ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning (Extended Version)

Tarique Siddiqui
Microsoft Research
tasidd@microsoft.com

Saehan Jo*
Cornell University
sj683@cornell.edu

Wentao Wu
Microsoft Research
wentwu@microsoft.com

Chi Wang
Microsoft Research
chiw@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Today's database systems include index advisors that recommend an appropriate set of indexes for an input workload. Since index tuning on large and complex workloads can be resource-intensive and time-consuming, workload compression techniques have been proposed to improve the scalability of index tuning. Workload compression techniques aim to efficiently identify a small subset of queries in the workload to tune such that the indexes recommended when tuning the compressed workload give similar performance improvements as when tuning the input workload. In this paper, we propose ISUM, a new workload compression algorithm that is based on two key ideas: a low-overhead technique for estimating the improvement in performance of the input workload when a subset of queries is selected for index tuning, and a novel method for concisely representing information across queries in the workload that improves scalability by avoiding pairwise comparisons between queries when choosing the set of queries to tune. Our evaluation over industry benchmarks and real-world customer workloads shows that ISUM results in a 1.4× of median and 2× of maximum performance improvements for the input workload when compared to prior techniques over similar compressed workload sizes.

CCS CONCEPTS

• **Information systems → Autonomous database administration**; • **Computer systems organization → Cloud computing**.

KEYWORDS

Index tuning, workload compression

ACM Reference Format:
Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning (Extended Version). In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3514221.3526152

1 INTRODUCTION

Indexes are critical for improving query performance in databases. Over the past three decades, a number of *index advisors* [15, 24, 38] have been proposed that search for appropriate indexes given an input workload and a set of constraints. However, index tuning can be resource-intensive and time-consuming when the input workloads are large and consist of complex SQL queries. Although this challenge exists for index advisors in on-premises databases, it is amplified in cloud database-as-a-service setting [1, 4] where large numbers of databases need to be tuned by the service provider [19].

To improve tuning efficiency, prior work [12, 21] has recognized the importance for *workload compression*, i.e., finding a smaller subset of queries (which we refer to as a compressed workload) from the input workload that can be used for index tuning. There are two requirements for any workload compression technique. First, the improvement in the performance of the input workload due to the indexes recommended on tuning the compressed workload should be close to that of the indexes obtained by tuning the input workload. Second, it is crucial that the compressed workload can be found *efficiently*, otherwise the advantage of selecting a smaller workload is negated.

Limitations of existing workload compression techniques. Prior work can be categorized into two categories: indexing-agnostic and indexing-aware techniques. Among indexing-agnostic approaches, a simple technique is to uniformly sample a subset of queries. However, sampling fails to capture similarities between queries and often misses out queries that may lead to substantial improvement in performance but may be less frequent in the workload. Recently, [21] proposes a greedy algorithm called GSUM that maximizes the coverage of features (e.g., columns) in the workload while also ensuring that the summary workload is representative (i.e., having similar distribution to that of the entire workload). Unfortunately, the proposed approach may select queries which are common in the workload but may not lead to a significant improvement in performance on index tuning.

Among indexing-aware techniques, [12] proposes a clustering-based approach that groups query instances sharing the same template (i.e., queries that are identical except for parameter bindings) and then clusters instances within the same group using a distance function that quantifies the loss in performance when indexes of one instance is chosen over the other. The queries in the compressed workload are then selected by uniformly sampling from each of the clusters. There are two issues with this approach. First, the clustering can be inefficient when there are a large number of instances per group due to *pairwise* comparison between queries. Approximations such as selecting random seeds to avoid pairwise

[*] Work done as an intern at Microsoft Research

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526152

comparisons, and limiting the number of iterations can help reduce the compression time but affect the quality of selected queries (see Section 8). Second, we observe that real workloads can have many more unique templates than the desired compressed workload sizes. For such cases, the proposed distance function cannot quantify the similarity between queries with different templates, making the approach less effective when we cannot select at least one instance per template.

Our Approach. In this work, we develop ISUM[1], an efficient and scalable workload summarization technique. ISUM can result in better index recommendations than existing indexing-aware techniques [12] while achieving similar efficiency compared to state-of-the-art indexing-agnostic approaches such as GSUM [21]. We make two main contributions. First, we develop a new technique that can efficiently estimate the performance improvement over the input workload when selecting a subset of queries for tuning. We show that the estimated performance improvements are highly *correlated* with the optimizer estimated improvements over the input workload. To be able to do so, we represent each query as a set of features such that two queries with similar values of the features will likely result in similar set of indexes. This representation also allows us to quantify the similarity between queries with different templates, thereby addressing the issue with the distance function in [12]. We further observe that selecting queries solely based on similarity is not always effective; rather the selected queries should be more similar to those queries in the workload that have high potential for performance improvement on adding indexes (e.g., queries with high costs and more selective predicates). To do so, we compute the feature values and adjust the similarity scores using query costs and statistics that are indicative of the potential improvement in the costs of queries.

We observe that a naive approach (called *all-pairs*) that compares each query with the other queries in the workload and selects the one that maximizes the estimated improvement has a prohibitively long running time over large workloads. To improve the efficiency, we develop a summarization technique that aggregates query-level features into workload-level features such that higher weight is given to features of queries with high potential for performance improvement. The single workload-level representation allows measuring the similarity of each query with the input workload without performing pairwise comparisons, thereby allowing us to develop a fast linear-time algorithm. We present a theoretical as well as an empirical analysis to show that the linear-time algorithm has bounded and small loss in the accuracy of estimated improvement compared to all-pairs algorithm.

We perform an extensive evaluation of ISUM with state-of-the-art workload compression techniques on multiple real and synthetic workloads. Our results show that ISUM results in a median of 1.4× and a maximum of 2× performance improvements compared to prior techniques for the same compressed workload sizes. Furthermore, given an input workload consisting of queries along with their costs, the time to select the compressed workload is small ($<$ 1%) compared to the tuning time of the compressed workload. Furthermore, we evaluate ISUM by varying the complexities of queries in the workload,the number of instances per template, as well as the configuration sizes of indexes, and find that ISUM outperforms baselines on a large majority of these settings while there is no one baseline that works well over many settings.

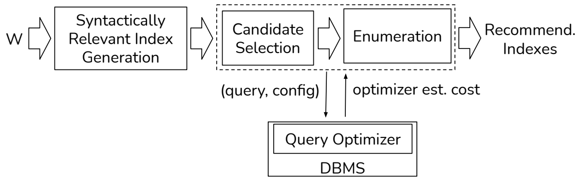
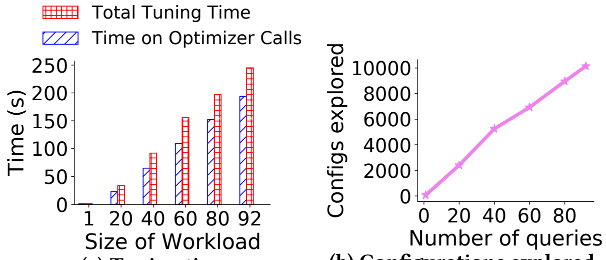


Figure 1: A typical architecture of index advisor as described in [15]



(a) Tuning time **(b) Configurations explored**

Figure 2: Scalability challenges in index tuning (TPC-DS workload)

Finally, in this paper, we have assumed that ISUM, as well as other workload compression techniques we compare with (e.g., [12, 21]), pre-process the input workload and output a subset of queries for index tuning. However, commercial index advisors such as DTA [3] need to report the estimated improvement of the recommended configuration on the entire input workload. For large input workloads, this estimation step can take a large fraction of the overall tuning time, and hence can significantly reduce the benefits of workload compression. Moreover, index advisors support tuning with a time-budget (e.g., see DTA [13]), and hence workload compression needs to identify queries for tuning incrementally as more queries from the input workload are consumed by the tuner. We discuss these practical challenges and limitations of integrating workload compression into index advisors and identify areas of future work in Section 10.

2 BACKGROUND

2.1 Scalability Challenges in Index Tuning

Figure 1 depicts the typical architecture of an index advisor as described in [15]. It consists of three steps: (1) generation of syntactically-relevant indexes by parsing and combining relevant columns in the query, (2) *candidate selection* to identify indexes that improve the performance of each query, and 3) *configuration enumeration* that searches the space of subsets of candidate indexes (from all queries) and picks a configuration (i.e., subset) that results in the maximum improvement (i.e., decrease in the cost) on the workload. Specifically, configuration enumeration is a combinatorial optimization problem that is even hard to approximate [11, 18]—the number of potential configurations grows *exponentially* with respect to the number of candidate indexes. To estimate the improvement for a given query-configuration pair, index advisors rely on a "what-if" API [16], which is an extended functionality of the query optimizer that can estimate the cost *without* building the indexes.

Essentially, the scalability and efficiency of an index advisor depends on the following factors: (1) the number of queries in

[1] ISUM stands for Index-based Workload Summarization

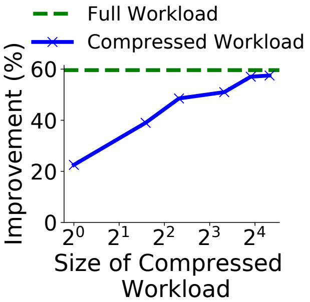


Figure 3: Impact of workload compression (TPC-DS workload)

the workload, (2) the number of configurations enumerated, and (3) the number of optimizer invocations or what-if calls [7, 19]. Figure 2a depicts the increase in tuning time for a state-of-the-art index advisor [7] as we increase the number of queries in the TPC-DS workload. As we can see, the tuning time grows significantly as we increase the size of the workload. This is primarily because the space of configurations to explore increases (Figure 2b), resulting in a large number of expensive optimizer calls (consuming 70% to 80% of the overall tuning time).

To show the efficacy of workload compression, Figure 3 depicts the impact of the compressed workload as selected by our compression algorithm (described in later sections). As we can see, using a compressed workload of carefully selected 20 queries (from 92 queries), we are able to achieve improvement in performance close to tuning the entire workload in much less time. Note that this time includes both the compression time as well as the time taken to tune the compressed workload.

2.2 Problem Formulation

Let $W = \{q_1, q_2, ..., q_n\}$ be an input workload of n queries. Let $C(q_i)$ be the optimizer estimated cost of q_i with the existing physical design (i.e., without adding or removing indexes), provided as part of the input workload. Many database systems typically log the plan details, e.g., Query Store [5] in Microsoft SQL Server, that can be provided along with query texts to reduce the number of optimizer calls during workload compression. Let $C(W)$ be the optimizer estimated cost for the input workload W with

$$C(W) = \sum_{i=1}^{n} C(q_i).$$

For a given index configuration of size m, let $I(W, A, m) = \{i_1, i_2, ..., i_m\}$ be a set of m indexes recommended by an index advisor A on tuning W. Let $C_I(q_i)$ be the optimizer estimated cost of q_i and correspondingly $C_I(W)$ be cost for the input workload W, when using the (hypothetical) indexes in $I(W, A, m)$.

We are interested in a subset of queries of W, tuning which can result in indexes that minimize the cost of W. Formally, let W_k be a set of k query-weight pairs, consisting of k queries from W. The weight indicates how representative a query in W_k is of the queries in W (discussed in more detail in subsequent sections). Let $I(W_k, A, m)$ be the m indexes recommended by an index tuner A when tuning W_k. Note that it is expensive to invoke index advisor A while evaluating subset of queries for W_k, hence during compression we need to use an efficient estimation technique in place of A as we discuss shortly.

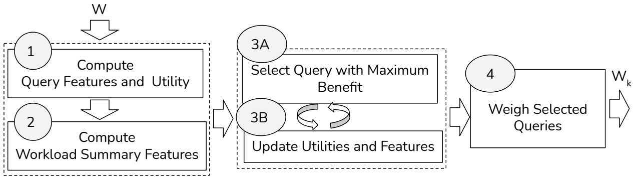

W
1
Compute
Query Features and Utility
2
Compute
Workload Summary Features
3A
Select Query with Maximum Benefit
3B
Update Utilities and Features
4
Weigh Selected Queries
W_k

Figure 4: Architecture of ISUM

Definition 1 (Improvement). *We define the improvement, Δ, as the* decrease *in the cost of the workload W when using the indexes in $I(W_k, A, m)$, i.e., $\Delta = C(W) - C_{I(W_k,A,m)}(W)$.*

To simplify notation, we omit A and m in the rest of the paper.

Problem 1 (Workload Compression for Index Tuning)**.** *Given as input the workload W consisting of n queries, their optimizer estimated costs $C(q_1), C(q_2), ..., C(q_n)$, and the number of queries k to be selected, return W_k consisting of k queries and their respective weights $w_1, w_2, ..., w_k$ s.t. Δ is maximized, i.e.,*

$$W_k = \underset{S_k \subset W}{\operatorname{argmax}} \Delta.$$

Or, equivalently, since $C(W)$ does not vary with W_k,

$$W_k = \underset{S_k \subset W}{\operatorname{argmin}} C_{I(W_k)}(W).$$

Note that $C_{I(W)}(W) \leq C_{I(W_k)}(W)$ since selecting indexes by tuning a subset of the workload in general cannot result in better performance improvement (on the entire workload) than when compared to the improvement obtained from indexes by tuning the entire workload. This is because the latter considers a larger space of indexes. Hence, optimizing the above problem will also minimize the difference between $C_{I(W)}(W)$ and $C_{I(W_k)}(W)$.

The computation of $C_{I(W_k)}(W)$ requires two steps: (1) running index tuner on W_k to determine $I(W_k)$, followed by (2) computing the cost of W using $I(W_k)$ as (hypothetical) indexes. Clearly, it is costly to invoke the index advisor to compute $I(W_k)$ and running these two steps negates the purpose of workload compression. Thus, in this work, one of the key challenges involves developing a mechanism to estimate $C_{I(W_k)}(W)$ without index tuning. Furthermore, we should be able to find the compressed workload W_k efficiently. In particular, the total time taken to compress the workload and tune the compressed workload should be smaller than the time it takes to tune the input workload. Lastly, the compression algorithm should be independent of index advisor and tuning constraints, and should select queries which are generally effective across a wide range of constraints.

3 OVERVIEW OF ISUM

We first describe our intuition on how we select a query in the compressed workload. Then, we discuss how we select a set of queries while accounting for inter-dependencies between selected queries. Figure 4 outlines the key steps.

