Comprehensive and Efficient Workload Compression

Shaleen Deep[†], Anja Gruenheid[‡], Paraschos Koutris[†], Jeffrey Naughton[‡], Stratis Viglas[‡]
[†]University of Wisconsin - Madison, [‡]Google Inc.
{shaleen,paris}@cs.wisc.edu
{anjag,naughton,sviglas}@google.com

ABSTRACT

This work studies the problem of constructing a representative workload from a given input analytical query workload where the former serves as an approximation with guarantees of the latter. We discuss our work in the context of workload analysis and monitoring. As an example, evolving system usage patterns in a database system can cause load imbalance and performance regressions which can be controlled by monitoring system usage patterns, i.e., a representative workload, over time. To construct such a workload in a principled manner, we formalize the notions of workload *representativity* and *coverage*. These metrics capture the intuition that the distribution of features in a compressed workload should match a target distribution, increasing representativity, and include common queries as well as outliers, increasing coverage. We show that solving this problem optimally is computationally hard and present a novel greedy algorithm that provides approximation guarantees. We compare our techniques to established algorithms in this problem space such as sampling and clustering, and demonstrate advantages and key trade-offs.

PVLDB Reference Format:
Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, Stratis Viglas. Comprehensive and Efficient Workload Compression. PVLDB, 14(3): 418 - 430, 2020.
doi:10.14778/3430915.3430931

1 INTRODUCTION

Performance tuning has been at the core of database system development and deployment since its inception. To facilitate effective system design and development, we need to understand how the system is used over time. For example, if a system is developed for transactional workloads but is increasingly used for analytical workloads, its usage patterns significantly shift, potentially resulting in performance regression. The first step towards a holistic understanding of system usage is to perform an in-depth analysis of the query workloads it is serving. However, logs from production database systems are far too large to examine manually. To be able to identify key components of the workload, we propose to create and monitor a subset of the input workload which *closely* represents the original workload. To that end, our work presents a semi-supervised framework to *compress* analytical workloads.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:10.14778/3430915.3430931

Problem Motivation. The need for our work stems from the complexity of contemporary database deployments, which have scaled with the number of customers and the workloads to serve. In order to predict the performance of a RDBMS on a large workload, it is common to evaluate it on a *benchmark workload* that resembles the target workload. Historically, benchmarked workloads are either standardized (such as TPC-H [2], SSB [30], YCSB [8], and Wisconsin Benchmark [12]) or created by domain experts who manually curate queries. If we wish to construct a custom benchmark for every use case of each customer, the second solution becomes unsustainable. Thus, automatic workload characterization and subsequently workload compression are a means to address this issue in an efficient and scalable way.

Approach. With commercial deployments serving billions of queries per day, the size of a system's workload-to-analyze quickly escalates. Instead of using every query of the workload, we propose to use a smaller sample of the workload while qualitatively not degrading the result of the process. We call this *workload compression* or *summarization*[1]. Constructing a compressed workload is challenging for several reasons. First, there is no universal set of goals to consider as representative for the workload: the output changes depending on the metric we optimize for. Second, the production workload to compress often does not fit any well-known statistical distribution, thereby making workload synthesis extremely challenging. Lastly, there are a variety of variables to take into account in a real production deployment: different job sizes, a wide range of query run times, observable skew due to temporal or spatial locality, query complexity. It is, therefore, unclear what features are the salient ones when it comes to characterizing a workload adequately. The two following examples illustrate the diverse characteristics of workload compression.

Example 1. *Consider the developers of an application who use a query engine in production. The developers would like to create performance accountability of the query engine, i.e., they would like to create compliance benchmarks to track the query engine performance. Suppose that application generates a workload mainly consists of short look-up queries but contains a handful of long-running queries, then both of these query types must be present in the benchmark. Thus, the benchmark workload must have* high coverage *by including queries with differing run times to track the query engine performance on both types of queries.*

Example 2. *System performance can be tuned by recommending indexes to speed up query processing. However, the complexity of index recommendation grows quadratically with the workload size. Therefore, we would like to find a compressed workload that is* highly

[1]We will be using the terms *workload compression* and *workload summarization* interchangeably, as we will for the terms *compressed workload* and *summary workload*

representative, *i.e., it has the same performance characteristics as the original input workload and use it for index recommendation instead.*

Finding a workload with both high representativity and high coverage is a challenging combinatorial problem for the following reasons: (*i*) *Defining metrics formally.* It is unclear what it means to have high coverage and representativity since these metrics are dependent on the application context. (*ii*) *High workload heterogeneity and variability.* While manual queries are smaller and easier to deal with, it is not uncommon that workload queries are machine-generated with more than 50 joins in production workloads. A good approximation of the input workload needs to contain all types of queries. (*iii*) *Increasing workload scale.* Production database systems routinely serve billions of queries every day, which makes any analysis challenging. Moreover, a significant fraction of these workloads is over ad hoc queries and tables rather than carefully designed schemas, making pattern mining over large, unstructured data sources even more difficult.

Prior work. Previous work on workload compression has used a variety of techniques ranging from random sampling and clustering to sophisticated ML models. For instance, [6, 7] employ clustering by defining a customized distance function for each application. More recently, [15] explores machine learning for workload compression. The insight here is to train a model specifically for SQL queries (similar to Word2Vec). Most closely related to our setting is the query log summarization framework, Ettu [18, 19]. Ettu summarizes query logs by parsing the syntax tree of queries and performing hierarchical clustering where the distance metric between queries is based on the number of common subexpressions. All of the above proposals have certain limitations. The techniques in [6, 7] are not scalable even for medium-sized workloads owing to high time complexity of $O(n^2)$ where n is the input workload size. While [15] does not suffer from quadratic complexity, it requires expensive preprocessing to train the machine learning model. The approach described in [19] ignores query execution statistics, templated queries, and stored procedures. Additionally, input workloads are often skewed in some way, and it is critical to ensure that the summary exhibits the same kind of characteristics as the input workload; this is not possible without some notion of representativity. Note that prior work, including our own, ignore any query execution impact of concurrently executed queries. For instance, they may compete for the same set of resources which in turn affects the performance. Designing representative workloads with provable guarantees that also take such effects into account remains a challenging open problem.

Contributions and organization. In this work, we introduce a novel, generic framework for workload compression of analytical queries that applies to a wide variety of performance tuning tasks. We design, implement, and evaluate our workload compressor with robust guarantees for representativity and coverage. More specifically, we

- formally define representativity and coverage to formulate workload compression as an optimization problem (Section 2) and propose a set of error metrics.
- prove that maximizing representativity is a hard problem even in restricted settings (Section 4).
- propose a novel objective that exploits submodularity to provide provable guarantees about the quality of a compressed workload (Section 5). Our algorithm allows for a smooth trade-off between representativity and coverage.
- apply optimizations to improve the performance of the algorithm and describe how submodularity can be exploited for efficient computation (Section 6).
- evaluate our approach by comparing to sampling and clustering-based methods in Section 8. We experimentally demonstrate that our framework and metrics are powerful enough by applying them to three practical use cases of (*i*) schema index tuning; (*ii*) materialized view recommendation; and (*iii*) index and view recommendation. We show that our techniques require two orders of magnitude less time to create a compressed workload with better representativity and coverage as compared to clustering-based approaches.

2 PROBLEM SETTING

In this paper, we assume that a query log L contains a finite collection of queries. For each query $q \in L$, we will create a representation that *featurizes* the query, such as finding predicates in the `WHERE` clause or table names present in the `FROM` clause. We assume that the universe of features in both a log and a query is enumerable and finite. This requirement is essential in order to define appropriate metrics. We also assume that the log contains execution statistics that can be looked over query-by-query. Such statistics are recorded by all DBMS ([1] shows the statistics recorded by SQL Server).

2.1 Notation

We define the input workload as a *multiset* $\mathbf{W} = \{q_1, \ldots, q_n\}$ that consists of n queries. A workload is a multiset, since the same query may occur many times in the workload. Each q_i is a log entry that contains the SQL text of the query and its execution information.

Features. We summarize a workload with respect to its feature set. We consider two types of features:

- *Categorical features* ($\mathbf{F}_{\text{categorical}}$): these features capture values derived from the syntax tree of a **SQL** query.
- *Numeric features* ($\mathbf{F}_{\text{numeric}}$): these features capture numeric values that are derived from profiling statistics of the query.

The feature set is defined as $\mathbf{F} = \mathbf{F}_{\text{numeric}} \cup \mathbf{F}_{\text{categorical}}$. We use $\text{dom}(\mathbf{S}, f)$ to denote the active domain of feature f for some workload $\mathbf{S}$ and use *token* to refer to feature values in the active domain.

