Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server

Kukjin Lee
Microsoft Research
kulee@microsoft.com

Anshuman Dutt
Microsoft Research
andut@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Cardinality estimation is widely believed to be one of the most important causes of poor query plans. Prior studies evaluate the impact of cardinality estimation on plan quality on a set of Select-Project-Join queries on PostgreSQL DBMS. Our empirical study broadens the scope of prior studies in significant ways. First, we include complex SQL queries containing group-by, aggregation, outer joins and sub-queries from real-world workloads and industry benchmarks. We evaluate on both row-oriented and column-oriented physical designs. Our empirical study uses Microsoft SQL Server, an industry-strength DBMS with a state-of-the-art query optimizer that is equipped with techniques to optimize such complex queries. Second, we analyze the sensitivity of plan quality to cardinality errors in two ways by: (a) varying the subset of query sub-expressions for which accurate cardinalities are used, and (b) introducing progressively larger cardinality errors. Third, query processing techniques such as bitmap filtering and adaptive join have the potential to mitigate the impact of cardinality estimation errors by reducing the latency of bad plans. We evaluate the importance of accurate cardinalities in the presence of these techniques.

PVLDB Reference Format:
Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. PVLDB, 16(11): 2871 - 2883, 2023.
doi:10.14778/3611479.3611494

1 INTRODUCTION

Cardinality estimation, the task of estimating the number of output rows for a SQL expression, is one of the challenging problems in query optimization. Due to the richness of SQL operators, limited data statistics available during query optimization, and the need to keep the time and resources used for query optimization small, today's query optimizers typically use simplifying assumptions for cardinality estimation [8, 15, 16, 29]. Therefore, large errors can occur in cardinality estimation of query sub-expressions during query optimization and may lead to *sub-optimal* execution plans.

Existing cardinality estimation (CE) benchmarks provide a set of select-project-join (SPJ) queries against real datasets - JOB [33] includes 113 queries on the IMDB dataset, and STATS [28] includes 146 queries on the Stats Stack Exchange dataset. These benchmarks use row-oriented physical design with B+Tree indexes on primary-key and foreign-key columns of tables to enrich the plan search space on the PostgreSQL database engine. The studies using these benchmarks [28, 33] demonstrate that plan quality can be significantly improved through accurate CE. While these studies are a valuable starting point to understand the impact of CE on plan quality, they leave several important open questions regarding the importance of CE.

First, real workloads often consist of complex queries that use SQL constructs that go well beyond SPJ (e.g., group-by, sub-queries) and the state-of-the-art query optimizers are equipped with techniques to handle such complex queries. Also, a "simpler" physical design that uses only columnar indexes is widely used, particularly in analytical workloads. Since the query optimizer faces different challenges in these realistic scenarios, we need to evaluate if the conclusions from prior studies [28, 33] hold. Second, since accurate CE can be quite expensive for complex queries [17, 40], understanding the extent to which CE errors can be tolerated without compromising plan quality is another worthy goal that has received limited attention. Finally, query optimizers on modern execution engines have the ability to use query execution techniques at runtime that can potentially *mask* (i.e., mitigate) the adverse impact of CE errors by reducing the latency of the plan obtained using estimated cardinalities. Examples of such techniques include adaptive join operator [6, 34] and bitmap (a.k.a. bitvector) filtering [5, 18]. Bitmap filters can reduce the work done in an otherwise sub-optimal join-order, and adaptive join can dynamically switch to Hash Join at runtime if Index Nested Loop Join turns out to be expensive. Prior studies have not evaluated the extent of such mitigation, and to what extent accurate CE is relevant in the presence of these runtime techniques.

We take a step towards answering these important open questions discussed above:

- First, we use Microsoft SQL Server, an industry-strength DBMS that uses a state-of-the-art query optimizer. It explores a large plan search space for queries due to transformation rules such as outer join reordering and group-by/aggregate push down.
- Second, we use a *variety* of query workloads with different SQL constructs including group by, outer joins, nested subqueries etc. Specifically, we complement the CE benchmarks noted above with 100+ queries across 6 real workloads, the industry benchmark TPC-DS [13], and modified industry benchmarks TPC-H with skew [2] and DSB [20], and additional queries on the IMDB dataset [38]. Thus our empirical studies go well beyond SPJ queries that were studied in [28, 33, 38].
- Third, we evaluate both kinds of physical database designs: the traditional row-oriented physical design with B+Tree indexes (*rowstores*) as well as columnar indexes (*columnstores*).

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611494

- Fourth, we measure: (a) whether it is sufficient to eliminate CE errors for only a subset of query sub-expressions without affecting plan quality, (b) how increasing CE errors in a controlled manner impacts plan quality.
- Finally, for bitvector filtering and adaptive join techniques in Microsoft SQL Server, we perform a set of controlled experiments to quantify the *additional* improvement in plan quality that arises through the use of accurate cardinalities in the presence of these techniques.

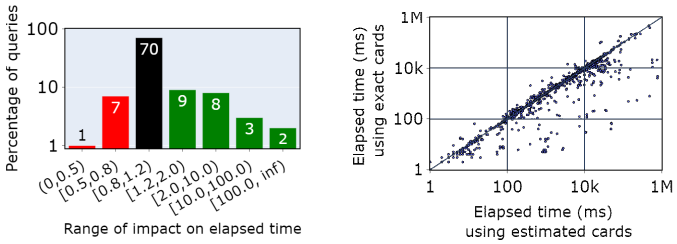
To conduct our evaluation on the Microsoft SQL Server query optimizer, we needed to develop a new *cardinality injection* API, such that when the query optimizer requests the cardinality of a logical sub-expression of the query, we are able to inject a value that overrides the cardinality estimated by the optimizer's built-in cardinality estimation model. Since Microsoft SQL Server is built using the Cascades [27] optimizer framework, the cardinality injection API needs to work with the existing data structures and enumeration architecture. The implementation of this API, as well as its limitations, are described in Section 2.3. We use the API to inject *exact* cardinality (i.e., no error) for each logical sub-expression whose cardinality is requested by the optimizer. This approach allows us to quantify the impact of cardinality estimation by comparing the elapsed times of the original plan and the plan obtained by injecting exact cardinalities. In the rest of this section, we present a few highlights of the findings and discuss open questions. A more complete description of the experimental results, along with explanations and examples of observed impact can be found in Sections 4– 6.

1.1 Key Findings

Results on JOB (Section 4): The results presented in [33] uses rowstore physical design in PostgreSQL. With the same setting on Microsoft SQL Server, we observe that about 30% of the queries in JOB improve by 2× or more when we inject exact cardinality. This is consistent with the results reported in [33]. However, on columnstores only around 5% JOB queries improve by 2× or more. This is because a vast majority of JOB queries on rowstores have index seeks as the most expensive operator (also noted in [33]) whereas in columnstores index seeks are absent.

Results on STATS (Section 4): STATS [28] queries contain a COUNT aggregate. Microsoft SQL Server, unlike PostgreSQL, leverages aggregate pushdown transformations to significantly bring down the elapsed times even with optimizer-estimated cardinalities. Consequently, STATS queries show significantly smaller magnitude of improvements on Microsoft SQL Server rowstores when the optimizer's cardinality estimates are replaced with exact cardinality for each sub-expression requested by the optimizer.

Variation across workloads (Section 4): We find that the impact of accurate CE *varies considerably* across different real workloads and industry benchmarks. The plan quality of all queries (i.e., across all workloads) we evaluated is shown in Figure 1 for rowstores. We find that 13% of the queries improve by 2× or more, and 5% of all queries improve by 10× or more. However, in 6 out of 17 workloads that we evaluated, more than 20% of the queries improved by 2× or more, whereas in 4 out of 17 workloads, fewer than 10% of the queries improved by 2× or more, including one workload where no queries improved at all. On columnstore, despite the simpler



(a) Distribution of plan quality improvements/regressions

(b) Log-Log scale scatter plot

Figure 1: Impact of exact cardinalities in Microsoft SQL Server: rowstore configuration

physical design, we find that accurate cardinalities help almost as many queries as in rowstore across all workloads (see Figure 8), e.g., 13% queries improve by 2× or more and 3% of queries improved by 10× or more. The variability across workloads is even more pronounced in columnstores where, for 9 out of 17 workloads, there were fewer than 10% of the queries that improved by 2× or more.

Regressions (Section 4.3): On both rowstores and columnstores, for a relatively small fraction of the queries (1% and 3% respectively), injecting accurate cardinalities *increases* their elapsed time by 2× or more . We determine the cause to be inaccuracies of the optimizer's cost model. This is consistent with findings in [33].