ISUM selects a query if (a) it has a high potential for reduction in its cost on adding indexes, referred to as the *utility* of the query, and (b) the indexes obtained on tuning the selected query has high potential for reducing the cost of other queries in the input workload, referred to as the *influence* of the query. As discussed earlier, it is expensive to make optimizer calls or use index advisor to determine the indexes or compute the reduction in cost. Instead, we use the costs of queries and the statistics such as selectivity to estimate the potential for reduction in the cost of the query on adding indexes. Specifically, a query with higher cost and more selective filters has higher potential for reduction in its cost. For measuring influence, we also need to characterize how useful are indexes obtained from the query to other queries in the workload. To do so, we represent each query using a set of features and assign weights to features using size of tables, position of columns in the query, and statistics such as selectivity such that two queries with a similar set of feature values (measured using *weighted Jaccard*) result in a similar set of indexes. A query with higher similarity with other queries that have high potential for reduction in their costs has higher *influence*. Putting together, we select a query with the highest sum of utility and influence, defined as the *benefit* of selecting the query.

In a compressed workload consisting of multiple queries, a) and b) also depend on the other queries that are selected. For instance, if a query helps select a given set of indexes, there is no additional advantage of selecting another query that helps select the same set of indexes. To address this, we develop a technique that updates the feature weights and potential for reduction in cost of queries such that queries that are dissimilar from already selected queries get higher values of a) and b). Specifically, we propose a *greedy* algorithm (Step 3 in Figure 4) that incrementally selects queries in decreasing order of their benefits (Step 3A in Figure 4). After every selection, the rest of the queries are updated to consider the impact of previously selected queries (Step 3B in Figure 4).

The above approach requires the computation of the similarity of each query with every other query in the workload and thus has prohibitively high runtime complexity (*quadratic* in the number of queries). To improve the efficiency, we propose a technique that summarizes query-level information into a single workload-level representation, called *summary features* (Step 2 in Figure 4). Our key idea in creating the summary features is to measure the importance of features of a query at the workload level such that the features of queries with high potential for reduction have higher weights in the summary features. The single workload-level representation allows measuring the similarity of each query with the input workload without performing pairwise comparisons, thereby allowing us to develop a linear-time algorithm which is significantly faster.

Finally, note that the queries in the compressed workload represent the input workload to a varying degree. Thus, we weigh queries (Step 4 in Figure 4) such that the weight of a query captures the relative importance of the query with respect to the input workload (Section 8). The compressed workload, consisting of queries and their corresponding weights, is then passed to the index tuner to generate index recommendations.

4 ESTIMATING IMPROVEMENT

In this section, we introduce a technique that estimates the performance improvement over the input workload for a set of selected queries such that it correlates with the actual performance improvement as measured by the optimizer in the presence of recommended indexes. We first discuss how we estimate the improvement for a single query (defined as the *benefit* of the query) and then extend it to a set of queries.

4.1 Benefit of Single Query

The computation of benefit of a query consists of two components: (1) the potential for reduction in cost of the queries (defined as *utility*) and (2) the similarity of the query with respect to other queries in the workload.

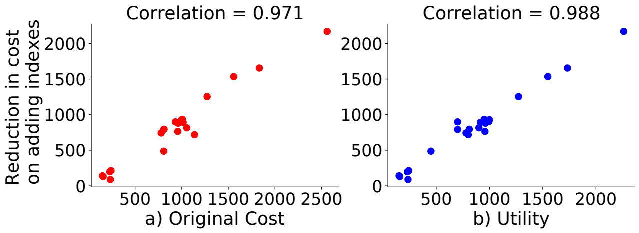


Figure 5: Relationship between utility and reduction in cost due to indexes when each query is tuned independently in the TPC-H workload. a) Utility is set to the original cost of the query, b) Utility considers both original cost and statistics such as selectivity of filter and join columns. Each point in the charts represents one query in the workload.

Utility. The potential for reduction in costs of queries due to indexes usually depends on the costs of *filter*, *join*, *order-by*, and *group-by* operators as well as the selectivity of filter and join operators. For instance, if the contribution of these operators to the cost of the query is high, or if the selectivity of filter and join predicates is low, the reduction in cost of the query may be high since indexes can help accelerate such operators. Let $C(q_i)$ be the original cost of the query q_i, and let $\Delta(q_i)$ be the estimated reduction in cost of q_i due to indexes. If $Sel(q_i)$ is the average selectivity of the filter and join columns in the query, then $\Delta(q_i) = (1 - Sel(q_i)) \times C(q_i)$ is often correlated with actual improvements. Here, we ignore the improvements due to grouping and ordering properties which is hard to quantify without query optimization. Notably, our analysis over standard benchmarks and real workloads shows that even only the cost of the query is often highly correlated with the actual performance improvement (see Figure 5), indicating that if the statistics are not available, one can use the original cost of the query as a proxy for the potential reduction in cost. For instance, on the TPC-H database (see Figure 5), we found that the reduction in cost of a query when using all indexes recommended by an index advisor after tuning that query is highly correlated (Pearson correlation = .97) with the cost of the query, and using selectivity of columns as described above only increases the correlation by a small percentage. In our experiments, we show the impact of estimating reduction on workload compression using both the scenarios.

We introduce the notion of utility that estimates the fraction of total reduction in cost of the workload on selecting a given query.

Definition 2 (Utility). *The utility of query q_i, denoted by $U(q_i)$, is the estimated reduction in cost of q_i (when all indexes on q_i are added) relative to the total estimated reduction in cost across all queries in the workload:*

$$U(q_i) = \frac{\Delta(q_i)}{\sum_{q_j \in W} \Delta(q_j)}.$$

Intuitively, if a query has higher utility, it contributes to a larger reduction in cost of the entire workload. While utility is a good indicator of the potential improvement for a query, we observe that it is less correlated with the improvement over the input workload. Figure 6a depicts the improvement over the entire workload as a function of utility of each query in the TPC-H workload. For each query, we selected the maximum number of indexes returned by a state-of-the-art index advisor [3], and measured the improvement over the entire workload using those indexes. While there is some correlation between utility of queries and improvements, we see

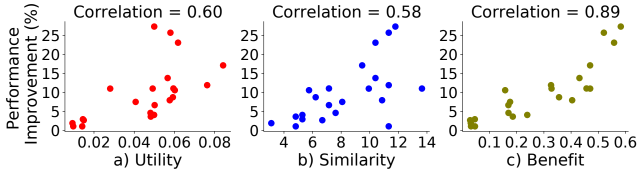


Figure 6: Impact of cost, similarity, and benefit on performance improvement of TPC-H workload

that indexes from queries with high utility are not always helpful in reducing the costs of other queries.

Similarity. Another way of measuring the benefit of a query is to look at its similarity with the other queries in the workload. Prior techniques on workload compression usually select queries that have high similarity with the queries in the workload (e.g., [21]). In order to be useful in the context of index tuning, the similarity here should be defined in terms of the usefulness of indexes between queries. We develop such a similarity function using indexable columns and weighted Jaccard [33] (described in Section 4.2).

Figure 6a depicts the performance improvement over the entire TPC-H workload for a query vs. the sum of its similarity scores with the other queries, i.e., its similarity with the workload. Once again, we observe that while there is some correlation between the similarities and the improvement, the queries with the highest similarity do not necessarily result in the maximum improvements.

Capturing Similarity and Utility Together. While utility measures the potential for reduction in cost for a query, say q_i, it does not capture the reduction in costs of other queries because of q_i. Likewise, similarity measures how useful the indexes selected from a query q_i are to other queries, but it does not capture the amount of reduction in cost of other queries if q_i is selected. To address this, we introduce the notion of 'influence'.

Definition 3 (Influence of a single query ($F_{q_i}(q_j)$)). *An influence of query q_i on query q_j is the reduction in the utility of query q_j when query q_i is selected for index tuning, computed using similarity and utility as follows:*

$$F_{q_i}(q_j) = S(q_i, q_j) \times U(q_j), \forall q_j \notin W_k, q_i \in W_k.$$

In other words, the influence of q_i on q_j captures the utility of q_j proportional to q_i's similarity with q_j. Furthermore, we assume that two queries both selected in W_k do not influence each other, since both queries can be tuned by the index tuner, i.e., $F_{q_i}(q_j) = 0$, $\forall q_j, q_i \in W_k$. We can now define the 'benefit' of selecting a query that estimates its performance improvement over the workload.

Definition 4 (Benefit of a single query ($B(q_i)$)). *The benefit of query q_i is the sum of its utility and its influence on other queries in the workload, defined as*

$$B(q_i) = U(q_i) + \sum\nolimits_{q_j \in W - \{q_i\}} F_{q_i}(q_j).$$

Intuitively, a query is more useful if it has high utility and high influence on the unselected queries in the workload. To verify the usefulness of this metric, we compared the benefit of a query (measured as sum of the benefit of the query with other queries in the workload) with the performance improvement over the workload on selecting the query for index tuning (Figure 6c). As we can see, using benefit is more correlated (0.89) than using utility (0.60) and similarity (0.58) alone. We observed similar results in our experiments over TPC-DS and real workloads.

Before we discuss how we compute the benefit for a *set* of selected queries, we describe the similarity measure.

4.2 Similarity Measure

A key component of our solution is the computation of indexing-aware similarity, $S(q_i, q_j)$, between a pair of queries q_i, q_j in the workload. Previously, [12] has proposed an indexing-aware distance function for measuring the similarity between two queries. However, the distance function only captures two queries that have the same set of table and join columns (referred as *signature*). Examples of such queries include multiple instances of an stored procedure that differ only in parameter bindings. For queries that have at least one mismatching table or join predicate, the distance function is undefined. As such, for workloads with high variance in query templates, the distance function is less effective. Here, we discuss a low-overhead similarity measure that works for queries with differing signatures and gives high correlation with the performance improvement as estimated by the optimizer.

Similarity using Candidate Indexes. For developing an indexing-aware similarity measure, an ideal option is to look at the overlap between the sets of selected indexes from both queries. While such a measure is effective in improving the correlation between benefits and improvements (e.g., we observe a high correlation of 0.93 on TPC-H), it requires index tuning for *every* query, and hence negates the very purpose of workload compression. Instead, an approximation could be to look at the set of *candidate indexes* assuming that the overlap between the sets of candidate indexes is indicative of the overlap between selected indexes. If CI_{q_i} and CI_{q_j} are the sets of candidate indexes generated for q_i and q_j respectively, the overlap can be measured using the Jaccard [33] similarity as follows:

$$S(q_i, q_j) = \frac{CI_{q_i} \cap CI_{q_j}}{CI_{q_i} \cup CI_{q_j}}.$$

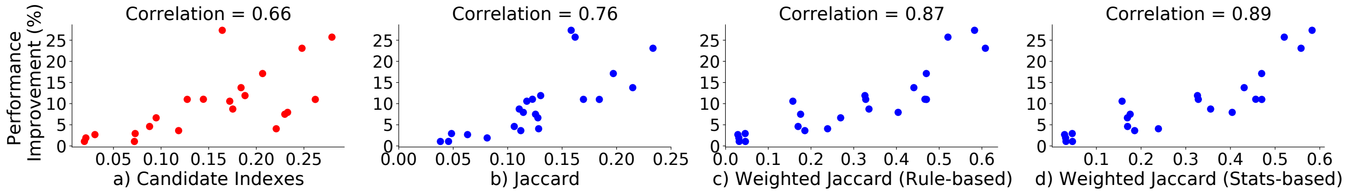
However, we observe that using candidate indexes as the similarity measure is often less correlated with the actual improvement (see Figure 7a). This is because a candidate index in one query may not exactly match any of the candidate indexes in the other query (e.g., the ordering of a same set of columns in candidate indexes may be different depending on selectivity), but may still lead to an improvement.

Similarity using Indexable Columns. To address the above issue as well as to reduce the overhead involved in generating candidate indexes, we instead consider the individual columns (called *indexable* columns) in candidate indexes for measuring similarity.

Definition 5 (Indexable Column). *A column in a query is indexable if it is part of a filter or join condition, or if it specifies the grouping or ordering of tuples.*

Specifically, we consider the following columns as indexable columns: (1) filter columns, (2) join columns, (3) group-by columns, and (4) order-by columns. Index advisors derive candidate indexes from indexable columns by combing them in different orders. Let C_{q_i} and C_{q_j} be the sets of indexable columns for q_i and q_j. The similarity between q_i and q_j can then be defined in terms of indexable columns (instead of selected/candidate indexes) as follows:

$$S(q_i, q_j) = \frac{C_{q_i} \cap C_{q_j}}{C_{q_i} \cup C_{q_j}}.$$



a) Candidate Indexes b) Jaccard c) Weighted Jaccard (Rule-based) d) Weighted Jaccard (Stats-based)

Figure 7: Impact of using different similarity measures for computing benefit on TPC-H workload.

The problem with the above measure is that all columns are assumed to be equally effective in reducing the cost of the query. To address this, we weigh columns based on statistics and rules that are indicative of how important they are in reducing the cost.

Measuring Index-Specific Importance of Columns. A large part of the query cost usually involves accessing, joining, or ordering a few expensive tables. Thus, index advisors typically select indexes on such tables. Let $n(t_i)$ be the size of the input table t_i, we assign a weight to t_i as follows:

$$w_{table}(t_i) = \frac{n(t_i)}{\sum_j n(t_j)}.$$

We discuss two approaches that combine weights of tables with additional factors to assign weights to individual columns.

(1) Rule-based. The importance of indexable columns can vary depending on whether they occur as part of filter/join predicates, or group-by/order-by clauses. Thus, an efficient way of measuring the importance is to count the proportion of candidate indexes that a column belongs to. Let $d(t_i)$ denote the total number of possible candidate indexes that can be generated by combining indexable columns in different orders, and $d(t_i, c)$ be the fraction of candidate indexes in $d(t_i)$ that contain the indexable column c. The weight of column c based on its position and weight of the corresponding table can then be computed as:

$$w(c) = \frac{d(t_i, c)}{d(t_i)} \times w_{table}(t_i).$$

The estimation of $d(t_i, c)$ and $d(t_i, c)$ does not require the invocation of an index advisor. Index advisors typically apply a set of rules that combine indexable columns in different orders to generate candidate indexes. For instance, Table 1 depicts a common set of such rules that we use in this work. Observe that not all columns generate equal number of candidates, e.g., order-by columns when used in the index should always be the leading columns. As such, a selection or join column is more likely to be part of the candidate indexes than an order-by column. Thus, given a set of common rules and the number of columns for each position, we can efficiently estimate $d(t)$ and $d(t, c)$.