Feature value multiplicity. To design a generic approach, we consider not only single-valued features but also multi-valued features. In other words, features of a query $\mathbf{q}$ can have multiple domain values associated with it (that can also occur multiple times). For example, consider the function calls present in the `SELECT` clause. Since a function call such as `SUM` can be present multiple times in the SQL statement, it is a multi-valued feature. To formally model multi-valued features, we represent the value of a feature f of query $\mathbf{q}$ as a multiset of tokens $f(\mathbf{q})$. The size of a query $\mathbf{q}$, denoted $\|\mathbf{q}\|$, is the total number of tokens across all features, $\|\mathbf{q}\| = \sum_{f \in \mathbf{F}} |f(\mathbf{q})|$. The size of a workload $\|\mathbf{S}\|$ is the sum of sizes of all queries in the workload. For a workload $\mathbf{S}$, we define $f(\mathbf{S}) = \biguplus_{q \in \mathbf{S}} f(\mathbf{q})$. Finally, for a token $t \in \text{dom}(\mathbf{S}, f)$, its *frequency*, denoted $\text{m}_{\mathbf{S}}(t, f)$ is the number of times the token appears in the multiset $f(\mathbf{S})$.

	f_1^c	f_2^c	f_3^c	f_4^c
Q_1	AVG, MAX, MAX	T_1	$T_1.a$	
Q_2	COUNT	T_1, T_2		
Q_3		T_1, T_2, T_3		$T_1.a, T_2.b, T_3.c$

Table 1: Categorical features.

	f_1^n	f_2^n	f_3^n	f_4^n	f_5^n	f_6^n
Q_1	5ms	4ms	5MB	100	2ms	0
Q_2	10ms	2ms	10MB	1000	3ms	1
Q_3	8ms	5ms	20MB	500	4ms	2
Q_1	0	6	0	0	0	0
Q_2	10	0	6	10	5	5
Q_3	6	10	10	4	10	10

Table 2: Extracted numeric feature values.

2.2 Encoding Queries

For the purpose of this paper, we consider a limited set of features that can be derived from a typical database system log entry, i.e., the SQL query text and its execution statistics. The features used throughout this paper are:

Categorical features. These are features derived from parsing the query statement, we choose: (1) Function calls in the SELECT clause (such as AVG, MAX or some stored procedure), (2) tables in the FROM clause of the (sub-)query, (3) columns in the GROUP BY clause, and (4) columns in the ORDER BY clause.

Numeric features. These are features that describe the performance of a query, we choose: (1) The total execution time of the query, (2) planning time of the query, (3) total size of the input to the query, (4) total output rows of the query, (5) CPU time spent executing the query, and (6) the number of joins as parsed from the query. It is likely that for some numeric features such as execution time, no two queries have identical values. To sparsify numeric features, we normalize the values such that they are in $[0, 1]$ by leveraging the largest and smallest values in the active domain. Using this methodology allows us to reason about their discrete distribution. We transform the scaled numeric values into a histogram by assigning a bucket id $b_i \in \{0, \ldots, H\}$ to each numeric value $v_i \in [0, 1]$ such that $b_i = \lfloor v_i \cdot H \rfloor$. Observe that this transformation also changes the active domain to $\{0, \ldots, H\}$ for the respective numeric features.

Example. Consider the workload $\mathbf{W} = \{Q_1, Q_2, Q_3\}$:

Q_1 = SELECT a, AVG(b), MAX(c), MAX(d) FROM T_1 GROUP BY a
Q_2 = SELECT COUNT(*) FROM T_1, T_2 WHERE $T_1.a = T_2.a$
Q_3 = SELECT * FROM T_1, T_2, T_3 ORDER BY $T_1.a, T_2.b, T_3.c$

The corresponding domains for the categorical features are:

$$\text{dom}(\mathbf{W}, f_1^c = \text{function\_call}) = \{\text{AVG}, \text{MAX}, \text{COUNT}\}$$
$$\text{dom}(\mathbf{W}, f_2^c = \text{table\_reference}) = \{T_1, T_2, T_3\}$$
$$\text{dom}(\mathbf{W}, f_3^c = \text{group\_by}) = \{T_1.a\}$$
$$\text{dom}(\mathbf{W}, f_4^c = \text{order\_by}) = \{T_1.a, T_2.b, T_3.c\}$$

Table 1 shows the feature values per query. Note that for query Q_1, the token MAX appears twice, since it occurs two times in the selection clause.
Additionally, we observe profile statistics as shown in Table 2. The numeric features $f_1^n, \ldots f_6^n$ correspond to the total execution time, planning time, size of the input, total output rows, total CPU time, and the number of joins. For all numeric features, we normalize the values into a histogram with $H = 10$, the resulting bucket assignment is shown in the lower part of Table 2.

The sizes of the queries are as follows: $\|Q_1\| = 11$, i.e., six numeric feature tokens, three tokens in f_1^c, and one in f_2^c and f_3^c respectively, $\|Q_2\| = 9$ and $\|Q_3\| = 12$. The size of the workload is $\|\mathbf{W}\| = 32$.
Extension to other features. Our techniques are not limited to presented features, but we simply choose these for demonstration purposes. Unlike the popular summarization scheme introduced by Aligon et al. [4], we do not restrict the features to relation names and columns in the WHERE, SELECT, or FROM clause. Thus, in principle, any feature can be used in our framework. However, in practice, the choice of features is limited by the hardware and available resources. For example, if a GPU is used for some of the queries, it would be useful to add features such as `average_memory_bandwidth_used`. One could also create a higher-order feature derived from two different features $f = f_1 \times f_2$ that captures the co-occurrence of $\langle t_1, t_2 \rangle$ where t_1 and t_2 are tokens of f_1 and f_2. Capturing cross-feature information may lead to improved summaries, but defining such features requires a principled approach to feature engineering (see Section 9 for more details). It is also possible to encode features such as query plan fragments, indexes used during query evaluation, and physical execution operators using standard techniques of 1-hot encoding and transforming a query plan into a tree of vectors (see Sec 3.2 in [24]) for more details.

2.3 Metrics

We next formalize the definitions of the coverage and representativity metrics that are subsequently used throughout this paper.
Coverage. Given a feature $f \in \mathbf{F}$, the coverage factor α_f is defined as the fraction of tokens covered by the compressed workload for feature f. To generalize this to a metric across all features, we can either compute the minimum or average α_f. Formally:

Definition 1 (Coverage). *Let* $\mathbf{S}$ *be a summary of the workload* $\mathbf{W}$. *The* coverage factor *for a feature* $f \in \mathbf{F}$ *is defined as* $\alpha_f = \frac{|\text{dom}(\mathbf{S}, f)|}{|\text{dom}(\mathbf{W}, f)|}$. *The* minimum coverage factor *and* average coverage factor *are respectively defined as:*

$$\alpha_{\min} = \min_{f \in \mathbf{F}} \alpha_f, \qquad \alpha_{avg} = \sum_{f \in \mathbf{F}} \alpha_f / |\mathbf{F}|$$

Observe that both the minimum and average coverage values are always in $[0, 1]$. A score of 1 means that the coverage is perfect.
Representativity. A representative summary of the workload must capture the structural properties of the original workload. Specifically, workload $\mathbf{W}$ induces a discrete distribution $p_{\mathbf{W}}(\cdot)$ over the

tokens present in the features of the queries in the workload. In particular, for any token t,

$$p_{\mathbf{W}}(t) = \frac{\mathrm{m}_{\mathbf{W}}(t, f)}{\sum_f \sum_{v \in \mathrm{dom}(\mathbf{W}, f)} \mathrm{m}_{\mathbf{W}}(v, f)}$$

In other words, $p_{\mathbf{W}}(t)$ denotes the probability of selecting token t if we choose a token from $\mathbf{W}$ uniformly at random. The summary $\mathbf{S}$ will induce a distribution $p_{\mathbf{S}}(\cdot)$; the representativity metric then measures the distance between $p_{\mathbf{S}}$ and $p_{\mathbf{W}}$. In general, we wish $p_{\mathbf{S}}$ to be as close to some target distribution $d(\cdot)$. Note that the target distribution can be different from $p_{\mathbf{W}}$ if wanted.

Definition 2 (Representativity). *Given a target token distribution d, the* representativity *w.r.t. d is defined using the following two metrics:*

$$\rho_1(d) = 1 - \frac{1}{2} \sum_f \sum_{t \in \mathrm{dom}(\mathbf{W}, f)} |p_{\mathbf{S}}(t) - d(t)|$$

$$\rho_\infty(d) = 1 - \max_f \max_{t \in \mathrm{dom}(\mathbf{W}, f)} |p_{\mathbf{S}}(t) - d(t)|$$

The ρ_1 metric essentially measures the total variation distance between the two distributions and is a popular distance metric for graph visualizations [22]. ρ_∞ metric captures the largest deviation in the distribution across all features. If the representativity score is 1, we say that the compressed workload is perfectly representative. Note that there are other possible definitions of representativity. We refer the reader to the full version [10] for more discussion.

User-specific modifications. If users have specific domain knowledge, they may want to use a (i) weighted version of computing these metrics and/or, (ii) target distribution for representativity. Both of these modifications are supported in our framework. For the former, we require that each feature is assigned a weight w_f such that $\sum w_f = 1$. For the latter, we define a general version of the metrics w.r.t. an arbitrary target distribution d; the case where d is the same as the input distribution becomes a special case. This functionality is useful in applications such as test workload generation. Developers frequently use queries to test their code while developing the functionality in RDBMS. However, instead of choosing from a set of predefined queries, it is more desirable to choose the test workload from a set of production queries, which increases more confidence in the testing of the functionality. Due to lack of space, we refer the reader to the full-report [10] for more details.