"Essential" query sub-expressions (Section 5.1): For queries that improve significantly (2× or more) on rowstores when all logical sub-expressions are injected with exact cardinality, we a posteriori identify an *essential subset* of logical sub-expressions, i.e., a minimal subset of logical sub-expressions such that injecting exact CE only for that subset and using optimizer-estimates for all other logical sub-expressions, is sufficient to yield the *same plan* as when exact CE are used for all logical sub-expressions. Intriguingly, we find that the essential subsets for the above queries are relatively small, i.e., only 1%-5% of the logical sub-expressions for queries with more than 100 logical sub-expressions.

Impact of increasing CE errors (Section 5.2): For queries that improve 2× or more, i.e., are sensitive to errors in CE, we use the cardinality injection API to inject *erroneous* cardinalities by systematically varying the magnitude of error for each logical sub-expression. We use q-error, an error metric widely used to evaluate CE techniques (e.g., [22, 30, 33, 35, 37]). Our experiment sheds more light on the relationship between q-error and plan quality that goes beyond previous studies.

Bitmap filtering (Section 6.1): We find that bitmap filtering is more frequently used in columnstores than rowstores because they require Index Scan and Hash Join operators, which are more predominant in columnstores. Bitmap filtering leads to significant improvement in elapsed time even without use of exact cardinalities (e.g., on columnstores 36% of queries improve by ≥ 2×), thereby significantly masking the negative impact of inaccurate CE. However, using exact cardinalities with bitmap filtering causes the above percentage to modestly rise to 42% due to better choice of physical

operators (e.g., Stream Aggregate vs. Hash Aggregate) and join orders, and pushdown of group by and aggregation.

Adaptive join (Section 6.2): Adaptive join is designed to mitigate the impact of choosing the wrong physical join operator. Specifically, it chooses between Hash Join and Index Nested Loops join at runtime depending on the actual number of rows observed from the execution of the first input. We find that use of Adaptive Joins alone improves elapsed time of less than 1% of queries by 2× or more, whereas when using both exact cardinalities and adaptive join 14% of the queries improve by 2× or more. The ability of adaptive join to mask the negative impact of CE errors is limited.

1.2 Open Questions

Based on our empirical study, we outline some potentially important questions for database system engineers and researchers:

- Since our empirical study is conducted on Microsoft SQL Server, follow-up studies on other DBMSs would be needed to understand the extent to which the above findings generalize.
- Our results point to the need to augment benchmarks used for evaluating query optimizers (e.g., [28, 33]) with new queries so that they can be more representative of data skew and correlation across predicates observed in real workloads. Additionally, queries in these benchmarks need to be modified since state-of-the-art query optimizers use transformation rules to optimize queries containing commonly used operators such as group-by and aggregation.
- Further work is required to understand the characteristics and significance of essential query sub-expressions.
- Since runtime execution techniques (e.g., bitmap filtering) are commonly used in practice, and can partially mask the negative impact of CE errors, empirical evaluations of the optimizer using CE benchmarks should also include use of these techniques.
- The choice of appropriate physical operators for aggregation and join is an important area where existing runtime execution techniques are unable to mask the impact of inaccurate CE. Thus developing new physical operators that are more resilient to errors in CE (see [19] for a survey) can be beneficial.

2 MICROSOFT SQL SERVER QUERY OPTIMIZER

In this section, we provide a brief overview of the relevant aspects of Microsoft SQL Server's query optimizer, and describe the extensions we made to obtain query plans with exact cardinality injection.

2.1 Overview

Microsoft SQL Server's query optimizer uses the Cascades framework's [27] extensible optimizer architecture. In this framework, rules are used to represent the knowledge of the search space. The optimizer uses two kinds of rules: transformation rules that map one algebraic expression into another (i.e., logical expression → logical expression) and implementation rules that map an algebraic expression into an operator tree (i.e., logical expression → physical operator tree). For efficiency, the optimizer uses a top-down enumeration algorithm with *memoization*, and performs cost-based pruning of the search space. The memo data structure compactly stores all explored alternatives by grouping together logically equivalent query sub-expressions, called *memogroups* (or groups for short), and their physical operator trees (i.e., plans). We use the terms memogroup and logical sub-expressions interchangeably in this paper. The memo also captures the parent-child relationships among memogroups, e.g., (A) and (B) are children of $(A \bowtie B)$, and $((A \bowtie B) \bowtie C)$ is a parent of $(A \bowtie B)$. Thus each memogroup has a set of ancestor nodes and a set of descendant nodes. Logical properties such as the cardinality of a group and physical properties such as the cost of a plan are stored in the memo. Additional information about Microsoft SQL Server's optimizer can be found in [24, 25]. Cardinality estimation in Microsoft SQL Server (see [8, 23] for more details) relies upon statistics such as *histograms* on individual columns of a table for estimating the selectivity of selection and join predicates, single and multi-column *density* information for estimating the number of distinct values, and information about constraints such as primary key and foreign key.

Bitmap filtering: Microsoft SQL Server uses bitmaps to perform a semi-join reduction for plans containing Hash Join operators [5]. This technique performs early filtering of rows that will not qualify a join [24]. At a high level, each Hash Join operator creates a single bitmap filter from the equi-join columns on the build side. This bitmap filter is then pushed down to the lowest possible level on the subtree rooted at the probe side so that it can eliminate tuples from that subtree as early as possible [26]. The optimizer also exposes a second mode in which it places bitmap filters in a physical execution tree based on estimated selectivity of the filter(s). In this mode, bitmap filters are not always pushed down as much as possible, but may be placed in alternative locations in the physical tree depending on the estimated selectivities. The results reported in this paper are for the first mode only. Bitmap pushdown in Microsoft SQL Server are performed on operators in the final plan chosen by the optimizer.

Adaptive join: The Microsoft SQL Server optimizer can use the adaptive join operator [6], which enables the choice of Hash Join or Nested Loops join method to be deferred to runtime until after the first join input has been scanned. If the row count of the first input is below a threshold, the Adaptive Join operator uses a Nested Loops join, otherwise it performs a Hash Join. A query plan can therefore dynamically choose a better join strategy during execution, and potentially reduce the impact of cardinality mis-estimation on join performance.

2.2 Physical design

Microsoft SQL Server supports a rich set of options for physical design of a database including indexes, materialized views and partitioning. In this study, we focus only on indexes. In Microsoft SQL Server, indexes fall into one of two categories: rowstore and columnstore described below.

Rowstore indexes: These are clustered and non-clustered B+-Tree indexes over data stored in row format. Indexes can have one or more key columns, and optional included (i.e., non-key) columns.

Columnstore indexes: Columnstore indexes store data in columnar format, broken into segments (of approximately 1M rows each). Columnstore indexes allows greater compression than B+Tree indexes and are thus effective in analytic workloads that need to scan

large amounts of data. Although columnstore indexes do not support index seek capabilities, optimizations such as zone maps allows skipping of segments of the data during scan. Details of columnstore indexes in Microsoft SQL Server can be found in [31, 32].

2.3 API for exact cardinality injection

To analyze the impact of cardinality estimation on plan quality, we compare the plan obtained by the optimizer using the its built-in cardinality estimator to the plan obtained when all cardinality values are exact i.e., have no errors. We extend the Microsoft SQL Server query optimizer with two APIs (see Figure 2): (1) *cardinality injection*, which allows a client to inject a cardinality value for a specific memogroup during optimization, thereby overriding the optimizer estimated cardinality, and (2) *SQL decoding*, which enables a client to obtain a valid and logically equivalent SQL statement corresponding to a memogroup. Note that the exact cardinality injection API is significantly more comprehensive in terms of supported SQL constructs such as group-by, aggregation, nested subqueries etc., compared to the API used in prior empirical studies [28, 33] that supported SPJ queries on PostgreSQL.

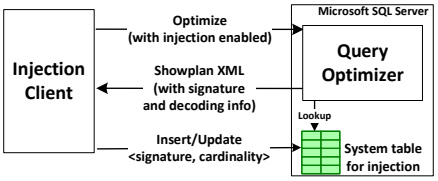


Figure 2: Cardinality injection API in Microsoft SQL Server

Extensions to the query optimizer: We extend the optimizer to derive two additional properties for each logical memogroup. First, we compute a *signature* (a 64-bit hash value) that uniquely identifies the memogroup. We derive this value by recursively traversing all nodes in the subtree rooted at that memogroup, and collecting unique properties of the group, e.g., columns references in the projections, filters, and grouping; outer vs inner join, filter predicate bindings, etc. Second, we construct a *valid SQL query* (aka SQL decoding) for the memogroup, using another recursive procedure that first computes decodings for the children memogroups and then combines these decodings based on the semantics of the memogroup's logical operator[1]. The signature and SQL decoding associated with each memogroup is returned via a newly added <*Memo*> element in the XML version of the output plan (i.e., Showplan output), which the client can then use to obtain the exact cardinality plan.