(2) Statistics-based. Instead of counting the number of candidate indexes generated from an indexable column, one can use statistics such as selectivity or density [6] of the columns. If u in the number of distinct values for the column, then density is measured as $1/u$. Two queries with similar values of these statistics tend to generate similar candidate indexes since index advisors use these statistics to estimate the importance of an index as well as to order the columns in an index. We decide whether to use selectivity or density depending on the position of the column. For filter and join columns, we use the selectivity, while for order-by and group-by columns we use the density since the costs of corresponding operations are usually correlated with the number of unique values in the column(s). In general, a smaller value of these statistics leads to a higher weight, indicating that building an index on such a column will lead to higher improvement in cost. Let $s(c)$ be the value of statistics for the column c, then the weight of c can be computed as:

R1	selection
R2	join
R3	selection + join
R4	join + selection
R5	order-by + selection + join
R6	group-by + selection + join
R7	order-by + join + selection
R8	group-by + selection + join

Table 1: A set of rules for combining indexable columns to estimate the number of candidate indexes.

$$w(c) = (1 - s(c)) \times w_{table}(t_i).$$

We call indexable columns as features for the query and use the normalized weights as the values of the features, computed using min-max normalization as follows:

$$\bar{w}_{c_i} = \frac{w_{c_i}}{\max_j(w_{c_j}) - \min_j(w_{c_j})}.$$

Definition 6 (Query features). *Query features is a set of normalized weights, one for each indexable column in the query.*

For simplicity, we use the same symbol q_i to represent the query features of query q_i. We use q_{ic} to denote the normalized weight of the column c in q_i.

Computing similarity between query features. Intuitively, the query features for each query represents a region in a multi-dimensional space where each feature (i.e., indexable column) represents a dimension. We measure the similarity between two queries as the overlap between their corresponding regions in this space, such that a higher overlap between the regions of two queries results in higher relevance of the indexes to each other. The overlap between the query features q_i and q_j can be measured using the *weighted Jaccard* [33], defined as follows:

$$S(q_i, q_j) = \frac{\sum_c \min(q_{ic}, q_{jc})}{\sum_c \max(q_{ic}, q_{jc})}.$$

More specifically, the weighted Jaccard measures the area of overlap between queries normalized by the total area covered by the queries. Our empirical results show that weighted Jaccard, when used as part of the benefit measure, correlates strongly with the performance improvement (Figure 7), giving better results than using candidate indexes or Jaccard similarity with unweighted columns. Furthermore, the differences in improvements when using statistics-based approach vs rule-based approach is not significant; however, the statistics-based approach requires additional overheads of storing and maintaining histograms for estimating selectivity and density of indexable columns.

4.3 Benefit of Set of Queries

So far, while computing the benefit of selecting a query, we assumed that no other query is selected. However, a compressed workload may consist of multiple queries and thus, the benefit of a query also depends on the other queries that are selected.

To address this, we update the costs and query features of the unselected queries. We use $q_j|q_i$ to denote the updated q_j after considering the influence of selected query q_i. If there are multiple queries $q_{i_1}, q_{i_2}, ..., q_{i_n}$ that are selected, we update q_j with these queries in their order of selection. We discuss how we decide the order of selection shortly. We use $q_j|q_{i_1}, q_{i_2}, ..., q_{i_n}$ to denote that q_j is updated using $q_{i_1}, q_{i_2}, ..., q_{i_n}$ in order. We now discuss how we update the cost and query features of q_j assuming that only one query q_i is selected.

Updating utility. When a query q_i is selected, we assume that the selected indexes on q_i would help reduce the utility of q_j *proportional to its similarity*, i.e.,

$$U(q_j|q_i) = U(q_j) - U(q_j) \times S(q_i, q_j).$$

In other words, we reduce $U(q_j)$ by $F_{q_i}(q_j)$.

Updating query features. Further, we update the query features of q_j so that queries similar to q_i have reduced similarity to each other. There are two ways of updating the query features. One option is to reduce the weights of features in q_i by $S(q_i, q_j)$, i.e.,

$$q_j|q_i = q_i - S(q_i, q_j).$$

Another option is to set the weights of the indexable columns that are present in the selected query q_j to 0, assuming that this column has been "covered" and the updated utility should be calculated based on the uncovered columns. Formally,

$$q_j|q_i : \quad \textbf{SET } q_{jc} = 0, \quad \forall q_{ic} > 0.$$

We empirically explore the performances of update strategies in Section 8.2 and observe that the second option is usually more effective.

Now that we have discussed how we update the query features and utility after selecting queries, we introduce the notion of *conditional influence*, which extends the notion of influence to take into account the impact of a sequence of selected queries. For ease of exposition, given a set of selected queries $Q = \{q_1, ..., q_K\}$, we use $\pi(Q) = q_{i_1}, ..., q_{i_K}$ to represent an arbitrary order/sequence of Q. We also use $q|\pi(Q)$ to denote the updated query q by following the sequence of queries in $\pi(Q)$.

Definition 7 (Conditional Influence). *Given a sequence of selected queries $\pi(Q)$, the conditional influence of a query $q \notin Q$ on a query $q' \notin Q$ is the reduction in cost of q' ($q' \neq q$) given the influence of queries in the order of $\pi(Q)$, defined as*

$$F_{q|\pi(Q)}(q'|\pi(Q)) = S(q|\pi(Q), q'|\pi(Q)) \times U(q'|\pi(Q)). \quad (1)$$

To simplify notation, in the following we use

$$F_{q|\pi(Q)}(q') := F_{q|\pi(Q)}(q'|\pi(Q)),$$
$$S_{\pi(Q)}(q, q') := S(q|\pi(Q), q'|\pi(Q)),$$
$$U_{\pi(Q)}(q') := U(q'|\pi(Q)).$$

As a result, Equation 1 can be alternatively written as

$$F_{q|\pi(Q)}(q') = S_{\pi(Q)}(q, q') \times U_{\pi(Q)}(q'). \quad (2)$$

Definition 8 (Cumulative Influence). *The cumulative influence of a sequence of selected queries $\pi(Q) = q_{i_1}, ..., q_{i_K}$ over a query $q' \notin Q$ is the sum of the conditional influence of q_{i_l} on q' with respect the prefix sequence $q_{i_1}, ..., q_{i_{l-1}}$, where $1 \leq l \leq K$. Formally,*

$$F_{\pi(Q)}(q') = \sum\nolimits_{l=1}^{K} F_{q_{i_l}|q_{i_1},...,q_{i_{l-1}}}(q'). \quad (3)$$

Definition 9 (Benefit of a set of queries). *The benefit $B(Q)$ of a set of queries $Q = \{q_1, ..., q_K\}$ is the sum of the utilities of the queries in Q and the* maximum *cumulative influence on queries not in Q w.r.t. any order $\pi(Q)$. Formally,*

$$B(Q) = U(Q) + \max_{\pi(Q)} \sum\nolimits_{q' \in W - Q} F_{\pi(Q)}(q'), \quad (4)$$

where $U(Q) = \sum_{q \in Q} U(q)$.

Problem 2 (Maximizing Benefit). *We can now define the original workload compression problem (Problem 1) in terms of the benefit of a set of queries as follows:*

$$W_k = \underset{S_k \subset W}{\operatorname{argmax}} B(S_k).$$

Hardness: We show that even a relaxed version of Problem 2 is NP-hard. To simplify, we ignore that $U(Q) = \sum_{q \in Q} U(q)$ in Equation 4 and assume that the influence of a query q_i on q_l does not depend on selection of other queries. In other words, conditional influence of a query is equivalent to its influence, i.e.,

$$W_k = \underset{S_k \subset W}{\operatorname{argmax}} \Big(\sum_{q_l \in W} F_{S_k}(q_l) \Big).$$

This problem is a variant of the k-center problem, where, given a set of n points in a metric space and an integer $k \leq n$, the goal is to find a set of k points such that the sum of distances of the n points to the k selected points is maximized or minimized. The k-center problem is known to be NP-hard [26].

Algorithm 1 FindMaxBenefitQuery

```
$maxbenefit$ = -1;
$maxbenefitquery$ = $null$
for all $q_i \in W$ do
    if all $feature$ in $q_j.features = 0$ then
        continue;
    end if
    $benefit = q_i.utility$;
    for all $q_j \in W$ do
        $benefit += S(q_i, q_j) \times q_j.utility$;
    end for
    if $benefit \geq maxbenefit$ then
        $maxbenefit = benefit$
        $maxbenefitquery = q_i$
    end if
end for
Return $maxbenefitquery$
```

5 AN ALL-PAIRS GREEDY ALGORITHM

A key requirement to scale to a large input workload is the efficient computation of the compressed workload. Given that the problem of maximizing the benefit of selected queries is NP-hard (Section 4.3), we develop a greedy algorithm (Algorithm 2) that trades-off accuracy to solve the optimization problem efficiently. We discuss the steps in the algorithm and then analyze its optimality.

The algorithm starts with an empty set. In each iteration, the algorithm selects the query with the maximum *conditional benefit*, defined as follows (and computed by Algorithm 1):

Algorithm 2 `Greedy Selection`

```
1:  for all $q_i \in W$ do
2:      $q_i.utility$ = ComputeUtility($q_i$)
3:      $q_i.features$ = ComputeQueryFeatures($q_i$)
4:  end for
5:  $W_k = \emptyset$
6:  $T \leftarrow W$
7:  while $W_k$.size()<$k$ do
8:      $q_s \leftarrow$ FindMaxBenefitQuery($W$)
9:      $W_k \leftarrow$ AddQuery($W_k$,$q_s$)
10:     $T \leftarrow$ RemoveQuery($T$,$q_s$)
11:     $T \leftarrow$ UpdateWorkload($T$, $q_s$)
12:     $T \leftarrow$ ResetQueryFeaturesIfAllWeightsZero($T$, $W$)
13: end while
14: Return $W_k$
```

Definition 10 (Conditional Benefit). *Given a sequence of selected queries $\pi(Q)$, the conditional benefit of a query $q \notin Q$ is the sum of the (discounted) utility of q and its conditional influence over the unselected queries (other than q), with respect to selected queries in the order of $\pi(Q)$. Formally,*

$$B_{\pi(Q)}(q) = U_{\pi(Q)}(q) + \sum\nolimits_{q' \in W - \left(Q \cup \{q\}\right)} F_{q|\pi(Q)}(q'). \quad (5)$$

Intuitively, at each greedy step, we select the query with the maximum conditional benefit over the remaining queries in the workload. After a query is selected, we update the query features and costs of unselected queries as discussed in Section 4.3.

Note that in some cases, the weights of all features of a query may be set to zero after selecting a few queries. In such scenarios, we assume that selected queries can cover all the indexes of a query with zero weights, and consider only those queries with at least one non-zero weight (Algorithm 2). However, for large sizes of compressed workload, it is possible that after certain point, all the remaining queries have zero-weight features. In such cases, we reset the query features of unselected queries to their original weights (line 12 of Algorithm 2).

5.1 Optimality Analysis

We show that under mild conditions the benefit function is a non-negative monotone *submodular* set function. As a result, the greedy algorithm can achieve worst-case $(1 - 1/e) \approx 0.63$ optimality guarantee [29, 32] under such conditions. Due to space constraints, we defer the full details, including all the proofs, to the complete version of this paper [8].

Specifically, given a set of queries $Q = \{q_1, ..., q_K\}$ and a particular order/sequence $\pi(Q) = q_{i_1}, ..., q_{i_K}$, we define

$$F(\pi(Q)) = \sum\nolimits_{q' \in W - Q} F_{\pi(Q)}(q').$$

Intuitively, $F(\pi(Q))$ measures the influence of the queries in Q as a whole (over queries not in Q) w.r.t. the order $\pi(Q)$. As a result, we can further define the benefit function $B(\pi(Q))$ *with respect to* the order/sequence $\pi(Q)$, in terms of $F(\pi(Q))$:

$$B(\pi(Q)) = U(Q) + F(\pi(Q)).$$

Moreover, our optimization goal now becomes

$$B(Q) = \max_{\pi(Q)} B(\pi(Q)) = U(Q) + \max_{\pi(Q)} F(\pi(Q)).$$

Theorem 1 (Monotonicity). *Given $X \subseteq Y$, if, for any permutations $\pi(X)$ over X and $\pi(Y)$ over Y, it holds that*

$$U(Y) - U(X) \geq F(\pi(X)) - F(\pi(Y)),$$

then $B(X) \leq B(Y)$, i.e., B is monotone.

Clearly, $U(Y) \geq U(X)$ since $X \subseteq Y$. Therefore, as long as the *gain* on the utility when expanding X to Y can offset the *loss* on the influence, B is monotone.

Theorem 2 (Submodularity). *Given $X \subseteq Y$ and $z \notin Y$, let $\pi(X)$ and $\pi(Y)$ be arbitrary orders of X and Y. We define*

- *$\pi^*(X)$, $\pi^*(Y)$ as the best orders on X and Y that maximize $B(X)$ and $B(Y)$, respectively;*
- *$\pi^*_z(X)$, $\pi^*_z(Y)$ as the orders on $X \cup \{z\}$ and $Y \cup \{z\}$ that append z at the end of $\pi^*(X)$ and $\pi^*(Y)$, respectively;*
- *$\pi^*(X \cup \{z\})$, $\pi^*(Y \cup \{z\})$ as the best orders on $X \cup \{z\}$ and $Y \cup \{z\}$ that maximize $B(X \cup \{z\})$ and $B(Y \cup \{z\})$, respectively.*

B is submodular *under the following conditions:*

(C1) For any $q' \in \overline{Y \cup \{z\}}$, $F_{z|\pi(X)}(q') \geq F_{z|\pi(Y)}(q')$;
(C2) $F_{\pi(Y)}(z) \geq F_{\pi(X)}(z)$;
(C3) $F(\pi^*(X \cup \{z\})) - F(\pi^*_z(X)) \geq F(\pi^*(Y \cup \{z\})) - F(\pi^*_z(Y))$.

