DIAMetrics: Benchmarking Query Engines at Scale

Shaleen Deep
University of Wisconsin-Madison
shaleen@cs.wisc.edu

Anja Gruenheid
Google Inc.
anjag@google.com

Kruthi Nagaraj
Google Inc.
kruthi@google.com

Hiro Naito
Google Inc.
kiroa@google.com

Jeff Naughton
Google Inc.
naughton@google.com

Stratis Viglas
Google Inc.
sviglas@google.com

ABSTRACT

This paper introduces DIAMetrics: a novel framework for end-to-end benchmarking and performance monitoring of query engines. DIAMetrics consists of a number of components supporting tasks such as automated workload summarization, data anonymization, benchmark execution, monitoring, regression identification, and alerting. The architecture of DIAMetrics is highly modular and supports multiple systems by abstracting their implementation details and relying on common canonical formats and pluggable software drivers. The end result is a powerful unified framework that is capable of supporting every aspect of benchmarking production systems and workloads. DIAMetrics has been developed in Google and is being used to benchmark a number of internal query engines. In this paper, we give an overview of DIAMetrics and discuss its design and implementation. Furthermore, we provide details about its deployment and example use cases. Given the variety of supported systems and use cases within Google, we argue that its core concepts can be used more widely to enable comparative end-to-end benchmarking in other industrial environments.

PVLDB Reference Format:
S. Deep, A. Gruenheid, K. Nagaraj, H. Naito, J. Naughton, and S. Viglas. DIAMetrics: Benchmarking Query Engines at Scale. *PVLDB*, 13(12): 3285-3298, 2020.
DOI: https://doi.org/10.14778/3415478.3415551

1. INTRODUCTION

The data management landscape has drastically changed over the last few years. The majority of database systems are no longer manually tuned and optimized for a specific application by well-versed administrators, instead, they are designed to support a variety of applications. To support all of these applications, a multitude of data models, storage formats and query engines have transformed the data management landscape from standalone, specialized deployments to entire ecosystems. Workloads are now a combination of machine-generated queries for both transactional and analytical workloads as well as ad-hoc queries, varying by application and use case. At the same time, the performance expectations of customers remain the same: They expect the system to be tuned for optimal performance on their workloads. This is commonly achieved in a manual process that first identifies the most important customer use cases which are then used to build curated benchmarks. This process is not principled and may not yield comprehensive benchmarks valid for a long period of time due to (*a*) the dynamic nature of continuously changing production workloads; (*b*) a tight coupling between the workload and underlying query engine, preventing customers from identifying queries that are important across multiple engines; and (*c*) a general lack of understanding how query performance is affected by small changes to the end-to-end system. Given such complex company-internal ecosystems, it is increasingly difficult to determine for example how well a specific system is performing, how it compares to alternative systems for the same use case, or whether modifying one of its components will negatively impact other parts of the system. However, answering these questions in a principled manner is crucial to companies. DIAMetrics[1] is our answer to this problem setting: a benchmarking framework built at Google with the goals to (*a*) deliver a general solution that is capable of benchmarking end-to-end a variety of query engines; (*b*) support every step of the benchmarking life-cycle; and (*c*) provide insights with respect to system performance and efficiency. It is a one-stop tool for all benchmarking needs including complex tasks such as benchmark generation, execution, and result visualization.

Prior work. Benchmarking data management systems is certainly not new; from the early efforts of the Wisconsin benchmark [2, 3], to the development of the industry-standard TPC-H [27] and TPC-DS [28] decision support and transaction processing [26] benchmarks for relational systems, to benchmarks for object-oriented [6] or object-relational [7] systems, or to larger cloud-scale serving benchmarks [11] and their derivatives [15]. All these benchmarks have been studied extensively and the knowledge gained [4] has been used to modify them in various ways (e.g., [13]) or

[1]The name stems from the unit within Google DIAMetrics was originally developed to provide metrics for, DIA: Data Infrastructure and Analysis.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415551

deliver new benchmarks altogether that address the shortcomings of the existing ones (e.g., [5]).

The two common aspects of any of these benchmarks have always been that: (*a*) the benchmark workload is statically defined: even if there are randomly seeded data and query generators their outputs all conform to well-defined patterns, i.e., schemas, value distributions, and queries; and (*b*) the system being benchmarked assumes complete control of the entire data management stack, from hardware to software configuration and to manual tuning for optimal performance. Though existing benchmarking efforts certainly serve their purpose for standalone deployments, they are not indicative of production-level data management use cases of an entire ecosystem. There, a query engine does not have control of the data and storage formats; it is expected to evaluate a wide spectrum of queries, from single-point lookups, to real-time analytics, to extremely large machine-generated queries over a multitude of formats, or any mixture of the above; and it has little to no statistics about the input a priori to guide the system's optimizer and execution engine to deliver robust performance. Static benchmarks can act as a measuring stick, so to speak, but only for the use case they have been designed to address. In all other use cases a static benchmark is often not representative of the actual system load.

Problem motivation. Our work is motivated by the observation that benchmarking is a key necessity to determining the efficiency and usefulness of specific systems for specific tasks. Not having a way to benchmark a production system in a dynamic and often unpredictable environment may prove detrimental not only to the system developer but also to the user. The system developer spends an inordinate amount of time tuning the system for particular use cases and may not have clear insight into the larger-scale problems of the system. For instance, the developer may spend effort optimizing a particular operator at the micro-level, whereas a comprehensive benchmark would have shown that there would be greater benefit optimizing a different part of the system's processing pipeline. Or, the developer may decide that more computing resources are necessary for a particular workload, when a targeted benchmark could showcase that the majority of time is spent on non-compute-intensive execution fragments. The user, on the other hand, benefits from knowing the level of performance a system delivers. For example, if that performance, is suboptimal, she can provide the system developer with examples of this suboptimality, or even move to a different engine that may be better suited to the workload requirements.

Problem solution. Instead of focusing on a specific benchmark workload and using that as the means to test performance and efficiency, we argue that we need a benchmarking framework. That is, an architecture for benchmarking that is capable of generating indicative benchmark workloads over production deployments, executing them, and measuring a system's performance on that workload. Moreover, to avoid duplicate effort, the architecture should be independent of the query engine and it should rely on generic reusable components that can be instantiated with minimal effort for every system that is to be benchmarked. At the same time, the framework should provide the means to track the performance per indicative benchmark workload and use that historical information to measure improvement

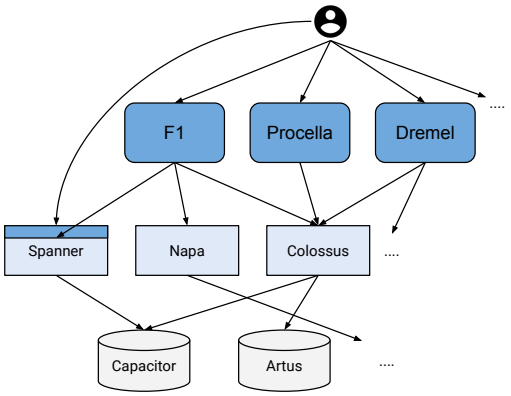


Figure 1: Part of the Google ecosystem.

over time. DIAMetrics provides all that functionality and has been used within Google to benchmark and reason about the end-to-end performance of internal query engines.

While DIAMetrics has only been used within Google, we posit that its architecture is powerful enough to support any query engine, as long as a minimal set of primitives are implemented. This is corroborated by the internal use of DIAMetrics: although Google is a single organization, it exhibits all the diversity characteristics we discussed earlier.

There are at least four internal query query engines, each designed for different use-cases: F1 [24, 25], Dremel [21], Spanner SQL [1], and Procella [8]. There exist specialized storage systems such as Mesa [19] and more generic ones such as Colossus [23] which are leveraged by different applications, supporting different storage formats. Figure 1 shows an overview of part of the Google-internal ecosystem of query engines and their dependencies. If every query engine would benchmark according to their own needs, there would be no accountability across engines and no way to determine which systems are useful for which use case. In contrast, DIAMetrics is specifically built to consolidate the benchmarking needs within Google, to provide an effective way to compare engines and at the same time provide means to improve them.

