Surprise Benchmarking: The Why, What, and How

Lawrence Benson[1], Carsten Binnig[2,3], Jan-Micha Bodensohn[2,3], Federico Lorenzi[4], Jigao Luo[2,3], Danica Porobic[5], Tilmann Rabl[6], Anupam Sanghi[2], Russell Sears[8], Pınar Tözün[9], Tobias Ziegler[2]

[1]Technical University of Munich, [2]Technical University of Darmstadt, [3]DFKI, [4]TigerBeetle, [5]Oracle, [6]Hasso Plattner Institute, [8]CrystalDB, [9]IT University of Copenhagen

ABSTRACT

Standardized benchmarks are crucial to ensure a fair comparison of performance across systems. While extremely valuable, these benchmarks all use a setup where the workload is well-defined and known in advance. Unfortunately, this has led to overly-tuning data management systems for particular benchmark workloads such as TPC-H or TPC-C. As a result, benchmarking results frequently do not reflect the behavior of these systems in many real-world settings since workloads often significantly vary from the "known" benchmarking workloads. To address this issue, we present *surprise benchmarking*[1], a complementary approach to the current standardized benchmarking where "unknown" queries are exercised during the evaluation.

1 INTRODUCTION

Benchmarking is prevalent. The conventional way of benchmarking data management systems relies on *static* standardized benchmarks. The Transaction Processing Performance Council (TPC) [13] has been standardizing benchmarks covering application domains such as OLTP and OLAP, amongst many other [3, 12, 16]. Similar efforts also exist for other types of data-intensive systems, including Linked Data Benchmark Council (LDBC), which targets graph analytics [1, 8, 15] or the Yahoo Cloud Serving Benchmark (YCSB) [6] which is quite popular for key-value stores. Furthermore, there have been many proposals for alternative benchmarks that serve more particular needs [2, 5, 9–11].

Databases systems are "overtuned" for benchmarks. The goal of all these existing benchmarks is to provide a fair way to compare the performance of different systems under expected scenarios [14]. While extremely valuable, these benchmarks all exercise a setup where the workload is well-defined and known in advance, even if the workload is defined as ad hoc queries. As such, existing database systems are often "overtuned" for existing benchmarks such as TPC-H and TPC-DS, and these benchmarks fall short in reflecting how the systems behave when faced with *unknown* scenarios, which arise in real-world applications. This oftentimes leads to complaints about the standardized benchmarks not being representative [17].

[1]The idea of surprise benchmarking was born by a subset of the authors at the Dagstuhl Seminar 23441: "Ensuring the Reliability and Robustness of Database Management Systems", https://www.dagstuhl.de/seminars/seminar-calendar/seminar-details/23441.

DBTest '24, June 9, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0669-1/24/06…$15.00
https://doi.org/10.1145/3662165.3662763

Towards a new class of benchmarks. We propose a new direction called *surprise benchmarking* to address this issue. Surprise benchmarking introduces a new benchmarking regime that is very different from existing benchmarks. It introduces unknown aspects to evaluate database systems, i.e., *elements of surprise*. On the one hand, surprise benchmarks will eliminate the overly-tuned nature of current standardized benchmarking practices. On the other hand, they will help to establish a larger set of benchmarks over time covering a more diverse set of data-intensive applications and their real-world deployments. However, realizing a surprise benchmark requires a design that departs from classical benchmarking.

The vision of surprise benchmarking. Surprise benchmarking comes with two important differences when compared to existing benchmarks: First, surprise benchmarks need to add an unknown aspect to a workload. To achieve this goal, in this paper we discuss a methodology that starts with an existing benchmark (e.g., TPC-H or any other "static" workload) and then adds surprises of different categories, which target different scenarios ranging from benign surprises (which we call on-road surprises) to more malicious surprises, which challenge the database in more extreme scenarios (which we call wrong-road surprises). Second, to add surprises to a benchmark, we also need to change the benchmarking procedure: We cannot disclose the full benchmarking workload to the participants as it is usually done with existing benchmarks, which come with a data and query generator that make the workload fully transparent to each participant. As such, we propose a new benchmarking procedure in which participants only get a representative subset of the workload that allows database vendors to get a rough understanding of the basic characteristics. For the actual evaluation, the database performance is examined by an independent benchmark organizer "secretly".