C1 means that, as we expand X to Y by including more queries, the *conditional influence* of z over q', both of which are outside Y, will decrease regardless of the orders $\pi(X)$ and $\pi(Y)$. This matches our intuition that the conditional influence of a query should in general decrease when following a larger number of queries. On the other hand, C2 states that the *cumulative influence* on z should in general increase when expanding X to Y, regardless of the orders $\pi(X)$ and $\pi(Y)$, which also matches our intuition. Moreover, C3 implies that the *marginal* gain on influence of including z diminishes as we expand X to Y, with respect to the best possible order and the best order that appends z at the end. Note that this trend is consistent with that stated in C1.

6 IMPROVING EFFICIENCY

A major issue with all-pairs algorithm is that its runtime is prohibitively high for large workloads. Specifically, if n is the number of queries in the workload, and k is the size of the compressed workload, the runtime complexity of the algorithm is $O(k \times n^2)$.

In order to improve the efficiency, we summarize all queries in the workload using a concise representation, called *summary features*. We then compute the utility of each query with respect to the summary features, avoiding all-pairs comparison. We first discuss how we generate the summary features and then discuss how we adapt the all-pairs algorithm to leverage the summary features.

6.1 Workload Summary Features

One of the key challenges in coming up with the summary features is to ensure that the salient characteristics of queries are preserved in the summary features. A simple summarization approach could be to sum up or take the average of query features across all the queries. The issue with this approach is that query features of all queries get equal importance, while in practice some queries contribute more to the influence depending on their utilities.

Key idea. The computation of summary features is inspired from how we measure the influence of a query (say q_s) over the entire workload. Recall

$$F_{q_s}(W) = \sum\nolimits_{q_j \in W - \{q_s\}} S(q_s, q_j) \times U(q_j).$$

Intuitively, this means that the influence is the weighted sum of similarity across queries in the workload.

For a given query q_i, the weight of a column c denoted by query q_{ic} represents the importance of c in q_{ic}. We modify the weight

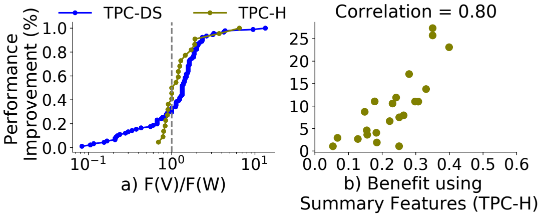


Figure 8: Impact of using summary features

such that it reflects the weight of the column at workload-level. The weight of a column c in query q_i at workload-level, denoted by V_{ic}, can be measured using the utility of q_i as follows:

$$V_{ic} = q_{ic} \times U(q_i).$$

Similarly, the weight of a column c at workload-level can be derived as the sum of the weights of c across all queries at workload-level:

$$V_c = \sum\nolimits_{q_i \in W} q_{ic} \times U(q_i).$$

Definition 11 (summary features). *Summary features is a set of all indexable columns in the workload where the value of each feature is the weighted (using normalized utility of the query) sum of values of the features across all queries.*

Now, the influence of a query q_s with respect to a summary features V that excludes q_s, can be computed as follows:

$$F_{q_s}(V) = S(q_s, V).$$

Thus, given a summary features, V, we can directly compute the influence of q_i as $F_{q_s}(V)$ instead of $F_{q_s}(W)$, thereby avoiding all-pairs comparisons. We prove the following result (see Appendix B in [8]) on the difference between $F_{q_s}(V)$ and $F_{q_s}(W)$ when using weighted Jaccard as the similarity measure.

Theorem 3.

$$\frac{R}{n \times U_L} \leq \frac{F_{q_s}(V)}{F_{q_s}(W)} \leq \frac{1}{n \times R \times U_S}.$$

Here, $U_S = \min_i\{U(q_i)\}$, $U_L = \max_i\{U(q_i)\}$, $n = |W|$,

$$R = \min_c \frac{\min_i\{q_{ic}\}}{\max_i\{q_{ic}\}},$$

i.e., the smallest ratio between any two values of the same column.

We make the following observations from this theorem. First, as R increases, i.e, as the similarity between two values of a column increases, both under-estimation and over-estimation decrease. In other words, the summary features closely approximates the workload as R increases. Second, assuming uniform distribution of utilities and values of columns, as the queries in the workload increases, the over-estimation increases faster than the under-estimation. This is because, as n increases, both U_L and U_S decrease. Finally, an increase in skew between the utilities of the queries may lead to more under-estimation and over-estimation.

To validate further, we empirically evaluate the estimation error of $F_{q_s}(V)$ w.r.t $F_{q_s}(W)$ over the TPC-H and TPC-DS datasets as depicted in Figure 8a. We find that for over 70% of the queries, the error is less than 2×, which is significantly smaller than the theoretical bounds. Furthermore, measuring benefit using the summary features still gives a very high correlation with performance improvement (0.83 vs 0.87 over the TPC-H workload.) We further evaluate the impact of summary features on the performance improvement and the improvement in efficiency in Section 8.

6.2 An Efficient Greedy Algorithm

Algorithm 3 depicts an adapted version of Algorithm 1 for finding the query with the maximum benefit using summary features. Note that while summary features help us directly compute the influence of a query; it is more erroneous to update the summary features directly for computing conditional influence. Thus, after selecting each query, we update all the query features and utility of queries as discussed in Section 4.3 (similar to all-pairs algorithm) and then regenerate the summary features.

Algorithm 3 `FindMaxBenefitQueryUsingSummaryFeatures`

```
$maxbenefit$ = -1;
$maxbenefitquery = null$
$V$ = ComputeSummaryFeatures($W$)
$totalutility$ = ComputeTotalUtility($W$)
for all $q_i \in W$ do
    if all $feature$ in $q_i.features = 0$ then
        continue;
    end if
    $contributiontoV = q_i.features$
    $reducedtotalutility = totalutility - q_i.utility$
    $V' = (V - contributiontoV) \times \frac{totalutility}{reducedtotalutility}$;
    $benefit = q_i.utility + S(q_i.features, V')$
    if $benefit \geq maxbenefit$ then
        $maxbenefit = benefit$
        $maxbenefitquery = q_i$
    end if
end for
Return $maxbenefitquery$
```

If n is the number of queries in the workload, and k is the size of the compressed workload, the runtime complexity of the algorithm is $O(k \times n)$, resulting in orders of magnitude more improvement in efficiency compared to all-pairs algorithm.

The optimality guarantee of Algorithm 3 w.r.t. the optimal choices when using $F_{q_s}(V)$ for measuring influence is the same (i.e., $1 - 1/e \approx 0.63$) as that of Algorithm 1, which uses $F_{q_s}(W)$ for measuring influence. While Theorems 1 and 2 hold for both $F_{q_s}(V)$ and $F_{q_s}(W)$, the derivation of the optimality guarantee requires that both the greedy and optimal algorithm use the same F. Given that finding the optimal solution is NP-hard, a direct comparison between choices made by the greedy algorithm using $F_{q_s}(V)$ vs. the choices made by an optimal algorithm using $F_{q_s}(W)$ is not feasible. Nevertheless, Theorem 3 offered some insights on the relationship between $F_{q_s}(V)$ and $F_{q_s}(W)$, and Figure 8a further validated the approximation error of $F_{q_s}(V)$ w.r.t. $F_{q_s}(W)$. We also empirically compare the impact on performance improvement and compression time between Algorithm 1 and Algorithm 3 in Section 8.

7 WEIGHING QUERIES IN COMPRESSED WORKLOAD

While passing the compressed workload to the index tuner, we need to assign weights to each of the queries based on their own utility and influence on the other queries in the workload. Given a weight $wt(q_i)$ for query q_i and the set of indexes I_k selected by tuning the compressed workload W_k, the improvement over the workload W is defined as $\Delta(W) = \sum_{q_i \in W} wt(qi) \times (C(q_i) - C_{I_k}(q_i))$.

Given a selected query q_i in W_k, a straightforward approach is to re-use the value of conditional benefit of queries during query selection. However, the benefits of queries only reflect their importance at workload-level *when* they are selected, which may have changed after *all* k queries are selected. Therefore, we need to adjust their relative importance afterwards.

Thus, once we have selected the k queries, we re-calibrate the benefits of individual queries using a variant of the summary feature-based greedy algorithm (Algorithm 5). Specifically, we consider only the queries from W_k at each step, and construct summary feature using only the unselected queries. The weight of each query is the normalized re-calibrated benefit.

Furthermore, in certain scenarios, we note that the workload may consist of multiple instances of the same query template. Thus, indexes selected for one of the instances may benefit all instances within the same template, though the amount of benefit may vary depending on the selectivity of predicates and costs of queries. For such scenarios, we note that it is more effective as well as efficient to modify the utility of a selected instance to the total utility of all instances in the input workload that have the same template as the selected instance (Algorithm 4). For the rest of the queries with non-matching templates, we compute their benefits following an approach similar to that of query selection (Algorithm 5).

Algorithm 4 `Template-based Utility Computation`

```
$T$ ← Compute unique query templates in $W_k$
for all $t \in T$ do
    $t.freq$ ← Compute number of queries matching template $t$ in $W_k$
    $t.totalutility$ ← Compute sum of utilities of queries in $W$ for template $t$
end for
$W'$ ← Remove queries in $W$ with matching templates in $T$
$W'_k \leftarrow \emptyset$
for all $q_i \in W_k$ do
    $template \leftarrow q_i.template$
    $q_i.utility = \frac{template.totalutility}{template.freq}$
end for
Return $W_k$,$W'$;
```

Algorithm 5 `Weighing of Selected Queries`

```
$W_k$ , $W_u$ ← TemplateBasedUtilityComputation($W_k$ , $W$)
$Q$ = ComputeSummaryFeatures($W_u$)
$queryweights = \{\}$, $totalbenefits = 0$
while $W_k$.size() > 0 do
    for all $q_i \in W_k$ do
        $benefit = q_i.utility + S(q_i.features, Q)$
        if $benefit \geq maxbenefit$ then
            $maxbenefit = benefit$
            $maxbenefitquery = q_i$
        end if
    end for
    $totalbenefits \leftarrow totalbenefits + maxbenefit$
    $queryweights$.Add($maxbenefitquery, maxbenefit$)
    $W_k$ ← RemoveQuery($W_k$,$maxbenefitquery$)
    $W_u$ ← UpdateWorkload($W_u$ , $maxbenefitquery$)
end while
for all $q_i \in queryweights$.Keys() do
    $queryweights[q_i] = queryweights[q_i]/totalbenefits$;
end for
Return $queryweights$;
```

8 EXPERIMENTS

Workloads. As summarized in Table 2, we use two standard benchmarks: TPC-H and TPC-DS, a recent variant of TPC-DS benchmark DSB [22], and a real customer workload Real-M. Both DSB and Real-M are complex, with a large variety of query templates and skewed data distribution. We instantiate the TPC-H, TPC-DS, and DSB workloads using the `qgen` tools of the benchmarks.

Name	#Queries	# Templates	#Tables
TPC-H (*sf*=10)	2200	22	8
TPC-DS (*sf*=10)	9100	91	24
DSB [22] (*sf*=10)	520	52	24
Real-M (26GB)	473	456	474

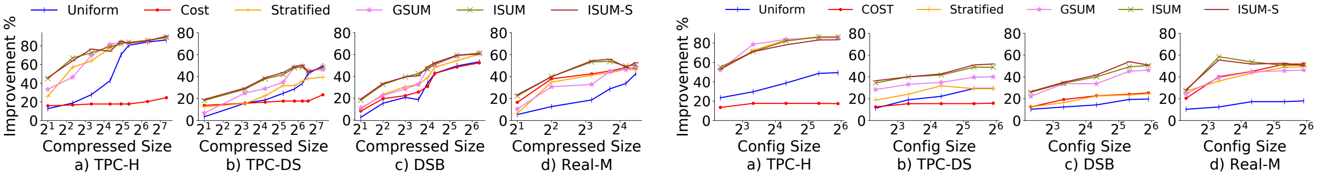
Table 2: Summary of workloads

Baselines. ISUM, by default, uses rule-based strategies to weigh columns using query costs, table sizes, and the number of candidate indexes they generate. We use **ISUM-S** to denote the variant where we use statistics such as selectivity and density to weigh columns as described in Section 4. We compare ISUM and ISUM-S with the following baselines: **1. Uniform Sampling**. It randomly samples k queries from the workload. **2. Cost**. It selects top-k queries with the highest costs. **3. Stratified**. It clusters queries based on templates and then uniformly samples equal number of queries from each cluster. **4. GSUM [21].** It is a recently proposed workload compression technique that maximizes both *coverage* and *representativity* as discussed in the Section 9. Note that all of the above baselines are efficient, i.e., given the input workload consisting of queries, their optimizer estimated costs and the statistics, the workload compression usually takes $< 1\%$ of the tuning time of compressed workload. Hence, for comparing with the above baselines, we use the linear-time algorithm for ISUM that has comparable compression time. We also compare ISUM with the following two algorithms using our proposed benefit measure. **5. All-pairs**. It selects top-k queries as described in Section 5. **6.** k**-mediod**. Using k random seeds, it clusters queries into k clusters until convergence as described in [12]. Both the algorithms have prohibitively high time complexity, which limits their benefits over large workloads.

Evaluation Metrics. We use the following two metrics. **(1) Improvement (%).** If $C(W)$ is the original optimizer estimated cost of the workload without adding or removing the existing indexes, and $C_k(W)$ is the optimizer estimated cost of the workload when using recommended indexes based on tuning of the compressed workload, we measure improvement (%) on W as: $\frac{C(W)-C_k(W)}{C(W)} \times 100\%$. Index advisors use a similar metric report the performance improvement [28]. We use the Database Tuning Advisor (DTA) tool [3] of Microsoft SQL Server as our default choice for index tuning. A recent benchmark study [28] compares the performances of various index tuning tools and reports that DTA can yield the state-of-the-art performance. To assess the generalizability, we also evaluate the effectiveness of proposed estimation techniques for improvement as well as well as compare ISUM and baselines over varying workload sizes using DEXTER [2], an open-source index tuning tool for PostgreSQL. **(2) Time**: The time (in seconds unless otherwise specified) it takes to compress the workload consisting of the queries, their optimizer estimated costs, and the statistics.

8.1 Comparison with Baselines

Improvement on varying compressed workload size. We compare different approaches as we vary the size of the compressed workload from a small size (< 5) to $2\sqrt{n}$ (n is the size of input workload). For compressed workload sizes $> 2\sqrt{n}$, we see insignificant difference between different approaches. As depicted in Figure 9a, we see that both ISUM and ISUM-S result in higher improvement w.r.t. other baselines. For these workloads, while both influence and



(a) Varying Compressed Workload Size.