Steps to obtain exact cardinality plan: The client computes the exact cardinality for each memogroup by executing its SQL decoding and inserts the *<signature, exact cardinality>* tuple into a system table. Then, it invokes the query optimizer along with a *flag* that redirects the optimizer to use cardinality from the system table, if available, rather than use its default cardinality estimate. Note that the query optimizer may encounter new memogroups for which exact cardinality information is not available in the system table, since they were not considered in previous optimizer invocations. Therefore, we repeat the above steps of signature generation, decoding, exact cardinality computation and optimizer invocation until no new memogroups are explored. Note that this iterative procedure can be quite expensive due to exact cardinality computations, which can be optimized e.g, using techniques described in [17].

Re-costing a plan with injected cardinality We can also use the above API to re-cost a plan with exact cardinalities by enabling cardinality injection while using the USE PLAN hint functionality of Microsoft SQL Server. We use such re-costing to identify whether the cost model of the query optimizer is responsible for regression in elapsed time after exact cardinality injection.

Injecting cardinality for a subset of memogroups We note that the cardinality injection API can also be used for injecting cardinalities for a subset of memogroups (Section 5). In this case, the cardinalities of *non-injected* memogroups are derived by the optimizer as usual. However, since existing methods in Microsoft SQL Server use the cardinality of a child memogroup to derive its cardinality, the cardinalities injected for a memogroup can affect the cardinality of its non-injected ancestor memogroups.

Limitations of cardinality injection: There are some cardinalities used by the optimizer for which we are unable to inject cardinalities, and therefore the optimizer falls back to its default cardinality estimation logic in these cases. We outline such cases below. First, our cardinality injection API only injects cardinalities for memogroups. However, before the memo is created, the optimizer performs an initial n-ary join reordering using cardinalities that can prune certain join orders. Second, for a memogroup that corresponds to a correlated sub-query its cardinality can vary with each invocation depending on the value from the outer relation, which is not known at compile time. Third, when we execute the decoded SQL expression for a memogroup to obtain its cardinality, in some cases the query times out[2], and therefore we are unable to obtain the exact cardinality. Finally, we note that in addition to the primary cardinality estimator of Microsoft SQL Server described in Section 2, it also uses two special estimators: (1) *rowgoal* selectivity estimator (e.g., when the query contains a TOP clause), and (2) bitmap selectivity estimator. While we can inject exact cardinalities for the primary estimator using the above API, the final cardinalities used by the optimizer can be modified by the other two estimators.

3 EVALUATION SETUP AND METHODOLOGY

3.1 Workloads

We use multiple workloads in our experiments: (1) **CE benchmarks**: JOB [33], CEB [38], STATS [28]. (2) **Industry benchmarks**: TPC-DS, modified industry benchmarks DSB [20] which uses the same schema as TPC-DS but varies the data distribution and query generation, and TPC-H (with data skew, Z=1 Zipfian skew factor). (3) **Real workloads**: a suite of real queries over 6 different real databases. Both databases and queries were collected from customers of Microsoft SQL Server, and were unmodified for this evaluation.

[1]We implemented these functions for ≈30 logical operators in Microsoft SQL Server - these functions accounted for the majority of our implementation effort. We skip a few infrequent logical operators in the current implementation.

[2]We use a timeout of 1500 seconds.

- REAL1: Analyzes sales and inventory data for a US bookstore chain using dimensions of time, item, store, merchant etc.
- REAL2: Analyzes products and orders for an international cosmetic retailer.
- REAL3: Internal application that stores and queries the catalog of a source control system.
- REAL4: ERP application that analyzes retail invoices and sales by time, customer, point of sale etc.
- REAL5 and REAL6: Internal applications tracking agreements, customers and sales of products/services.

(4) **Real templates**: We generate additional queries by treating each real query above as a *template* and instantiating bindings in their filter predicates. This way we systematically introduce variations in the selectivity of predicates of real templates without modifying the database or the template. To create a template corresponding to each real query we convert each filter predicate on an attribute of a base table into a parameterized predicate. We then generate new instances of each template by choosing parameter binding(s) for each predicate independently, and choosing binding values such that the resulting filter has geometrically increasing selectivities.

This ensures that across instances, we cover a large range of selectivities for that predicate with more emphasis on small selectivities. We obtain binding values from the histogram of that column which is stored as part of Microsoft SQL Server's statistics object for that column. We handle equality, range, and IN predicates.

These workloads are summarized in Table 1. In contrast to queries in the CE benchmarks that focus on SPJ queries with *count* aggregate, real queries and industry benchmark queries contain outer joins, nested sub-queries, group by and other aggregate functions. Moreover, data sizes and the number of relations per query tend to be significantly larger. Finally, we note that for each workload we only include queries for which exact cardinality injection is possible for all memogroups in the query (see Section 2.3).

Table 1: Workloads used in the evaluation

Workload (DB)	Data/Index sizes(GB)	Query Count	#relations (min-max)	Group By	Outer-join	Sub-query	CTE	Union /Intersect
JOB (IMDB)	5.4/3	88	4-17	-	-	-	-	-
STATS (Stacks -Exchange)	0.2/0.03	138	2-7	-	-	-	-	-
CEB (IMDB)	6/3	38	4-17	17	-	-	-	-
TPC-H (z=1)	12.7/28	126	1-7	92	11	34	-	-
TPC-DS	16.3/0.03	174	3-32	143	5	56	23	19
DSB	11.2/20	63	3-34	30	11	5	8	3
REAL1 (Real DB1)	97/0.3	14	1-9	6	3	2	10	-
REAL2 (Real DB2)	49.4/39	13	2-10	11	-	11	-	-
REAL3 (Real DB3)	262.4/129	11	1-6	-	4	-	-	-
REAL4 (Real DB4)	2888/265	8	1-20	7	2	1	-	-
REAL5 (Real DB5)	377/325	16	5-33	16	15	-	16	2
REAL6 (Real DB6)	10.4/13	47	5-25	47	45	-	47	7

3.2 Methodology

Our evaluation varies the following dimensions: (1) *Cardinality:* {Optimizer-Est, Exact} (2) *Physical Design*: {Rowstore, Columnstore}. For rowstore, for real workloads, we retain the original physical design present in the database. For all other workloads, we create indexes recommended by Microsoft SQL Server Database Engine Tuning Advisor (DTA) tool. For columnstores, we create one clustered columnstore index on each table. (3) For Parallelism, Bitmaps and Adaptive Join, we run with each feature turned On/Off. We run each query in isolation with a cold cache i.e., the buffer pool is reset prior to each execution using dbcc dropcleanbuffers [11]. We record the plan, elapsed time and CPU time of each query execution. The experiments were carried out on Windows Server machines with AMD EPYC 7352 2.3 GHz processors (24 cores, 2 sockets), 256 GB RAM and 1.6TB SSD.

For each query, we measure the elapsed time of the original plan obtained with optimizer-estimated cardinalities (P_{orig}), and the elapsed time of the plan obtained with exact cardinality injection (P_{exact}). Then, we define the *Impact* on plan quality of using exact cardinality for a query as:

$$Impact = \frac{Elapsed\_Time(P_{orig})}{Elapsed\_Time(P_{exact})} \quad (1)$$

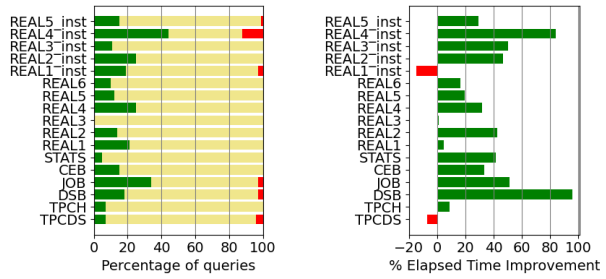
Observe that $Impact > 1$ represents an improvement due to cardinality injection whereas $Impact < 1$ is a regression.

4 RESULTS: EXECUTION-TIME OPTIMIZATIONS TURNED OFF

In this section we discuss the impact of injecting exact cardinalities on plan quality in the setting where all execution time techniques are disabled. In particular, we use serial plans only (i.e., disable multicore parallelism), and turn off bitmap filtering and adaptive join techniques. This is also the setting used in prior studies including JOB [33] and STATS [28]. We first present our results for rowstore databases in Sections 4.1- 4.3 followed by results for columnstore databases in Section 4.4. We conclude this section with takeaways in Section 4.5.