Example. Consider the setup from our running example and let $\mathbf{S} = \{Q_1, Q_3\}$. The normalizing factor for $\mathbf{W}$ and $\mathbf{S}$ is $14 + 18 = 32$ (14 categorical tokens and 18 numeric feature tokens) and $11 + 12 = 23$ respectively. Table 3a and Table 3b show the $p_{\mathbf{W}}$ and $p_{\mathbf{S}}$ distributions (target d is set to be the input distribution) for the function call and table reference features.

The reader can verify that $\rho_1 = 1 - \frac{1}{2}\left\{18(\frac{1}{23} - \frac{1}{32}) + \frac{14}{32} - \frac{5}{23}\right\} \approxeq 0.779$ and $\rho_\infty = 1 - \frac{1}{32} \approxeq 0.96$.

Similarly, Table 3c shows α_f for each feature. For example, only two tokens of the function call feature are covered by $\mathbf{S}$ and ⟨COUNT⟩ is missed since $Q_2 \notin \mathbf{S}$. All numeric features have $\alpha_f = \frac{2}{3}$. Thus, the minimum coverage factor is $\alpha_{\min} = \frac{2}{3}$ and the average coverage factor is $\alpha_{avg} = \frac{23}{30}$.

token	$p_{\mathbf{W}}(t)$	$p_{\mathbf{S}}(t)$
AVG	0.031	0.043
MAX	0.062	0.086
COUNT	0.031	0

(a) Function call feature.

token	$p_{\mathbf{W}}(t)$	$p_{\mathbf{S}}(t)$
T_1	0.093	0.086
T_2	0.062	0.043
T_3	0.031	0.043

(b) Table reference feature.

cov	f_1^c	f_2^c	f_3^c	f_4^c	f_1^n	f_2^n	f_3^n	f_4^n	f_5^n	f_6^n
α_f	$\frac{2}{3}$	1	1	1	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$

(c) Feature coverage for workload S.

Table 3: Representativity and coverage computation.

2.4 Problem Statement

Given an input workload $\mathbf{W} = \{q_1, \ldots, q_n\}$ where each query $\mathbf{q}$ in $\mathbf{W}$ is associated with a non-negative cost $c(\mathbf{q})$ such as the size of the summary workload.

Assuming a target distribution $d(\cdot)$ over these tokens, a budget constraint $B \geq 0$, and a parameter $\beta \in [0, 1]$, our goal is to construct a *summary workload* $\mathbf{S} \subseteq \mathbf{W}$ such that:

- the cost of the summary workload is less than the budget, $\sum_{\mathbf{q} \in \mathbf{S}} c(\mathbf{q}) \leq B$; and
- the quantity $\beta \cdot \alpha + (1 - \beta) \cdot \rho(d)$ is maximized, where $\alpha \in \{\alpha_{min}, \alpha_{avg}\}$ and $\rho \in \{\rho_1, \rho_\infty\}$.

Here, the user-specified parameter β controls the trade-off between the coverage and representativity metrics. If $\beta = 0$, we optimize for representativity only, and if $\beta = 1$, we optimize for coverage. The compression ratio, $\eta = 1 - c(\mathbf{S})/c(\mathbf{W})$, is the fraction of queries that have been pruned. Observe that the larger the value of η, the smaller the compressed workload.

3 DESIGN CONSIDERATIONS

Several desiderata are important to consider when compressing a workload, and these form a rich design space. We now describe these desiderata and their corresponding trade-offs.

High Coverage. High coverage is desirable to ensure that long-tail feature values are part of the compressed workload. Ideally, we would like to maximize the coverage subject to certain budget constraints. Maximizing coverage is an NP-hard problem [14, 29] that can be efficiently approximated [29].

High Representativity. High representativity implies that the compressed workload must faithfully reproduce the target distribution, which can be either derived from the input workload's feature distribution or specified by the user. This is a key requirement for successful workload compression when used, for example, in the context of performance analysis in a database system.

Customizability. Representativity and coverage are competing objectives, which makes the task of maximizing both metrics simultaneously hard. For instance, simple random sampling achieves high representativity but may miss long-tail queries. On the other hand, set cover algorithms maximize coverage but will not pick queries whose features have already been covered. It is therefore desirable that the user can control this trade-off smoothly, which we realize through the parameter β.

User-Specific Constraints. Users may want to specify constraints on some property of the compressed workload. For instance, the user may want to limit the size, total execution time, or the representativity target distribution of the compressed workload. The framework should be flexible enough to allow users to specify these constraints on-the-fly. For simplicity, we restrict ourselves to two types of constraints: (i) specifying the desired feature distribution of the summary workload; and (ii) non-negative modular constraints (i.e., a knapsack constraint) of the form $\sum_{\mathbf{q}\in\mathbf{S}} c(\mathbf{q}) \leq B$, where $c(\cdot)$ can be any cost function.

Scalability. Efficient computation of the compressed workload is a key requirement for any framework to be deployed in practice. Ideally, the compression algorithm must compute the compressed workload fast and scale effectively to large input workloads. This will also allow the user to find the correct parameter settings for fine-tuning the representativity and coverage trade-off dynamically.

Incremental Computation. Consider a user who wants to analyze how the workload is changing over time with respect to a set of features. For this case, we want to avoid computing the compressed workload from scratch every day; instead, it would be better to create a summary for each day, and then merge them. In other words, we would like to construct *mergeable* compressed workloads.

4 HARDNESS RESULTS

In this section, we show that our problem is computationally hard for any parameter $\beta \in [0, 1]$, even for the simple case where the cost function is the same constant for every query, *i.e.*, $c(\mathbf{q}) = 1$. We note that [14] already shows that the problem is NP-complete when $\beta = 1$ (*i.e.*, we want to maximize only coverage) for both coverage metrics α_{min} and α_{avg}. The next theorem shows that the NP-hardness result extends for any choice of the parameter β.

Theorem 1. *Let $\alpha \in \{\alpha_{min}, \alpha_{avg}\}$, $\rho \in \{\rho_1, \rho_\infty\}$, $\beta \in [0, 1]$, and $d(\cdot)$ be a target distribution. Then, the problem of finding a summary $\mathbf{S} \subseteq \mathbf{W}$ such that $|\mathbf{S}| \leq B$ and the quantity $\beta \cdot \alpha + (1 - \beta) \cdot \rho(d)$ is maximized is NP-complete. In particular, the problem remains NP-complete when d is the input distribution $p_{\mathbf{W}}$, and there exists only one multi-valued feature.*

Since the problem of maximizing the objective function with respect to a cost constraint is NP-hard, an approximation algorithm with theoretical guarantees is required to solve the problem. Indeed, if we restrict to optimize only for coverage (so $\beta = 1$) and a single feature, there exists a greedy algorithm (by [14]) that achieves an $(1 - 1/e)$-approximation ratio. Next, we show that the problem is APX-hard for any choice of parameter β.

Theorem 2. *Let $\alpha \in \{\alpha_{min}, \alpha_{avg}\}$, $\rho \in \{\rho_1, \rho_\infty\}$, $\beta \in [0, 1]$, and $d(\cdot)$ a target distribution. Then, the problem of finding a summary $\mathbf{S} \subseteq \mathbf{W}$ such that $|\mathbf{S}| \leq B$ and the quantity $\beta \cdot \alpha + (1 - \beta) \cdot \rho(d)$ is maximized is APX-hard.*

The problem gets even more complex if representativity is taken into account. As the next lemma shows, neither ρ_1 or ρ_∞ metrics satisfy desirable properties from an optimization perspective.

Lemma 1. *The ρ_1 and ρ_∞ metrics are not monotone or submodular.*

5 PROBLEM SOLUTION

There are two often-applied methods to solve the summarization problem: *clustering* and *random sampling*. While clustering-based methods (such as k-medoids[2] and hierarchical clustering[3]) identify the patterns in the workload, they suffer from the following drawbacks: (i) $O(n^2)$ time complexity, (ii) sensitivity to the distance function and (iii) the number of clusters k is required as an input. The best value for k is not known a priori. To address this problem, one commonly used idea is to run the clustering algorithm several times, where the cluster size is doubled in every iteration. However, this may be far from optimal because of the sensitivity of the metrics to the size constraint. To address the drawbacks of clustering and random sampling, we present a new approach to summarization. We define a new objective function (Section 5.1) that can be parametrized to control the trade-off between coverage and representativity followed by efficient algorithm (Section 5.2).

5.1 A New Objective Function

Instead of using the initial objective of the summarization problem, we replace it with the following objective, where $\gamma \in (0, 1]$ is a smoothing parameter that controls the trade-off between representativity and coverage:

$$G(\mathbf{S}, \gamma) = \sum_f \sum_{t \in \mathsf{dom}(\mathbf{W}, f)} d(t) \cdot \log\left(\frac{\mathsf{m}_\mathbf{S}(t, f) + \gamma}{\gamma}\right) \qquad (1)$$

Before we explain the intuition of choosing this objective, we show that it satisfies several properties that make it amenable to optimization. In particular, $G(\mathbf{S}, \gamma)$ is a non-negative, monotone and submodular set function.

