') \in E$ **then**
    Append c to JO
  end
end

The motivation for these can be found in the extended versions of this work [6, 7]. In addition, our analysis encompasses results from DuckDB [24] and HEAVY.AI [25], both of which operate as main-memory database systems. Furthermore, we extend our experiments to MonetDB [26], which utilizes a columnar storage architecture and mainly relies on main memory for data processing.

4.1 Setup

In alignment with prior work [14, 15], we specifically configure PostgreSQL by increasing the memory limit per operator `work_mem` from 4 megabytes to 128 gigabytes, `shared_buffers` from 32 megabytes to 128 gigabytes, and `effective_cache_size` from 4 gigabytes to 128 gigabytes. We raise the `geqo_threshold` parameter to 18 joins, which adjusts the threshold for switching from dynamic programming to heuristic search. Lastly, we set the query optimizer not to reorder joins or unnest subqueries for Simpli^2 by setting `from_collapse_limit` and `join_collapse_limit` to 1.

Hardware. Each database is configured within its own Ubuntu 20.04 LTS Docker container. These containers are deployed on a machine featuring an Intel Xeon E5-2660 v4 (2.00GHz) processor with 28 CPU cores, 256 GB of RAM, and HDD storage.

Dataset. Many prior works on query processing and optimization use standard benchmarks like TPC-H, TPC-DS, or the Star Schema Benchmark (SSB) [2, 16, 18]. However, Leis et al. [15] argue that, while these benchmarks serve well in assessing the performance of query engines, they are not good for evaluating query optimizers. We use JOB (Join Order Benchmark) [19] derived from the IMDB (Internet Movie Database) dataset [1], a widely recognized benchmark for assessing query optimizer performance in real-world scenarios. JOB consists of 113 queries of varying complexity with up to 28 join predicates. The IMDB dataset features skewed attributes and cross-table correlations.

Methodology. In our research methodology, we have structured the experimental evaluation into two distinct phases to thoroughly assess the performance characteristics of systems with and without cardinality estimates. In the first phase, we compare methods that utilize cardinality estimation — including PostgreSQL [28], Pessimistic [3], Simplicity [11], Compass [12] — and that doesn't — Simpli^2. This comparison is conducted within the context of the Join Order Benchmark (JOB). Each method is run independently to collect their join orders and explicit query statements, which are then executed in PostgreSQL to measure runtime performance. Building on the insights gained from the first phase, we delve deeper to better understand the performance traits between systems with and without estimates. PostgreSQL is chosen to present the cardinality estimation based approach, while Simpli^2 illustrates the systems where cardinality estimation is absent. To anchor our analysis, we utilize TrueCard, a modified version of PostgreSQL calibrated to utilize actual cardinalities instead of estimates, setting it as our comparative baseline. Initially, we examine these systems on a per-query basis, assessing both cost and runtime. Then, we collect 10k random plans for three queries of different complexity using the quickpick algorithm [20] and analyze their cost and runtime compared with the TrueCard plan, assuming that's the best plan we can get. Finally, we analyze how the plan is produced using different optimization strategies that benefit from multi-threaded and main-memory settings.

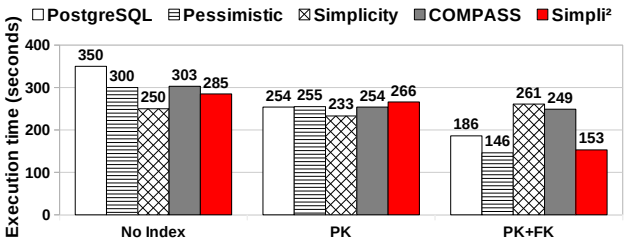


Figure 3: Execution time for JOB queries in PostgreSQL with various index configurations: No Index, Primary Key (PK) Only, and Primary + Foreign Key (PK+FK).

4.2 Results

For phase 1, we conduct experiments in three configurations - configuration with no index, primary key index(PK), and primary and foreign key (PK+FK) - for every method. We report the median between five runs. Figure 3 depicts the JOB workload runtime. In the following, we analyze the results organized by configurations.

Non-Indexed. In the non-index configuration, where database systems rely on full table scans, the execution time of queries is mainly influenced by both the order and type of joins employed. Figure 3 depicts the performance of various query optimization methods in a non-indexed setting. Among these methods, the standard PostgreSQL takes the longest execution time of 350 seconds

to run all the queries from JOB. This performance is primarily due to inaccurate estimates that lead to suboptimal join order decisions. Simplicity performs marginally better compared with other methods. This performance is attributed to its efficient use of heuristics, which, when combined with upper bounds and functional dependencies, find better join orders in most cases.