4.1 Overview in rowstores

Query-level improvements and regressions: Figure 1(a) (in Section 1) shows the distribution of *Impact* across queries from all workloads. In 70% of the queries the *Impact* is between 0.8× and 1.2×. In other words, these queries improve or regress by less than 20%. In 8% of the queries we see 2×-10× improvement, in 3% of the queries we see 10×-100× improvement, and in 2% of the queries we see over 100× improvement in elapsed time. We also observe regressions, although they are significantly fewer than improvements - 1% queries regress by 2× or more. Figure 1(b) shows a scatter plot (log-log scale) of elapsed time of each query for P_{orig} (x-axis) and P_{exact} (y-axis). Points below the diagonal are queries that improve due to exact cardinality injection, whereas points above the diagonal are queries that regress. First, as the plot shows, there is a large range of elapsed times of the queries across workloads. They vary between just a few milliseconds to 1000's of seconds. Second, the number of queries that improve as well as the magnitude of their improvements is significantly higher than those of queries



(a) Percentage of queries improving/regressing by ≥ 2×

(b) Workload level improvement or regression

Figure 3: Impact of exact CE across workloads (rowstore)

that regress. Third, a large fraction of the largest improvements that arise due to use of exact cardinality occur in queries with large original elapsed time. We drill down into reasons for improvement and regressions in Sections 4.2 and 4.3 respectively.

Workload-level improvements and regressions: Figure 3(a) shows for each workload the percentage of queries that improve (green bar) or regress (red bar) 2× or more when using exact cardinalities. The impact of using exact cardinalities varies significantly with no queries improving in REAL3 and more than 30% of the queries improving in JOB. Regressions are less frequent, absent in 11 workloads and happen only in a few workloads such as TPC-DS, and REAL4_inst. Figure 3(b) shows the improvement or regression in *total elapsed time*, i.e., sum of elapsed time over all queries in the workload. In 7 workloads we observe more than 40% improvement in total elapsed time, and in 2 workloads, DSB and REAL4_inst, we observe more than 80% improvement. In contrast, the regressions result in 12% or less degradation in total elapsed time for 2 workloads.

Significant regression for TPC-DS/REAL1_inst: We did deeper analysis of workload level regression for TPC-DS and found that it is due to large absolute time regression of a small percentage of queries that nullify the improvements in much larger percentage of queries. To elaborate, 19% of TPC-DS queries improve by at least 1.2×, however the resulting 60% improvement in execution time is cancelled by significant absolute time regression for only 3% regressing queries. It is worth noting that the improvements in TPC-DS queries are limited to a small number of query templates, i.e., 26 out of 32 improving queries were instances of 7 query templates. Similarly, all regressing queries correspond to only 2 TPC-DS query templates. Our observations are similar for REAL1_inst, that large absolute regression of a small number of queries nullify the gains in larger fraction of queries.

Limited impact of accurate CE for STATS/REAL3: Recall that in the STATS benchmark, the table sizes are small, but intermediate result sizes of joins are very large due to FK-FK joins, making join size estimation very important. In [28] the authors note that in PostgreSQL, using exact CE results in huge improvement (50% end-to-end gains ≈5.5 hours). In contrast, the Microsoft SQL Server optimizer takes advantage of aggregate pushdown optimization to push the COUNT aggregate computation below joins. Thus, for many STATS queries the intermediate join sizes are small, leading to efficient plans even with its native cardinality estimation model - most queries execute in few hundred milliseconds with estimated cardinalities. The scope of improvement due to exact CE is therefore limited in Microsoft SQL Server, only 7% queries improve by 2× or more and maximum per query improvement is only ≈ 7 seconds.

We also analyzed the reason why REAL3 does not exhibit any improvement due to exact CE. We found that, while some of the tables referenced in these queries are large, many queries contain a conjunction of $4-5$ selective predicates on these large tables. As a result, the actual number of rows that qualify after selection predicates are applied are small (usually below 10) and the optimizer estimates for base table expressions are also quite accurate. The original plan uses Index Seeks and index intersections in some cases to access rows in these large tables. Since join sizes are also very small, there is minimal scope (with the exception of one query) for improving the elapsed times of such queries by making CE more accurate.

4.2 Drill-down into improvements

We now drill down to characterize in what ways exact cardinalities help in finding a better plan. We find that an explanation such as P_{exact} has a "better join order" than P_{orig} often does not capture real reason for improvement even though the join order changes. Our approach is triggered by the observation in [33] that for most queries in JOB, the reason for a slow plan P_{orig} on PostgreSQL is *excessive use of index seeks*. This motivated us to understand whether the queries from industry benchmarks and real workloads also follow a similar pattern, where the improvements in the elapsed time of P_{exact} when compared to P_{orig} is dominated by speeding up or eliminating expensive index seek operators, or whether there also exist other patterns of improvement. Our analysis therefore focuses on the most resource consuming operations in P_{orig}, and we ask how using accurate CE helps avoid or mitigate them in P_{exact}.

Methodology: For all queries that improved by 2× or more and whose elapsed time is $\geq 50ms$, we analyze the plans P_{orig} and P_{exact}, and compute the actual time spent in seek, scan and non-leaf operators for both P_{orig} and P_{exact}. With this information, we categorize each query into one of the following four categories based on which class of operators contributes to the majority, i.e., > 50%, of the observed improvement: (1) Seek operators, i.e., operators that access base tables using random I/O, (2) Scan operators, i.e. , operators that access base tables using sequential I/O, (3) Non-leaf operators (e.g., physical join operators, sort, hash aggregate, ...) (4) A catch-all category where none of the conditions (1)-(3) apply and the elapsed time improvement is *spread across multiple* leaf and/or non-leaf operators.

Note that each category can have examples of changes in join order. For example, a query where P_{orig}'s join order $((A \bowtie B) \bowtie C)$ changes to $((A \bowtie C) \bowtie B)$ in P_{exact}, can be assigned to category 1 if access path for C was an expensive seek and majority of the observed improvement happened because this expensive seek on C is avoided, or category 2 if access path for B was an expensive scan and majority of the observed improvement happened because expensive scan on B is avoided, or category 4, if none of the two

reason was individually dominant (more than 50%) and the execution cost improvement was spread across table access path and join operators.

Table 2: Queries across improvement categories

	Category 1	Category 2	Category 3	Category 4
	Dominant improvement in:			
Workload set	Seek operators	Scan operators	Non-leaf operators	Other (catch all)
CE benchmarks	33	13	2	4
Industry-benchmarks	14	14	0	3
Real workloads	8	4	4	2
Real templates	12	23	17	1

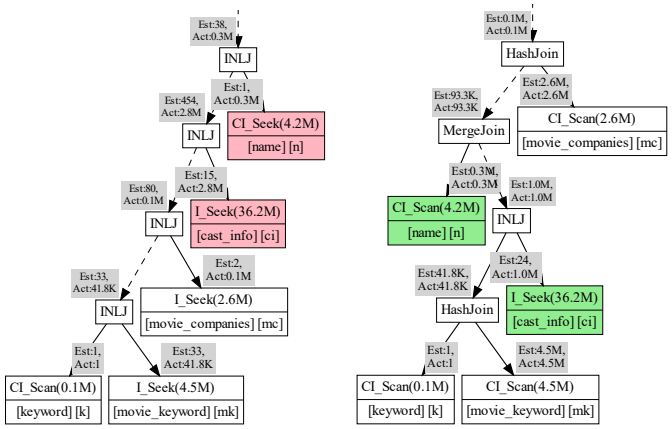
The distribution of queries in these categories is shown in Table 2. We find that for CE benchmarks, the number of queries in Category 1 (Seek operators) exceeds the number of queries in all other categories combined. Specifically, in JOB, we often find heavy use of index seeks in P_{orig} due to a series of severe underestimations starting from the first join and hence the dominant cause of improvement is replacing seeks with scan operators, or changing join order such that the number of seeks are significantly reduced. In STATS queries, severe underestimates originate much later in the plan due to a many-to-many join that either results in excessive seeks (category 1) or *spills to disk* during hash-build operator (category 3 or 4). In both of these cases for STATS, Microsoft SQL Server's ability to push-down COUNT(*) as partial aggregate operator leads to a faster plan P_{exact}. In contrast to CE benchmarks where Category 1 is predominant, for industry benchmarks, real workloads, and real templates, we find that Category 1 constitutes a significantly smaller percentage, and the speed up of index scans and other non-leaf operators often are the dominating reasons for plan improvements.

In the rest of this subsection we describe one example query from each category, and we show P_{orig} and P_{exact}. For clarity, we only show plan fragments with relevant operators - missing operators are represented with *dashed* edges.

Example 4.1. An example query of first category, JOB Q17a, with its before and after plans are shown in Figure 4. The expensive Index Seek operators in P_{orig} which contributes most to its elapsed time are shown in color. In P_{exact} the Index Nested Loops Join with table "name" is replaced with a Merge Join with an Index Scan of the "name" table, and the number of seeks for the table "cast_info" are significantly reduced due to join order change.