(b) Varying Configuration Size.

Figure 9: Impact of different compression algorithms on workload performance improvement on index tuning

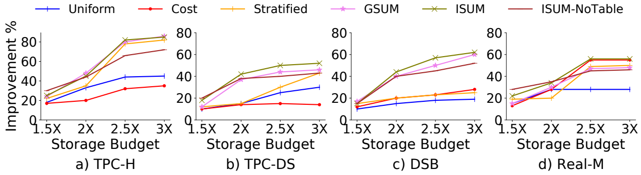


Figure 10: Impact of varying storage size.

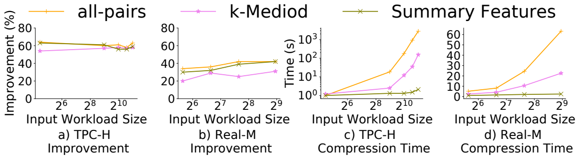


Figure 11: Comparing summary-features based algorithm with all-pairs and k-mediod [12] algorithms.

utility of queries are important to maximize the improvement (see our discussion in Sec 4.2), the baselines capture only one of these two properties. Furthermore, we see that there is no one baseline that performs consistently well over all the workloads. For instance, the cost-based approach does well for Real-M, however it does not perform that well for TPC-H and TPC-DS. The reason is that the queries in Real-M are more similar to each other, and the cost of queries is a more dominant factor. Finally, we see that ISUM variants perform better on both Real-M and DSB, which consists of more complex and larger variety of query templates. Notably, the difference between ISUM-S and ISUM is insignificant, even though ISUM-S uses statistics such as selectivity and density to measure utility and weigh columns. From our analysis of these workloads, we find that (a) the cost of a query is often strongly correlated with potential improvement as we also see in Figure 5; (b) most of the frequently used filter and join attributes in the workload are highly selective for a large number of queries; and finally (c) weighing of indexable columns according to the number of candidate indexes they participate is helpful in capturing similarity across queries.

Improvement on varying configuration size. Next, we fix the compressed workload size to $0.5\sqrt{n}$ and measure the improvement as we vary the size of the index configuration from 5 to 60 (Figure 9b). We see that the improvement (%) generally increases as the size of the configuration increases; however ISUM and ISUM-S perform better than other baselines over a wide range of index configuration sizes. We also note that for most of the workloads, the improvement after selecting 30 indexes is negligible, indicating that additional indexes do not have significant impact on the performance. For a few configuration sizes (e.g., > 10 on Real-M), we see some unexpected decrease in improvement. This is because the behaviour of index tuners like DTA is not always monotonic as they use greedy algorithms and other approximations to efficiently search the indexes. This has also been observed in prior work [28].

For many compressed workload sizes (see Figure 9a) and configuration sizes (Figure 9b), ISUM gives 30% more improvement than GSUM. For complex real workload like Real-M, GSUM performs worse than other baselines, with ISUM giving significantly better improvements. ISUM also consistently performs better than all the baselines over many settings. Furthermore, ISUM is able to identify promising queries that lead to higher improvement with smaller compressed workload sizes than GSUM and converges faster, e.g., on TPC-H we see that ISUM gives 60% improvement in half the number of queries and tuning time than that of GSUM.

Improvement on varying storage. We assess the impact of varying the storage size from 1.5× to 3× of the original database size on the improvement for different algorithms (Figure 10). Note that DTA uses the default storage budget of 3× the original size of the database. We omit the results of ISUM-S as it shows results similar to that of ISUM. To assess the impact of using table size as a feature for weighing columns, we consider another variant of ISUM-S (denoted by ISUM-NoTable) where we skip the table size and only weigh columns based on statistics. Among the baselines and ISUM variants, we make similar observations as when varying configuration sizes. However, for 1.5× we see that ISUM-NoTable results in better improvement due to selection of indexes over smaller tables. Nonetheless, the performance of this approach is significantly worse as we increase the storage budget to 2× and beyond.

Effectiveness of summary features. Figure 11 depicts the effectiveness of summary-features based greedy approach w.r.t. to the all-pairs greedy (Section 5) and k-mediod [12] approaches. We use our weighted Jaccard based benefit measure for k-mediod since the proposed distance function in [12] is not defined for queries across different templates (e.g., differing in tables). First, we see that the improvements using summary-features based approach is close to all-pairs approach, indicating that summary features capture the key characteristics of the workload. However, the time taken by all-pairs increases rapidly as the size of the workload (close to 1,000 seconds for 2,000 queries) increases while using summary-features is significantly more efficient (< 10s). k-mediod [12] is faster than all-pairs as it avoids all-pairs comparison by randomly selecting k queries as starting points for each cluster and then comparing all other queries with k queries in each iteration. However, the number of iterations for convergence can still be very large, taking much more time than summary-features. In terms of quality, we find that k-mediod has generally the worst improvement results as the clusters chosen by k-mediod are prone to local minima.

8.2 Sensitivity of ISUM

We next explore the sensitivity of ISUM on varying workload characteristics as well as the effectiveness of different strategies ISUM employs during query selection and weighing.

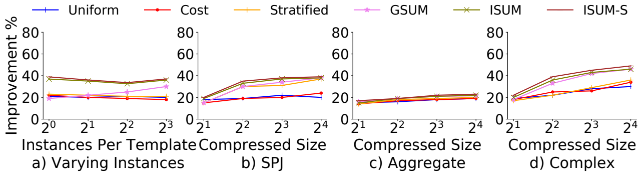


Figure 12: Impact of Complexity of Queries (DSB)

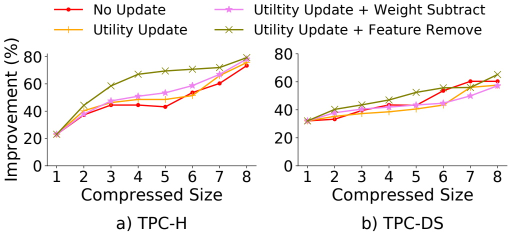


Figure 13: Impact of update strategies

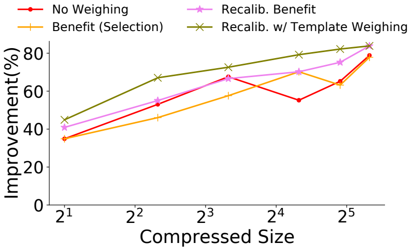


Figure 14: Impact of weighing strategies on TPC-H

Varying characteristics of workload. Figure 12a depicts the impact of increasing the number of instances per template on improvement over the DSB benchmark. We see that ISUM is less affected due to the changes in the number of instances; however the performance of baselines varies significantly as we vary the number of instances. For example, we see that the improvement increases for GSUM as we add more instances as it is able to discriminate between similar queries. On the other hand, the improvement for the cost-based approach decreases as it selects multiple similar instances from the same template, which do not help add new indexes. However, ISUM performs reasonably well as it is able to capture the best of both scenarios.

Figure 12b-d depict the performance of algorithms for varying query complexities: SPJ, Aggregate, and Complex (as categorized in DSB benchmark). For SPJ and Complex queries, we see similar trends as in Figure 9a. However, for Aggregate queries, we see less overall improvement and insignificant differences between different algorithms, indicating that indexes are less effective in improving the performance of such queries.

Effectiveness of benefit metric for a set of queries and update strategies. Figure 13 depicts the efficacy of benefit computation over a set of queries and the impact of different update strategies on the improvement (%) on the TPC-H and TPC-DS workloads. We use the all-pairs greedy algorithm to select queries incrementally. As we can see, selecting queries without any update has worst improvement, indicating that defining benefit for a set of queries while updating queries is helpful. Furthermore, we see that updating only utility is not sufficient—we also need to update the query features. Among the two ways of updating query features discussed in Section 8.2, we see that setting weights to zero is more effective than reducing the weights.

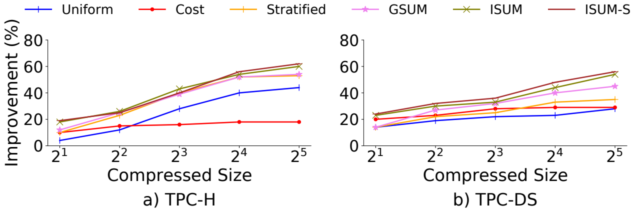


Figure 15: Comparing compression algorithms using DEXTER on PostgreSQL

	TPC-H		TPC-DS	
Estimation Technique	**DTA (SQL-Server)**	**DEXTER (PostgreSQL)**	**DTA (SQL-Server)**	**DEXTER (PostgreSQL)**
Utility (only cost)	.54	.40	.33	.28
Utility (cost + selectivity)	.60	.41	.44	.35
Similarity (Rule-based)	.61	.53	.55	.51
Similarity (Stats-based)	.68	.50	.62	.48
Benefit (Rule-based)	.87	.59	.70	.54
Benefit (Stats-based)	.88	.62	.73	.59

Table 3: Correlation of improvement estimation techniques with respect to the actual improvement reported by DTA and DEXTER.

Effectiveness of weighing. Figure 14 depicts the impact of different weighing strategies on improvement. We see that no weighing results in indexes that lead to low improvement in performance. This is because all queries get equal importance during tuning even though they do not equally represent the workload. Furthermore, using benefits computed during greedy selection of queries is not representative, as queries selected earlier get much higher weights as discussed in Section 7. We fix this by recomputing the benefits of selected queries while ignoring their influence on each other, which improves the results. Finally, we see that changing the utility of selected queries based on the utilities of other queries with the same query template significantly improves the performance, demonstrating the effectiveness of template-based weight readjustment discussed in Section 7.

8.3 Effectiveness over Another Index Advisor

To assess the generalizability of ISUM, we compared ISUM and baselines using DEXTER [2], an open-source index advisor for PostgreSQL. DEXTER exposes a parameter, called minimum improvement, for a candidate index to be considered for selection, which we set to 5% to let it consider a large number of candidate indexes. As depicted in Figure 15, we see that ISUM outperforms the baseline algorithms for a large majority of the compressed workload sizes. To further understand, we perform micro-benchmarks over the TPC-H and TPC-DS workloads that compares the correlation of different techniques (discussed in Section 4) for estimating improvement with the actual improvement reported by each of the index tuning tools (see Table 3). We observe that our proposed "benefit" metric that captures both utility and similarity between queries has higher correlation with the actual performance improvement compared to only using utility or similarity. The baseline algorithms either optimize for utility or similarity but not both. Note that the improvements are in general smaller than DTA, which is likely because the tuning algorithm leveraged by DEXTER is simpler than DTA and misses optimizations such as index merging [28]. Furthermore, we observe that DEXTER is significantly limited in terms of supported features such as constraining tuning based on the number of indexes or storage budget, which prevents us from doing a more thorough comparison.

9 RELATED WORK

A compressed workload on index tuning should lead to indexes that result in high performance improvement for the input workload. In order to achieve this, a compression technique should select queries that are similar to queries in the input workload that have potential for performance improvement. Furthermore, the technique much be efficient. We observe that prior techniques on workload compression do not effectively capture both of these requirements. For instance, sampling-based approaches miss out queries that lead to high performance improvement but are less frequent in workload. This is because sampling lacks information about query structure and the similarity between queries. On the other hand, clustering-based approaches [7, 12, 30] have been proposed that group similar queries and sample from each cluster. Unfortunately, clustering is costly (quadratic in the number of queries), and thus do not scale to large workloads. Furthermore, the distance functions[12] employed for comparing queries in these techniques are less effective in characterizing similarity between queries with varying templates.

Prior work have proposed metrics such as coverage [21, 31], representativity [21, 35], and diversity [20] for compressing workloads. Among them, GSUM [20] is the most relevant to SQL workloads. It uses an efficient greedy algorithm that maximizes coverage of features (e.g., columns) in the workload while also ensuring that the summary workload is representative (i.e., having similar distribution to that of the entire workload). There are two issues with this approach. First, the featurization and the computation of coverage and representativity metrics is agnostic of the features that are more relevant to index tuning. For instance, not all columns in a query lead to useful indexes. Second, it is also agnostic of the potential improvement in performance of queries. For instance, more importance should be given to queries that can lead to large improvement in performance as against minimizing the difference between the distributions of the summary workload and the original workload. [27] proposes a machine learning approach for workload compression by training a model specifically for SQL queries. However, this technique requires expensive prepossessing to train the models. In addition, [14] proposes a new SQL operator specifically for summarizing workloads. However, it also suffers from the quadratic complexity like clustering-based approaches.

To improve the scalability of index advisors, there has been previous work that reduces the number of optimizer calls during tuning by trading off accuracy. For instance, [25, 34] avoid unnecessary optimizer calls by caching and reasoning about reuse of costs of sub-expressions across queries. Index merging [17] extends the basic design framework for performing candidate selection through merging candidate indexes, thereby reducing optimizer calls. [9, 10] compute the bounds on costs of queries based on query optimization of past configurations, which can be used for pruning optimizer calls. While most of these works share our goal of improving the scalability of index advisors, in this work we focus on a solution that does not rely on the index advisor or the query optimizer.

10 LIMITATIONS AND FUTURE WORK

Index advisors such as Database Tuning Advisor (DTA) [3] report the actual improvement on the entire (uncompressed) input workload due to the recommended set of indexes, along with drill-downs on which indexes were used by each query. This involves making an optimizer call for each query in the input workload. For large input workloads, we observe that making these calls can consume a significant proportion of the tuning time, thereby limiting the benefits of workload compression. It is an open question as to whether the above contract with users could be relaxed without affecting the interpretability of the index recommendations output by the tool. One direction to explore is reporting the estimated improvement and drill-downs on the compressed workload, while providing additional details and how each query in the compressed workload represents queries in the input workload that were not tuned.

In this work, we assume that the optimizer estimated cost of each query in the workload is provided as input. We observe that most DBMSs expose functionality to collect historical workload information including the execution plan for a query, e.g., Query Store [5] in Microsoft SQL Server. Such information can be leveraged by ISUM for analyzing queries without making optimizer calls. However, in cases where such logs are not available, ISUM needs to make an optimizer call for each query in the workload to get its plan and cost. For large input workloads, such calls may dominate the compression time of the algorithm.