The Pessimistic and COMPASS methods show comparable performances, with times of 300 and 303 seconds, respectively, suggesting they possess optimizations that marginally improve over the baseline PostgreSQL. Simpli^2 registers a moderately better execution time at 285 seconds, standing out as a middle ground between the efficiency of Simplicity and other approaches. These observations collectively underscore the impact of different query optimization techniques, which leads to marginal improvement comparable with the baseline in the absence of indexes.

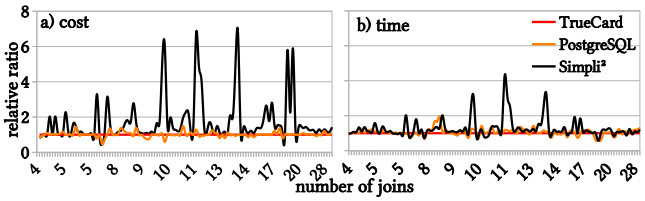


Figure 4: Cost and time [non-indexed setting] — as a normalized ratio to TrueCard.

Indexed. The impact of indexing on query performance is profound, as evidenced in figure 3. Including primary key indexes alone leads to noticeable enhancements in performance across all evaluated methods, compared to scenarios without indexing. Specifically, the Simplicity method slightly outperforms its counterparts. However, with only primary key indexes at play, the systems are compelled to conduct full-table scans for foreign keys, a factor that predominantly dictates the overall query runtime. This necessity for full-table scans on foreign keys explains limited variation in runtime across different methods.

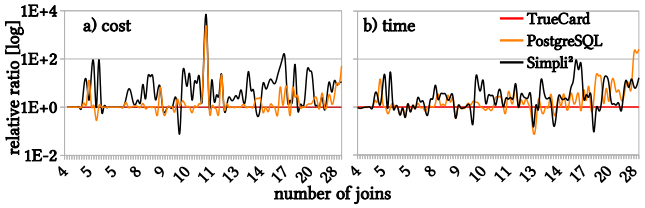


Figure 5: Cost and time [indexed setting] — as a normalized ratio to TrueCard.

Including foreign key indexes alongside primary key indexes markedly impacts the performance outcomes among the evaluated methods. Specifically, Pessimistic, PostgreSQL, and Simpli^2 see their runtimes improve by factors of 1.42x, 1.26x, and 1.42x, respectively. This enhancement suggests that these systems are finely tuned for environments rich in indexing, leveraging the available indexes to formulate query execution plans that optimize the utilization of these indexes.

In contrast, the Simplicity and COMPASS do not exhibit any benefit from the availability of foreign key indexes. This discrepancy can be attributed to the inherent design of these two approaches, which do not factor in indexes during the optimization phase. Instead, their focus is primarily on optimizing the order of joins.

5 ANALYSIS OF RESULTS

In phase 1, we observed marginal differences in performance across different systems in both non-indexed and primary key indexed settings. This outcome is intriguing and worth investigating further to understand the reason behind this behavior. Given the statistics-free nature of Simpli^2, it is expected to perform worse. However, this wasn't clearly the case in Figure 3.

In this section, we dive deeper and try to understand the reason behind it. We begin by comparing Simpli^2 with PostgreSQL and TrueCard, focusing on their performance at the query level in terms of cost and runtime 5.1. We assume TrueCard, which relies on actual cardinality, should theoretically deliver the best plans and fastest execution times. Using TrueCard as a baseline helps us better understand the performance differences when actual, estimated, or no cardinality estimates are used. Given no index setting and setting with only primary key indexes exhibit similar trends in performance, for the rest of the experiments, we used a non-indexed setting to represent without an index setup and an indexed setting to represent a setup where both primary and foreign key indexes are present. And, for non-indexed setting, hash join is used exclusively as it yields more consistent performance and mitigates the potential for selecting suboptimal join operators as a consequence of cardinality estimation errors [7, 15].

Next, in section 5.2, we examine whether Simpli^2's performance is random or driven by specific factors. We do this by analyzing 10,000 random query plans generated using the quickpick algorithm, then comparing these results with the plans produced by PostgreSQL, TrueCard, and Simpli^2.

Lastly, in section 5.3, we assess how parallel processing affects the query runtime for both Simpli^2 and PostgreSQL in these different settings, aiming to understand the role of parallel execution in their performance.