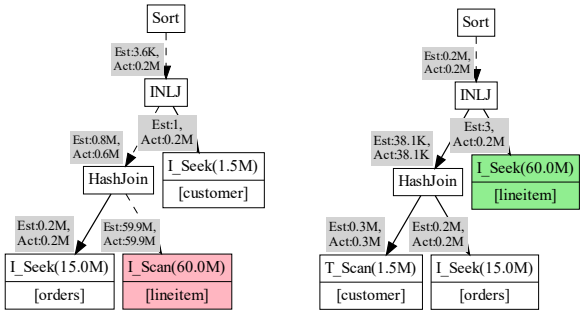
Example 4.2. As an example for category 2 improvements, Figure 5 shows P_{orig} and P_{exact} for an query instance of TPC-H Q3. P_{orig} spends a large fraction of its total elapsed time scanning the "lineitem" table (with 60M total rows). Once exact cardinalities are injected, the optimizer finds a different join order where it is sufficient to seek on the "lineitem" table.

Example 4.3. Figure 6 shows an example of category 3 improvement, using plans P_{orig} and P_{exact} for a query from REAL1. The most expensive operators in P_{orig} are the *Sort* and *StreamAgg* operators highlighted in color, due to severe underestimation (est=100k, exact=6M) in the input of the *Sort* operator. P_{exact} uses a partial aggregate push down operator after the second join that drastically reduces the input size of the *Sort* operator later on in the plan.



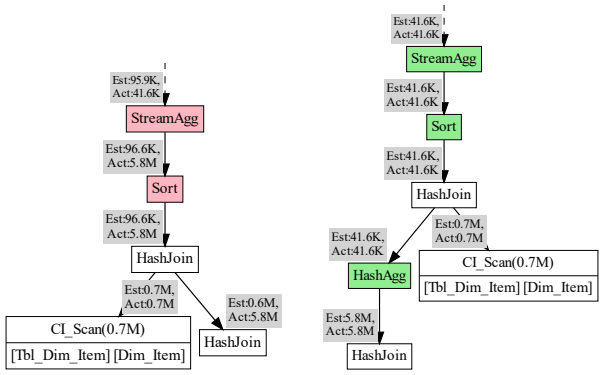
(a) With estimated cards **(b) With exact cards**

Figure 4: Plan fragments for category 1 example (JOB Q17a): Majority of improvement is due to avoiding expensive seeks to access tables "name" and "cast_info"



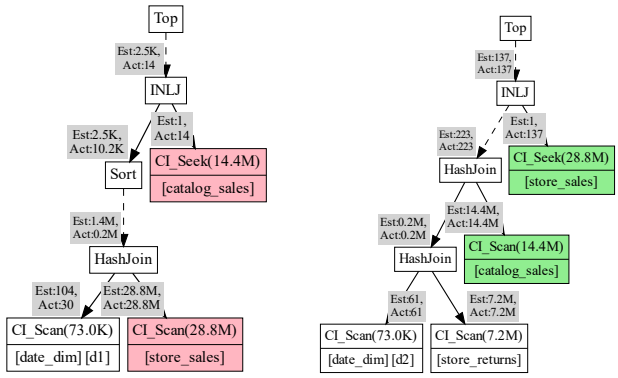
(a) With estimated cards **(b) With exact cards**

Figure 5: Plan fragments for category 2 example (TPC-H Q3-i6 query): Scan operator for the "lineitem" table is eliminated



(a) With estimated cards **(b) With exact cards**

Figure 6: Plan fragments for category 3 example (REAL1 query): Non-leaf operators (Sort and StreamAgg) process much smaller input due to push down aggregate



(a) With estimated cards **(b) With exact cards**

Figure 7: Plan fragments for category 4 example (TPC-DS Q29): change in join order leads to improvement across multiple operators

Example 4.4. An example for category 4 improvement where join order change helps in improving multiple expensive operators, the before and after plans for query TPC-DS Q29 are shown in Figure 7. The most expensive operators in P_{orig} are clustered index scan of "store_sales" (29M rows) and an index nested loop operator with seek on "catalog_sales". With exact cards, the optimizer find a new join order in P_{exact} where it avoids both expensive scan of "store_sales" and expensive seek of "catalog_sales" table.

4.2.1 Observed root causes of CE errors. The root causes of CE errors that lead to bad plans in JOB and STATS have been noted earlier. Here we describe the root causes of CE errors we find in real workloads and industry benchmarks. In real workloads, the most common reason we see for CE errors that lead to poor plans is data skew in one or more join columns. Data skew in join columns can lead to heavy underestimation or overestimation based on the skew value and whether infrequent or frequent values are filtered out by the filter conditions. Unlike real workloads, the data skew and filter conditions in JOB queries typically favor underestimations. In DSB Q101, we see a complex join predicate spanning two date dimension tables "d2.d_date between d1.d_date and dateadd(day, 90, d1.d_date)" which leads to large cardinality underestimation in the output of the join of these two dimension tables, and in turn leads to a bad plan involving an Index Seek on the large *web_sales* table. In TPC-H Q12 we observe that even single-table predicate "l_commitdate < l_receiptdate and l_shipdate < l_commitdate" lead to large overestimation due to assumptions made by the optimizer about correlation across columns.

4.3 Drill-down into regressions

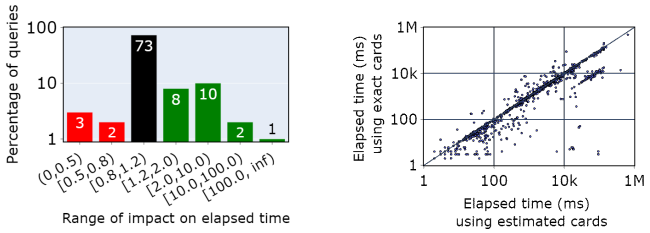
The three major components of the query optimizer are cardinality estimation, cost model and search [15]. With exact cardinality injection we eliminate cardinality estimation as the cause of a regression. To reduce the likelihood of regressions due to plan search, we disable the early exit criteria used in plan search based on the time spent on query optimization exceeding a threshold or optimizer estimated cost of the best plan dropping below a threshold. Therefore, two potential causes of regression are: (a) errors in cost modeling such as relative cost of Index seek vs scan, or errors in deciding whether to use row-mode or batch-mode operators and (b) cases where injected cardinalities are overridden with another heuristic estimator during optimization, such as rowgoal estimator [3, 9].

We analyzed queries where the elapsed time of P_{exact} is 2× or more *higher* than the elapsed time of P_{orig}. The most common cases we see are an incorrect "crossover point" in the estimated cost between a pair of physical operators: (a) Index Seek vs. Index Scan or (b) Index Nested Loop Join vs. Hash Join, since the seek to scan cost ratio is overestimated. This is because the machines we run the experiments on are equipped with SSDs having relatively low ratio of random access to scan cost, whereas the optimizer's cost model is calibrated for storage devices with a higher ratio. In other cases, the plan determined by the optimizer after exact cardinality injection has a different join order than original plan with smaller optimizer estimated but has larger execution cost. Finally, we find cases where the rowgoal estimator, invoked for queries with TOP clause (as noted in Section 2.1), overrides the injected cardinalities and leads to a worse plan in execution cost.

4.4 Columnstores

The overall distribution of improvements and regressions for queries in columnstore databases is shown in Figure 8(a). We observe that around 13% of queries improve in elapsed time by 2× or more and about 3% of the queries regress by 2× or more. The scatter plot in Figure 8(b) shows that large improvements occur for both inexpensive and expensive queries. Next, we breakdown the impact of exact cardinalities on plan quality by workload. Figure 8(a) shows the percentage of queries for each workload with 2× or more improvement in elapsed time (green bar) and 2× or more regression in elapsed time (red bar). Overall, these aggregate improvement and regression numbers are similar to what we see on rowstore databases (Figure 1). Thus, the impact of accurate cardinality estimation remains significant even for columnstore databases. We do however observe an even larger variation in impact across workloads when compared to rowstores with large improvements in CEB and almost none to negative improvements in 5 workloads.