Contributions. In this paper, we present an overview of DIAMetrics, a novel extensible framework for engine-agnostic, repeatable benchmarking that is indicative of large-scale production performance.

Framework Architecture. We present the generic architecture of DIAMetrics in Section 2. We show a high-level description of its components and discuss how its design allows for extensions with little effort while seamlessly supporting its core functionalities. DIAMetrics departs from existing state-of-the-art benchmarking frameworks in the following ways: (*a*) Instead of focusing on a specific benchmark workload and using that as the means to test performance and efficiency, DIAMetrics provides an end-to-end benchmarking framework. The system is capable of generating indicative benchmark workloads over production deployments, executing them, and measuring a system's performance on that workload. (*b*) In order to avoid duplicate effort, DIAMetrics is query engine independent and relies on a handful of generic reusable components that can be instantiated with minimal effort for every system that is to be benchmarked. (*c*) DIAMetrics provides the means to

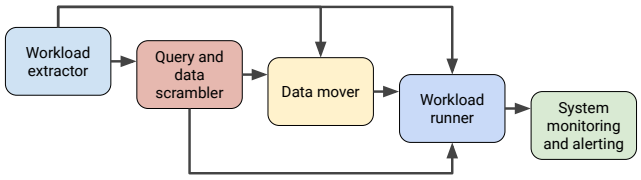


Figure 2: An overview of the DIAMetrics components.

track the performance per indicative benchmark workload and use that historical information to measure improvement over time.

Modular Components. We present the design and implementation of the components of our system in detail in Section 3. Each of the components is highly customizable to cater to customer specific requests while being general enough to handle a variety of use cases.

Use Cases. We discuss the deployment and varying use cases of the DIAMetrics framework within Google in Section 4. We discuss in detail the challenges faced by customer teams, how existing benchmarking solutions fail to capture their requirements and how DIAMetrics is able to help support their requirements. We outline some of the key research challenges faced and insights from our experience of working with our customers. Finally, we present related work in Section 5 and conclude and identify future work directions in Section 6.

2. OVERVIEW

DIAMetrics has two primary goals: (*a*) to be fully composable and rely on enhanced reusability in order to facilitate benchmarking at scale; and (*b*) to be able to benchmark and profile any internal system capable of evaluating queries and any customer workload of that system producing these queries. These goals are realized through two key notions:

- *Canonical exchange formats*: for extensive abstraction, whenever two components need to communicate, they do so through well-defined exchange formats that we term *canonical*. The formats are component-dependent, but the intuition is that the module that facilitates the transition from one format to the other can now be 'plugged in': if the component respects these formats all DIAMetrics pipelines remain functional.
- *System drivers*: to interact with all supported query engines, DIAMetrics employs drivers, i.e., modules that are capable of translating canonical workload representations into query processing requests for each supported query engine, gathering profiling metrics from the execution of that query on the query engine, and translating these metrics to the framework's own canonical profiling format for further processing.

2.1 Components

Using canonical exchange formats and system drivers, we construct the DIAMetrics pipeline as depicted in Figure 2. We use five components when benchmarking a variety of query engines and workloads: (*a*) the workload extractor, (*b*) the query and data scrambler, (*c*) the data mover, (*d*) the workload runner, (*e*) and the system monitoring and alerting component. An overview of these components follows.

Workload extractor. The first component extracts a representative workload from a 'live' production workload. It is often the case that customers experience a large mixture of prepared and/or ad-hoc queries, but they cannot readily identify which subset of queries is the best indicator for the needs they would place on a particular database engine. To help those users, DIAMetrics employs a workload extractor and summarizer, which is a feature-based way to 'mine' the query logs of a customer and extract a subset of queries that adequately represent the workload of the customer.

Data and query scrambler. A benchmark is not only the query workload, but also the data that queries process. The data, however, may be sensitive user data that cannot be used verbatim for benchmarking. The data scrambler anonymizes data in various ways (e.g., by masking data values, permuting column values in ways that do not alter the value distribution of the column but break any correlations between columns—to name but a few such ways). Once the data has been scrambled, the query scrambler implements a similar functionality at the query level by altering the queries so that they use the scrambled versions of the input data.

Data mover. In practice, users may want to benchmark is the efficiency of different storage back-ends, or different storage formats on the same back-end, and so on. The data mover undertakes the task of moving data between back-ends and formats so that the same query can be executed over multiple representations of its input.

Workload runner. The execution component within the DIAMetrics framework is the workload runner. In essence, it allows users to specify various combinations of workloads and systems to be benchmarked. For instance, we may want to run TPC-H on various query engines over various storage formats to see which storage format is the best option for which engine. The workload runner periodically schedules these runs over either specific production engines within our infrastructure, or by bringing up a new standalone and hermetic instance of an engine, executing the workload, and shutting the instance down after workload completion. The workload runner profiles the execution of each query and exports its metrics in an internal canonical format so that profiling metrics can be permanently stored for historical analysis; or in a generic format used by various monitoring systems within our infrastructure.

System monitoring and alerting. The last component of DIAMetrics is externally visible and responsible for presenting the consolidated performance reports to users. Additionally, its services can be used to alert interested parties for potential performance issues. The component is divided into a visualization framework, which brings up monitoring dashboards for the captured profiling metrics; and an alerting framework that, upon execution of a workload, compares its performance to historical data and triggers alerts whenever there is cause for concern, e.g., the performance of a query has degraded, or a query have started failing.

Each component described above can act as an entry point to the DIAMetrics framework. For instance, some user may not need to create a production workload since they may have one readily available through other means; or they want to use a standard benchmark like TPC-H. Alternatively, another user may only need to test different storage

back-ends for the same query workload, so they only need to use the data mover to generate multiple instances of the same workload. Essentially, the components described here are designed in a way that they can be mixed and matched specific to each benchmarking use case.

2.2 Workflow

Google-internal query engines are highly scalable and are capable of serving billions of queries per day from multiple customers, both internal and external. Each query served, along with a number of internal system-specific information, is logged for example in a distributed logging system running on Colossus, Google's file system [12, 17]. The log formats of each query engine are different, but there is a lot of common information between them stored in different ways. DIAMetrics builds on top of the idea that log entries in essence contain the same information, presented in different ways. Specifically, it uses a canonical representation of a query log, which treats a query as a combination of its query text and a number of features and their values that describe its profile. This representation is leveraged by DIAMetrics to drive the workload extraction and summarization process for custom benchmark generation. In essence, the workload extractor connects to the respective log system, extracting all relevant log entries that may contribute to the benchmark. The summarizer then uses that information to select an optimized subset of these logs that can be used as a custom, representative query workload.

However, these queries are often based on sensitive user data and are thus not available to any outside application or benchmarking system. To address this problem, users can choose to anonymize their data using DIAMetrics's data scrambler. The data scrambler scans the original customer data and applies various anonymization techniques on it in order to ensure no sensitive information is leaked to the benchmark dataset. In the simplest case, the data scrambler will arbitrarily permute the values of a column independently of other columns. Such permutation will ensure that per-column value distributions remain the same, but correlations across columns are broken, thereby reducing the likelihood of disclosure. Additionally, the scrambler may further obfuscate values by hashing them, by mapping them to a different domain, or by adding a small amount of noise so that the resulting dataset has approximately the same statistical properties but over different values; and so on. Finally, depending on the user's benchmarking use case, they might choose to compare their benchmark on a variety of storage layers or with different file formats. The data mover allows DIAMetrics to prepare the benchmark for execution on a variety of back-ends, allowing the user to get a comprehensive understanding of their execution patterns.