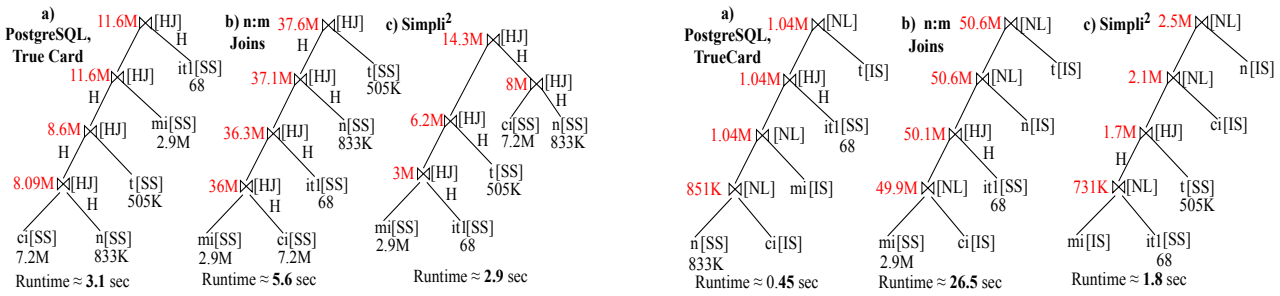
5.1 Query-level Analysis

In this subsection, we conduct a comparative analysis of TrueCard, PostgreSQL, and Simpli^2 within the PostgreSQL framework, focusing on both cost and runtime. Join orders from Simpli^2 are integrated into the PostgreSQL system via query rewriting. Exact cardinalities for TrueCard are collected by executing all subqueries for each query, which are then injected during query runtime. The cost of each query is calculated using the cost function defined in [15]. Although PostgreSQL's default cost function was an option, we opted for Leis' [15] cost function due to its suitability for main-memory setups focused on operator cardinality. Query runtime was collected by running each query five times and taking its median. These results are illustrated in Figures 4 and 5.

Non-Indexed. We evaluate the costs of JOB plans for PostgreSQL, Simpli^2, and TrueCard, with the cost ratio relative to TrueCard depicted in Figure 4a. The cost of PostgreSQL plans are usually similar to TrueCard's. Even in the worst case, they are up to 1.5

		TrueCard (ratio)	PostgreSQL (ratio)	Simpli^2 (ratio)
non-indexed	cost(million)	994.8 (1)	1006.6 (1.01)	1680.4 (1.68)
	runtime(seconds)	324.6 (1)	349.4 (1.07)	416.3 (1.28)
indexed	cost(million)	257.03 (1)	290.02(1.13)	1041.6(4.05)
	runtime(seconds)	102.3 (1)	192.5 (1.88)	309.2(3.02)

Table 2: Summary of aggregated cost (in millions) and runtime (in seconds) relative to TrueCard.



(a) Non-indexed **(b) Indexed**

Figure 6: Execution plans for a representative sub-query extracted from query JOB 18a.

times higher. The cost of Simpli^2 plans is also similar to TrueCard's, except for a few instances where they can be as much as 6.5 times higher. This is due to the bushy join structure of Simpli^2 method, which generates more intermediate join results compared with a linear structure, i.e., left-deep. In contrast, the join orders utilized by TrueCard and PostgreSQL are primarily left-deep, minimizing the production of join results.

While the cost of a query plan can indicate its execution time, it is not always an accurate predictor. Factors like database configuration and available resources also influence runtime but may not be accurately represented in the cost model. The cumulative cost ratio of JOB plans between PostgreSQL and TrueCard is 1.01, while the runtime ratio is 1.08 – which is 8% slower than TrueCard. Conversely, the cost ratio between Simpli^2 and TrueCard is 1.69, yet the runtime ratio is 1.28 – 28% slower than TrueCard. Overall, the cost function tends to overestimate.

Indexed. Figure 5a depicts the impact of indexing on query plan costs. In the most unfavorable scenarios, the cost of plans using cardinality estimates (PostgreSQL) can escalate to as much as 1370 times greater than those using TrueCard's exact cardinalities. However, the ratio of these costs exhibits considerable variability across different plans, more so than in non-indexed settings. While some plans do incur elevated costs, the aggregate workload cost for PostgreSQL plans is merely 1.13 times that of TrueCard. This is primarily due to the majority of plans having costs similar to, or occasionally even lower than, TrueCard's, thus counterbalancing the effects of costlier outliers. No outliers were observed regarding query runtime, and the cumulative workload runtime for PostgreSQL plans is 1.88 times that of TrueCard's.

In extreme scenarios, Simpli^2 plans can incur costs that are up to 3,000 times greater than TrueCard. Despite being optimized for maximum index utilization, the absence of cardinality estimates often leads to suboptimal query plans. This manifests as a cumulative workload cost that is 4.05 times higher when using Simpli^2 plans, compared to TrueCard, while total runtime is 3.02 times more than those of TrueCard. However, for certain complex queries involving multiple joins, Simpli^2 surprisingly outperforms TrueCard in terms of runtime. This advantage is likely due to the relatively low optimization time required by Simpli^2, even for complex, join-heavy queries. In contrast, plans utilizing cardinality estimates (PostgreSQL) and TrueCard necessitate additional time for optimization, especially when more joins are involved, consequently affecting query runtime.