Proposition 1. *For any value $\gamma \in (0, 1]$, the set function $G(\mathbf{S}, \gamma)$ is non-negative, monotone and submodular.*

Analysis of the Objective. To understand the intuition behind the choice of the objective function, we first discuss how $G(\mathbf{S}, \gamma)$ behaves for very small values of γ.

Lemma 2. *Let $\mathbf{S}_1, \mathbf{S}_2$ be two summaries of a workload $\mathbf{W}$ such that $\bigcup_f \mathsf{dom}(\mathbf{S}_1, f) \subsetneq \bigcup_f \mathsf{dom}(\mathbf{S}_2, f)$. Then, for $\gamma \to 0$ we have $G(\mathbf{S}_1, \gamma) < G(\mathbf{S}_2, \gamma)$.*

Lemma 2 tells us that when the parameter γ tends to zero, a summary that covers strictly more tokens will always have a better value for the objective G, independent of the representativity of each summary. This implies that if there exists a summary $\mathbf{S}$ within the budget B that covers all tokens of $\mathbf{W}$, then an optimal solution for G will always cover all tokens as well. Now, let us consider a summary $\mathbf{S}$ that achieves perfect coverage. We can then write:

$$\begin{aligned}
&\lim_{\gamma \to 0} \{G(\mathbf{S}, \gamma) + \log \gamma\} = \sum_f \sum_{t \in \mathsf{dom}(\mathbf{W}, f)} d(t) \cdot \log \mathsf{m}_\mathbf{S}(t, f) \\
&= -\sum_f \sum_{t \in \mathsf{dom}(\mathbf{W}, f)} d(t) \cdot \log \frac{d(t)}{p_\mathbf{S}(t)} - H(d) + \log \|\mathbf{S}\| \\
&= -KL(d \| p_\mathbf{S}) - H(d) + \log \|\mathbf{S}\|
\end{aligned}$$

[2] K-medoids is an iterative greedy algorithm that chooses k cluster centers, assigns all points to the closest center and iteratively refines the points in each cluster.
[3] Hierarchical clustering is a top-down approach where all points start in a single cluster and the algorithm recursively splits the points into k disjoint clusters.

where $KL(d\|p)$ is the Kullback-Leibler (KL) divergence, a metric that captures the difference of the two distributions:

$$KL(d\|p) = \sum_{x \in \Omega} d(x) \ln \frac{d(x)}{p(x)}$$

Thus, when $\gamma \to 0$, among all summaries with the same size and perfect coverage, the objective prefers the one that minimizes the KL divergence between the target distribution and the summary. We should note here that KL divergence is related to the total variation distance by the well-known Pinkser's inequality: $TV(d,p) \leq \sqrt{\frac{1}{2} KL(d\|p)}$. If the summaries do not have the same size, then the summary size will also influence the objective.

As γ increases from 0 to 1, the penalty for not covering a token decreases. Hence, an optimal solution will focus less on maximizing coverage and more on maximizing representativity. For larger values of γ, the objective function will choose the summary that minimizes the KL divergence between the target distribution d and the 'smoothed' summary distribution where the probability of a token is proportional to $m_S(t,f) + \gamma$ instead of the frequency $m_S(t,f)$. Intuitively, one can think of the case of $\gamma = 1$ as if each token already starts with a count of 1 as the summary is constructed.

5.2 A Greedy Algorithm

We now present an algorithm that solves our optimization problem which can be formally stated as follows:

$$\textbf{maximize} \quad G(\mathbf{S}, \gamma)$$
$$\textbf{subject to} \quad \sum_{\mathbf{q} \in \mathbf{S}} c(\mathbf{q}) \leq B, \quad \mathbf{S} \subseteq \mathbf{W}$$

We solve the above optimization problem greedily in Algorithm 1. The algorithm starts with an empty summary $\mathbf{S}_0 = \emptyset$. At the i-th iteration of the main loop, it adds the query from the workload that maximizes the *normalized marginal gain* $\Delta(\mathbf{q} \mid \mathbf{S}_{i-1})$ to the current summary $\mathbf{S}_{i-1}$. The normalized marginal gain is defined as

$$\Delta(\mathbf{q} \mid \mathbf{S}) = \frac{G(\mathbf{S} \cup \{\mathbf{q}\}, \gamma) - G(\mathbf{S}, \gamma)}{c(\mathbf{q})}.$$

In other words, the algorithm greedily chooses the query with the best gain per unit of cost. To increase the efficiency of the algorithm, we apply a common optimization [21] which skips the computation of the normalized gain $\Delta(\mathbf{q} \mid \mathbf{S}_{i-1})$ of a query $\mathbf{q}$ at round $i-1$ if we know that the gain can not be larger than the query with highest gain so far (line 6). This optimization works because submodularity tells us that the gain can only decrease as the size of the summary grows (hence, values of $\Delta(\mathbf{q} \mid \mathbf{S}_k)$ for $k < i-1$ are an upper bound to the gain). Experiments in [10] show that this lazy strategy can substantially speed up execution. Algorithm 1 considers additionally the best single element solution, and chooses the best of the two (line 15). Since G is a monotone, non-negative and submodular function, it can be shown that Algorithm 1 achieves an $1/2(1-1/e)$ approximation guarantee [16, 21].

Runtime Analysis. The runtime cost of the algorithm is dominated by the cost of the main loop. During each iteration, the algorithm needs to compute the normalized marginal gain for each of the n queries (in the worst case). Since the feature vector is sparse, each iteration of the main loop can be performed in $O(n)$ time. The number of iterations can be as large as n, resulting in a worst-case runtime of $O(n^2)$. However, we can obtain better bounds depending on the budget constraint B and the cost function $c(\cdot)$. For example, if $c(\cdot)$ is the unit cost function, then the number of iterations can be at most B, and the runtime becomes $(n \cdot B)$. In general, if c_{min} is the smallest possible cost of the query, then the number of iterations is upper bounded by B/c_{min}. If we want to optimize for our original score function $\alpha\beta + (1-\beta)\rho(q)$, observe that the summary we obtain at the end of the algorithm may not be the best one. We can slightly modify Algorithm 1 by recording the best score and the corresponding set that achieves it at every iteration without any impact on the total running time.

Algorithm 1: Greedy algorithm

```
INPUT  : input workload W, cost function c, budget B,
         parameter γ ∈ (0, 1]
OUTPUT : summary workload S
1  S ← ∅
2  ∀q ∈ W : Δ(q) ← 0
3  while W ≠ ∅ do
4      Δ* ← −1
5      foreach q ∈ W do
6          if Δ(q) > Δ* then
7              Δ(q) ← (G(S∪{q},γ) − G(S,γ)) / c(q)
8          if Δ(q) > Δ* then
9              Δ* ← Δ(q)
10             q* ← q
11     if c(S) + c(q*) ≤ B then
12         S ← S ∪ {q*}
13     W ← W \ {q*}
14 S' ← argmax_{q∈W} {G({q}, γ) | c(q) ≤ B}
15 return argmax_{S,S'} {G(S, γ), G(S', γ)}
```

6 PARALLELIZATION AND INCREMENTAL COMPUTATION

Our algorithm is inherently parallelizable and well suited for incremental computation.

Parallelization. Consider the problem where our cost function is $c(\mathbf{q}) = 1$, and the budget is B. In order to parallelize Algorithm 1, we partition the input workload into B machines and run the algorithm on each machine. Each run results in B different summary workloads, one from each machine. We then merge each machine's summary workload, which will act as the new input workload to generate the final summary. Observe that the first step of generating B different summary workloads takes $O(\frac{n}{B} \cdot B) = O(n)$ time in parallel, while the second step of merging requires time $O(B^2 \cdot B) = O(B^3)$. Hence, we obtain a faster runtime if $B^3 \leq n \cdot B$, i.e $B \leq \sqrt{n}$. For values of $B \geq \sqrt{n}$, there exist algorithms that allow for parallelization with slightly worse approximation guarantees. We refer the reader to [5, 26–28] for a more detailed discussion on parallelizing submodular maximization problems.

Incremental Computation. Supporting incremental computation of a summary is critical in the case where the workload that needs

to be summarized is not provided at once, but instead constantly grows. Since we want to perform summarization *across* multiple workloads over time, we need to normalize for numeric features in a consistent way, *i.e.*, the maximum and minimum values used to normalize need to be fixed a priori. In order to do so, we fix the largest and smallest value for all numeric features explicitly. Although this assumption may feel restrictive, in our experience, setting the maximum and minimum values for a feature by looking at historical workloads works very well. For instance, it is safe to assume that the number of joins in a query will be smaller than 1000 in ad-hoc workloads. Suppose now that we have computed a summary $\mathbf{S}$ of $\mathbf{W}$, and a new set of queries $\mathbf{W}'$ is added (with the same feature set) to the current workload. Instead of computing directly the summary of $\mathbf{W} \uplus \mathbf{W}'$, we can compute a summary $\mathbf{S}'$ of $\mathbf{W}'$, and merge the two summaries to obtain $\mathbf{S} \uplus \mathbf{S}'$. The next two lemmas describe how the merged summary behaves:

Lemma 3. *Let* $\mathbf{S}$ *be a summary for* $\mathbf{W}$ *with* $\alpha_{min} = \alpha$ *and* $\rho_\infty(p_{\mathbf{W}}) = \rho$. *Also, let* $\mathbf{S}'$ *be a summary for* $\mathbf{W}'$ *with* $\alpha_{min} = \alpha'$ *and* $\rho_\infty(p_{\mathbf{W}'}) = \rho'$. *Then,* $\mathbf{S} \uplus \mathbf{S}'$ *is a summary of* $\mathbf{W} \uplus \mathbf{W}'$ *that has* $\alpha_{min} \geq \min\{\alpha, \alpha'\}/2$, $\rho_\infty \geq \min\{\rho, \rho'\}$ *and cost* $c(\mathbf{S}) + c(\mathbf{S}')$.