Once queries and data are stored in the correct place(s), independent of whether they are derived from the above pipeline or provided by the user, we deploy a workload runner that reads a set of configuration files describing the execution parameters and automatically runs the benchmarks on the specified systems with the specified execution constraints. Following the modular principles explained above, we allow users to write pluggable configurations, i.e., the same system configuration may be used for a set of different benchmarks. Note that by defining these configurations, the user determines the parameters of the benchmark. For example, they can decide to run the benchmark on a production server or in isolation by using different system setups. Similarly, they may choose to compare the generic execution of the TPC-H workload to a platform-optimized version to examine choices made by the query optimizer. In all of DIAMetrics's benchmark executions, we follow standard experimental procedure and allow users to execute the same workload and system configurations multiple times to provide realistic results.

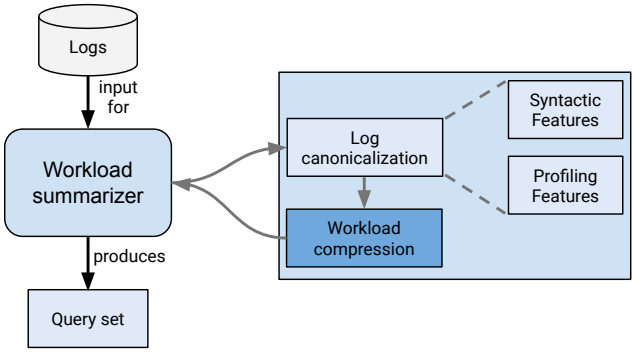


Figure 3: Overview of the workload summarizer.

Finally, the last step in the end-to-end workflow is the interpretation of the execution results. The monitoring component of DIAMetrics provides dashboards to the framework users that allow them to easily interpret the historic results of their benchmarks. It is triggered periodically and automatically updates its dashboards whenever new execution results have become available. If desired, users can furthermore use alerts to get notified when their execution patterns change significantly from previously observed or expected patterns. Our end-to-end framework for workload benchmarking has simplified and streamlined benchmarking within Google across different systems. It allows users to set up automatic benchmarking in a matter of minutes without needing to worry about the specific implementation details of executing repeatable benchmarks. In essence, DIAMetrics enables efficient and consistent benchmarking at scale within Google.

3. FRAMEWORK COMPONENTS

We next present the components of the DIAMetrics framework in detail. Each component can be thought of as a stand-alone facility, but it is their interaction that delivers an end-to-end solution.

3.1 Workload extractor

One of the main problems when benchmarking any system is defining the benchmark that appropriately evaluates the system. Indeed, a single system may experience multiple types of workload at different times. For instance, the majority of queries may be long-running resource-intensive analytics queries; or they may be single point lookup queries for record retrieval; or anything in between when a user is using a database in exploratory mode. These can be created by a single or multiple customer(s) using the system for different types of applications. Traditionally, system deployments have been tailored for different application needs, each deployment being optimized for the types of queries it

is expected to evaluate. With the move to distributed, large-scale, federated, and cloud-based deployments, however, the advantage of fully controlling the architecture of a system is no longer given. A query engine is treated simply as an end-point and is expected to be able to process user queries with little to no optimization from the user. It is therefore a requirement for the query engine providers to cater to different needs at the same time, which makes it imperative to have a way to gauge the system's performance on the user's workload. Whereas for relational systems we have had benchmarks like TPC-H, TPC-C, or TPC-DS, mostly stemming from the general division of relational workloads into OLTP and OLAP, there are no representative benchmarks for these (user-specific) mixed workloads.

To process not only standardized benchmarks but also user-specific benchmarks, we developed techniques that compress a user's workload into a small set of representative queries that can then be used as a benchmark workload [14]. Our framework for workload extraction and summarization roughly undertakes the following tasks:

Log canonicalization. To create a user-specific benchmark, log entries are first extracted and transformed to a canonical representation that contains a set of *features* necessary to drive summarization. Features can be anything that characterizes the specifics of a query that are deemed useful for benchmark creation. In DIAMetrics, we support two types of features: syntactic and profiling features. Syntactic features can be extracted by parsing the query, e.g., the number of joins in the query statements or the aggregate functions used in the query. Profiling features on the other hand may encompass characteristics such as query latency, CPU usage or amount of data read/written to disk.

Workload summarization. Once the workload features have been extracted, we can leverage them to identify a subset of queries for benchmarking this workload. The choice of queries in the subset is driven by two metrics: *representativity* and *coverage*. Representativity determines how closely the distribution of features in the subset matches the original workload. In contrast, coverage determines how well the features in the subset cover the features observed in the original workload. To an extent, coverage describes the completeness of the benchmark. During workload summarization, we optimize the selection of queries according to these metrics and greedily pick the benchmark queries which can then be used for realistic production benchmarking.

3.1.1 Summarization algorithm

In essence, the summarization problem described above is an optimization problem. To solve it, we first define the metrics that drive the optimization, representativity and coverage as follows. Consider a feature f and let $\mathsf{dom}(W, f)$ (respectively $\mathsf{dom}(S, f)$) denote the domain of f in the input workload W (respectively summary workload S). Coverage α_f is defined as the fraction of domain values covered by the compressed workload for feature f, i.e, $\alpha_f = |\mathsf{dom}(S, f)|/|\mathsf{dom}(W, f)|$ and the overall coverage α is the average of α_f over all features f. Observe that α and α_f are always in $[0, 1]$ where a score of 1 means that the coverage is perfect. At the same time, workload W induces a discrete distribution $p_W(\cdot)$ over the tokens present in the features of the queries in the workload. Let $\mathsf{m}_W(t, f)$ (multiplicity of t in W) denote the number of times a domain value t of feature f appears in the entire workload W. Then, for any domain value t of some feature f,

$$p_W(t) = \frac{\mathsf{m}_W(t, f)}{\sum_f \sum_{d \in \mathsf{dom}(W,f)} \mathsf{m}_W(d, f)}$$

In other words, $p_W(t)$ denotes the probability of selecting token t if we choose a token from W uniformly at random. The workload summary S will induce a distribution $p_S(\cdot)$; the representativity metric then measures the distance between p_S and p_W.

Analogous to coverage, the representativity score is in $[0, 1]$ where a score of 1 signifies that the summary workload is perfectly representative. For numeric features, we discretize the entire space by normalizing the numerical values into histogram buckets within the range $[B]$ for some predefined constant B which becomes $\mathsf{dom}(W, f)$ for the numerical feature.

Key algorithmic ideas. Our goal is to generate the summarized workload $S \subseteq W$ while maximizing coverage and representativity, subject to budgetary constraints (such as $|W| = k$). We accomplish this by employing a novel instantiation of a submodular maximization algorithm that minimizes the KL divergence between input workload distribution $p(\cdot)$ and the compressed workload target distribution $d(\cdot)$. Note that $d(\cdot)$ can be arbitrary and can be specified by the user. The case when $d(\cdot)$ is the same as input distribution, it is simply a specific instantiation of the algorithm. Our algorithm is easily parallelizable and supports incremental computation. A full description of the algorithm is beyond the scope of this paper and we refer the interested reader to [14] for more details.

Summarization desiderata. To accommodate different types of benchmarks, the following desiderata describe the design space of workload summarization as leveraged in practice by DIAMetrics.

High Coverage and Representativity. High coverage is desirable to ensure that long-tail feature values are included in the summary workload whereas high representativity ensures 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.

Scalability. Efficient computation of the summary workload is a key requirement for any framework to be deployed in practice. Ideally, the summarization algorithm must compute the summary workload fast and scale effectively to large input workloads.

Customizability. Note that the two metrics may well be competing: high representativity may imply low coverage if the original workload is skewed. The non-skewed subset of the original workload will most likely contain outliers in terms of our distance function, i.e., queries with low representativity. Thus, we allow users to control the trade-off between these two metrics.