Several rounds of surprises. Finally, we argue that surprise benchmarking should be conducted more like a contest where we establish periodic (e.g., quarterly or yearly) benchmarking rounds. This allows us to achieve several goals: (1) We can constantly examine new "themes" per round. For example, themes can differ in the general workload class they address (e.g., OLAP, OLTP, key-value, graph processing) and the specific challenges they add (e.g., UDFs or recursive queries). (2) Instead of keeping a surprise workload secret forever, we can release the full workloads of the past rounds, including the surprises. As such, surprise benchmarking will act as an interesting repository of workloads that industry and academia can use as static workloads that target different database capabilities.

Contributions of this paper. To summarize, the contributions of this paper are as follows: (1) We define a methodology for surprise benchmarking through surprise types and benchmarking procedure (Sections 2 & 3). (2) We evaluate the effectiveness of our methodology through a case study, *a first round*, that resembles TPC-H but with surprises from each category. Our results demonstrate how surprise benchmarking captures system behavior that is hard to observe with static and pre-defined benchmarks (Section 4). (3) We identify the challenges of ensuring the longevity of surprise benchmarking and argue for a community effort based on various forms of automation in the process (Sections 5 & 6).

2 SURPRISE CATEGORIES

For a surprise benchmark, it is important to add "unknown" elements to the workloads. We categorize these elements of surprise into three levels of surprise. We expect each benchmarking round to have surprises covering each category. However, the manner in which we mix the surprises, and the percentage of surprise queries may change from round to round.

2.1 On-road Surprises

This type of surprise represents queries that fall in the broader class of queries that come naturally with a certain workload (e.g., OLAP). In other words, these are the queries or scenarios the system was designed to cover, and they do not intend to "drive" the system into "uneven" terrain (and hence we call this category "on-road" surprises). The goal is to observe how the system behavior changes compared to standardized benchmarking.

Examples. For example, queries similar to, but not the same as, the TPC-H benchmark queries for a system specifically designed for OLAP workloads. Our case study shows that these could be queries that exercise typical OLAP scenarios but use not-so-commonly used features in TPC-H, such as common table expressions. Similarly, introducing data skew that mimics real-world data distributions as the surprise would fall under this category.

Workload generation. As part of this work, we plan to develop automatic query generation techniques for this class of surprises as much as possible. In our case study, we use an automated method to generate such queries based on LLMs.

2.2 Wrong-road Surprises

A typical case in real-world scenarios is that users draft schemas or queries that are not an "ideal" use of the database. In contrast to the previous category, this category thus represents scenarios that are not the intended use of the system under test (i.e., "wrong-road"). The important fact is that these scenarios still represent common queries, the user's intentions are not harmful, and the aim is not to strain the database. The goal is to observe how the system handles the performance impact of such misaligned scenarios, and possibly whether it can correct them.

Examples. Examples for this category are the use of `SELECT *` or the use of `string`-columns in joins as often done by applications. Other examples are the use of UDFs instead of doing the data processing with some built-in SQL functions (e.g., use of a UDF instead of `LIKE` for text filter); or instead of using JSON support in a system, casting the data to string and performing regex operations on the string would be misaligned uses of a database system.

Workload generation. While this class of surprises clearly goes beyond the intended use of databases, we will still investigate auto-generation of scenarios for these types of surprises.

2.3 Off-road Surprises

The final surprise category represents scenarios that aim at driving the database system to the edge by testing "extreme" (i.e., "off-road") scenarios. The goal here is to observe the robustness of the systems under extreme scenarios.