Lemma 4. *Consider two summaries* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *for* $\mathbf{W}_1, \mathbf{W}_2$ *respectively) with identical budget B. Then, we can produce a summary* $\mathbf{S}$ *that is a subset of* $\mathbf{S}_1 \uplus \mathbf{S}_2$, *such that its cost is at most B and its objective value is at most an* $1/2(1 - 1/e)$ *factor away from the optimal solution of G for* $\mathbf{W}_1 \uplus \mathbf{W}_2$.

As we will see in the experimental evaluation, the worst-case bounds do not occur in practice and incremental merging of the summaries works well.

7 END-TO-END FRAMEWORK

Benchmarking is an important problem to solve in a structured manner as it allows developers and users to reason about the performance of a system over time. `DIAMetrics` [11] is an end-to-end benchmarking system developed at Google for query engine-agnostic, repeatable benchmarking that is indicative of large-scale production performance. In essence, it allows users to (*a*) anonymize production data, (*b*) move data between different file storage systems, (*c*) execute preset workloads on specific systems automatically, and (*d*) visualize the results of executed benchmarks. `DIAMetrics` provides the context for which GSum was prototyped.

One of the biggest barriers of entry to benchmarking a system is that teams are often unable to provide a concise benchmark that represents their production workload. Although clustering and simple frequency-based analysis has been sufficient for some cases, in a majority of the cases it is infeasible to manually create accurate benchmarks. Workload compression provides a powerful means to generate a subset of production queries with formal guarantees. In addition to its usefulness for benchmarking, GSum can be deployed daily to build workload summaries which are used to monitor workload patterns over time. A shift in these patterns can entail several issues such as changing resource usage or execution regressions which need to be addressed in a timely manner. A full detailed description of `DIAMetrics` is beyond the scope of this paper and we refer the interested reader to [11] for more details and use cases.

8 EVALUATION

In this section, we empirically evaluate the techniques discussed throughout this paper. More specifically, we

- validate that the summarization framework is useful for index tuning, materialized view recommendations, and test workload generation.
- evaluate the runtime of all algorithms for varying workloads and summary size constraints.
- compare the coverage and representativity metrics of our solution with k-medoids, hierarchical clustering, and random sampling both on real production workloads and standardized workloads and its scalability.
- evaluate the trade-off between representativity and coverage.

For all workloads, we assume that the query log is available through the DBMS. All running time related experiments report the mean of the three observations that are closest to the median over a total of five runs. To normalize the numeric features, we set $H = 1000$. Unless specified otherwise, we choose $\gamma \to 0$ and $\beta = 0.5$. We refer to the compressed workload generated by our technique as GSum (short for Google SUMmarized workload). We perform our experiments over 3 datasets:

(1) `DataViz`: A dataset of 512796 ad-hoc data visualization queries issued against F1. This workload contains references to 2729 relations in total. The largest join query contains 19 joins and the workload has 106 unique function calls. Most expensive query in the workload takes 6 hours to execute.
(2) `TPC-H` [2]: A benchmark for performance metrics over systems operating at scale. We use a workload of 2200 queries with `SF=1` and uniform data distribution.
(3) `SSB` [30] : A benchmark designed to measure performance of database in support of data warehousing applications.

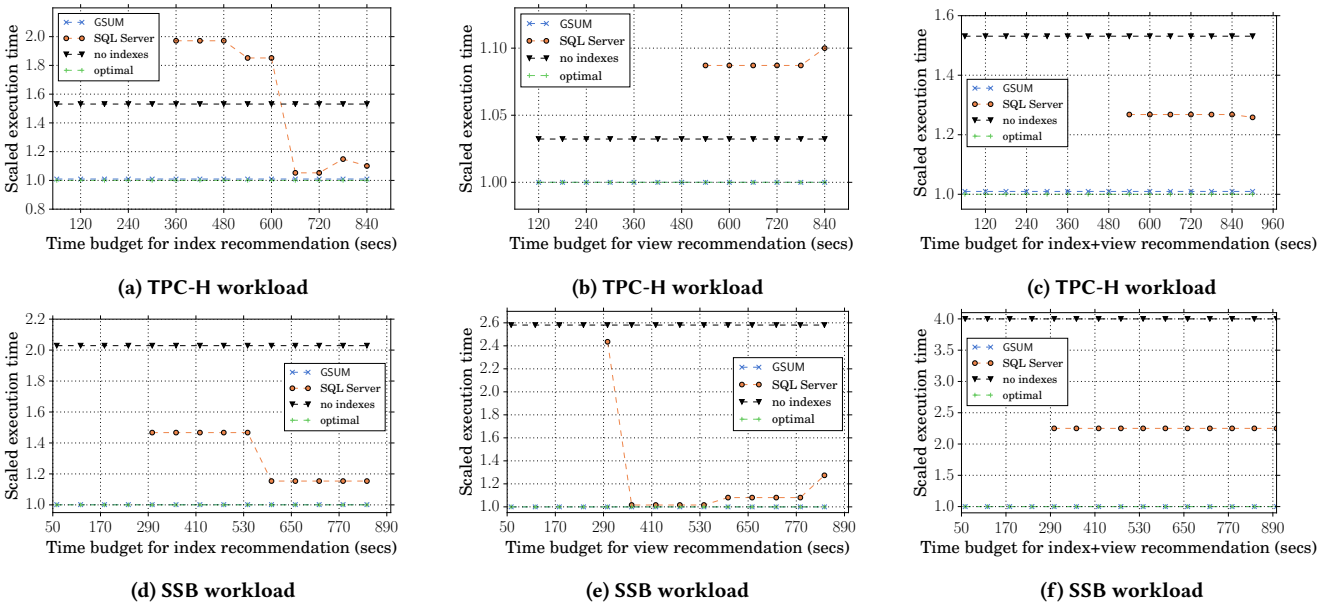
8.1 Use cases

As described earlier in this paper, there are several use cases for a summarization framework. We now explore three of these use cases, index tuning, materialized view recommendation and both of them together, to show the validity of our framework and demonstrate that coverage and representativity metrics are useful in practice. Due to a lack of space, we defer the experiments for test workload generation to the full version [10].

Experimental Setup. We use the SQL Server DTA utility as the baseline, which is a state-of-the-art industrial-strength tool that has been shipped in SQL Server for more than a decade [3]. All experiments in this section are run on a *m5a.8xlarge* AWS EC2 instance using a single core.

8.1.1 Index Tuning. Index tuning is the task of selecting appropriate indexes for a workload that improve its overall runtime. Summarization can be used in this context to determine a subset of relevant queries from the input workload and then generating indexes from the subset rather than the whole workload.

Methodology. To evaluate summarization in the context of index tuning, we leverage the same evaluation strategy as Chaudhuri et al. [7] and subsequently used by Jain et al. [15]. That is, we first measure the execution time of a workload without indexes (t_{orig})



(a) TPC-H workload (b) TPC-H workload (c) TPC-H workload

(d) SSB workload (e) SSB workload (f) SSB workload

Figure 1: Experimental results for three use cases: index tuning (a,d), materialized views (b,e) and indexes and indexed views (c,f). Execution time is scaled using the total running time when using optimal indexes to show the comparative slowdown.

and then apply an index recommendation engine, determine the recommended indexes, create these indexes, and then measure the runtime again (t_{sub}). As a baseline, we use SQL Server 2016, which comes with a built-in Database Engine Tuning Advisor. Our experiments for SQL Server show our measurements for t_{sub}^{SQL} under different temporal budget constraints for the tuning advisor, i.e., we vary the time allotted to the advisor that determines the indexes to create. For comparison, we run GSum to create a summary workload to generate $\mathbf{S}$ with constraint $|\mathbf{S}| \leq \sqrt{|\mathbf{W}|}$ while maximizing representativity and coverage, which we then use as input for the SQL Server tuning advisor. Using these indexes, we can then measure t_{sub}^{GSUM}, i.e., the time it takes to run the input workload while using index suggestions based on the compressed workload generated by our algorithm. We use all categorical and numeric features described in Section 2.