Constraints. Users may want to specify constraints on some property of the summary workload like the number of queries that form the benchmark; expected execution duration; or input/output size restrictions that the benchmark is executed under.

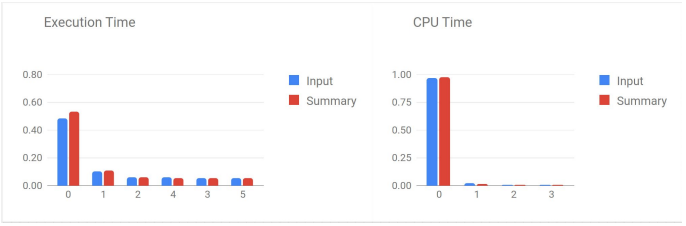
Table 1: Summarization experiments.

Algorithm ↓ | $\|S\|$ →	50	100	200	500	1000
DIAMetrics	6	10	26	60	125
random sampling	10	12	15	22	25
k-medoids (syntax)	20177	20315	11739	12221	10396
k-medoids (profile)	1311	706	327	188	50
hierarchical (syntax)	21194	21283	21020	21574	21123
hierarchical (profile)	866	888	886	879	873

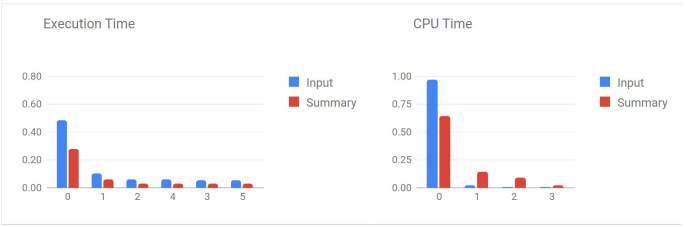
(a) Runtime (in s), $|W| = 5000$

Algorithm ↓ | Metrics →	coverage	representativity
DIAMetrics	0.30	0.94
random sampling	0.09	0.99
k-medoids (syntax)	0.38	0.55
k-medoids (profile)	0.31	0.51
hierarchical (syntax)	0.27	0.51
hierarchical (profile)	0.32	0.45

(b) Metrics, $|W| = 5000$ and $|S| = 100$.



(a) Representativity



(b) Coverage

Figure 4: Dashboards for visualizing workload distributions of varying features.

3.1.2 *Summarization experiments*

We next highlight some key experimental results comparing our algorithm to state-of-the-art summarization algorithms, namely K-Medoids, hierarchical clustering [9, 10] and random sampling. Table 1a shows the runtime for all algorithms on a small sample of 5000 production queries. Our algorithm is only marginally more expensive than random sampling and up to two orders of magnitude faster than clustering algorithms are in $\Omega(N^2)$. Further experiments have shown that our technique can easily handle millions of queries as input and is thus highly scalable. This is a key requirement for our use case, allowing users to customize their benchmarks iteratively. Going back to Table 1a, we observe a difference in runtime when using syntactic vs. profile features for clustering algorithms. In essence, distance computation for profile features is cheaper than computing the Jaccard distance between sets obtained from syntactic features. Random sampling and our submodular algorithm do not suffer from this drawback since they do not use a distance metric for comparing two queries. Table 1b compares the trade-off between coverage and representativity for summary size $|S| = 100$. While clustering algorithms achieve good coverage but low representativity, random sampling results in low coverage but high representativity. Our algorithm is able to maximize for both and can achieve a better cumulative result than any alternative method.

To understand how well an input workload matches a summary workload, we leverage the same visualization capabilities of DIAMetrics that are described in detail in Section 3.5. Figure 4a shows the dashboard when we maximize representativity for profile features *execution time* and *CPU time* of a production workload while Figure 4b shows the histograms when we maximize coverage instead of representativity. The x-axis shows the domain of a particular feature (denoted $0, 1, 2 \ldots$ to anonymize) and the y-axis denotes the fraction of queries that contain the corresponding feature value. Comparing these two visualizations, we observe that the summary in Figure 4a more closely represents the input while maximizing coverage requires picking queries to maximize domain coverage which in turn manifests as low representativity leading to a wider gap in the histogram bars. A user can utilize these dashboards to adjust the input configuration, modifying the targeted metrics until the desired outcome is achieved.

3.2 Data and query scrambler

In addition to finding a representative set of queries to execute for benchmarking, DIAMetrics also needs to ensure that the data it is using for these benchmarks is representative. The choice of dataset will drive storage and query processing decisions depending on the query patterns being executed, the storage back-end, the complexity of the data, and the data value distributions, to name but a few factors.

The data scrambler is a step towards addressing the problem of representative data generation, as it provides a simple and efficient way to use production data for query benchmarking. The intent is to have a facility that would allow one to quickly sanitize a representative production dataset and use actual production queries over the sanitized version for performance benchmarking. Once workload summarization identifies the queries that are representative of a workload, we can use the inputs these queries process to snapshot the production data and use that snapshot to build a version of the input data to be used for benchmarking. This is not always straightforward, mainly because production data may contain fields, values and correlations between them that are sensitive and should not be used for benchmarking purposes. In the data scrambler we solve that problem by breaking correlations between values; by protecting data through hashing their values to obfuscate them; and by adding small amounts of noise to the data so that their distributions are not significantly altered, whereas their original values airare. While the scrambler does not provide formal guarantees with respect to privacy or non-disclosure, it has been found to alter the input data in a reasonable way that might be good enough for performance benchmarking in a secure industry setting. No formal guarantees notwithstanding, the scrambler is also extremely customizable and may well provide these guarantees implicitly if configured properly by data owners.

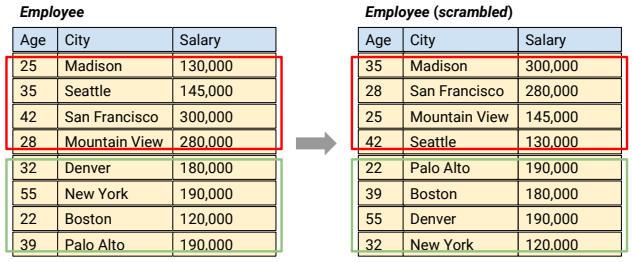


Figure 5: The data scrambler in action: the input table is split into chunks and the values in each column of each chunk are individually permuted.

3.2.1 *Scrambling techniques*

The data scrambler, at its core, performs a column-wise, row-bounded, permutation of data values across the fields of a table. That is, a table is first split into chunks each with the same number of rows. For each column of each row in the chunk, its values are permuted arbitrarily and independently of one another. The chunks are then collated to produce a new version of the original table. The scrambler is shown in Figure 5, where we show a potential scrambling of an example *Employee* table. In the example, we assume a chunk-size of four rows. The values of the columns within a chunk will be independently permuted. We call the mapping between old and new positions of the values of a single column a permutation order for that column. For instance, the Age column in the first chunk has a different permutation order than the Age column in the second chunk: the permutation order in the first chunk is $[1 \rightarrow 3, 2 \rightarrow 1, 3 \rightarrow 4, 4 \rightarrow 1]$, whereas the permutation order in the second chunk is $[1 \rightarrow 4, 2 \rightarrow 3, 3 \rightarrow 1, 4 \rightarrow 2]$. Moreover, the permutation order across columns of the same chunk is not the same. For the City column of the first chunk, its permutation order is $[\rightarrow 1, 2 \rightarrow 4, 3 \rightarrow 2, 4 \rightarrow 3]$, which, again, is different than the permutation order of the Age column for the same chunk.