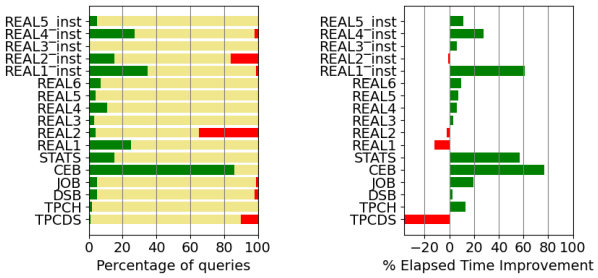
Improvement categories: In comparison to rowstores (see Section 4.2), we find a few differences for queries where elapsed time improves by 2× or more. First, Category 1 (Seek operators) is absent and most of the queries fall in category 2 or 3. For CE benchmarks, 90% of the queries lie in category 3 where underestimated cardinalities lead to spills in non-leaf operators (hash build or aggregate). In contrast, industry benchmarks have more than 30% queries and real workloads have more than 80% queries in category 2 (scans) where P_{exact} has a different join order found by fixing overestimated cardinalities, which helps in avoiding expensive scans in P_{orig} , e.g., the query has empty output or a row mode index scan changes to batch mode index scan (which processes multiple rows at a time rather than one row at a time) [4] or vice versa. Compared to rowstore, we see that JOB sees much fewer queries that benefit from use of exact cardinalities. As noted in Section 4.2 virtually all queries that improved significantly were in Category 1. In columnstore, Index



(a) Distribution of plan quality improvement or regression

(b) Log-Log scale scatter plot

Figure 8: Impact of exact cardinalities: columnstore configuration



(a) Percentage of queries improving/regressing by ≥ 2×

(b) Percentage improvement in total elapsed time

Figure 9: Impact by workloads on columnstore

Seek and Index Nested Loops join are not typically considered by the optimizer and the optimizer usually picks Index Scan and Hash Join plans in P_{orig}. Hence, the opportunities to improve plans with exact cardinality are significantly reduced.

4.5 Takeaways

(1) Accurate CE results in significant improvements both in terms of percentage of queries and total workload execution time improvement. However, there are large variations in degree of improvement depending on the workload. (2) There is opportunity to enhance our CE benchmarks to: (a) increase coverage of additional categories of improvement that appear in real workloads and industry benchmarks. For example, include queries on IMDB (or other real datasets) where the optimizer also has to deal with significantly overestimated cardinalities due to join skew and complex filter/join predicates similar to DSB [20] and TPC-H. (b) Include queries that challenge state-of-the-art query optimizers that are capable of GROUP BY and aggregation push-down. For example, in STATS, using an aggregate function such as MEDIAN or a user-defined aggregate, can ensure that accurate CE for joins is important even in DBMSs with sophisticated query optimizers. (3) Columnstore physical design are generally considered "simpler" than rowstore. Our experiments show that accurate CE remains important even in such simpler physical designs. However, large improvements (over 60%) in total elapsed time occurs in fewer workloads than in rowstores.

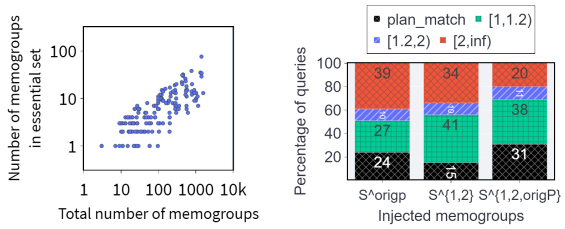


Figure 10: (a) Sizes of essential sets are much smaller than total number of memogroups. (b) Plan quality evaluation of simple heuristics for identifying essential memogroups (see Appendix).

5 SENSITIVITY TO CARDINALITY ERRORS

We take a first step towards quantifying the sensitivity of plan quality to CE errors with two experiments: (1) How few memogroups can we inject with exact CE and still get to the same plan as injecting all memogroups with exact CE? (2) How large CE errors can be tolerated by the optimizer without significantly affecting plan quality?

5.1 Essential set of memogroups

We define an *essential* set of memogroups as any subset (E) of memogroups of the query that satisfies two conditions. First, injecting exact cardinalities only for E and using optimizer-estimated cardinalities for all other memogroups[3], yields the *same plan* as when exact cardinalities are used for all memogroups (P_{exact}). Second, E is minimal in the sense that injecting exact cardinalities for only a proper subset of E does *not* yield P_{exact}.

The above definition leads to the following straightforward algorithm to identify potentially one of potentially many essential sets for a query *a posteriori*, i.e., after all exact cardinalities have been computed. The algorithm initializes the essential set to the set of all memogroups and then iteratively *shrinks* the set by attempting to remove a memogroup in each iteration. For each query that improves at least 2× in elapsed time due to injection of exact CE, we use the above algorithm to identify an essential set for the query. While different order of removal of memogroups can lead to different essential sets for the same query, in our experiments we rarely find queries with more then one essential set. Even in cases where we find multiple essential sets for a given query, there are typically only 1-2 memogroups that are not in common across all essential sets. Figure 10(a) shows a scatter plot of total number of memogroups vs. essential set size. The median essential set size is 5 and the 95th percentile is 26. For queries where the memo is "large" (i.e., contains more than 100 memogroups), a median of 3% and 95th percentile of 10% of all groups are essential.

While the above observation is intriguing, more work is needed along a few key dimensions to understand the importance of essential memogroups: (a) Assess how accurate the cardinalities must be for essential memogroups before we no longer obtain P_{exact}. Although we know that if we use optimizer estimated cardinality

[3]Note that, in Microsoft SQL Server, the optimizer estimated cardinality of a memogroup can change when we inject exact cardinality for a descendent memogroup.

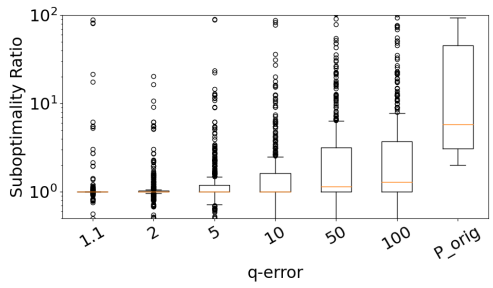


Figure 11: Sensitivity of plan quality to cardinality errors

for any essential memogroup, then we no longer get P_{exact}, however for any essential memogroup there is potentially a large gap between its exact cardinality and its optimizer estimated cardinality. (b) We know that injecting exact cardinalities for all essential memogroups is *sufficient* to obtain P_{exact} when we use optimizer estimated cardinalities for other memogroups, but we do not know if injecting exact cardinalities for all memogroups in the essential set is *necessary*. (c) Thus far, by examining essential memogroups in our experiments, we do not find any obvious patterns (e.g., number of relations in memogroup, logical operator etc.). It remains an open question as to whether the characterization of essential memogroups can be learned automatically.

Finally, given the high cost of obtaining exact cardinalities for all memogroups, we tried a few simple heuristics to identify essential memogroups a priori, and evaluated the impact of injecting exact cardinalities only for memogroups identified by each heuristic – see Figure 10(b) and Appendix A for details. However, these heuristics lead to obtaining P_{exact} in only between 15% to 31% of the queries. Therefore, although the cost of experiments using essential sets are expensive, to ensure accuracy of the results reported in this paper, we do not use these heuristics.

5.2 Impact of varying CE error

In this experiment, we evaluate the sensitivity of plan quality (i.e., elapsed time) to cardinality estimates by introducing errors in the injected cardinalities of all memogroups. As in most previous results presented in this paper, we use rowstore databases, and focus on queries which improved by 2× or more when using exact cardinalities. These are queries sensitive to accurate cardinality estimates. We use q-error [36], since it is a widely used error metric for evaluating CE techniques e.g., [22, 30, 33, 35, 37]. We vary the *magnitude* of error ϵ of q-error over the values: 1.1, 2, 5, 10, 50, 100 and for each memogroup we inject a cardinality such that its error is ϵ. Thus we empirically bound the error for each memogroup. Since q-error is symmetric, for a given value of ϵ we can potentially inject one of two values, each in a different *direction*. Thus, for each memogroup, there are two choices possible, leading to many possible combinations of cardinality injection for the query for a given value of ϵ. For the experiments reported in this section, we invoke four such combinations for every value of ϵ. In two invocations, we choose the direction of error with respect to exact cardinality for each memogroup at random. In the other two invocations, we choose the direction of error with respect to exact cardinality for *all* memogroups to be the same, one in each direction. We measure the elapsed time for each execution. Note that each of these invocations can potentially lead to a different plan for a given ϵ value. The impact of varying q-error on plan quality is captured using box plots for each ϵ value in Figure 11. The outliers are shown as dots in the figure. The y-axis is the *suboptimality ratio* of the plan with respect to elapsed time of P_{exact} and x-axis is ϵ. The rightmost box plot in each figure shows the distribution of suboptimality ratio of P_{orig}, the plan with optimizer estimated cardinalities. For clarity, we truncate the y-axis at 100, and hence a few points above that are omitted.