Additionally, index advisors (e.g., see DTA [7]) support tuning with a time-budget, where queries from the input workload are consumed and tuned incrementally. However ISUM requires preprocessing all the queries from the input workload before it can select queries for tuning. Thus, more work is needed to make the ISUM algorithm incremental when it only has information available about a subset of queries from the input workload.

Finally, there are opportunities to extend our approach to design workload compression algorithms for other physical design tuning problems, e.g., selecting materialized views or choosing data partitioning strategy. Investigating these problems is an interesting area of future work.

11 CONCLUSION

In this work, we developed ISUM, an efficient workload compression algorithm that identifies a subset of salient queries in large workloads for scalable index tuning. In contrast to prior work on workload compression that focuses on syntactic relevance, ISUM also considers the potential for performance improvement while selecting queries. Furthermore, it introduces novel indexing-specific query featurization and weighing that helps measure the similarity between queries in terms of index usage. We show that ISUM is efficient, and the recommended indexes based on queries selected by ISUM significantly improve the performance of the input workload compared to state-of-the-art workload compression techniques.

Acknowledgments: We thank the anonymous reviewers, Lukas Maas, Arnd Christian König, and Bailu Ding for their valuable feedback.

REFERENCES

[1] Apr 01, 2022. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database/.

[2] Apr 01, 2022. DEXTER Index Tuning Tool on PostgresSQL. https://github.com/ankane/dexter.

[3] Apr 01, 2022. DTA utility. https://docs.microsoft.com/en-us/sql/tools/dta/dta-utility?view=sql-server-ver15.

[4] Apr 01, 2022. Google Cloud SQL. https://cloud.google.com/sql.

[5] Apr 01, 2022. Query Store. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver15.

[6] Apr 01, 2022. Statistics. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver15.

[7] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 930–932.

[8] Anonymous. [n.d.]. ISUM: Technical Report. https://bit.ly/3kdSZvI. https://bit.ly/3kdSZvI

[9] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.

[10] Nicolas Bruno and Surajit Chaudhuri. 2006. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the 32nd international conference on Very large data bases*. Citeseer, 499–510.

[11] Surajit Chaudhuri, Mayur Datar, and Vivek R. Narasayya. 2004. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Trans. Knowl. Data Eng.* 16, 11 (2004), 1313–1323.

[12] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. 2002. Compressing SQL workloads. In *SIGMOD*. 488–499.

[13] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server. https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/.

[14] Surajit Chaudhuri, Vivek Narasayya, and Prasanna Ganesan. 2003. Primitives for workload summarization and implications for SQL. In *Proceedings 2003 VLDB Conference*. Elsevier, 730–741.

[15] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.

[16] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378.

[17] Surajit Chaudhuri and Vivek R. Narasayya. 1999. Index Merging. In *ICDE*.

[18] Douglas Comer. 1978. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.* 3, 4 (1978), 440–445.

[19] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD*. 666–679.

[20] Anirban Dasgupta, Ravi Kumar, and Sujith Ravi. 2013. Summarization through submodularity and dispersion. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1014–1022.

[21] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow.* 14, 3 (2020), 418–430.

[22] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[23] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*. 1241–1258.

[24] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.* 13, 1 (1988).

[25] Antara Ghosh, Jignashu Parikh, Vibhuti S Sengar, and Jayant R Haritsa. 2002. Plan selection based on query clustering. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 179–190.

[26] Teofilo F Gonzalez. 1985. Clustering to minimize the maximum intercluster distance. *Theoretical computer science* 38 (1985), 293–306.

[27] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. 2018. Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics. *arXiv preprint arXiv:1801.05613* (2018).

[28] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.

[29] Andreas Krause and Daniel Golovin. 2014. Submodular Function Maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 71–104.

[30] Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2016. Ettu: Analyzing query intents in corporate databases. In *Proceedings of the 25th international conference companion on world wide web*. 463–466.

[31] Rishabh Mehrotra and Emine Yilmaz. 2015. Representative & informative query selection for learning to rank using submodular functions. In *Proceedings of the 38th international ACM sigir conference on research and development in information retrieval*. 545–554.

[32] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. 1978. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.* 14, 1 (1978), 265–294.

[33] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

[34] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. 2007. Efficient Use of the Query Optimizer for Automated Database Design. ACM.

[35] Sayan Ranu, Minh Hoang, and Ambuj Singh. 2014. Answering top-k representative queries on graph databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1163–1174.

[36] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*. IEEE, 166–171.

[37] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.

[38] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.

A ANALYSIS OF GREEDY ALGORITHM

Let $Q = \{q_1, ..., q_K\}$, and let $\pi(Q) = \{q_{i_1}, ..., q_{i_K}\}$ be a *permutation* of the queries in Q. Define $\overline{Q} = W - Q$. For a given query $q' \in \overline{Q}$, we define the *cumulative influence* of $\pi(Q)$ on q' as

$$F_{\pi(Q)}(q') = \sum_{l=1}^{K} F_{q_{i_l} | q_{i_1},...,q_{i_{l-1}}}(q'), \tag{6}$$

where $F_{q_{i_l} | q_{i_1},...,q_{i_{l-1}}}(q')$ is the *conditional influence* of q_{i_l} on q' given the sequence $q_{i_1}, ..., q_{i_{l-1}}$. Moreover, we define the *residual utility* of a query $q_{i_l} \in \pi(Q)$ given the sequence $q_{i_1}, ..., q_{i_{l-1}}$ as

$$\begin{aligned} U_{q_{i_1},...,q_{i_{l-1}}}(q_{i_l}) &= U(q_{i_l}) - \sum_{t=1}^{l-1} F_{q_{i_t} | q_{i_1},...,q_{i_{t-1}}}(q_{i_l}) \\ &= U(q_{i_l}) - F_{q_{i_1},...,q_{i_{l-1}}}(q_{i_l}). \end{aligned} \tag{7}$$

(Problem Statement) Now define

$$B(\pi(Q)) = U(Q) + \sum_{q' \in \overline{Q}} F_{\pi(Q)}(q'), \tag{8}$$

and define

$$B(Q) = \max_{\pi(Q)} B(\pi(Q)). \tag{9}$$

Our optimization problem can be stated as

$$\max_{Q \subseteq W} B(Q), \text{ s.t. } |Q| \leq K.$$

Our goal is to understand, under which conditions, that $B(Q)$ can be a non-negative monotone *submodular* set function. If so, a greedy approach can yield $1 - 1/e$ worst-case optimality guarantees, if in each greedy step it selects a query z that maximizes

$$\partial_B(z) = B(Q \cup \{z\}) - B(Q),$$

where Q is the current of selected queries. Our greedy algorithm does not exactly do this, since the exact computation of B requires enumeration of all possible orders $\pi(Q)$ of Q, which is infeasible in practice unless Q is very small. Instead, our greedy algorithm selects a query z that maximizes

$$\partial'_B(z) = B(\pi_z(Q)) - B(\pi(Q)),$$

where $\pi(Q)$ is the specific order of the queries selected in previous steps, and $\pi_z(Q)$ appends z at the end of $\pi(Q)$, i.e., $\pi_z(Q) = \pi(Q), z$. For the purpose of this analysis, let us assume that $\partial'_B(z)$ is a fair approximation of $\partial_B(z)$. We can further show that $\partial'_B(z)$ is just the *conditional benefit* of z defined by Equation 5. To see this, we have

$$B(\pi_z(Q)) = U(Q \cup \{z\}) + \sum_{q' \in \overline{Q \cup \{z\}}} F_{\pi_z(Q)}(q'),$$

$$B(\pi(Q)) = U(Q) + \sum_{q' \in \overline{Q}} F_{\pi(Q)}(q').$$

As a result, it follows that

$$\partial'_B(z) = \Big(U(z) - F_{\pi(Q)}(z)\Big) + \sum_{q' \in \overline{Q \cup \{z\}}} \Big(F_{\pi_z(Q)}(q') - F_{\pi(Q)}(q')\Big).$$

By Equation 7, we have

$$U(z) - F_{\pi(Q)}(z) = U_{\pi(Q)}(z).$$

Furthermore, by Equation 6, we have

$$F_{\pi_z(Q)}(q') - F_{\pi(Q)}(q') = F_{z|\pi(Q)}(q').$$

Therefore, it follows that

$$\partial'_B(z) = U_{\pi(Q)}(z) + \sum_{q' \in \overline{Q \cup \{z\}}} F_{z|\pi(Q)}(q'),$$

which matches our definition of condidtional benefit.

Clearly, B is non-negative. It stands to reason about the monotonicity and submodularity of B. For ease of exposition, we define

$$F(\pi(Q)) = \sum_{q' \in \overline{Q}} F_{\pi(Q)}(q'). \tag{10}$$

As a result, we can further express $B(\pi(Q))$ in terms of $F(\pi(Q))$:

$$B(\pi(Q)) = U(Q) + F(\pi(Q)). \tag{11}$$

Intuitively, F measures the influence of the queries in Q as a whole (over queries not in Q) w.r.t. the order $\pi(Q)$.

A.1 General Analysis

A.1.1 Monotonicity. We first consider the monotonicity of B. Let $X \subseteq Y$. More specifically, let $X = \{x_1, ..., x_m\}$ and $Y = \{y_1, ..., y_n\}$. We have $m \leq n$ since $X \subseteq Y$. Let $\pi(X) = x_{i_1}, ..., x_{i_m}$ be a *permutation* of X, and let $\pi(Y) = y_{j_1}, ..., y_{j_n}$ be a *permutation* of Y. By the definition of the set function $B(Q)$, we have

$$B(X) = \max_{\pi(X)} B(\pi(X)), \quad B(Y) = \max_{\pi(Y)} B(\pi(Y)),$$

where by Equation 11,

$$B(\pi(X)) = U(X) + F(\pi(X)), \quad B(\pi(Y)) = U(Y) + F(\pi(Y)).$$

As a result, we have

$$B(\pi(Y)) - B(\pi(X)) = \Big(U(Y) - U(X)\Big) + \Big(F(\pi(Y)) - F(\pi(X))\Big).$$

Therefore, we have the following lemma:

Lemma 4. *Given $X \subseteq Y$, let $\pi(X)$ be any permutation of X and $\pi(Y)$ be any permutation of Y. If*

$$U(Y) - U(X) \geq F(\pi(X)) - F(\pi(Y)), \tag{12}$$

then $B(\pi(Y)) \geq B(\pi(X))$.

Clearly, $U(Y) \geq U(X)$ since $X \subseteq Y$. The intuitive explanation of Lemma 4 is that, as long as the *gain* on the utility when expanding X to Y can offset the *loss* on the influence, B will be monotone with respect to any permutations $\pi(X)$ and $\pi(Y)$.

Since $\pi(X)$ and $\pi(Y)$ in Lemma 4 are arbitrary, we can just use $\pi^*(X)$ that maximizes $B(\pi(X))$ and $\pi^*(Y)$ that maximizes $B(\pi(Y))$. As a result, we have

$$B(X) = \max_{\pi(X)} B(\pi(X)) = B(\pi^*(X)),$$

$$B(Y) = \max_{\pi(Y)} B(\pi(Y)) = B(\pi^*(Y)).$$

Under the condition of Lemma 4, we have $B(\pi^*(X)) \leq B(\pi^*(Y))$. Therefore, we have proved the following theorem:

Theorem 5. *Given $X \subseteq Y$, if, for any permutations $\pi(X)$ over X and $\pi(Y)$ over Y, it holds that*

$$U(Y) - U(X) \geq F(\pi(X)) - F(\pi(Y)),$$

then $B(X) \leq B(Y)$, i.e., B is monotone.

A.1.2 Submodularity.

We now consider the submodularity of B. We need to show that, given $X \subseteq Y$ and $z \notin Y$, we have

$$B(X \cup \{z\}) - B(X) \geq B(Y \cup \{z\}) - B(Y).$$

By the definition of $B(Q)$, we have

$$B(X) = \max_{\pi(X)} B(\pi(X)), \quad B(Y) = \max_{\pi(Y)} B(\pi(Y)),$$
$$B(X \cup \{z\}) = \max_{\pi(X \cup \{z\})} B(\pi(X \cup \{z\})),$$
$$B(Y \cup \{z\}) = \max_{\pi(Y \cup \{z\})} B(\pi(Y \cup \{z\})),$$

where

$$B(\pi(X)) = U(X) + F(\pi(X)),$$
$$B(\pi(Y)) = U(X) + F(\pi(Y)),$$
$$B(\pi(X \cup \{z\})) = U(X \cup \{z\}) + F(\pi(X \cup \{z\})),$$
$$B(\pi(Y \cup \{z\})) = U(Y \cup \{z\}) + F(\pi(Y \cup \{z\})).$$

("Submodularity" for Permutations) Consider an arbitrary permutation $\pi(X)$ on X and the corresponding permutation $\pi_z(X)$ on $X \cup \{z\}$ that "extends" $\pi(X)$ by appending z. Define

$$\Delta_X = B(\pi_z(X)) - B(\pi(X)).$$

By Equation 11, it follows that

$$\begin{aligned}\Delta_X &= \Big(U(X \cup \{z\}) + F(\pi_z(X))\Big) - \Big(U(X) + F(\pi(X))\Big) \\ &= U(z) + \Big(\sum_{q' \in \overline{X \cup \{z\}}} F_{\pi_z(X)}(q') - \sum_{q' \in \overline{X}} F_{\pi(X)}(q')\Big).\end{aligned}$$

Since $\overline{X} - \overline{X \cup \{z\}} = \Big(X \cup \{z\}\Big) - X = \{z\}$, it follows that

$$\begin{aligned}\Delta_X &= U(z) + \sum_{q' \in \overline{X \cup \{z\}}} \Big(F_{\pi_z(X)}(q') - F_{\pi(X)}(q')\Big) - F_{\pi(X)}(z) \\ &= \Big(U(z) - F_{\pi(X)}(z)\Big) + \sum_{q' \in \overline{X \cup \{z\}}} \Big(F_{\pi_z(X)}(q') - F_{\pi(X)}(q')\Big).\end{aligned}$$

By Equation 6 we have

$$F_{\pi_z(X)}(q') - F_{\pi(X)}(q') = F_{z|\pi(X)}(q').$$

As a result,

$$\Delta_X = \Big(U(z) - F_{\pi(X)}(z)\Big) + \textstyle\sum_{q' \in \overline{X \cup \{z\}}} F_{z|\pi(X)}(q'). \tag{13}$$