Examples. For example, using extremely long SQL query string, introducing extreme (unnatural) data skew, or cutting down the hardware resources by running a heavy stream of collocated queries would fall under this category.

Workload generation. While some automation is possible, this category requires the most careful crafting, and hence might require manual effort.

3 THE BENCHMARKING PROCEDURE

Key idea. The foundation of every benchmark is a comparable setup across systems. Without this, the obtained results are hard to interpret meaningfully. As discussed before, for surprise benchmarking, we use a setup where participating systems are evaluated "secretly". For this, we crafted a first benchmarking environment that showcases how this can be realized. The main idea is that participants submit their systems to the benchmarking organization, which exercises them under a surprise workload.

To realize such a setup, we reuse as much existing infrastructure as possible. Existing benchmarks and infrastructure are more likely to be supported by evaluated systems than a custom setup. In our first iteration, this means we leverage Docker, which allows participants to submit a pre-configured system to the benchmark organization and *BenchBase* [7] as the benchmark driver.

By sharing the full setup (i.e., BenchBase) as well as a representative workload with the systems to be evaluated before they submit[2] their system to be executed under the actual surprise benchmark, we ensure that all systems can execute the benchmark queries. At the same time, we take care not to reveal the nature of surprise in the representative workload to prevent cheating. In the actual benchmark, we reuse the setup but run *surprise* queries. Our code is open-source and available on GitHub.[3]

BenchBase as a driver. BenchBase [7] (formerly *OLTPBench*) is a JDBC-based benchmarking framework written in Java. As many systems support JDBC directly or are PostgreSQL-compatible, we can instruct BenchBase to use the necessary drivers. BenchBase can execute a wide range of standard database benchmarks out of the box,[4] including TPC-H and TPC-C, which we can leverage for reusability. To specify a benchmark, BenchBase uses XML configuration files containing JDBC connection details and the benchmark suite to run. It also allows us to specify custom queries to run, which we use for our surprise queries that are not part of a standard benchmark suite. To support custom queries without parameterization, we slightly modify BenchBase's query templating.

Benchmark setup. Setting up a surprise benchmark requires work from both the organizers and the participants. Below, we outline which steps both need to take.

[2]In our pilot run, we "submitted" PostgreSQL and MonetDB as exemplary systems.
[3]https://github.com/lawben/surprise-benchmark
[4]17 at the time of writing.

3.1 Organizer Setup

The majority of the work for a surprise benchmark is on the side of the benchmark organizers. Currently, this means reliably setting up infrastructure to execute BenchBase against a system and finding suitable surprise queries. Our setup is initially oriented towards single-node databases. This avoids issues such as container-level networking overheads for server-based DBMS systems such as PostgreSQL and also allows in-process databases to run as intended. We note that open challenges and opportunities around different setups exist (e.g., how to involve cloud databases in our benchmark), which we discuss in Section 6.

For our first test benchmark run (see Section 4), the setup contained the following steps:

1) **Setup Dockerfile.** To run all systems in a consistent environment, we prepare a Dockerfile that builds and installs BenchBase for the required JDBC driver and copies it to a target image that contains the system under test. To execute BenchBase, we additionally need to install a Java runtime in the target image.
2) **Copy standard queries.** We copy the benchmark suite's default queries to the target image in the format that we use for our surprise queries. This allows participants to test and configure their system to our setup.
3) **Run benchmark queries.** Given the test setup with the default queries, we execute BenchBase as the benchmark driver against the system. We ensure correct functionality by testing this on an example system, e.g., PostgreSQL.
4) **Hidden surprise queries.** Once the basic setup works, we create a second Dockerfile that copies the surprise queries in addition to the default ones. This Dockerfile remains hidden from the participants and is only used for the actual benchmark.