Results. Figure 1a to Figure 1d show the execution time of varying benchmarks given different index recommendation budgets with two baselines: *no indexes* and *optimal*, which uses indexes based on the 22 TPC-H query templates. Diving deeper into Figure 1a, we note that under tight index generation budget constraints (between 6 and 11 minutes), SQL Server may give recommendations that result in worse performance than using no indexes at all. The reason is that the TPC-H workload is large, so the advisor is unable to recommend appropriate indexes within these time constraints. We further note that the tuning advisor's behavior is non-monotonic, which explains an increase in t_{sub}^{SQL} with a larger index recommendation time budget. In contrast, using GSum results in a much smaller workload for the advisor to interpret, reducing the time it takes to find index suggestions significantly. With GSum, we can obtain the first suggestion for indexes within 1 minute, while it takes SQL Server 6 minutes to derive its first result. SQL Server suggests same indexes as GSum at 11 minutes. However, due to its non-monotonic behavior, an additional time budget worsens t_{sub}^{SQL} marginally. For SSB (Figure 1d), we observe that that the first index is recommended after 5 minutes which is subsequently improved when the budget is 10 minutes. We also verified that even after running the full workload with a budget of 60 minutes, the recommended indexes were no better than the indexes recommended under 1 minute. This demonstrates the benefit of using a compressed workload as opposed to the full workload. For both TPC-H and SSB workloads, the compression ratio is $\eta > 0.95$ since the compressed workload is always between size 10 to 50.

8.1.2 Materialized view recommendation. We utilize GSum to suggest materialized views using the inbuilt materialized view recommendation tool of SQL Server.

Results. Figure 1e and 1b show the results for SSB and TPC-H. For SSB, using GSum results in materialized views within 1 minute that are better up to 2.5× faster than the *no indexes* baseline. For both the workloads, even allowing up to 15 minutes of time does not improve the recommended views. After 15 minutes, SQL Server recommends the same views as GSum. In fact, for some time budgets, using the recommended views is slower as evidenced by the increasing execution time for SQL Server for both TPC-H and SSB.

8.1.3 Index and view recommendations. Finally, SQL Server allows a third setting where both indexes and views can be recommended together. This usually allows for more indexes over the recommended views that further improve the workload performance.

Results. For SSB and TPC-H datasets, the performance is 2.5× and 1.5× better respectively when using compressed workloads from GSum by using 5× lesser time for generating recommendations

Task ↓ Feature →	execution_time	output_size	#joins	#joins+output_size	execution_time+#joins	execution_time+output_size	all numeric
SSB index	1.1×	1.23×	1.23×	1.23×	1.1×	1×	1×
SSB views	1.08×	1.31×	1.58×	1.29×	1.08×	1.09×	1.09×
SSB indexes+views	1.13×	1.35×	1.85×	1.35×	1.13×	1.08×	1.08×

Table 4: Using Numeric Features: Slowdown compared to using all categorical and numeric features for compression

Task ↓ Feature →	function_call	table_reference	group_by	order_by
SSB index	1.56×	1.23×	1.36×	1.36×
SSB views	1.88×	1.58×	1.17×	1.41×
SSB indexes+views	3.2×	1.85×	1.95×	1.5×

Table 5: Using Categorical Features: Slowdown compared to using all categorical and numeric features for compression

as compared to when using the full workload. For TPC-H, we observed that even after 30 minutes, the recommendations produced using the full workload are no better than the recommendation generated after 9 minutes. We also observed the non-monotonic behavior of SQL Server tuning advisor when using the full workload. For some values of tuning time close to 30 mins, the indexes and recommended views increases the workload execution time, again highlighting that allowing more time does not necessarily improve the quality of recommendations, further evidencing the advantage of using GSum.

8.2 Impact of Features

Next, we perform an ablation study to see the impact of using a subset of features for the purpose of compression and compare the quality of the compressed workload using the same experimental setup as in the previous section. The metric we will use is slowdown in workload execution time using the indexes and views recommended using the compressed workload obtained with subset of features vs using all categorical and numeric features. Table 4 and Table 5 show the impact of using a subset of features for compression. The first observation is that all features (except `execution_time`) when used in isolation fail to identify a good compressed workload. `function_call` performs the worst because all queries in the SSB workload contain exactly one function call (SUM) which gives no useful information to the summarization task. The `table_reference` feature is able to extract 4 templates from the total 13 SSB templates, while the `order_by` and `group_by` features extract the largest number of templates from the input workload. We remark that even when using all categorical features together, the average slowdown for all three tasks is 1.4×. On the other hand, the numeric feature `execution_time` performs very well and is able to identify almost all templates. This is because even for queries with high syntactic similarity (ex. $Q1.1, Q1.2, Q1.3$), the execution time varies enough for the algorithm to identify that they originate from different templates. This demonstrates the importance of numeric features in the algorithm. Using `execution_time` with `output_size` further improves the compressed workload quality slightly. However, using `output_size` with `#joins` does not perform well. Our conclusions for TPC-H are similar and we defer the experiments to the full tech-report [10].

8.3 Microbenchmarks

In this section, we study the impact of different algorithms on the metrics, explore the effect of parameter γ on the objective function, study the impact of algorithmic optimization, and provide empirical evidence of GSum's scalability.

Experimental Setup. For all experiments in this section, we have implemented our summarization framework on top of the F1 database [33] within Google. It consists of two distinct modules: the featurization module that transforms and materializes the feature vectors of all queries that have been executed on the DBMS, and the summarization module that uses the materialized feature vectors and the input from the user to generate the workload summary. We use the `DataViz` dataset as the basis for our comparison, as it is a representative production workload. All experiments in this section are executed on a single machine running Ubuntu 16.04 with 60GB RAM and 12 cores.

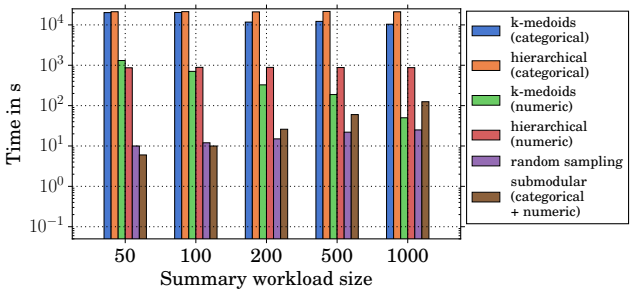
8.3.1 Algorithm Comparison. In the previous sections, we have compared against an industry system. However, there are several other algorithms that can be used in the context of workload summarization such as clustering techniques. To examine these, we have implemented k-medoids and hierarchical clustering (average linkage) in addition to random sampling and GSum.

Methodology. In this set of experiments, we use 5000 randomly chosen `DataViz` queries[4] and vary the summary workload between 50 and 1000. We compute the Euclidean distance over numeric features and the Jaccard distance over categorical features as distance function for clustering.

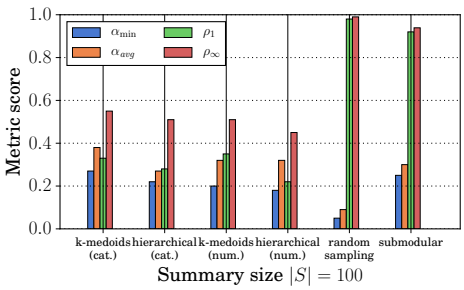
Results. Figure 2a shows the runtime of the different algorithms. Our first observation is that using clustering algorithms with categorical features is the most expensive choice (time $> 10,000$ seconds). This is because even though the feature vectors are materialized, finding the Jaccard distance takes $O(\|\mathbf{q}\|)$ time, as compared to $O(1)$ for the distance computation for single-valued numerical features. Further, in order to find the representative of each cluster, the number of operations required is quadratic in the cluster size, which amplifies the performance difference. Our second observation is that as the summary size increases, the execution time decreases for k-medoids, it increases for GSum, and stays approximately the same for hierarchical clustering and random sampling. Finally, we observe that compared to the two clustering algorithms whose performance depends heavily on the type of features used, the submodular algorithm is less sensitive to the chosen features since the submodular gain computation depends on a single query.

8.3.2 Representativity and Coverage. We now compare the metrics for the same workload and fix $d(\cdot)$ to be the input distribution.

[4] A small sample is chosen to make sure that clustering algorithms can finish running.



(a) Runtime



(b) Coverage and representativity scores (|S| = 100).

Figure 2: Runtime and metric scores for varying algorithms with |W| = 5000 on `DataViz`.

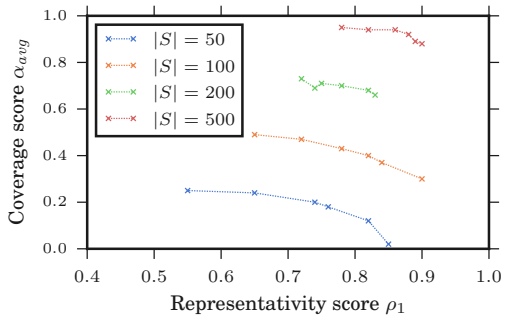


Figure 3: Trade-off between metrics; γ is the smoothing parameter that varies between 0 and 1 for each curve.

# processors →	1	2	3	4	5	6
runtime (min)	95	53	36	26	21	18
speedup obtained	1x	1.8x	2.7x	3.6x	4.5x	5.5x
coverage	1.0	1.0	1.0	1.0	1.0	1.0
representativity	0.91	0.88	0.86	0.85	0.85	0.85

Table 6: Runtime (in minutes) and metrics obtained using parallel processing.