Key algorithmic ideas. There are two advantages to the approach we follow. The first one is that the value distributions per column in the scrambled table will remain exactly the same as those in the original table. The values themselves will not change; they will merely appear in a different order. The second advantage is that any correlations between columns will be broken, as the values of the two columns will be permuted independently of one another. In the example of Figure 5, if we knew that there was only one single 22-year old employee in Boston we could have identified their salary; whereas in the scrambled version that correlation has not been preserved, therefore it is not possible to make that association. Note that depending on the use case for the benchmarked dataset, this is not necessarily a good idea, as these correlations may be important. If that is the case, the scrambler can be configured so that groups of columns have the same permutation order and correlations are preserved in the output, while it can still guarantee the property of the correlations being split between a group of correlated columns and the rest of the columns of the row. In essence, the scrambler works as follows:

1. Split the input into chunks of some pre-determined number of records.
2. Within each chunk, we scan its records maintaining the current path with each traversal within the record.
   - Whenever we come across a value, if the path that this value corresponds to is to be scrambled, we enumerate the value by storing it in a map of the appropriate type. The map is a mapping from the path expression to a pair of value and frequency count (i.e., if we come across a value for the second time we only increment its counter).
   - If the value is not to be scrambled, we either ignore or obfuscate it, depending on configuration.
3. Once the chunk is exhausted and all its values have been enumerated, the chunk is scanned again and scrambled. For every path to be scrambled and whenever we come across a value in that path, we draw a random number to land in the enumeration map; that will be the new value of the field. We update the map accordingly, decrementing the frequency for every value we use and removing exhausted values.
4. After all records in the chunk have been exhausted, we output the chunk and move on to the next one.

Note that the process described above works for both sequential and parallel scrambling, since it is trivial to process each chunk in parallel. The sequential and parallel implementations will produce identical results as long as the sizes of the chunks are the same and the same random number generators per chunk are used. Additionally, the scrambler can be used to sample from an input, or expand it without changing its statistical properties. While scrambling each chunk, and when processing a single row, we can either sample the row with some probability; or re-scramble it a number of times to reach a desired output size.

Scrambling desiderata. The approach outlined above works for flat data, in addition to arbitrarily nested datasets. For instance, consider a table with an array field. We can apply the permutation-based approach by collapsing all values across all array fields of a chunk of rows, and then performing the permutation. An additional advantage here is that the length distribution of the array fields will not change, so the output will still be a representative version of the input. In addition to permuting data values, it is possible to add extra functionality across three other dimensions:

- Leaving values unchanged, perhaps because the fields containing them are insignificant to a particular workload and are therefore irrelevant to scrambling.
- Adding small amounts of noise to values, potentially in addition to permuting them, in a way that does not significantly alter their distribution.
- Obfuscating input values by hashing them, again potentially in addition to permuting them, since the values themselves are significant and should not appear in the scrambled output verbatim.

We identify the parts of an input to apply specific transformations on through path expressions from root to leaf. Working with path expressions is necessary to account for the complexity of production schemas, which have array fields, or nested relations at arbitrary depths. For instance, if we blacklist path `/root/to/value/left/unchanged/`, then whenever we come across values that belong to that path we

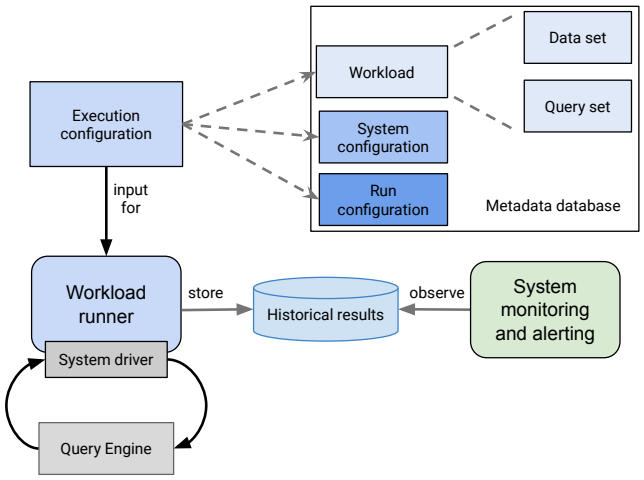


Figure 6: Overview of the workload runner's components.

can leave them unchanged. Likewise, we can use a path expression to designate that the values of other paths will have small amounts of noise added to them, or obfuscated.

3.3 Data mover

The data mover acts as an intermediary between formats. The intuition behind the data mover is to give DIAMETRICS the ability to generate multiple benchmarks from the same workload by converting the same input source to fit different storage back-ends. This is far from trivial as there are multiple aspects to take into account when designing data transformation mechanisms. First, different storage formats imply different schema definitions, which in turn implies potential type conflicts. For instance, the target format supports dates only as milliseconds in the epoch, whereas the format we want to move data from stores these dates as strings; thus, the data mover needs to apply the transformation from one format to another. Second, some input format may have additional statistical information embedded into its sources, or even value indexes incorporated. If that is the case, the data mover deploys a best-effort mechanism to replicate the original input structure with as many auxiliary structures transferred to the output as possible. Other information that the data mover attempts to preserve is sharding information, e.g., the number of shards and the partitioning scheme; input storage properties like the input being sorted; data definition properties like functional dependencies if these are supported by the target storage back-end; and, in general, any optimizations that are present in the input dataset and might affect the performance of the storage back-end if they are not preserved. Once the requested data movements have taken place, the input workload will be rewritten so that instead of using the original input sources, it uses the newly generated data sources.

3.4 Workload runner

The benchmark execution component of DIAMETRICS is the workload runner. The runner accepts multiple *execution configurations* as input, with each execution configuration containing the following four elements: (a) a number of systems and their configurations to use for benchmarking; (b) a number of benchmark configurations; (c) a number of workloads to benchmark; (d) a number of alert configurations to trigger if there are any issues detected when running a benchmark. The workload runner will then run each execution configuration by deploying every workload over every benchmark configuration and over every system configuration. We next describe each of these configurations in detail:

System configuration. This configuration encapsulates the endpoint of a query engine in the Google infrastructure. This may be a production instance of a query engine, or a configuration for a new hermetic instance. In the latter case, the workload runner will bring up the instance before proceeding and tear it down upon completion of the benchmark. In both cases, if there are any system-specific options to pass along to the endpoint, they are embedded into this configuration and passed on to the system at benchmark-time.

Benchmark configuration. This configuration describes various benchmark-specific parameters like the number of iterations for each query, a location to store the profiling metrics for the benchmarked workloads, or query-specific options to apply before sending a benchmark query for execution.

Workload configuration. The workload is effectively a set of self-contained queries that every system referred to in the overall execution configuration is assumed to be capable of evaluating. Each query has certain identifying information, along with a set of potential parameter values that are instantiated at query execution time. Parameters allow for a single query template to result in multiple concrete queries at run-time, which, in some cases, drastically reduces the length of a workload configuration. Workloads may be further divided in logical entities termed *query sets* if the user of the system wants to have a high-level grouping of the benchmark queries.

Alert configuration. The alert configuration places conditions on the metrics captured during query execution and, if that condition is true after workload completion, the runner will fire off the alert. We will present the alerting framework in more detail in Section 3.5.

All of these configurations are uniquely identifiable in the system as well as reusable. For instance, one may have the same system configuration referred to from multiple execution configurations; or the same workload executed on multiple systems. Configurations are stored in the *metadata database* of DIAMETRICS. When the workload runner is requested to process an execution configuration, it determines the specifics of that configuration in the metadata database, and retrieves all required system, benchmark, workload, and alert configurations it refers to. Next, the runner will deploy an intermediate orchestrator to configure systems and benchmark, run queries, save their profiling metrics, and evaluate any potential alerts as shown in Figure 6. While the default execution is sequential, the workload runner also offers various degrees of parallelism at any point of a configuration. For example, the runner may send the same workload of all targeted systems for execution in parallel; but within a system it can be configured to issue queries sequentially for better isolation.