3.2 Participant Setup

The main task of participants is to ensure that their system runs in our benchmark setup. This is why we provide the complete setup to the participants, and just omit the surprise queries. Currently, setting up a system for our benchmark comprises the following steps:

1) **Create Docker image.** As we copy BenchBase and our benchmark queries into the environment that the system is running in, we require participants to provide a Docker image containing their system. This image is used as the base image (`FROM` in Dockerfile) for our benchmark.
2) **System setup.** As each system is started differently, we require a system setup script. This script should start the system and return once it is ready.
3) **Test default queries.** Participants must ensure that our benchmark setup works correctly with the provided default benchmark suite setup and queries.
4) **Submit Docker image.** Finally, the participants must submit their Docker image.

3.3 Benchmark Execution

During the benchmark execution, we collect the query run times and additional metrics that are relevant to the benchmark. As we use BenchBase, this process can easily be automated. Based on the collected data, we calculate the desired metrics for a leaderboard. Similarly to existing benchmarks, this does not only have to be query latency but could also contain other metrics such as the number of query errors or crashes, e.g., as shown in our case study in Table 1.

4 CASE STUDY: A FIRST ROUND

In this section, we introduce our initial case study with a twofold goal: (1) demonstrate the general methodology we proposed for the benchmark by running concrete example surprises and (2) illustrate initial opportunities that surprise benchmarking presents for system developers and database researchers alike, to evaluate the system under new workloads.

Experimental design & setup. The case study uses the benchmarking methodology as described in Section 2 and Section 3. We build on the schema, data, and queries from the existing TPC-H benchmark, which we augment with additional tables and queries to cover the three surprise categories. Our experiments compare the query execution latencies of PostgreSQL and MonetDB. It is important to note that these systems are used illustratively to show the benchmark design, not to determine which system is better. We conduct all experiments on a single node with four Intel Xeon Platinum 8268 CPUs (24 cores each) and 792 GB main-memory split between sockets.

4.1 On-road: New TPC-H-style Queries

The *on-road* surprise category addresses use cases that database systems are expected to handle but are often overlooked in standard benchmarks. We designed analytical queries that use the TPC-H schema. Unlike traditional TPC-H queries, our queries diverge by including more advanced features, including set predicates and common table expressions. The goal is to observe how databases behave when we deviate slightly from the standardized benchmarks and challenge the systems' capabilities more rigorously.

Generating on-road surprise queries. We use OpenAI's GPT-4-Turbo large language model to automatically generate analytical queries based on the TPC-H schema. Our prompt includes the TPC-H schema, an example query ($Q8$), and instructions on which SQL features to use. We focus on nested queries, set expressions, and common table expressions. We ensure that the resulting queries represent meaningful information needs, such as inquiring about the average total cost of supplies provided by suppliers in each region.[5] From the collection of generated queries, we carefully select five surprise queries $G1-5$ (see Appendix A.1) to examine in our initial case study.

Results. Figure 1 compares the latency of the newly-generated queries ($G1-5$) to the original TPC-H queries ($Q1-22$). We can observe that the newly-generated queries have two outliers ($G2$ and $G4$), which run surprisingly long on Postgres. Furthermore, we see that in contrast to the other queries, $G3$ takes longer on MonetDB than on Postgres. In general, a first interesting observation is that our generated queries typically run longer than the existing TPC-H queries. This demonstrates that our surprise queries are either less optimized or are more compute-heavy than the more data-heavy

[5] About 2/3 of the generated queries were executable by sqlite. We started to see the same queries multiple times when generating more than 30 queries.