5.2 Good Plans Despite Bad/No Cardinalities

Table 2 presents an aggregated analysis of costs and runtimes for the Join Order Benchmark, comparing the TrueCard method with PostgreSQL and Simpli^2 in indexed and non-indexed settings. In non-indexed settings, the PostgreSQL and Simpli^2 costs are 1.01 and 1.68 times that of TrueCard, respectively, while runtimes are 1.07 and 1.28 times greater. Despite the potential inaccuracies in cardinality estimation and the absence of any estimates in the Simpli^2 approach, the performance of PostgreSQL and Simpli^2 closely aligns with the TrueCard baseline. Moreover, the performance disparity among the methods is less pronounced in non-indexed settings compared to indexed ones. A detailed examination of Figure 4 reveals that, for the majority of queries, all methods under comparison perform similarly, barring a few outliers. Contrary to expectations that, inaccuracies in cardinality estimates(PostgreSQL) and the absence of cardinality estimates in Simpli^2 would lead to sub-optimal execution plans and extended runtimes. Our observations do not substantiate this in most cases. This subsection delves into the underlying reasons for this observation.

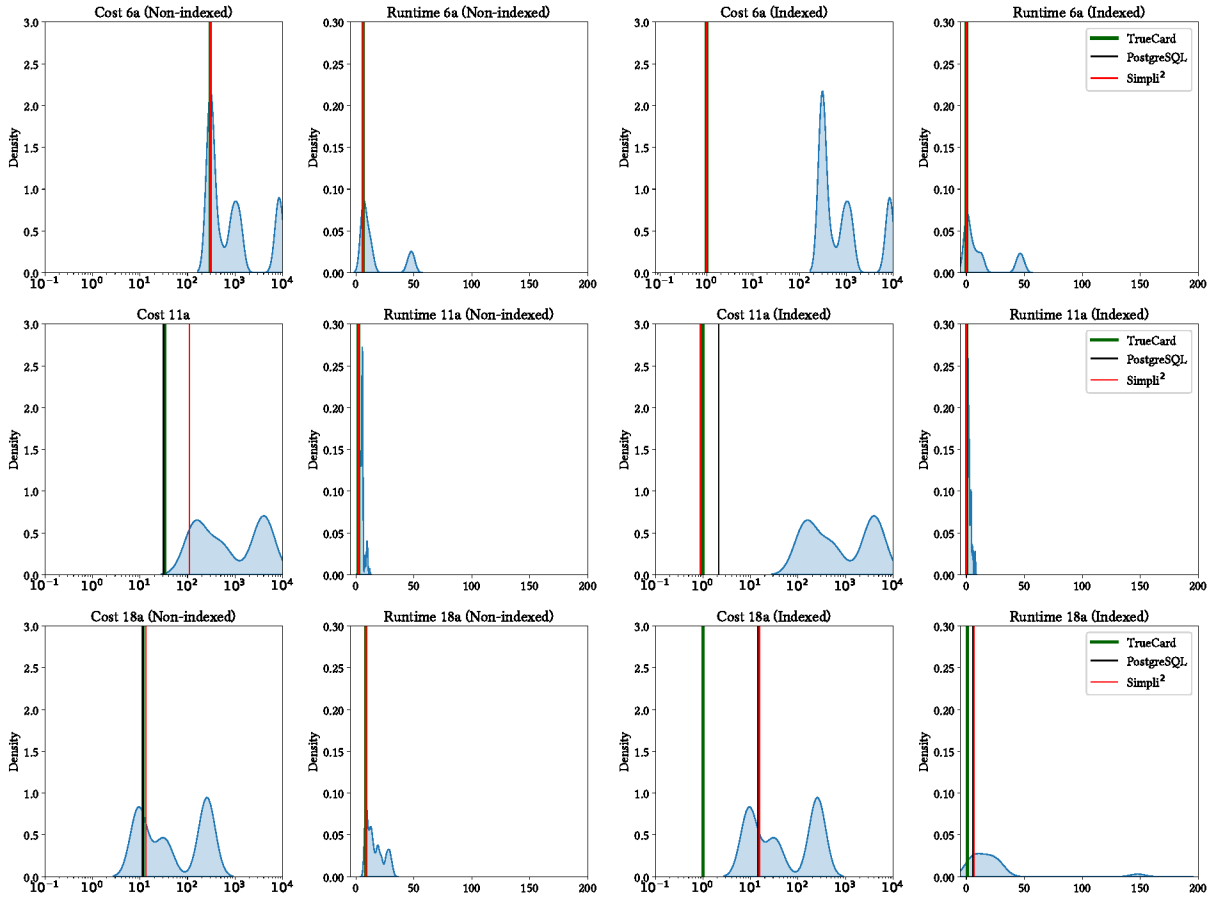


Figure 7: Cost (log-scale) and runtime (seconds) distribution for query 6a, 11a and 18a relative to TrueCard.

