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 various 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.

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 of 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], to the development of industry standards like TPC-H [22], to benchmarks for object-oriented [4] systems, or to larger cloud-scale serving benchmarks [9] and their derivatives. All these benchmarks have been studied extensively and the knowledge gained has been used to modify them in various ways or deliver new benchmarks altogether that address the shortcomings of the existing ones.

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

© Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. This is a minor revision of the paper entitled DIAMetrics: Benchmarking Query Engines at Scale, published in PVLDB, Vol. 13, No. 12, 3285-3298.
DOI: https://doi.org/10.14778/3415478.3415551

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

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 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 engines, each designed for different use-cases: F1 [20, 21], Dremel [17], Spanner SQL [1], and Procella [6]. There exist specialized storage systems such as Mesa [15] and more generic ones such as Colossus [19] which are leveraged by different applications, supporting different storage formats. Figure 1 shows an

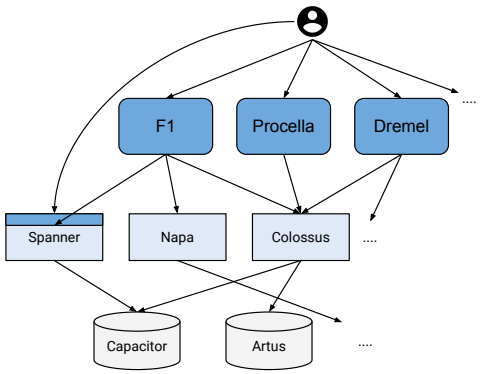


Figure 1: Part of the Google ecosystem.

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.

Modular Components 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 as well challenges addressed by the DIAMetrics framework within Google in Section 4.

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

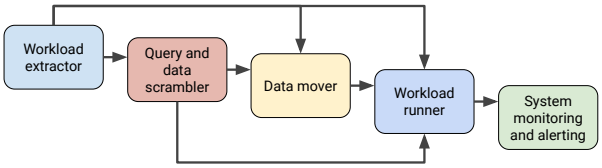


Figure 2: An overview of the DIAMetrics components.

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, and (*e*) the system monitoring and alerting component. An overview of these components is given in Section 3. Each component 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 specifically 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 [19]. 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.

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

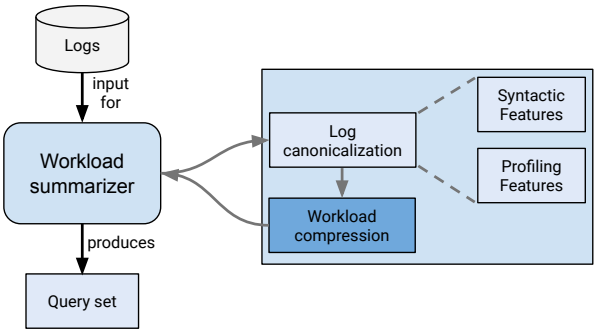


Figure 3: Overview of the workload summarizer.

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 [11]. 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. For further details on how to realize workload summarization, please refer to [11].

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. 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 extremely customizable and may well provide these guarantees implicitly if configured properly by data owners.

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

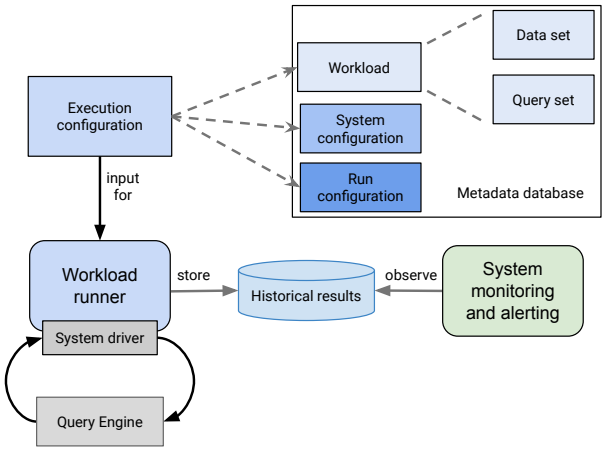


Figure 4: Overview of the workload runner's components.