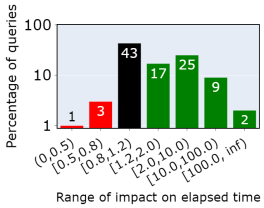
First, we observe that as the q-error increases so does the median suboptimality ratio compared to P_{exact}. At q-error of 10 the median and 75th percentile suboptimality ratios are 1.2 and 2.1 respectively. We also note that the variability of plan quality is significant for a given q-error. Further, consider the case when we empirically limit q-error for each memogroup to a small value, e.g., $\epsilon = 1.1$. We observe that although the median and 75th percentile suboptimality ratios are 1 and 1 respectively, for 3.5% of the queries the suboptimality ratios are large: ranging from 2× up to almost 100×. By examining these queries we find that plan changes typically correspond to cases where the plans have only a small difference in optimizer estimated cost compared to P_{exact}, but exhibit a large difference in elapsed time due to errors in cost model. Thus, while we find as expected that reducing q-error is beneficial to plan quality, given the large variability observed for a given q-error, the above experiment reinforces the point that it is not sufficient to measure q-error alone [28, 38] when evaluating CE techniques, and that evaluation of plan quality is critically important.

Finally, we see that when the q-error for each memogroup is large, e.g., $\epsilon = 100$, the median and 75th percentile suboptimality ratios are 2.5 and 4.6 respectively. For a small number of queries the suboptimality ratio reaches as high as 200. While these suboptimality ratios are significant, we also observe that for P_{orig} the corresponding median suboptimality ratios are 5.8 and 46.6 respectively, and the largest suboptimality ratios we observed are in excess of 500. When we examine queries where the plan at $\epsilon = 100$ is significantly improved compared to P_{orig}, not surprisingly we find that this is due to memogroups where the optimizer's estimates are significantly worse, e.g., with q-errors exceeding 1000.

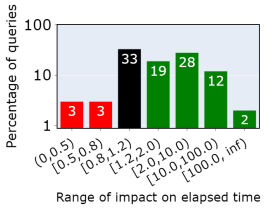
6 EFFECT OF RUNTIME TECHNIQUES: BITMAP FILTERING AND ADAPTIVE JOIN

In this section, we focus on the the impact of accurate CE in the presence of: (1) the bitmap filtering technique [5, 18], and (2) the adaptive join operator [6]. We refer the reader to Section 2 for a description of these techniques in Microsoft SQL Server. Each of these techniques has the potential to mitigate the impact of inaccurate CE that leads to choice of bad plans. Thus, our goal is to quantify the extent to which these techniques can mask the shortcomings of inaccurate CE.

For each technique (bitmap filtering, adaptive join), our methodology is as follows. We measure the elapsed time of the query in four different configurations: (i) {Opt-Est cardinalities, Technique-off} (ii) {Exact cardinality, Technique-off} (iii) {Opt-Est cardinalities,

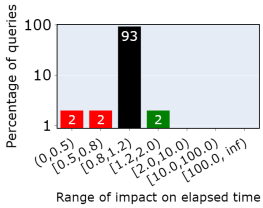


(a) Plan quality impact of enabling bitmap filtering with estimated cardinalities

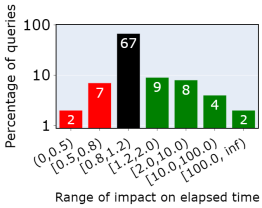


(b) Combined plan quality impact of bitmap filtering and exact CE

Figure 12: Impact of enabling bitmap filtering and injecting exact CE in Microsoft SQL Server, columnstore configuration



(a) Plan quality impact of enabling adaptive joins with estimated cardinalities



(b) Combined impact of adaptive joins and exact CE

Figure 13: Impact of enabling adaptive joins and injecting exact CE for serial plans in Microsoft SQL Server, rowstore configuration

Technique-on} (iv) {Exact cardinality, Technique-on}. We use configuration (i) as the baseline, and compare the elapsed time of the query in each of the remaining configurations with (i). Using this methodology allows us to compare the *individual* impact of using exact cardinalities and the technique, as well as the *combined* impact of using both exact cardinalities and the technique; thereby helping to quantify the extent to which runtime techniques mask the inadequacies of inaccurate CE.

6.1 Bitmap filtering

For this experiment, we use columnstores since bitmap filtering is used more extensively in columnstores when compared to rowstore. This is because bitmap filtering is applicable when the plan contains Hash Joins and Index Scans, which is typical of plans on columnstore databases, whereas on rowstore databases, other join operators find considerable usage. Figure 8(a) (discussed in Section 4.4) and Figure 12(a) respectively show the individual impact of using exact cardinalities only and bitmap filtering only. We find that using bitmap filtering alone results in 36% of the queries improving by 2× or more. In contrast, when using exact cardinalities alone, 13% of the queries improve by 2× or more. As expected, bitmap filtering results in almost no regressions since the overheads of creating and evaluating bitmap filters are relatively low, and they can often reduce the number of input rows to a join very significantly. Thus, they are a safe technique when applied to the final plan chosen by the optimizer.

When we turn on *both* exact cardinalities and bitmap filtering (see Figure 12(b)), we find that 42% of the queries improve by 2× or more. Thus, while the shortcomings of inaccurate CE are indeed masked by bitmap filtering to a significant degree, accurate CE does bring additional benefits. Among the 6% of queries that improve by 2× or more when exact CE and bitmap filtering are both used but not when using bitmap filtering alone, we find that in more than 75% of such queries the inability of bitmap filtering to mask the negative impact of inaccurate CE can be largely attributed to one of the following three reasons: (i) aggregation or group by pushdown (ii) change in aggregation operator (Steam Aggregate vs. Hash Aggregate), and (iii) a better join order and/or join method is chosen when intermediate results sizes are severely mis-estimated by the optimizer. A special case of (iii) that occurs frequently is when the actual result size of an intermediate result is 0. When exact CE is available, the optimizer places a scan of a large table on the inner side of a Nested Loops join operator, which is never invoked at runtime, whereas in the original plan the expensive scan occurs on the outer side of a Hash Join operator and cannot be avoided.

Finally, recollect from Section 2 that the placement of bitmap filters in Microsoft SQL Server is done on the final plan chosen by the optimizer. There are other techniques in the literature (e.g., [21]) where placement of bitmaps is part of plan enumeration. In such cases, the degree to which bitmap filtering masks the impact on inaccurate CE needs to be evaluated.

6.2 Adaptive join

An adaptive join operator aims to mitigate the impact of CE errors that could lead the optimizer to pick the worse operator between Index Nested Loops join and Hash Join. Note that adaptive join can potentially make the plan resilient to CE errors only when CE error can be detected at runtime in the outer child of the join. In this experiment, we focus on rowstores where Index Seek is available as an option to the query optimizer.

When using only exact cardinalities (see Figure 1(a)) 13% of the queries improve by 2× or more. In contrast, when using only adaptive join (see Figure 13(a)), we find only a few queries (less than 1%) that improve by 2× or more. This indicates that while adaptive join has clear benefits with few regressions, its ability to mask the negative impact of inaccurate CE is limited. This is also reflected when comparing Figure 13(a) with Figure 13(b), where we observe a substantial increase in percentage of queries (from less than 1% to 14%) that improve when using *both* exact cardinalities and adaptive join. We find several reasons why adaptive join is unable to mask the impact of inaccurate CE, e.g., change in join order, change in physical operator, aggregation push-down (as noted in Section 4.2). Such changes that lead to plan improvements are not achievable using adaptive join alone.

6.3 Other techniques and combinations

Microsoft SQL Server and other DBMSs support additional runtime techniques to speedup execution of query plans. Two examples of widely-used techniques are multi-core parallelism [1, 7, 10, 12] and batch mode vs. row mode versions of physical operators [4]. Query optimizers rely upon cardinality estimation to decide whether to

use techniques such as parallelism, and whether to use row mode or batch mode version of operators. Although the interaction of CE with these other techniques is beyond the scope of our study, we did perform an evaluation of impact of exact cardinalities in the presence of parallelism. We find that exact CE brings additional benefits beyond the use of parallelism. We find two reasons for such complementary improvements: (i) the decision to use parallel vs. serial operators is cost-based, and hence relies on accurate cardinality estimation. In our study, for many queries, the optimizer uses parallelism only after exact cardinality injection. (ii) Some of the largest improvements in elapsed time (e.g., 100× or more) arise due to better join orders, which is not possible through use of parallelism alone. Observe that in contrast to adaptive join and bitmap filtering which achieve speedup by doing less work and *reducing* the resources consumed by the plan, parallelism increases the resources (e.g., CPU, memory) consumed by the query to achieve speedup. Finally, we also experimented with turning on bitmap filtering, adaptive join and parallelism *together*, which is the default setting of Microsoft SQL Server. In rowstores we find that the additional benefits of exact CE continue to be significant despite the use of all three techniques, although the impact of exact CE is less significant in columnstores due to the masking effect of bitmap filtering.