In our study, we generated 10,000 random plans for queries 6a, 11a, and 18a from the Join Order Benchmark (JOB) [14, 15] using the QuickPick algorithm [20]. These plans are analyzed and visualized in Figure 7, where we display the distributions of plan costs and runtimes in both non-indexed and indexed settings. Each row in the figure depicts the cost and runtime distributions for a given query under both non-indexed and indexed settings. The costs are normalized against the cost of TrueCard's indexed plan and are represented on a logarithmic scale. In each subplot, we highlight the costs or runtimes of plans derived from TrueCard, PostgreSQL, and Simpli^2 approaches with vertical lines in green, black, and red, respectively. The vertical lines coinciding indicate instances where these methods yield comparable costs or runtimes, as observed in sub-figure 7 a), which focuses on the cost distribution for query 6a.

Non-Indexed. Figure 7, column a, represents the cost distribution of all considered queries in a non-indexed setting. Similar cost of plans formed clusters. There are multiple such clusters of plans for the same query. For instance, query 6a demonstrates three distinct clusters of such plans. The cluster adjacent to the green line represents plans that are closest in cost to the optimal plan, whereas clusters to the right indicate progressively costlier plans. A closer inspection of the plans in the third cluster — the one farthest from the optimal cost — revealed a common pattern: many either commence with or incorporate a foreign key/foreign key join. This pattern persists across other queries as well.

Figure 6a focuses on a specific subquery from query 18a to illustrate the effect of foreign key/foreign key joins on cost. It demonstrates that plans commencing with a join between foreign keys – for example, `ci` and `mi` in this context – tend to incur higher costs by generating large intermediate results early in the execution path. Contrastingly, in the other two scenarios analyzed within a non-indexed setting, such cost inflations do not occur.

Minimizing the occurrence of foreign key/foreign key joins could reduce the likelihood of generating plans that fall within this third, less cost-effective cluster for query 6a – figure 7. Leis et al. [14, 15] argued that cardinality estimates are generally good for avoiding such joins. However, our research extends this discourse by illustrating how a Simpli^2 approach can successfully avoid these joins even in the absence of cardinality estimates.

Column b of Figure 7, which examines runtime, reflects a pattern analogous to that of cost, albeit with generally reduced distances between the clusters. This is particularly evident in the runtime distributions for queries 11a(b) and 18a(b), where the clusters are more

method	setting	<0 %	0.1 - 39 %	40-94 %	improvement percentage
PostgreSQL	non-indexed	0	0	100	% of queries
		0	0	627.04	runtime decrease in seconds
	indexed	62	13.20	23.90	% of queries
		-12.50	1.96	48.78	runtime decrease in seconds
Simpli^2	non-indexed	0	1.76	98.23	% of queries
		0	1.48	748.82	runtime decrease in seconds
	indexed	39.80	6.10	54	% of queries
		-2.31	0.81	490.3	runtime decrease in seconds

Table 3: Variation in query execution time (seconds) as a function of the thread count increment from 1(t1) to 5(t5), with the proportion of queries per category represented as a percentage of the total 113 JOB queries.

tightly grouped compared to their cost counterparts, suggesting a lesser variance in the runtime efficiency of the plans.

Indexed. In Figure 7, column c depicts the cost distribution of all considered queries in an indexed setting. Similar to non-indexed settings, similar plans cost form clusters for all queries. Notably, in the indexed context, these clusters tend to be positioned significantly to the right of the green line, representing the TrueCard cost. This displacement suggests that the randomly selected plans are generally associated with substantially higher costs than the TrueCard plan, a trend consistent across the queries studied. We assume that this phenomenon is largely attributable to the selection of physical operators, where errors in cardinality estimates can lead to sub-optimal choices, adversely affecting the cost of the plans. Conversely, in non-indexed settings, the choice of join physical operator was constrained to hash joins.