Next, let $\pi(Y)$ be an arbitrary permutation of Y, and similarly let $\pi_z(Y)$ be the permutation that "extends" $\pi(Y)$ by appending z, i.e., $\pi_z(Y) = \pi(Y), z$. We can similarly define

$$\Delta_Y = B(\pi_z(Y)) - B(\pi(Y)),$$

which follows similarly to the derivation of Δ_X to yield

$$\Delta_Y = \Big(U(z) - F_{\pi(Y)}(z)\Big) + \textstyle\sum_{q' \in \overline{Y \cup \{z\}}} F_{z|\pi(Y)}(q'). \tag{14}$$

Since $\overline{X \cup \{z\}} - \overline{Y \cup \{z\}} = Y - X$, it follows that

$$\sum_{q' \in \overline{X \cup \{z\}}} F_{z|\pi(X)}(q') = \sum_{q' \in \overline{Y \cup \{z\}}} F_{z|\pi(X)}(q') + \sum_{q' \in Y-X} F_{z|\pi(X)}(q').$$

Define

$$\Delta_{X,Y,z} = \textstyle\sum_{q' \in \overline{Y \cup \{z\}}} \Big(F_{z|\pi(X)}(q') - F_{z|\pi(Y)}(q')\Big),$$

and consider

$$\Delta' = \Delta_X - \Delta_Y = \Big(F_{\pi(Y)}(z) - F_{\pi(X)}(z)\Big) + \Delta_{X,Y,z} + \sum_{q' \in Y-X} F_{z|\pi(X)}(q').$$

We have arrived at the following lemma:

Lemma 6. *Given $X \subseteq Y$ and $z \notin Y$, let $\pi(X)$ and $\pi(Y)$ be arbitrary permutations of X and Y. Let $\pi_z(X)$ and $\pi_z(Y)$ be corresponding permutations that extend $\pi(X)$ and $\pi(Y)$ by z, respectively. If the following two conditions holds, namely*

$$F_{z|\pi(X)}(q') \geq F_{z|\pi(Y)}(q'), \qquad \forall q' \in \overline{Y \cup \{z\}}, \tag{15}$$

and

$$F_{\pi(Y)}(z) \geq F_{\pi(X)}(z), \tag{16}$$

then $\Delta' \geq 0$, namely, we have

$$B(\pi_z(X)) - B(\pi(X)) \geq B(\pi_z(Y)) - B(\pi(Y)).$$

Intuitively, Equation 15 means that, as we expand X to Y by including more queries, the *conditional influence* of z over q', both of which are outside Y, will decrease regardless of the orders $\pi(X)$ and $\pi(Y)$. This matches our intuition that the conditional influence of a query should decrease when following a larger number of queries. On the other hand, Equation 16 states that the *cumulative influence* on z should increase when expanding X to Y, regardless of the orders $\pi(X)$ and $\pi(Y)$, which also matches our intuition. We assume that Equations 15 and 16 hold in the rest of our analysis.

("Submodularity" of B) Consider the permutations $\pi^*(X \cup \{z\})$ and $\pi^*(X)$ that maximize $B(\pi(X \cup \{z\}))$ and $B(\pi(X))$, namely,

$$B(X \cup \{z\}) = \max_{\pi(X \cup \{z\})} B(\pi(X \cup \{z\})) = B(\pi^*(X \cup \{z\})),$$
$$B(X) = \max_{\pi(X)} B(\pi(X)) = B(\pi^*(X)).$$

Define

$$\Delta_X^* = B(X \cup \{z\}) - B(X) = B(\pi^*(X \cup \{z\})) - B(\pi^*(X)).$$

Consider the permutation $\pi_z^*(X)$ that "extends" $\pi^*(X)$ by appending z, i.e., $\pi_z^*(X) = \pi^*(X), z$. It follows that

$$\Delta_X^* = \Big(B(\pi^*(X \cup \{z\})) - B(\pi_z^*(X))\Big) + \Big(B(\pi_z^*(X)) - B(\pi^*(X))\Big).$$

Similarly, let $\pi^*(Y)$ be the permutation that maximizes $B(\pi(Y))$, and let $\pi_z^*(Y)$ be the permutation that "extends" $\pi^*(Y)$ by appending z. By Lemma 6, we have

$$B(\pi_z^*(X)) - B(\pi^*(X)) \geq B(\pi_z^*(Y)) - B(\pi^*(Y)).$$

Now consider the permutation $\pi^*(Y \cup \{z\})$ that maximizes $B(\pi(Y \cup \{z\}))$. Our next goal is to show

$$B(\pi^*(X \cup \{z\})) - B(\pi_z^*(X)) \geq B(\pi^*(Y \cup \{z\})) - B(\pi_z^*(Y)), \tag{17}$$

with which it follows that

$$\Delta_X^* \geq B(\pi^*(Y \cup \{z\})) - B(\pi^*(Y)),$$

and thus the submodularity of B.

For this sake of proving Equation 17, define

$$\delta_X^* = B(\pi^*(X \cup \{z\})) - B(\pi_z^*(X)),$$
$$\delta_Y^* = B(\pi^*(Y \cup \{z\})) - B(\pi_z^*(Y)).$$

Let $\delta^* = \delta_X^* - \delta_Y^*$. By Equation 11 we have

$$\delta_X^* = F(\pi^*(X \cup \{z\})) - F(\pi_z^*(X)),$$
$$\delta_Y^* = F(\pi^*(Y \cup \{z\})) - F(\pi_z^*(Y)).$$

As a result, it follows that

$$\delta^* = F(\pi^*(X \cup \{z\})) - F(\pi_z^*(X)) + F(\pi_z^*(Y) - F(\pi^*(Y \cup \{z\})).$$

Rearranging the terms yields

$$\delta^* = \Big(F(\pi^*(X \cup \{z\})) - F(\pi_z^*(X))\Big) - \Big(F(\pi^*(Y \cup \{z\})) - F(\pi_z^*(Y)\Big).$$

We therefore have the following result:

Theorem 7. *Given $X \subseteq Y$ and $z \notin Y$, in addition to the two conditions of Lemma 6 if it further holds that*

$$F(\pi^*(X \cup \{z\})) - F(\pi_z^*(X)) \geq F(\pi^*(Y \cup \{z\})) - F(\pi_z^*(Y)), \tag{18}$$

then $\delta^* \geq 0$ and thus

$$B(X \cup \{z\}) - B(X) \geq B(Y \cup \{z\}) - B(Y),$$

i.e., B is submodular.

The intuition behind Theorem 7 is the following. If the *marginal* gain on influence by including z diminishes as we expand X to Y, with respect to the best possible order and the best order that appends z at the end, then B is submodular. By the two conditions stated in Lemma 6, in general the conditional influence of z diminishes as we expand X to Y. As a result, we expect that the gain on influence would also diminish.

A.2 Case Study

We now study the specific form of the influence function F used in our implementation, namely, for any $q' \in \overline{Y \cup \{z\}}$, we have

$$F_{z|\pi(Y)}(q') = S_{\pi(Y)}(z, q') \cdot U_{\pi(Y)}(q'), \tag{19}$$

where $S_{\pi(Y)}(z, q')$ and $U_{\pi(Y)}(q')$ represent the *similarity* and *utility* conditioned on an arbitrary permutation $\pi(Y)$ over Y. Specifically, let $\pi(Y) = y_1, ..., y_n$. By Equation 7, we have

$$U_{\pi(Y)}(q') = U(q') - F_{\pi(Y)}(q') = U(q') - \sum\nolimits_{l=1}^{n} F_{y_l|y_1,...,y_{l-1}}(q').$$

By Equation 19, we have

$$F_{y_1}(q') = S(y_1, q') \cdot U(q'),$$
$$F_{y_2|y_1}(q') = S_{y_1}(y_2, q') \cdot U_{y_1}(q'),$$

and so on so forth until

$$F_{y_n|y_1,...,y_{n-1}}(q') = S_{y_1,...,y_{n-1}}(y_n, q') \cdot U_{y_1,...,y_{n-1}}(q').$$

For ease of exposition, we define $\alpha_1^Y = S(y_1, q')$, $\alpha_2^Y = S_{y_1}(y_2, q')$, and so on so forth until $\alpha_n^Y = S_{y_1,...,y_{n-1}}(y_n, q')$. As a result,

$$\begin{aligned} U_{\pi(Y)}(q') &= U(q') - \sum\nolimits_{l=1}^{n} F_{y_l|y_1,...,y_{l-1}}(q') \qquad (20) \\ &= U(q') - \sum\nolimits_{l=1}^{n} \alpha_l^Y \cdot U_{y_1,...,y_{l-1}}(q'). \end{aligned}$$

Moreover,

$$\begin{aligned} U_{y_1,...,y_l}(q') &= (1 - S_{y_1,...,y_{l-1}}(y_l, q')) \cdot U_{y_1,...,y_{l-1}}(q') \\ &= (1 - \alpha_l^Y) \cdot U_{y_1,...,y_{l-1}}(q'). \end{aligned}$$

As a result, by induction, we have

$$U_{y_1,...,y_l}(q') = U(q') \cdot \Pi_{t=1}^{l}(1 - \alpha_t^Y).$$

Therefore, it follows that

$$U_{\pi(Y)}(q') = U(q') \cdot \Big(1 - \sum\nolimits_{l=1}^{n} \alpha_l^Y \cdot \Pi_{t=1}^{l-1}(1 - \alpha_t^Y)\Big).$$

We can expand the summation $\sum_{l=1}^{n} \alpha_l^Y \cdot \Pi_{t=1}^{l-1}(1 - \alpha_t^Y)$ to obtain

$$1 - \Big(\alpha_1^Y + \alpha_2^Y(1 - \alpha_1^Y) + \cdots + \alpha_n^Y(1 - \alpha_1^Y) \cdots (1 - \alpha_{n-1}^Y)\Big),$$

which is equivalent to

$$(1 - \alpha_1^Y)(1 - \alpha_2^Y) \cdots (1 - \alpha_n^Y) = \Pi_{l=1}^{n}(1 - \alpha_l^Y).$$

As a result, it follows that

$$U_{\pi(Y)}(q') = U(q') \cdot \Pi_{l=1}^{n}(1 - \alpha_l^X),$$

and therefore,

$$F_{\pi(Y)}(q') = U(q') - U_{\pi(Y)}(q') = U(q') \cdot \Big(1 - \Pi_{l=1}^{n}(1 - \alpha_l^Y)\Big),$$

$$\begin{aligned} F_{z|\pi(Y)}(q') &= S_{\pi(Y)}(z, q') \cdot U_{\pi(Y)}(q') \qquad (21) \\ &= U(q') \cdot S_{\pi(Y)}(z, q') \cdot \Pi_{l=1}^{n}(1 - \alpha_l^Y). \end{aligned}$$

These are the $F_{\pi(Y)}(q')$ and $F_{z|\pi(Y)}(q')$ that we will use. Similarly, we can have $F_{\pi(X)}(q')$ and $F_{z|\pi(X)}(q')$ for $X = \{x_1, ..., x_m\}$:

$$F_{\pi(X)}(q') = U(q') \cdot \Big(1 - \Pi_{l=1}^{m}(1 - \alpha_l^X)\Big),$$

$$F_{z|\pi(X)}(q') = U(q') \cdot S_{\pi(X)}(z, q') \cdot \Pi_{l=1}^{m}(1 - \alpha_l^X).$$

For ease of exposition, we define

$$A^Y = \Pi_{l=1}^{n}(1 - \alpha_l^Y), \quad A^X = \Pi_{l=1}^{m}(1 - \alpha_l^X). \tag{22}$$

As a result,

$$F_{\pi(Y)}(q') = U(q') \cdot \Big(1 - A^Y\Big), \quad F_{\pi(X)}(q') = U(q') \cdot \Big(1 - A^X\Big), \tag{23}$$

$$F_{z|\pi(Y)}(q') = S_{\pi(Y)}(z, q') \cdot A^Y, \quad F_{z|\pi(X)}(q') = S_{\pi(X)}(z, q') \cdot A^X. \tag{24}$$

Moreover, we will use A_l^Y and A_l^X to denote the *partial* sums:

$$\mathrm{A}_l^Y = \Pi_{t=1}^{l}(1 - \alpha_t^Y), \quad \mathrm{A}_l^X = \Pi_{t=1}^{m}(1 - \alpha_t^X).$$

A.2.1 Monotonicity. We next revisit Lemma 4 in the context of Equation 19. Lemma 4 requires

$$U(Y) - U(X) \geq F(\pi(X)) - F(\pi(Y)).$$

On one hand, we have

$$U(Y) - U(X) = \sum\nolimits_{q \in Y - X} U(q).$$

On the other hand, we have

$$F(\pi(X)) - F(\pi(Y)) = \delta_F^E = \sum\nolimits_{q' \in \overline{X}} F_{\pi(X)}(q') - \sum\nolimits_{q' \in \overline{Y}} F_{\pi(Y)}(q').$$

By definition (i.e., Equation 23), we have

$$\begin{aligned} \delta_F^E &= \sum\nolimits_{q' \in \overline{X}} U(q') \cdot (1 - A^X) - \sum\nolimits_{q' \in \overline{Y}} U(q') \cdot (1 - A^Y) \\ &= \sum\nolimits_{q' \in Y - X} U(q') \cdot (1 - A^X) - \sum\nolimits_{q' \in \overline{Y}} U(q') \cdot (A^X - A^Y). \end{aligned}$$

Since all α's are between 0 and 1, by Equation 22 we have

Assumption 1. *If $X \subseteq Y$, then $A^X \geq A^Y$ for any $\pi(X)$ and $\pi(Y)$.*

As a result, it follows that

$$\delta_F^E \leq \sum\nolimits_{q' \in Y - X} U(q') \cdot (1 - A^X) \leq \sum\nolimits_{q' \in Y - X} U(q').$$

Therefore, we have proved

$$F(\pi(X)) - F(\pi(Y)) \leq U(Y) - U(X).$$

We state this condition formally in the following theorem:

Theorem 8. *Given $X \subseteq Y$, let Equation 19 be the conditional influence. Under Assumption 1, B is monotone.*

A.2.2 Submodularity. We now revisit Theorem 7 in the context of Equation 19. We start by checking the two conditions in Lemma 6. Consider Equation 15 first. By Equation 24, we have

$$F_{z|\pi(X)}(q') = S_{\pi(X)}(z, q') \cdot A^X,$$
$$F_{z|\pi(Y)}(q') = S_{\pi(Y)}(z, q') \cdot A^Y.$$

Since $X \subseteq Y$, by Assumption 1, we have $A^X \geq A^Y$. Moreover, one would expect $S_{\pi(X)}(z, q') \geq S_{\pi(Y)}(z, q')$, as similarity in general decreases (similar to Assumption 1). As a result, $F_{z|\pi(X)}(q') \geq F_{z|\pi(Y)}(q')$ and therefore Equation 15 holds under Assumption 1.