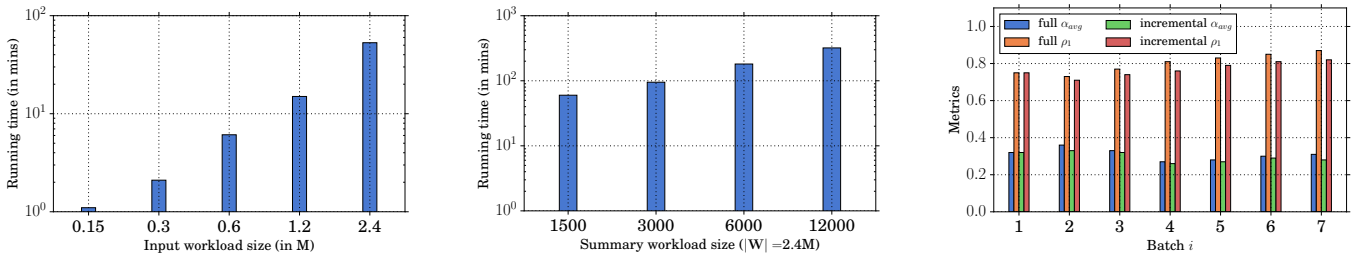
Results. Figure 2b shows the resulting coverage and representativity scores for all algorithms and summary size |S| = 100. As expected, random sampling has the best representativity score and the lowest coverage. All clustering algorithms have low representativity scores but achieve good coverage. This is not surprising because the cluster initialization step identifies *outlier* queries as cluster centers since they have the largest distance from most other queries. In comparison, GSum has a slightly lower coverage score than the best clustering algorithms, but performs significantly better in terms of representativity. Note that since $\gamma \to 0$, the submodular algorithm focuses on maximizing coverage first and the chosen input workload of 100 queries is not able to cover the active domain (as are the clustering techniques). Once GSum has reached the best possible coverage for a fixed γ, ρ improves even further. Given the faster running time of random sampling, it is natural to ask why GSum is better than random sampling. Note that random sampling provides poor coverage, since in the presence of skew, outlier queries are likely missed from the sample. These drawbacks were also identified by [6, 7]. Secondly, it is not clear how to incorporate user-specified constraints, e.g., the execution time of the summary is at most 1 hour, or a custom target distribution. Finally, the right sample size for random sampling is *unknown* a priori. Similar to the clustering methods, random sampling needs the sample size as the input. This means that we may need to run random sampling for all possible sample sizes which makes it a less attractive choice.

8.3.3 Trade-off between Metrics. In the next experiment, we empirically verify the impact of the γ parameter on DataViz workload and study how it can be used to trade-off between the two metrics.

Results. Figure 3 shows the trade-off between coverage and representativity (by controlling γ) for different summary sizes. Observe that as γ decreases, representativity starts dropping and coverage starts to increase. Note that the increase in coverage or decrease in representativity is not necessarily monotone due to the complex interaction between features. Since all features are uniformly weighted, features with more tokens tend to dominate both α and ρ. In order to balance that, we can set the weight for each feature inversely proportional to its active domain. However, in almost all our experiments, no feature dominated a different feature even when each feature had the same weight. In other words, the representativity score for each feature shows an empirical monotone behavior as γ changes. Table 7 shows the impact of γ on the full `DataViz` workload. With |W| = 512796 and |S| = 1000, we observe that as the value of γ decreases, ρ_1 increases at the expense of α_{avg}. Using the generated summary, we are able to identify recurring patterns in the input workload. Finally, we observe that the adjustments to γ depend on the skew of the workload and should be examined on a case-by-case basis.

8.3.4 Scalability and Parallel Computation. To benchmark scalability, we execute GSum on a single thread on a single machine and use all available categorical and numeric features. We use a workload consisting of 2.4M TPC-H and SSB queries.

Results. Figure 4a shows the runtime in minutes when the input workload size |W| varies and the summary size is fixed to |S| = $\sqrt{|W|}$. If |W| = 2.4 million queries, it takes GSum 53 minutes to execute compression. Since the compression algorithm is executed



(a) Runtime varying |W|, |S| = $\sqrt{|W|}$ **(b) Runtime varying |S|, |W| = 2.4M** **(c) Coverage and representativity scores**

Figure 4: Scalability (a),(b), and incremental computation (c) experiments on `DataViz`.

γ	f_1^s	f_1^p	f_2^p	f_4^p	f_5^p
10^0	0.75 | 1.0	0.67 | 1.0	0.57 | 1.0	0.9 | 1.0	0.5 | 1.0
10^{-3}	0.76 | 1.0	0.74 | 0.81	0.66 | 0.88	0.90 | 1.0	0.6 | 0.84
10^{-10}	0.8 | 1.0	0.81 | 0.68	0.73 | 0.75	0.90 | 1.0	0.69 | 0.70
10^{-15}	0.84 | 1.0	0.87 | 0.57	0.80 | 0.61	0.95 | 0.56	0.74 | 0.64
10^{-20}	0.88 | 1.0	0.98 | 0.32	0.94 | 0.26	0.98 | 0.30	0.94 | 0.18
10^{-25}	0.91 | 1.0	0.99 | 0.07	0.94 | 0.07	0.98 | 0.07	0.99 | 0.02

Table 7: Tradeoff between metrics for production workload. Each cell shows ρ_1 | α_{avg}

daily, this performance is acceptable in practice. As we will see later, using multiple cores can further improve the runtime. We generally observe a linear increase in runtime when increasing the workload size. Figure 4b shows how the choice of summary size impacts scalability. Here, we set |**W**| = 2.4 million queries and observe that creating a summary with |**S**| = 12000 takes roughly 5.2 hours. Analogous to the results observed when increasing |**W**|, we see a linear increase in runtime with an increase in summary size |**S**|. Table 6 shows the runtime and the impact of parallelization on the summary workload metrics. The first column shows the metrics when a single processor calculates the summary. As the number of processors increases, the speed-up obtained is near linear. We further observe no impact on coverage and representativity only drops marginally from 0.91 to 0.85.

8.3.5 Incremental Computation. Recall that incremental computation computes a local summary for each batch of input queries and merges them, instead of recomputing the summary from scratch. To test this behavior, we split `DataViz` into 7 different batches by partitioning the query log sequentially and summarize the workload incrementally, adding one batch at a time. We compare our incremental results to a from-scratch execution of GSum on the same subset of queries. For this experiment, we set |**S**| = $\sqrt{|W|}$.

Results. Figure 4c shows our findings. With an increase in the number of processed batches, we observe incremental computation providing marginally worse results than creating a summary from scratch. At the same time, we observe that the cumulative time spent in creating a summary from scratch ($\approx$ 480 minutes) is much larger than the cumulative time of merging summaries across batches ($\approx$ 60 minutes). After receiving batch i, the total input workload size has increased by a factor of i and the summary size by a factor of $\sqrt{i}$, increasing the runtime by a factor of $i^{3/2}$ as compared to $i-1^{\text{th}}$ batch. Using incremental computation, we can avoid this runtime increase as each batch is treated independently.

8.4 Discussion

Choice of Features. So far, we have observed how different knobs and configuration parameters impact various performance metrics and the output of GSum but we have not discussed how to choose them in practice. The answer to this question depends on (*i*) the chosen application, and (*ii*) how the chosen features interact with each other. While adding more features certainly provides more signals, it is not necessarily the case that it will guarantee a better result. For instance, the addition of more features will require a larger summary size to obtain better coverage, which in turn can possibly decrease representativity. For our experiments, the choice of features was driven mainly by iterating over the available choices and understanding their impact. Two heuristics that were useful to us are: (*i*) We found multiple production workloads that contained > 10^5 table references, most of which were temporary tables. For such workloads, including `table_reference` and attributes in WHERE clause as a feature is not a good idea; (*ii*) if two features f_1 and f_2 are highly correlated, then it suffices to include only one of these features. We found it beneficial to perform multiple iterations by generating multiple workloads with different features and then look at the compressed workload to understand how the summary has changed by using the visualization tools present in `DIAMetrics`. We found it useful to change feature weights and introduce weights for each token (initially uniform) and then change in each iteration to boost metric scores. For features with large domain, setting γ closer to 1 to dampen the effect of coverage also worked well. For tasks such as choosing platform alternatives to execute a workload, using categorical features is not necessary. Indeed, the execution performance of a workload has little do with SQL syntax and more to do with physical execution plans and operators available on different systems.

Choice of β. The right choice of β is determined by the application for which the compressed workload will be used. Applications that require outliers in the compressed workload (such as test workload generation and benchmarks for creating compliance benchmarks) set β closer to 1 whereas applications such as index recommendation require representative workloads. However, even within a specific application the optimal choice of β can change. As an example, consider two index recommendation algorithms *A* and *B*.

A may choose to recommend indexes that optimize the execution time of the most expensive queries first. In this case, the compressed workload must contain the most expensive queries in the workload. On the other hand, B may choose to recommend indexes that benefit the common-case queries in the workload but ignore the uncommon long running queries. This example demonstrates the need for a formal specification of the application context that can be integrated into the compression algorithm. Currently, we choose β by constructing multiple summaries and then test the performance to find the right threshold.