In the case of query 11a, the PostgreSQL demonstrates a higher cost than the TrueCard. With query 18a, both the PostgreSQL and Simpli^2 methods result in costs that exceed those associated with TrueCard. This underscores the pivotal role of precise cardinality estimates in settings where indexes are employed. Throughout our experiment in the indexed setting, we fixed the join order in advance; however, the choice of physical operators was left to the discretion of the database management system, which in our experiments was PostgreSQL.

Column d of Figure 7 presents the runtime analysis for the respective queries. The runtimes of similar plans tend to form clusters. Notably, multiple clusters are evident within the results for queries 6a and 18a. For query 11a, the clusters are more closely spaced, indicating a smaller variation in runtimes among the different plans.

Observations. We observe that plans with comparable cost form clusters, and the same pattern holds for runtime. In no-indexed settings, columns a and b in Figure 7, TrueCard, PostgreSQL, and Simpli^2 yield plans that fall into the same cluster, indicating little to no variation in the plans cost and runtime. We also observe that plans with significantly higher costs than TrueCard often exhibit a common pattern - many either start with or incorporate a foreign key/foreign key join. Avoiding foreign key/foreign key join whenever possible reduces the likelihood of generating plans with higher cost and runtime. Simpli^2 effectively sidesteps foreign key/foreign key join whenever possible by prioritizing primary key/foreign key joins. However, in scenarios with indexes, plans derived from actual and estimated cardinalities tend to outperform those generated without such estimates.

5.3 Parallel Processing and Runtime

Contemporary query execution engines, including PostgreSQL version 9.6 and later, have embraced the capability for parallel processing. The degree of parallelism within a given query in PostgreSQL is governed by two parameters: `max_parallel_workers` and `max_parallel_workers_per_gather`. During the query optimization phase, PostgreSQL evaluates both parallel and sequential execution plans, in contrast to some other databases that may allocate threads dynamically at execution time.

In this section, we examine the impact of thread concurrency on query runtime, comparing the performance in both indexed and non-indexed settings. We analyze the proportional disparity in runtime between multi-threaded and single-threaded (t1) executions as depicted in Table 3, 4. We further investigate to identify which queries benefit from parallel execution and which do not for both the PostgreSQL and Simpli^2 methods. Our experiments span a range from single-threaded (t1) to five-threaded (t5) execution, with the empirical finding that extending beyond five threads yields negligible performance improvements when using the PostgreSQL execution engine.

Non-Indexed. In the non-indexed environment, as demonstrated in Table 3, we note consistent improvement in the execution times of all queries, with speed-ups ranging from 40% to 94% when the number of operating threads is increased from a single thread (t1) to five threads (t5). The performance variance is quantified using the following formula.

$$\text{Performance Change} = \left(\frac{\text{Runtime}_{t1} - \text{Runtime}_{t5}}{\text{Runtime}_{t1}} \right) \times 100\%$$

Transitioning from a single-threaded(t1) to a five-threaded(t5) environment results in a runtime reduction of 627.04 seconds for PostgreSQL and approximately 750.3 seconds for Simpli^2 as detailed in Table 3. This increase in efficacy primarily stems from the adept parallelization of physical operations, particularly in Sequential Scans and Hash Joins, within non-indexed scenarios. These findings underscore that in the absence of indexes, an elevation in thread count exerts a positive impact on query runtimes, though the magnitude of acceleration varies across individual queries.

method	improvement	threads	Index Scans	Heap Scan	Seq Scan	Nested Loop	Hash Join
PostgreSQL	(a) <0 (71 queries)	1 (t1)	450	66	209	573	15
		5 (t5)	441	61	211	569	19
	(b) >0 (42 queries)	1 (t1)	212	22	106	231	45
		5 (t5)	218	11	107	214	62
Simpli^2	(a) <0 (45 queries)	1 (t1)	294	29	116	344	21
		5 (t5)	290	24	120	339	26
	(b) >0 (68 queries)	1 (t1)	304	14	263	320	179
		5 (t5)	302	2	265	300	199

Table 4: Impact of thread count on operator statistics for queries with improvement and decline in execution times.

Indexed. In an indexed configuration, the comparative analysis between a five-threaded (t5) and a single-threaded (t1) execution yields a diverse spectrum of results. Specifically, while using PostgreSQL, 62% of queries experience a marginal degradation in performance, 13.20% exhibit negligible changes, and a noteworthy enhancement in runtime – exceeding 40% – is observed in 23.90% of queries. Similarly, while using Simpli^2 approach, a marginal performance decrement is observed in 39.80% of queries, while 6.10% remain essentially stable, and an improvement exceeding 40% is discernible in 54% of queries. This result indicates, for both PostgreSQL and Simpli^2 fails to exhibit performance improvements for lot of queries, a pattern that is documented in Table 3. To elucidate the underlying dynamics at play, we turn our attention to the operator statistics presented in Table 4.