Key framework ideas. The workload runner connects to a variety of query engines through *system drivers*. That is, given a query, the runner needs to issue that query against a target system. It will therefore need to use a system-specific client to generate a processing request for that system. Once the query is evaluated, it will return various profiling information for its execution. That profiling information may contain irrelevant information for further processing within DIAMETRICS, but the system driver will distill it into a canonical representation that is common across all supported systems, aiming to support as many elements of that canonical representation as possible. Captured profiling information includes, but is not limited to: latency, resource consumption, scheduling and planning time, time the query spent queuing for execution slots, number of execution fragments, number of input bytes, number of output bytes, number of input tables, number of physical input files, number of remote procedure calls generated during query execution, types of operators in the query and their count. In general, there is a multitude of information each system may export; the driver of the system captures that information and, where applicable, converts it into the canonical profiling format.

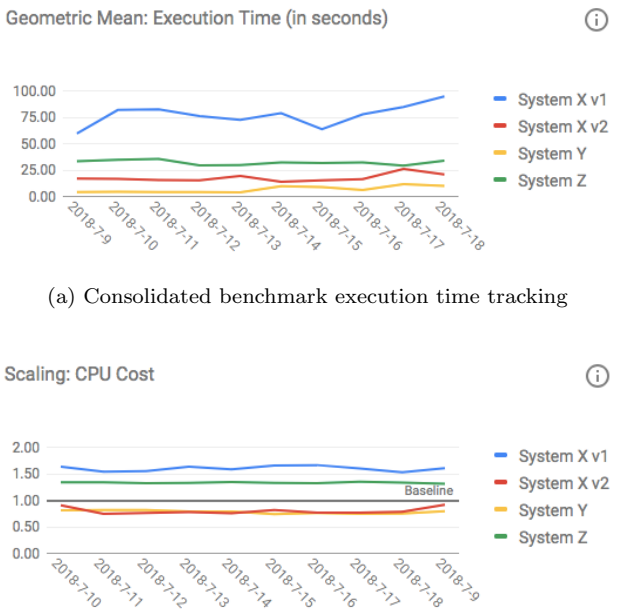
Finally, the benchmarking results are persistently stored for system monitoring and historical analysis, and additionally to drive the alerting framework. To better aid with the analysis, the results of a benchmark execution not only contain the profiling metrics, but also all references to all parameters of the execution like the system configuration that was used, the benchmark configuration, the queries themselves and the values of any parameters expanded during query set up. The results themselves can be analyzed using any of the database engines DIAMETRICS itself benchmarks: they have a well-defined schema and are stored in a format accessible through any of Google's query engines.

3.5 System monitoring and alerting

After the workload runner has executed a workload, we export the output into DIAMETRICS-specific logs. These logs are then used to (*a*) allowing users to monitor benchmarking performance through result visualization, and (*b*) automatically monitor performance regressions and issue alerts. System monitoring is a core objective of DIAMETRICS, as it helps users to track the performance of the system. At the same time, it is useful to system developers to track incremental changes of the same workload, visualizing whether changes to the codebase improved a system's performance. Alerting, on the other hand, can signal to developers and workload owners if there exists a significant performance degradation in the most recent snapshot of the system, if there were any failures, and so on.

3.5.1 Performance monitoring

DIAMETRICS automatically retrieves the logs that the workload runner generates and uses the logged profiling metrics for visualization. Specifically, we use static dashboards to visualize the workload execution over time in terms of essential statistics such as latency, CPU time, spilled bytes and any other metric that is captured by the executed system and deemed important by the client. An example of performance tracking of execution time using the same benchmark for various systems is shown in Figure 7a. Here, we observe the execution of three different systems, one of which is executed with two different system settings. Their performance is tracked over a timespan of ten days and the average execution time is reported. We primarily monitor aggregate metrics like the geometric mean of Figure 7a) for latency, but this is not the only capability of the monitoring substrate of DIAMETRICS. The dashboard visualizations allow for an intuitive comparison of the different systems and, at the very least, signal (*a*) how stable a system is, and (*b*) how a system fares in comparison to alternative deployments of the same system, or other internal query engines.



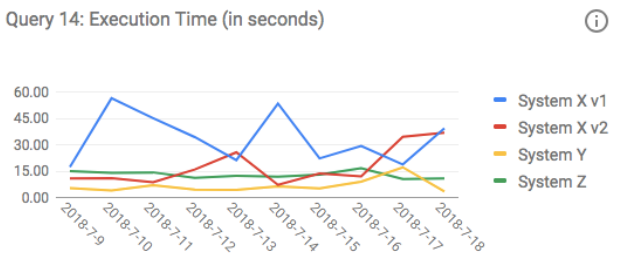
(a) Consolidated benchmark execution time tracking

(b) Scaling comparison

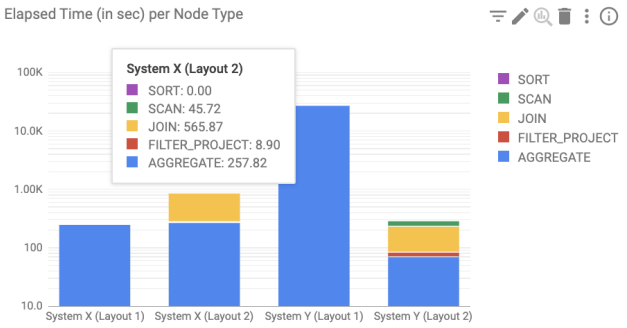
Figure 7: Example dashboards for per-benchmark performance monitoring.

In addition to simple tracking dashboards, we also developed more insightful dashboards, such as dashboards that look at the scalability of a system. Recall from Section 3.1 and Section 3.2 that we can compress workloads and input data to different sizes. We can therefore use these different samples of the same workload to observe the relative difference of performance as we scale the input size. In Figure 7b we show such a dashboard for CPU time. Here, we observe that the same three systems scale differently in terms of their CPU time. Ideally, we would want to scale a system (sub-)linearly. If the ratio of the scaling factors is equivalent to the ratio of the CPU time, the scalability plot will center around one. If the ratio is below one, the system scales sublinearly. In this specific case, we observe that only 'System X v2' and 'System Y' show the desirable scaling performance and are able to alert users of 'System X v1' accordingly.

Finally, DIAMETRICS is able to show the per-query performance breakdown for each query in the workload. This is essential information, as shown in Figure 8a, where the execution time for this specific query is noisy for some of the systems. A historical performance tracking framework allows system designers to see these kind of inconsistencies and eventually correct them. Note that these inconsistencies are only apparent if we drill down into the per-query execu-



(a) Query-specific latency tracking



(b) Operation breakdown computation

Figure 8: Examples dashboards for per-query benchmark performance monitoring.

tion of the workload and may be cushioned when looking at the overall performance of a system.

Exploring this direction even further, DIAMetrics is able to break down per query performance by query operation, in this case using a SQL engine, too. In Figure 8b, we visualize two different systems that run the same query on two different data layouts to compare performance. We observe that System X performs comparatively better on data layout 1 while System Y is preferable on data layout 2. Furthermore, these different data layouts lead to different utilization of SQL operators: While data layout 1 results in query execution being dominated by aggregation operations, data layout 2 results in join operations also taking a comparative fraction of query execution. This type of information is invaluable when evaluating different systems as well as storage layers, optimization mechanisms, and so on.