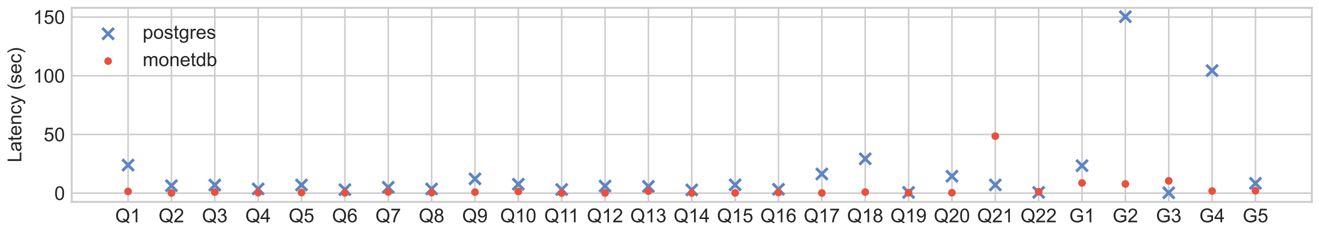

Figure 1: On-road scenario: Latency of the newly-generated queries $G1-5$ compared to the original TPC-H queries $Q1-Q22$.

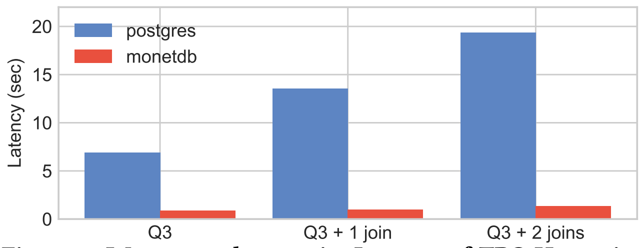

Figure 2: Wrong-road scenario: Latency of TPC-H queries with superfluous joins.

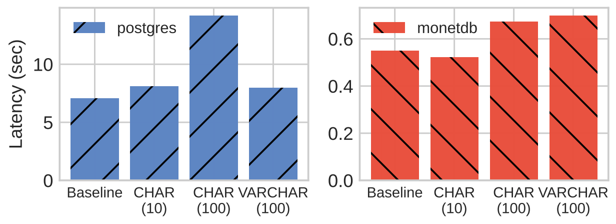

Figure 3: Wrong-road scenario: Latency of TPC-H queries with joins over string keys.

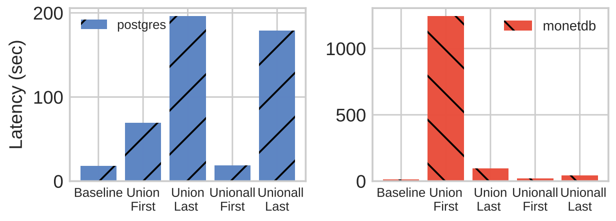

Figure 4: Wrong-road scenario: Latency of TPC-H queries with horizontally-split tables.

queries of TPC-H. Especially Postgres seems to be more sensitive towards our queries, as can been seen by queries $G2$ and $G4$, which exhibit significantly higher latency.

4.2 Wrong-road: Unintentional Misuse

The *wrong-road* surprise category focuses on the unintentional misuse of a database – queries that are not necessarily incorrect but do not follow best practices. For instance, superfluous joins caused by unnecessary overly-normalized schemas, joins over string keys, or horizontally-split tables. The authors have observed all three examples in real-world applications.

Superfluous joins. To examine the effects of superfluous joins, we increase the number of joins required for $Q3$ by distributing the attributes of the `orders` table across multiple tables. This showcases a suboptimal use of a database, which can often happen in real-world scenarios where users design a schema without the optimal execution in a database in mind. As shown in Figure 2, increasing the number of joins leads to a large increase in latency for PostgreSQL, but not for MonetDB. We speculate that this is due to MonetDB's column-major storage model as opposed to PostgreSQL's row-major storage layout.

Joins over strings. Another suboptimal use of a database is to execute joins over string columns. To quantify the effects of this common wrong-pattern using strings as keys, we alter all keys in the TPC-H schema from integers to various string data types. In SQL, `CHAR` represents a fixed-length data type, while `VARCHAR` is variable-length. We select $Q5$ as the benchmark query for this misuse type since it involves 5 tables to join, comparing its performance on the original schema with that on modified schemas where keys are in different string data types. As depicted in Figure 3, transitioning from integers to `CHAR(100)` introduces a significant performance overhead in PostgreSQL. Interestingly, MonetDB's performance remains relatively stable across all cases. We speculate that MonetDB's internal dictionary encoding for string data types makes it more robust against this type of misuse. This again showcases that surprise queries can reveal interesting and maybe unexpected behavior of database systems.