Table 4 categorizes the JOB workload into two distinct groups based on performance trajectory: (a) <0, where performance wanes, and (b) >0, where performance is improved relative to a single-threaded baseline (t1). Within the first category (a), the adoption of physical operators remains invariant from t1 through t5. Index Scans are the preferable choice for scan operation in most cases, followed by Sequential Scans. Concurrently, join operations are primarily dominated by Nested Loop Joins for both thread configurations t1 and t5. The invocation of Hash Join operations is relatively infrequent, registering at 15 instances for t1 and 19 for t5. We observe a similar trend for the Simpli^2 approach.

Category (b) reflects a pattern akin to category (a), with Index Scans and Nested Loop Joins being the prevalent methods for scan and join operations, respectively. However, category (b) distinguishes itself by a greater incidence of Hash Joins—constituting 22% of joins in PostgreSQL (t5) and 40% in Simpli^2 (t5) — figures that surpass those observed in category (a), where Hash Joins account for a mere 3% in PostgreSQL and 7% in Simpli^2. In PostgreSQL (t5) setting, a notable 23.9% of JOB queries exhibit performance gains ranging from 40% to 94% over the single-threaded baseline (t1), culminating in a runtime reduction of 48.78 seconds. In contrast, Simpli^2 witnesses 54% of queries outperforming t1 by 40% to 94%, resulting in a substantial runtime decrease of 491 seconds.

Despite the inherent capability of all operators listed in Table 4 to capitalize on parallel processing, category (a) does not demonstrate any performance gains, whereas category (b) shows a moderate improvement for PostgreSQL and a more pronounced enhancement for Simpli^2. We assume that the observed performance improvements are attributable to the heightened utilization of hash join operations, which appear to be more amenable to parallel execution.

5.4 In-memory Experiments

To ensure a comprehensive evaluation, our study includes analysis in modern main-memory databases, specifically in DuckDB, HEAVY.AI, and MonetDB, a columnar database. While MonetDB does not offer guaranteed join ordering, our analysis aligns with prior work [11] and incorporates its findings. We present the execution times for non-indexed configurations in Table 5. We omitted the indexed settings from our results due to the absence of comprehensive index support in HEAVY.AI, coupled with minimal or no performance gains observed in DuckDB and MonetDB when utilizing indexes. Our evaluation considers both the default runtime for each database system, indicated as "default" in Table 5, as well as execution plans chosen by the Simpli^2 algorithm.

DuckDB utilizes a basic query optimizer, considering join types and column distinct value counts. In our experiments, DuckDB executed the Join Order Benchmark (JOB) workload in 30.38 seconds, while Simpli^2's duration was 37.89 seconds. HEAVY.AI struggled with the same workload, timing out on 37 out of 113 queries based on a 120-second threshold. Accounting for these timeouts by assigning a 120-second penalty for each, the cumulative time for the JOB workload amounted to 4744 seconds. When we run Simpli^2 queries in HEAVY.AI, the JOB workload completion time was 282.50 seconds. MonetDB, on the other hand, employs a more advanced, cost-based optimizer that utilizes both statistics and heuristics, thereby distinguishing itself from both DuckDB and HEAVY.AI. In our tests, MonetDB processed the JOB queries in 114.46 seconds, when augmented with Simpli^2 method, the time was further reduced to 88.79 seconds.

In the context of main-memory databases, we find that the performance of the Simpli^2 method is on par with the default optimization strategies while using Join Order Benchmark. This observation is likely attributable to the databases minimal to no reliance on cardinality estimates during the optimization process.

6 RELATED WORK

Cardinality estimation is crucial for determining optimal query execution plans. Leis et al. [14, 15] explored the impact of cardinality

	default	Simpli^2
DuckDB	30.38	37.87
HEAVY.AI	>4744	282.50
MonetDB	114.46	88.79

Table 5: Execution time (seconds) in DuckDB, HEAVY.AI, and MonetDB.

estimates on query optimization. Estimates often suffer from inaccuracies due to oversimplified assumptions like uniformity, independence, and inclusion. PostgreSQL [28] uses histograms for data representation, relying on formulas based on these assumptions. Although histograms work well for single attribute estimations, they struggle with join-crossing correlations. Cai et al. [3] introduced Pessimistic, employing count-min sketches to capture foreign key join-crossing correlations, but the sketch-building process introduces significant overhead as join numbers increase. Hertzschuch et al. [11] maintained pessimistic cardinality estimation properties while substituting sketches with an upper bound formula leveraging statistics already available to PostgreSQL. Izenov et al. [12] used Fast-AGMS sketches to capture join-crossing correlations, reducing overhead during sketch-building compared to Pessimistic.