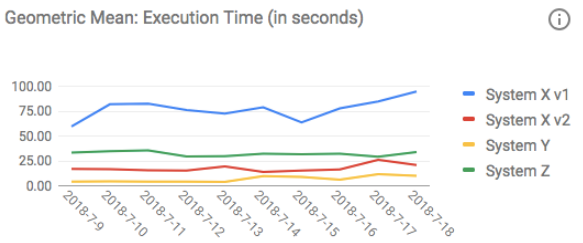
structures transferred to the output as possible. Other information that the data mover attempts to preserve is sharding information, e.g., the number of shards and the partitioning scheme; input storage properties like the input being sorted; data definition properties like functional dependencies if 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

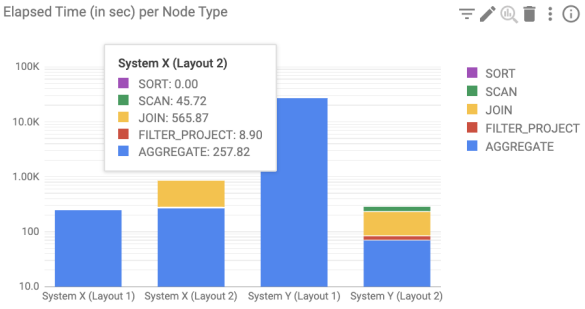
The benchmark execution component of DIAMETRICS is the workload runner. The runner accepts multiple *execution configurations* as input, with each execution configuration containing the following four elements: (*a*) a number of systems and their configurations to use for benchmarking; (*b*) a number of benchmark configurations; (*c*) a number of workloads to benchmark; and (*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. 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 4. While the default execution is sequential, the workload runner also offers various degrees of parallelism. 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.

3.5 System monitoring and alerting

After the workload runner has executed a workload, we export the output into DIAMETRICS-specific logs. These



(a) Consolidated benchmark execution time tracking



(b) Operation breakdown computation

Figure 5: Example dashboards for per-query benchmark performance monitoring.

logs are then used to (*a*) allow 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. 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 5a. 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 5a 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.

In addition to simple tracking dashboards, we also developed more insightful dashboards, such as dashboards that look at the scalability of a system or give insights into specific queries. For example, in Figure 5b, 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.

Finally, DIAMetrics 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. 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. 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. We next sketch 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, and Dremel. Some of their use cases are described here:

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 means to identify important patterns. 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.

Workload optimization. DIAMetrics is capable of producing tracking dashboards for various combinations of system configurations over different versions of the same 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, doing so on a compressed, representative version of a workload 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.

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.

System comparison and choice. As the implementation of DIAMetrics progressed, a novel 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.

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, 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 from 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. 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.

5. RELATED WORK

Benchmarking is not a novel problem, especially in the context of data management [2, 3, 5, 9, 10, 12], 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, [18] discusses robustness for changing datasets while [26] 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 or worse impact on the performance of data management systems if not tested appropriately and continuously.

From a research perspective, the work that is closest to some of the ideas implemented in DIAMetrics is workload compression [7] and particularly its application to index selection for relational databases [8]. 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 [25]; (*b*) as a means to identify interesting queries to help in debugging SQL performance [14]; or (*c*) as a way to identify data cleaning primitives in large datasets [16].

Industry-wise, there exist commercial products that allow customers to replay entire workloads [13] in order to analyze performance [23]. 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. Their 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 [13, 23] 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 on reactive optimization as opposed to proactive end-to-end benchmarking, which is our intention.

The effort that is most related to ours is Snowtrail [24]. 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 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. 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, many 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 *ACM SIGMOD*, pages 331–343, 2017.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *VLDB*, pages 8–19, 1983.

[3] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, pages 61–76, 2014.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *ACM SIGMOD*, pages 12–21, 1993.

[5] 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 *ACM SIGMOD*, pages 135–146, 1997.

[6] 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.

[7] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *ACM SIGMOD*, pages 488–499, 2002.

[8] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[10] A. Crolotte and A. Ghazal. Introducing Skew into the TPC-H Benchmark. In *TPCTC*, pages 137–145, 2012.

[11] S. Deep, A. Gruenheid, P. Koutris, J. Naughton, and S. Viglas. Comprehensive and efficient workload compression. *PVLDB*, 14(3):418–430, 2020.

[12] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW*, pages 223–230, 2014.

[13] 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 *SIGMOD*, pages 1159–1170, 2008.

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

[15] 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, 2014.

[16] S. Jain and B. Howe. Data cleaning in the wild: Reusable curation idioms from a multi-year sql workload. In *QDB*, 2016.

[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1–2):330–339, 2010.

[18] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, pages 1167–1182, 2015.

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

[20] 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.

[21] 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.

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

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

[24] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee. Snowtrail: Testing with production queries on a cloud database. In *DBTest*, pages 4:1–4:6, 2018.

[25] 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.

[26] 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.