Next consider Equation 16. By Equation 23 we have

$$F_{\pi(Y)}(q') = U(q') \cdot \Big(1 - A^Y\Big),$$
$$F_{\pi(X)}(q') = U(q') \cdot \Big(1 - A^X\Big).$$

Again, by Assumption 1, we have $A^X \geq A^Y$ and therefore $F_{\pi(Y)}(q') \geq F_{\pi(X)}(q')$. As a result, Equation 16 holds under Assumption 1.

Our next goal is to check Equation 18, which requires

$$F(\pi^*(X \cup \{z\})) - F(\pi_z^*(X)) \geq F(\pi^*(Y \cup \{z\})) - F(\pi_z^*(Y)).$$

To simplify notation, we define

$$\delta_F^{X*} = F(\pi^*(X \cup \{z\})) - F(\pi_z^*(X)),$$
$$\delta_F^{Y*} = F(\pi^*(Y \cup \{z\})) - F(\pi_z^*(Y)).$$

By Equation 10, we have

$$\delta_F^{Y*} = \sum\nolimits_{q' \in \overline{Y \cup \{z\}}} \Big(F_{\pi^*(Y \cup \{z\})}(q') - F_{\pi_z^*(Y)}(q')\Big).$$

By definition (i.e., Equation 23), we have

$$F_{\pi^*(Y \cup \{z\})}(q') = U(q') \cdot (1 - A^{Y \cup \{z\}}),$$
$$F_{\pi_z^*(Y)}(q') = U(q') \cdot (1 - A^{Y,z}).$$

As a result, it follows that

$$\delta_F^{Y*} = \sum\nolimits_{q' \in \overline{Y \cup \{z\}}} U(q') \cdot (A^{Y,z} - A^{Y \cup \{z\}}).$$

Similarly, we can have

$$\delta_F^{X*} = \sum\nolimits_{q' \in \overline{X \cup \{z\}}} U(q') \cdot (A^{X,z} - A^{X \cup \{z\}}).$$

Moreover, we have the following assumption:

Assumption 2. *The best order always results in the smallest A, i.e.,*

$$A^{Y,z} \geq A^{Y \cup \{z\}}, \quad A^{X,z} \geq A^{X \cup \{z\}}.$$

By Assumption 2, we further define

$$\Delta^Y = A^{Y,z} - A^{Y \cup \{z\}} \geq 0, \quad \Delta^X = A^{X,z} - A^{X \cup \{z\}} \geq 0.$$

We have

$$\delta_F^{X*} - \delta_F^{Y*} = \sum\nolimits_{q' \in \overline{X \cup \{z\}}} U(q') \cdot \Delta^X - \sum\nolimits_{q' \in \overline{Y \cup \{z\}}} U(q') \cdot \Delta^Y,$$

which is equivalent to

$$\delta_F^{X*} - \delta_F^{Y*} = \sum\nolimits_{q' \in \overline{Y \cup \{z\}}} U(q') \cdot \Big(\Delta^X - \Delta^Y\Big) + \sum\nolimits_{q' \in Y - X} U(q') \cdot \Delta^X.$$

Since $X \subseteq Y$ and all similarities are between 0 and 1, we expect $\Delta^Y \leq \Delta^X$. As a result, we have $\delta_F^{X*} - \delta_F^{Y*} \geq 0$, namely, $\delta_F^{X*} \geq \delta_F^{Y*}$. Therefore, we have proved the following result:

Theorem 9. *Given $X \subseteq Y$ and $z \notin Y$, let Equation 19 be the conditional influence. Under Assumptions 1 and 2, B is* submodular.

B ERROR WHEN USING SUMMARY VECTOR

Theorem 10 (Difference between $F_{q_s}(W)$ and $F_{q_s}(V)$ when weighted Jaccard is used as the similarity measure.)**.**

$$\frac{R}{n \times U_L} \leq \frac{F_{q_s}(V)}{F_{q_s}(W)} \leq \frac{1}{n \times R \times U_S}$$

Here, $U_S = \min_i\{U(q_i)\}$, $U_L = \max_i\{U(q_i)\}$, $n = |W|$,

$$R = \min_c \frac{\min_i\{q_{ic}\}}{\max_i\{q_{ic}\}},$$

i.e., the smallest ratio between any two values of the same column.

PROOF. We define

$$S = \max_c \max_{i \neq j, q_{ic} \leq q_{jc}} \frac{q_{ic}}{q_{jc}},$$

i.e, the largest ratio between two values of the same column such that the numerator is smaller than the denominator. Furthermore, we make use of the following result: $\forall a_i, b_i > 0$,

$$\min_i\{\frac{a_i}{b_i}\} \leq \frac{a_1 + a_2 + ... + a_n}{b_1 + b_2 + ... + b_n} \leq \max_i\{\frac{a_i}{b_i}\}. \qquad (1)$$

By definition, we have

$$F_{q_s}(W) = \sum_i^n \frac{\sum_c \min\{q_{sc}, q_{ic}\}}{\sum_c \max\{q_{sc}, q_{ic}\}} \times U(q_i)$$
$$\geq \sum_i^n R \times U_S \quad \text{(using (1))}$$
$$\geq n \times R \times U_S.$$

Similarly,

$$F_{q_s}(W) \leq n \times S \times U_L.$$

Therefore,

$$n \times R \times U_S \leq F_{q_s}(W) \leq n \times S \times U_L$$

Again, by definition, we have

$$F_{q_s}(V) = \frac{\sum_c \min\{q_{sc}, V_c\}}{\sum_c \max\{q_{sc}, V_c\}},$$

where

$$V_c = \sum_i U(q_i) \times q_{ic}.$$

As such,

$$\min_i\{q_{ic}\} \leq V_c \leq \max_i\{q_{ic}\}.$$

Therefore, it is easy to see that

$$R \leq F_{q_s}(V) \leq S.$$

Now, looking at the ratio, $\frac{F_{q_s}(V)}{F_{q_s}(W)}$, we get

$$\frac{R}{n \times S \times U_L} \leq \frac{F_{q_s}(V)}{F_{q_s}(W)} \leq \frac{S}{n \times R \times U_S}.$$

We know that $S \leq 1$, it follows that

$$\frac{R}{n \times U_L} \leq \frac{F_{q_s}(V)}{F_{q_s}(W)} \leq \frac{1}{n \times R \times U_S}$$

□

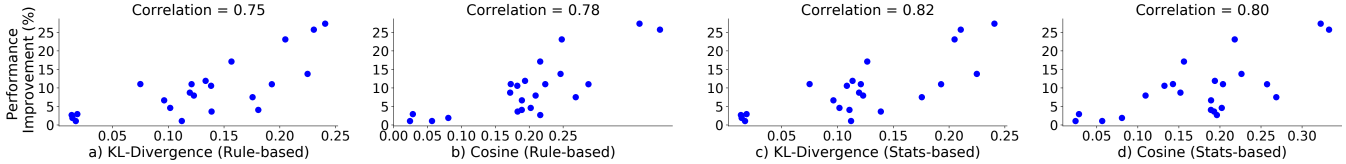
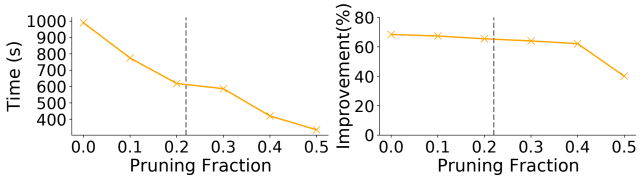
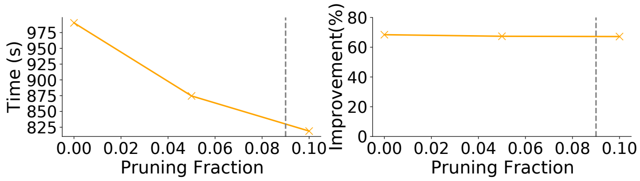


Figure 16: Impact of using KL-Divergence and Cosine similarity measures for computing benefit on TPC-H workload



a) Reduction in Time b) Reduction in Improvement

Figure 18: Impact of pruning queries on TPC-H workload



a) Reduction in Time b) Reduction in Improvement

Figure 19: Impact of pruning features on TPC-H workload

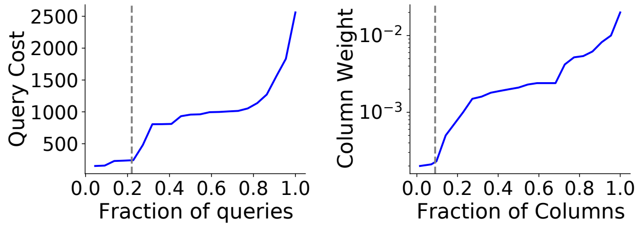


Figure 17: Distribution of query costs and columns weights for TPC-H workload

C OTHER SIMILARITY MEASURES

In this section, we present empirical results to show the effectiveness of *KL-divergence* and *cosine similarity* for measuring similarity between queries.

Cosine Similarity computed as follows:

$S(q_i, q_j) = \frac{\sum_c (q_{ic} \times q_{jc})}{||q_i|| \times ||q_j||}$.

KL-Divergence: We first compute the probability of the column q_{ic} being used, $P(q_{ic})$, as $\frac{\bar{w}_{c_i}}{\sum_j \bar{w}_{c_j}}$. We then define similarity using KL-divergence as follows:

$$S(q_i, q_j) = 1 - \sum_c P(q_{ic}) \times \log\left(\frac{P(q_{ic})}{P(q_{jc})}\right).$$

Figure 16 depicts the correlation of the above measures using rule- and statistics-based featurization. We see that these measures are also effective in capturing similarity between queries, giving high correlations. However, our analysis shows the weighted Jaccard measure results in relatively better correlation. We make similar observations on other workloads such as TPC-DS and Real-M.

D PRUNING OPTIMIZATIONS

We further propose pruning optimizations that remove queries and indexable columns that do not lead to a significant improvement after index tuning. For instance, a workload may have many queries with extremely small utilities. The contribution of such queries to the improvement over the workload is very small, upper-bounded by their original costs, and can thus be pruned. Similarly, the efficiency of the compression algorithm also depends on the size of the workload features. For complex workloads, the size of the features can be large, making the computation of similarity between pairs of queries expensive. Thus, we can remove features whose weights are extremely small. A low weight of a column in the workload features indicates that either the index column is not frequent in the workload or its contribution to the improvement in costs of queries is very small.

Specifically, we remove α fraction of the queries (or columns) with the smallest costs (or column weights) in the workload. Given the distribution of cost of queries and column weights (as used in the summary features), one can identify the value of α for pruning. We notice that workloads typically have a tail of low cost queries and low-weight columns followed by a rise in these values. One option is to set the value of α to the first knee point [36]. For instance, Figure 17 depicts such distributions for the TPC-H workload. We see that about .22 fraction of the queries have small costs (< 350) compared to the rest of the workload. Similarly 9% of the columns have significantly small weights (< 10¯3). While currently we assume that α is provided as input to ISUM, we can also automatically identify the value of α using efficient knee point detection algorithms such as the one proposed in [36].

Empirical Evaluation: To show the effectiveness of pruning, we perform an experiment over a TPC-H workload consisting of 100k queries. We set the compressed workload size to 200. Figures 18 and 19 depict the impact on compression time and improvement as we vary the fraction of pruning (x-axis). Without any pruning, ISUM gives an improvement of 68.8% taking approximately 17 minutes. We observe that when we set $\alpha = .22$ (i.e., the first knee point in Figure 17), the compression time reduces to 12 minutes while the improvement decreases to 67.2%. Similarly, while pruning columns, when we set $\alpha = .09$, the compression time reduces to 14 minutes while the change in improvement is negligible (< 1.5%) very high. Thus, we see that with our pruning techniques, we are able to see a significant reduction in compression time (> 20%) with small reduction in improvements (< 5%). We believe that the proposed pruning techniques will be more effective in massive cloud workloads consisting of millions of queries.

E IMPACT ON EXECUTION TIME

For empirical evaluation in Section 8, we use the same improvement metric (computed using optimizer estimated costs) that index tuning

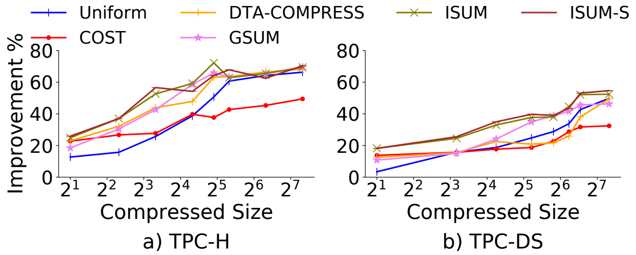


Figure 20: Impact of compression algorithms when improvement is measured using execution time

advisors [2, 15, 28] use for selecting indexes. To further validate, we measure the improvement using execution time of the queries.

Figure 20 depicts the improvement when measured using actual runtime as we vary the size of the workload. We use the similar settings as in Section 8. We see that both ISUM and ISUM-S result in higher improvement w.r.t. other baselines for a large number of compressed workload sizes. ISUM performs better than baselines when the size of the workload is small indicating it is able to identify important queries earlier. The improvement, however, is smaller with high variance compared to when estimating improvement using optimizer estimate costs. Furthermore, we see that 16 (out of 92 queries) in TPC-DS and 3 out of 22 queries in TPC-H have regressions in execution time when using the recommended indexes. Both variance and regression in execution time can attributed to the fact optimizer cost models do not accurately measure the actual execution time.

Note that the execution time of queries for different configurations is expensive to obtain and requires "building" of indexes (instead of using hypothetical indexes) which is costly and often infeasible. Hence although more accurate, execution time is rarely used by index tuning tools. We believe that making optimizer estimated cost more accurate or close to actual runtime is an orthogonal problem. Prior work [23, 37] have observed similar findings and proposed techniques that can be leveraged to improve the accuracy of optimizer cost models.