Heuristic-based query optimizers employ predefined heuristics to identify the optimal plan for query execution. Several heuristic-based systems [4, 8–10, 17, 21, 23, 27, 29] have developed their own rule languages and execution environments to avoid compatibility issues. Held et al. [10] introduced Ingres, the first rule-based system, where the original query is divided into single-valued sub-queries and executed separately using a greedy approach. While effective for simple queries, this method struggles with complex queries. In contrast, Pirahesh et al. [17] developed Starburst, a Query Graph Model (QGM) based system that represents a SQL query as a graph. Query rewriting rules transform one QGM into an equivalent QGM, and during the plan optimization phase, each equivalent QGM is assigned an estimated cost, with the lowest cost QGM selected for query execution. Integrating query graphs with join types and functional dependencies may help to find efficient plans for execution in a main-memory setup [6]. Graefe et al. [8] created EXODUS, where a query is represented as an algebraic tree and employs rule-based reordering and plan optimization techniques similar to Starburst. However, the simplistic search strategy and cost function used by Starburst and EXODUS introduce limitations for complex queries. To address these limitations, Graefe et al. [9] presented Volcano, which utilizes directed dynamic search rather than rules for enumeration.

7 CONCLUSIONS

This paper presents an in-depth analysis of query optimization with and without cardinality estimates across a broad experimental spectrum. Our findings suggests that the performance gap between different optimization strategies is minimal in non-indexed settings and main-memory databases. This insight emerged from a detailed analysis of plan costs and execution times, supplemented by an evaluation of around 10,000 random query plans generated using the quick pick algorithm. Our investigation extends to assessing the effects of parallel processing on query performance, examining its implications across both indexed and non-indexed settings. Overall, our findings contribute to a deeper understanding of query optimization in the presence and absence of estimates.

Acknowledgments. This work is supported by NSF award number 2008815.

REFERENCES

[1] Peter Boncz. The IMDB Dataset. homepages.cwi.nl/~boncz/job/imdb.tgz.
[2] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC 2013*.
[3] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *SIGMOD 2019*. 18–35.
[4] Dinesh Das and Don S. Batory. Praire: A Rule Specification Framework for Query Optimizers. In *ICDE 1995*. 201–210.
[5] Asoke Datta. Analyzing the Impact of Cardinality Estimates on Query Optimization. github.com/Asoke26/Estimates-and-No-Estimates/.
[6] Asoke Datta, Yesdaulet Izenov, Brian Tsan, and Florin Rusu. 2021. Simpli-Squared: A Very Simple Yet Unexpectedly Powerful Join Ordering Algorithm Without Cardinality Estimates. *CoRR* abs/2111.00163 (2021).
[7] Asoke Datta, Brian Tsan, Yesdaulet Izenov, and Florin Rusu. 2023. Analyzing Query Optimizer Performance in the Presence and Absence of Cardinality Estimates. *CoRR* abs/2311.17293 (2023).
[8] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. In *SIGMOD 1987*. 160–172.
[9] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.
[10] G. D. Held, M. R. Stonebraker, and E. Wong. INGRES: A Relational Data Base System. In *AFIPS 1975*.
[11] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Simplicity Done Right for Join Ordering. In *CIDR 2021*.
[12] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. In *SIGMOD 2021*. 804–816.
[13] Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proc. VLDB Endow.* 16, 11 (2023), 2871–2883.
[14] Victor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.
[15] Victor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDB Journal* 27 (2018), 643–668.
[16] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC 2009*.
[17] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD 1992*. 39–48.
[18] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *VLDB 2007*. 1138–1149.
[19] Greg Rahn. Join Order Benchmark (JOB). github.com/gregrahn/join-order-benchmark.
[20] F. Michael Waas and Arjan Pellenkoft. Join Order Selection - Good Enough Is Easy. In *BNCOD 2000*.
[21] Lane B. Warshaw and Daniel P. Miranker. Rule-Based Query Optimization, Revisited. In *CIKM 1999*.
[22] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case With In-memory Star Schema Data Warehouse Workloads. *Proc. VLDB Endow.* 10, 8 (2017), 889–900.
[23] Apache Calcite. calcite.apache.org.
[24] DuckDB. duckdb.org.
[25] Heavy.AI. www.heavy.ai.
[26] MonetDB. www.monetdb.org.
[27] Oracle query optimization. www.oreilly.com/library/view/oracle-essentials-oracle9i/0596001797/ch04s07.html.
[28] PostgreSQL. www.postgresql.org.
[29] Presto. prestodb.io.