3.5.2 Alerts

DIAMetrics implements an alerting framework that is inherently coupled to system monitoring. At an abstract level, the alerting framework uses the metrics produced by the latest run of an execution configuration and compares them to their historical behavior to identify potential regressions. Alert triggers are configured at a per-metric level. For instance, the most frequently used alert compares the latest value of a metric with the aggregate value of the metric over the last time period; if the difference between the last measured value and the historical aggregate exceeds some user-defined threshold, then the alert is triggered. Another type of alert only looks at the latest measured value and checks that is within specific bounds—usually set up to simulate the service-level objectives of the system on the benchmark being evaluated. Upon an alert being triggered, the system generates a record containing the alert itself, its triggering conditions, and the pointers to the queries that caused the alert to be triggered in the first place so the recipient of the alert can quickly identify the offending query and triage the issue. The alert record is then delivered to various interested parties either immediately, by being embedded in an email to the team that owns the benchmark, or by being filed as an issue for immediate attention. Additionally, the alert record itself is also stored so that other consumers of the alert can pick it up asynchronously by monitoring the alert log. At the same time, and in addition to the framework's own alert evaluation primitives, DIAMetrics exports in an internal monitoring format all the metrics gathered during benchmarking to external production monitoring systems for other teams. Production monitoring teams can then use these metrics to set their own monitoring. One added provisioning of DIAMetrics is that the historical performance data can be used for further analysis like statistical processing, or identifying correlations between metrics, or any other type of more complicated regression analysis. Overall, alerts are an important facility of DIAMetrics in order to identify any deviations from the expected norm and focus the attention of the development, production, and benchmark-owning teams to problematic situations that can be exemplified through a handful of queries exhibiting the problem.

4. DEPLOYMENT & LESSONS LEARNED

DIAMetrics is capable of benchmarking all production-ready generally-available SQL engines within Google as well as selected internal, non-SQL engines. With every workload run it evaluates thousands of queries across multiple systems, gathering and storing performance metrics for immediate and later analysis. It has enabled query engine production teams to set performance goals and know where they stand with respect to alternative systems, while it has also helped teams to migrate between query engines by identifying problematic cases and setting up roadmaps for the migration. In what follows, we will sketch out the most widely encountered use cases for DIAMetrics, some of them expected, but others being off the beaten track with respect to its original design.

4.1 Benchmarking

The core idea behind DIAMetrics is to provide users with an intuitive means to benchmark their systems. To that effect, we have developed DIAMetrics to be modular, customizing the benchmarking experience to the team's use case. Tooling for daily benchmarking is currently used by a variety of Google query engines such as F1, Procella, or Dremel. Comparing the performance of these engines but also being able to reason about the performance of each system has become a crucial part of evaluating the success of these engines. When using DIAMetrics for benchmarking within Google, we have specifically encountered the following use cases:

Workload characterization. One of the highest barriers of entry to DIAMetrics has been that teams often do not have a clear grasp on what their query workload looks like; or, if they know all the query patterns that they employ, they have no clear way to identify which patterns are more

important than others. In some cases, a simple frequency-based clustering of queries is enough to identify a rough approximation of the workload; but in the majority of cases that is not possible. Workload summarization is a powerful method to compress a workload into a benchmark, providing guarantees about the output in terms of its representativity and coverage. Moreover, the summarizer is capable of delivering the benchmark workload under specific constraints in terms of the profile of the extracted benchmark. Having such a facility in place allows teams to quickly turn their workload into a benchmark with minimal manual log mining and configuration on their part. As such, it allows these teams to focus on their own mission without having to internally benchmark the performance of their query engine.

Workload optimization. DIAMetrics is capable of producing tracking dashboards for various combinations of system configurations over different versions of the same workload. As a result, workload owners can measure the impact of these configurations on their workload. Internal teams have used DIAMetrics to test their optimizer's performance by comparing out-of-the-box and manually optimized versions of a workload; or to compare the performance of different storage configurations; or to measure the impact of a feature upon a workload by comparing system performance with the feature being turned on or off. Having the ability to do this with minimal configuration and over production workloads in addition to standard benchmarks, improves the confidence of development teams in their decisions and not only optimizes specific use cases, but also reduces the management and financial cost of deploying these workloads. Moreover, being able to do so on a compressed version of a workload that is representative of the original one with robustness guarantees allows the data owners to quickly perform these optimizations at scale and extrapolate from the performance of the compressed workload to the expected performance of the system on the actual workload.

Performance accountability. Tracking the historical performance of a query engine on a workload is a two-way street. Not only is it useful for a query engine to track how well it performs on a specific workload, it also works in the inverse direction: the developers of an application using a query engine can hold the engine accountable for the performance it delivers on their application. DIAMetrics can be used to deliver compliance benchmarks for service level objectives between data owners and query engine users, and the production team of the query engine. Such accountability bridges the gap between teams and leads to a common understanding of the expected level of performance. Whenever there is any cause for concern, tracking dashboards act as the proof for that concern and aid towards its resolution.

Data anonymization. The DIAMetrics framework enables the use of actual production-like data for benchmarking, thus eschewing the need to come up with synthetic data generators or not being able to benchmark a system with production-like workloads altogether. Often, internal teams have a good idea of benchmark queries, but it is impossible to run these queries over production data as the data contains sensitive user information that only the owning team should be able to access; adding DIAMetrics as a data accessor is simply not an option, nor is it an option to provide access to the data through some other role. The data scrambler can help in these cases as it can reformat the data in various ways and with user-controlled degrees of anonymization. Moreover, scrambling takes place in ways that preserve the input value distributions, thus making the scrambled data a good representation of the original production data.

4.2 Software development

In addition to traditional benchmarking, we also offer support for developers to run their benchmarks on experimental instances. DIAMetrics has improved awareness of how different changes to the codebase impact different query engines and has started to integrate large-scale benchmarking into the developer's workflow. Our tool has impacted product development as follows:

System comparison and choice. Our original intention was not to compare the performance of query engines but to provide a single benchmarking framework that supports all internal implementations with the ability to provide historical tracking of multiple performance metrics. As the implementation of DIAMetrics progressed, a new use case emerged: helping new teams decide on the most appropriate query engine for their workload. Whereas for well-established teams it is hard to migrate to a new query engine, newer teams do not have such tie-ins. It is therefore possible for a team to come along with a representative workload and test that workload on the internal query engines that can support it. They can then make an informed decision as to which query engine provides the best support for their workload. Additionally, if they are keen to work with a specific query engine but that engine is not optimized for the workoad, they can provide the engine's developers with example queries where performance suffers. The developers can then integrate these performance enhancement requests into their own roadmaps and have an immediate way to measure their progress towards satisfying these requests.

Performance-driven development. DIAMetrics has been frequently used to set performance goals for development teams. One of the typical use cases is to identify problematic worklads for a particular query engine and then set a roadmap for implementing improvements for these workloads. Development teams will then use DIAMetrics to track their performance on those workloads, observing how their modifications improve the system's performance. At the same time, and given that DIAMetrics may be using other workloads for benchmarking as well, the development team has assurances that newly introduced improvements do not degrade the performance of other workloads.

Release blocking. Monitoring and alerting give rise to the production of compliance tests for the query engines that DIAMetrics supports. Recall that our framework can target any existing query engine deployment. Some of these deployments may be staging ones, running a version of the system's binary that is different to the official one; most frequently the latter is a release candidate version. By comparing the performance of a benchmark on the current binary with that of the release candidate, teams can identify potential problems before releasing the candidate and block the release in the presence of a potential regression. Again, one of the welcome side-effects of DIAMetrics is that it exemplifies the regression through a handful of queries in which the regression manifests. By having this information, development teams can quickly start addressing the regression, continuously checking the convergence between

the degraded and expected performance. At the same time, avoiding a bad release has measurable performance and financial impact.

5. RELATED WORK

Benchmarking is not a novel problem, especially in the context of data management [2, 3, 4, 5, 6, 7, 11, 13, 15, 26, 27, 28], but has become increasingly important over the last years with the increase in available data, the move to hosted management and data services, and the need for low latency processing regardless of data size. All systems need to be robust, i.e., they need to consistently execute their workloads without performance degradation due to changes in the data or the underlying codebase. Robustness has been discussed in several lines of research in the broader context of database systems. For example, [22] discusses robustness for changing datasets while [32] addresses robustness in the context of query plan optimization. Our use-case is not so much data-driven as it is development-driven. Code changes have similar, if not worse, impact on the performance of data management systems if not tested appropriately and continuously. This is especially true in environments with rapid code development and release iterations.