7 RELATED WORK

JOB [33] focuses on impact of CE, search and cost model of the PostgreSQL query optimizer. The study introduces a new benchmark – the Join Order Benchmark (JOB) using hand-crafted queries over the IMDB database. CEB [38] provides additional queries over the IMDB schema. STATS [28], uses a real-world dataset similar to JOB, a snapshot of Stats Stack Exchange data, and also define a set of benchmark queries to compare the effectiveness of different CE techniques in PostgreSQL query optimizer. These queries contain both one-to-many and many-to-many joins, as well as a large number of filter predicates. Other than the above CE specific benchmarks, there are multiple traditional benchmarks (TPC-H [14], TPC-DS [13] and SSB [39]) that, however, are not well suited to study impact of cardinality estimation due to lack of skew and correlations. We include TPC-DS, in addition to TPC-H with skew [2] and DSB [20], due to their non-uniform data distributions, to represent the industry benchmarks. As noted in the introduction, in addition to the variety of query workloads, our study covers new and important facets of CE evaluated not covered in prior work including: (i) evaluation on a industry-strength DBMS Microsoft SQL Server with state-of-the-art optimizer and execution engine with run-time optimizations, and (ii) quantification of the degree to which CE errors impact plan quality. Unlike JOB, we do not focus on the cost model and search components of the optimizer. Also, CEB [38] presents an error metric to approximate important cardinalities for a query, while we focus on empirically identifying an essential query sub-expressions. Finally, while earlier works have considered improving efficiency of exact cardinality query optimization [17, 40], our work focuses primarily on the impact of exact CE on plan quality.

8 CONCLUSION

Our empirical study quantifies the importance of accurate CE with both rowstore and columnstore physical designs, and commonly used query runtime techniques such as adaptive join and bitmap filtering. Some of our key findings are: (i) The impact of accurate CE on plan quality is significant on both rowstore and columnstore physical designs. (ii) There is large variability in impact of CE on plan quality across workloads. (iii) Bitmap filtering noticeably masks the shortcoming of inaccurate CE on columnstores, but accurate CE additionally improves plans. (iv) Adaptive join is limited in terms of diminishing the negative impact of inaccurate CE. (v) For most queries, we find that it is sufficient to use accurate CE for a small subset of logical sub-expressions of the query to ensure that plan quality is not degraded when compared to using accurate CE for all logical sub-expressions. Finally, we refer the reader to Section 1.2 for a few open questions arising from our empirical study.

ACKNOWLEDGMENTS

We thank our colleagues Nico Bruno and Cesar Galindo Legaria for their insightful comments on the paper.

A HEURISTICS TO IDENTIFY ESSENTIAL MEMOGROUPS

We experimented with a few simple heuristics to identify essential memogroups. These heuristics share the property that the percentage of memogroups they inject grows slowly as a function of the number of memogroups of the query. The first technique limits cardinality injection to the set of memogroups that occur in the original plan P_{orig}, denoted by $S^{\{orig\}}$ The second technique restricts injection of exact CE to memogroups containing one or two relations only ($S^{\{1,2\}}$). The third technique injects exact cardinalities for the union of above two sets ($S^{\{1,2,orig\}}$). Among queries with 100 or more memogroups, the median sizes of the above three sets are 5%, 11%, and 15% of all memogroups respectively, and the plan quality impact is summarized in Figure 10(b).

First, the percentage of queries with the same plan as P_{exact} is captured by the lowest bin named *plan_match*. These percentages vary between 15% and 31% across the different heuristics. Second, for the queries where these heursitics lead to a *different* plan than P_{exact}, the bin marked $[1, 1.2)$ captures the percentage of the queries where the elapsed time is within 1.2× of the elapsed time of P_{exact}. We observe that by removing the requirement of having to match P_{exact}, the coverage of the above techniques increases significantly. In particular, $S^{\{1,2,orig\}}$ results in a plan that has no more than 20% slowdown compared to P_{exact} for 69% of the queries. Third, we illustrate a limitation of the above heuristics using the example query JOB 17a shown in Figure 4, where injecting all exact cardinalities results in avoiding expensive index seeks on both tables $cast\_info$ and *name*. In contrast, when we restrict injection to $S^{\{1,2,orig\}}$, we can only avoid expensive seeks on $cast\_info$ but the plan still incurs expensive seeks on the table *name*, since we do not fix the severe underestimation of the three table memogroup $((keyword \bowtie movie\_keyword) \bowtie cast\_info)$, which is not captured by memogroups in $S^{\{1,2,orig\}}$.

REFERENCES

[1] 2011. Understanding parallel query plans. https://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc00743.1570/html/queryprocessing/CHDHHIIF.htm. accessed on 07/25/2023.

[2] 2016. Program for TPC-H Data Generation with Skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430. last accessed on 07/25/2023.

[3] 2018. https://sqlperformance.com/2018/02/sql-plan/setting-and-identifying-row-goals. last accessed on 07/25/2023.

[4] 2019. Columnstore Index Performance: BatchMode Execution. https://techcommunity.microsoft.com/t5/sql-server-blog/columnstore-index-performance-batchmode-execution/ba-p/385054. last accessed on 07/25/2023.

[5] 2019. Intro to Query Execution Bitmap Filters. https://techcommunity.microsoft.com/t5/sql-server-blog/intro-to-query-execution-bitmap-filters/ba-p/383175. last accessed on 07/25/2023.

[6] 2019. Introducing Batch Mode Adaptive Joins. https://techcommunity.microsoft.com/t5/sql-server-blog/introducing-batch-mode-adaptive-joins/ba-p/385411. last accessed on 07/25/2023.

[7] 2019. Parallel Execution with Oracle Database. (Feb 2019). https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-parallel-execution-fundamentals-133639.pdf last accessed on 07/25/2023.

[8] 2021. Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator. https://learn.microsoft.com/en-us/previous-versions/dn673537(v=msdn.10)?redirectedfrom=MSDN. last accessed on 07/25/2023.

[9] 2022. https://www.erikdarlingdata.com/a-row-goal-riddle/. last accessed on 07/25/2023.

[10] 2022. Parallel Query (PostgreSQL 13). https://www.postgresql.org/docs/13/parallel-query.html. last accessed on 07/25/2023.

[11] 2023. DBCC DROPCLEANBUFFERS (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-dropcleanbuffers-transact-sql?view=sql-server-ver16. last accessed on 07/25/2023.

[12] 2023. Query processing architecture guide (Parallel Query Processing). https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver16#parallel-query-processing. last accessed on 07/25/2023.

[13] 2023. TPC-DS decision support benchmark. https://www.tpc.org/tpcds/. last accessed on 07/25/2023.

[14] 2023. TPC-H decision support benchmark. https://www.tpc.org/tpch/. last accessed on 07/25/2023.

[15] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.

[16] Surajit Chaudhuri. 2009. Query Optimizers: Time to Rethink the Contract?. In *ACM SIGMOD*. 961–968.

[17] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2009. Exact Cardinality Query Optimization for Optimizer Testing. *PVLDB* 2, 1 (aug 2009), 994–1005.

[18] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query Optimization in Oracle 12c Database In-Memory. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1770–1781. https://doi.org/10.14778/2824032.2824074

[19] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. 2007. Adaptive query processing. *Foundations and Trends® in Databases* 1, 1 (2007), 1–140.

[20] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[21] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-aware query optimization for decision support queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2011–2026.

[22] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[23] Campbell Fraser, Leo Giakoumakis, Vikas Hamine, and Katherine F. Moore-Smith. 2012. Testing Cardinality Estimation Models in SQL Server. In *DBTest*. Article 12.

[24] Cesar A Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. 2008. Optimizing star join queries for data warehousing in microsoft sql server. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 1190–1199.

[25] Cesar A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. 2008. Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server *(ICDE '08)*. IEEE Computer Society, USA, 1190–1199. https://doi.org/10.1109/ICDE.2008.4497528

[26] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[27] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[28] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (dec 2021), 752–765. https://doi.org/10.14778/3503585.3503586

[29] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *ACM SIGMOD*. 268–277.

[30] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[31] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L Price, Srikumar Rangarajan, Remus Rusanu, et al. 2013. Enhancements to SQL server column stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1159–1168.

[32] Per-Åke Larson, Eric N Hanson, and Susan L Price. 2012. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.* 35, 1 (2012), 15–20.

[33] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (nov 2015), 204–215.

[34] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing through Progressive Optimization. In *ACM SIGMOD*. 659–670.

[35] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[36] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (aug 2009), 982–993.

[37] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved selectivity estimation by combining knowledge from sampling and synopses. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1016–1028.

[38] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[39] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[40] Immanuel Trummer. 2019. Exact Cardinality Query Optimization with Bounded Execution Cost. In *ACM SIGMOD*. 2–17.