9 LIMITATIONS AND FUTURE WORK

We now discuss limitations of our work and ideas that will drive the agenda for this line of research.

Feature Engineering. One limitation of our framework is that it only looks at features at *query level* but does not incorporate workload level features such as contention between queries for resources. Choosing the right set of features for each applications is also a bottleneck. Currently, our features are hand-picked by domain experts such as application and database developers, database administrators and support personnel. Finding the right set of features requires an iterative analysis of the query logs to understand the variability in feature values. Since GSum is much faster than clustering algorithms, it allows us to build multiple benchmarks with different sets of features and β. As a part of future work, we plan to utilize machine learning techniques to identify the best features for a given application.

Transactional Workloads. In this work, we focus only on analytical workloads, ignoring the impact of data. For transactional workloads, the runtime of a query changes as the skew in the data changes. Thus, we need more sophisticated techniques to construct compressed workloads that take data updates into account.

Auto-tuning Knobs. Since the framework contains many knobs such as the choice of β, budget constraint and different application contexts, it is worth exploring how we can find the optimal configuration of the knobs for each application. Deep reinforcement learning has had considerable success in performing this task.

10 RELATED WORK

Most prior work focuses on maximizing coverage of information in summary workload as the primary optimization criteria while incorporating notions such as quality, efficiency as additional constraints. The property of representativity is more nuanced in comparison to coverage since it is highly dependent on the application context. In most cases, a representative summary minimizes the average distance from all items in the input workload [17, 32, 34], maximize mutual information between summary and input [31], maximizes saturated coverage [25] or maximizes coverage and diversity [9, 13, 26, 35, 36, 38]. In all these cases, the representative metric function is well behaved, i.e, it is monotonic and submodular by definition. Our problem setting departs from these works in our definition of representative where we would like the summary workload to mimic a target feature distribution. This makes the summarization problem more challenging. We note that our definition of representative has been proposed in previous work but has only been studied empirically as a quality metric whereas our algorithms are designed specifically to optimize for this metric.

Compressing Workloads. Compressing or summarizing SQL workloads has been studied by Chaudhuri et al. [6, 7]. In [7], the authors propose multiple summarization techniques including K-Medoids clustering, stratified/random sampling and all pairs query comparison. As the authors themselves note, random sampling ignores valuable information about statements in the workload and misses queries that do not appear often enough in the workload. [6] proposes a new SQL operator specifically for summarizing workloads but also suffer from the quadratic complexity. More recently, [15, 20] propose query structure based clustering methods for workload summarization but both proposed approaches are not scalable and rely only on the syntactic information in the text of the query. In contrast, our framework is more general in the sense that we also incorporate query execution statistics. Ettu [18, 19] presents a promising approach that clusters workload queries based on query syntax but it has restrictive assumptions with respect to defining query similarity which is based only on the subtree similarities of the query syntax. It is geared towards clustering queries written by humans with the purpose of identifying queries that may constitute a security attack. While the framework is scalable, it has restrictive assumptions with respect to defining query similarity which is based only on the subtree similarities of the query syntax. This is not true in our setting where most queries are generated by a pipeline of processes that increase the size of the query making it difficult to find out if two queries have the same 'intent'.

Counting Workload Patterns. An orthogonal but related problem to our setting is creating a compressed representation that allows for counting *patterns* in a query. The key idea explored in [23, 37] is to develop a maximum entropy model over feature values that then allows us to query pattern counts by simply computing the product of probabilities that each feature value is present.

11 CONCLUSION

In this paper, we propose a novel and tunable algorithm that allows us to summarize workloads for various application domains. We show that the proposed solution provides provable guarantees and solves the underlying problem efficiently by exploiting its submodular structure. Our solution supports parallel execution as well as incremental computation model and is thus highly scalable. We show through extensive experimental evaluation that our solution outperforms clustering algorithms and random sampling. We view this work as the first in an exciting research direction to develop automated solutions for workload monitoring at scale. We believe our solution can be extended to tackle interesting problems such as (but not limited to) resource prediction and production workload analysis that can be applied for a variety of (database) systems.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation grants CRII-1850348 and III-1910014. We are also grateful to the anonymous referees for their detailed and insightful comments that have greatly aided in improving this work.

REFERENCES

[1] 2020. SQL Server execution statistics. Retrieved November 17, 2020 from https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-query-stats-transact-sql?view=sql-server-ver15

[2] 2020. TPC-H Benchmark. Retrieved November 17, 2020 from http://www.tpc.org/tpch

[3] 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.

[4] Julien Aligon, Matteo Golfarelli, Patrick Marcel, Stefano Rizzi, and Elisa Turricchia. 2014. Similarity measures for OLAP sessions. *Knowledge and information systems* 39, 2 (2014), 463–489.

[5] Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. 2014. Streaming submodular maximization: Massive data summarization on the fly. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 671–680.

[6] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 488–499.

[7] Surajit Chaudhuri, Vivek Narasayya, and Prasanna Ganesan. 2003. -Primitives for Workload Summarization and Implications for SQL. In *Proceedings 2003 VLDB Conference*. Elsevier, 730–741.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[9] 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)*, Vol. 1. 1014–1022.

[10] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. (2020). Retrieved November 17, 2020 from https://arxiv.org/abs/2011.05549

[11] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2020. DIAMetrics: Benchmarking Query Engines at Scale. *Proceedings of the VLDB Endowment* 13, 12, 3285–3298.

[12] David J DeWitt. 1993. The Wisconsin Benchmark: Past, Present, and Future.

[13] Christoph F Eick, Nidal Zeidat, and Ricardo Vilalta. 2004. Using representative-based clustering for nearest neighbor dataset editing. In *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*. IEEE, 375–378.

[14] Uriel Feige. 1998. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)* 45, 4 (1998), 634–652.

[15] Shrainik Jain and Bill Howe. 2019. Query2Vec: NLP Meets Databases for Generalized Workload Analytics. *CIDR* (2019).

[16] Andreas Krause and Carlos Guestrin. 2005. *A note on the budgeted maximization of submodular functions*. Carnegie Mellon University. Center for Automated Learning and Discovery.

[17] Lun-Wei Ku, Yu-Ting Liang, and Hsin-Hsi Chen. 2006. Opinion extraction, summarization and tracking in news and blog corpora. In *Proceedings of AAAI*. 100–107.

[18] 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*. International World Wide Web Conferences Steering Committee, 463–466.

[19] Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2016. Summarizing Large Query Logs in Ettu. *arXiv preprint arXiv:1608.01013* (2016).

[20] G. Kul, D. T. A. Luong, T. Xie, V. Chandola, O. Kennedy, and S. Upadhyaya. 2018. Similarity Measures for SQL Query Clustering. *IEEE Transactions on Knowledge and Data Engineering* (2018), 1–1. https://doi.org/10.1109/TKDE.2018.2831214

[21] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 420–429.

[22] Stephen Macke, Yiming Zhang, Silu Huang, and Aditya Parameswaran. 2018. Adaptive sampling for rapidly matching histograms. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1262–1275.

[23] Michael Mampaey, Jilles Vreeken, and Nikolaj Tatti. 2012. Summarizing data succinctly with the most informative itemsets. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 16.

[24] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.

[25] 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*. ACM, 545–554.

[26] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, and Amin Karbasi. 2016. Fast Constrained Submodular Maximization: Personalized Data Summarization.. In *ICML*. 1358–1367.

[27] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrák, and Andreas Krause. 2015. Lazier Than Lazy Greedy.. In *AAAI*. 1812–1818.

[28] Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. 2013. Distributed submodular maximization: Identifying representative elements in massive data. In *Advances in Neural Information Processing Systems*. 2049–2057.

[29] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functionsâĂŤI. *Mathematical programming* 14, 1 (1978), 265–294.

[30] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB).

[31] Feng Pan, Wei Wang, Anthony KH Tung, and Jiong Yang. 2005. Finding representative set from massive data. In *Data Mining, Fifth IEEE International Conference on*. IEEE, 8–pp.

[32] 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*. ACM, 1163–1174.

[33] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, et al. 2018. F1 query: declarative querying at scale. *Proceedings of the VLDB Endowment* 11, 12, 1835–1848.

[34] Kamal Sarkar. 2009. Sentence clustering-based summarization of multiple text documents. *TECHNIA–International Journal of Computing Science and Communication Technologies* 2, 1 (2009), 325–335.

[35] Balaji Vasan Srinivasan and Ramani Duraiswami. 2009. Efficient subset selection via the kernelized Rényi distance. In *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, 1081–1088.

[36] Sebastian Tschiatschek, Rishabh K Iyer, Haochen Wei, and Jeff A Bilmes. 2014. Learning mixtures of submodular functions for image collection summarization. In *Advances in neural information processing systems*. 1413–1421.

[37] Ting Xie, Varun Chandola, and Oliver Kennedy. 2018. Query log compression for workload analytics. *Proceedings of the VLDB Endowment* 12, 3 (2018), 183–196.

[38] Jie Xu, Dmitri V Kalashnikov, and Sharad Mehrotra. 2014. Efficient summarization framework for multi-attribute uncertain data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 421–432.