From a research perspective, the work that is closest to some of the ideas implemented in DIAMetrics is workload compression [9] and particularly its application to index selection for relational databases [10]. This is merely part of what our framework supports and any compression algorithm can be 'plugged in' to DIAMetrics so long as its inputs and outputs are translated to the canonical representations the various components of DIAMetrics expect. At the same time, DIAMetrics does not aim to provide insight into different storage configurations of a dataset to optimize its run time; rather, it provides the support necessary to compare and contrast the performance of a query engine on these configurations. Similarly, while workload characterization has received attention from the database community, it has often been used for limited-scope purposes: (*a*) as a tool to help with physical design [31]; (*b*) as a means to identify interesting queries to help in debugging SQL performance [18]; or (*c*) as a way to identify data cleaning primitives in large datasets [20].

Industry-wise, there exist commercial products that allow customers to replay entire workloads [16] in order to analyze performance [29]. The users of these products are expected to replay an entire workload, whereas we can filter it through our summarizer, in order to have something to measure the performance of an SQL engine on. The goal is to completely replay a workload trace down to the sequence and timing of queries issued. This is not our focus: instead, we aim to provide repeatable benchmarking for a variety of systems and not the means to debug any performance issues faced by a particular deployment. Additionally, products like [16, 29] are specific to a system and lack the ability to compare and contrast multiple metrics across systems. Overall, and while certainly related to some of the components of DIAMetrics, that line of products is less general and focuses at reactive optimization as opposed to proactive end-to-end benchmarking, which is our intention.

The effort that is most related to ours is Snowtrail [30]. While the objective is similar, i.e., testing with production data to identify performance regressions, the approach is much more limited in scope compared to DIAMetrics. Performance regression is but one of the use-cases supported by DIAMetrics, which is (*a*) far more general in its architecture, (*b*) provides more stand-alone components, each alleviating a particular benchmarking problem, as opposed to the monolithic design of Snowtrail, (*c*) is capable of supporting more metrics than latency, and (*d*) supports cross-system benchmarking. To the best of our knowledge, DIAMetrics is the first system to provide a disciplined and generic end-to-end solution for benchmarking multiple query engines in a single framework.

6. CONCLUSIONS AND OUTLOOK

We presented DIAMetrics: a framework for benchmarking query engines within Google. DIAMetrics employs a modular architecture of fully reusable components and configurations to simplify the deployment of benchmarks in production environments. Its capabilities extend from generating benchmarking workloads from representative subsets of customer queries, to anonymizing production data for benchmarking purposes, to scheduling the execution of these workloads across multiple query engines and storage backends, and, finally, to system monitoring and alerting. It has been used in various ways within Google, including, but not limited to, historical benchmark performance tracking, system performance comparison, performance-driven development, and release blocking.

DIAMetrics is a relatively new effort, that has already shown strong potential and we believe could be used in various more ways than it was originally designed for. For starters, it would be interesting to apply these techniques not only to internal customers, but also to external customers using Google's infrastracture and query engines that are interested in custom benchmarks to track the performance of Google systems on their workloads. Although internal workloads can be immensely complicated they are also under our complete control. So if there are any issues with any part of the DIAMetrics pipeline we can manually intervene and ensure progress; this is not always the case with external customers. Another interesting application of DIAMetrics would be to use it to make configuration recommendations for new customer workloads. By measuring the similarity of a new customer's workload to existing ones we can set expectations for the performance an internal query engine will deliver. These expectations can be used to set service-level objectives for the engine itself with respect to the customer's workload. Furthermore, workload similarity may imply configuration similarity so a new customer can have a head-start with respect to optimizing a query engine's performance on their workload. Alternatively, various sample sizes of a target summarized workload can be used to estimate the scalability of an engine for that workload, and even extrapolate to the performance of the engine as the size of the workload grows; such capability is very helpful for provisioning and planning.

Overall, DIAMetrics solves the key problem of system benchmarking at the query engine level by providing a uniform way to develop benchmarks for multiple systems without worrying about the intricacies of each individual system. It does so in a scalable and extensible way and we believe that its modular architecture renders it as a framework that is truly greater than the sum of its parts.

7. REFERENCES

[1] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a SQL system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, New York, NY, USA, 2017. ACM.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 8–19, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[3] D. Bitton and C. Turbyfill. A retrospective on the Wisconsin benchmark. In M. Stonebraker, editor, *Readings in Database Systems*, pages 280–299. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[4] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*, pages 61–76, Berlin, Heidelberg, 2014. Springer-Verlag.

[5] P. A. Boncz, A. Anatiotis, and S. Kläbe. JCC-H: adding join crossing correlations with skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers*, pages 103–119, 2017.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 12–21, New York, NY, USA, 1993. ACM.

[7] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke, and D. N. Shah. The bucky object-relational benchmark. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 135–146, New York, NY, USA, 1997. ACM.

[8] B. Chattopadhyay, P. Dutta, W. Liu, A. Mccormick, A. Mokashi, O. Tinn, N. McKay, S. Mittal, H. ching Lee, X. Zhao, N. Mikhaylin, P. Harvey, V. Lychagina, T. Xu, B. Elliott, H. Gonzalez, L. Perez, F. Shahmohammadi, D. Lomax, and A. Zheng. Procella: A fast versatile SQL query engine powering data at YouTube. Data Works Summit, 2018.

[9] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 488–499, New York, NY, USA, 2002. ACM.

[10] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[13] A. Crolotte and A. Ghazal. Introducing Skew into the TPC-H Benchmark. In *Proceedings of the Third TPC Technology Conference on Topics in Performance Evaluation, Measurement and Characterization*, TPCTC'11, pages 137–145, Berlin, Heidelberg, 2012. Springer-Verlag.

[14] S. Deep, A. Gruenheid, J. Naughton, S. Viglas, and P. Koutris. Comprehensive and efficient workload compression. `http://pages.cs.wisc.edu/~shaleen/drafts/compression_fullversion.pdf`, 2020.

[15] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW)*, volume 00, pages 223–230, March 2014.

[16] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, and G. Wood. Oracle database replay. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1159–1170, New York, NY, USA, 2008. ACM.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[18] T. Grust and J. Rittinger. Observing sql queries in their natural habitat. *ACM Trans. Database Syst.*, 38(1):3:1–3:33, Apr. 2013.

[19] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[20] S. Jain and B. Howe. Data cleaning in the wild: Reusable curation idioms from a multi-year sql workload. In *Proceedings of the 11th International Workshop on Quality in Databases, QDB 2016, at the VLDB 2016 conference, New Delhi, India, September 5, 2016*, 2016.

[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

[22] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *Proceedings of the 2015 ACM*

SIGMOD international conference on management of data, pages 1167–1182. ACM, 2015.

[23] M. Pasumanskyl. Inside capacitor, bigquery's next-generation columnar storage format. In *Google Cloud Blog*, 2016.

[24] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, F. Weigel, D. Wilhite, J. Yang, J. Xu, J. Li, Z. Yuan, C. Chasseur, Q. Zeng, I. Rae, A. Biyani, A. Harn, Y. Xia, A. Gubichev, A. El-Helw, O. Erling, Z. Yan, M. Yang, Y. Wei, T. Do, C. Zheng, G. Graefe, S. Sardashti, A. M. Aly, D. Agrawal, A. Gupta, and S. Venkataraman. F1 query: Declarative querying at scale. *PVLDB*, 11(12):1835–1848, 2018.

[25] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[26] Transaction Processing Performance Council. TPC Benchmark C, 2010.

[27] Transaction Processing Performance Council. TPC Benchmark H (decision support), 2017.

[28] Transaction Processing Performance Council. TPC Benchmark DS, 2018.

[29] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer, 2008.

[30] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[31] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE Trans. Softw. Eng.*, 18(4):347–355, Apr. 1992.

[32] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *PVLDB*, 10(8):889–900, 2017.