Horizontally-split tables. In real-world sales applications, instead of maintaining a single monolithic table for bookkeeping sales records, another option is to have horizontally-split sales tables (i.e., create a separate table for each time interval and use a union to reconstruct the result).

To simulate this wrong-pattern, we begin with a baseline query, a modified $Q3$, which excludes the `LIMIT` to force the database to execute the query on all partitions. Furthermore, we remove the `ORDER BY` clause to focus solely on presenting the wrong-pattern, rather than sorting algorithms. Following this, we horizontally partition the `lineitem` table based on the `l_shipdate` column. To obtain identical results to the baseline query, two dimensions must be considered: concatenating *first* the monthly tables before query execution or concatenating *last* the results of sub-queries, and performing either `UNION` or `UNION ALL` for the concatenation operation.

Figure 4 shows the result for this wrong-pattern. It shows that performing the concatenation first with `UNION ALL` is the approach with the least overhead in both systems compared to the baseline. Despite significant differences in behavior between the two systems, a common takeaway is that the query optimizers in both systems are unable to rewrite unnecessary `UNION` to `UNION ALL` in the workload of horizontally-split tables.

Table 1: Off-road scenario: Latency (in seconds) for different values of n. Errors: †Could not resize shared memory segment. ‡Query too complex: running out of stack space.

	long SQL strings			complex arithmetic			complex filters		
	1M	10M	100M	10K	100K	1M	1K	10K	100K
postgres	0.06	0.63	27.95	0.03	0.56	†	0.01	0.06	1.05
monetdb	0.09	0.67	28.51	0.37	4.82	120.77	0.10	‡	‡

4.3 Off-road: Intentionally Breaking Systems

The *off-road* surprise category is designed to challenge database systems deliberately with extreme cases, probing their robustness by pushing them to their operational limits. We focus on three common off-road surprises in our case study: long SQL strings, complex arithmetic expressions, and complex filter predicates. While this category strains the database systems intentionally, it is not uncommon for real-world applications. For example, long SQL strings or complex filter predicates described below could occur if queries are programmatically-generated by application logic. In contrast to the previous experiments, we intentionally choose a scale factor of 0.01 for our off-road experiments to isolate the internals of the database rather than the results being influenced by data processing.

Long SQL strings. Our first setting deals with queries that are logically simple, but have long SQL strings. To achieve this, we use the following SQL statement, in which we replace *alias* with a string with $n \in \{1\text{M}, 10\text{M}, 100\text{M}\}$ characters:

```
SELECT l_returnflag AS alias FROM lineitem LIMIT 10;
```

As shown in Table 1, the large aliases lead to large increases in latency for both PostgreSQL and MonetDB despite the fact that the queries are conceptually simple. We observe similar results for long table aliases.

Complex arithmetic expressions. Next, we experiment with complex arithmetic expressions in the select clause. We achieve this using the following SQL query, in which we replace *expr* with a randomly-generated arithmetic expression with $n \in \{10\text{K}, 100\text{K}, 1\text{M}\}$ components that only depend on `l_quantity`:

```
SELECT expr AS val FROM lineitem LIMIT 10;
```

Table 1 shows that the large arithmetic expressions also lead to large increases in latency. Interestingly, PostgreSQL fares much better than MonetDB for arithmetic expressions with 10K and 100K components. However, it fails to evaluate the expression with 1M components, which we speculate is due to a stack overflow.

Complex filter predicates. Finally, we experiment with large filter predicates. We use the SQL statement shown below, in which we replace *pred* with a large ($n \in \{1\text{K}, 10\text{K}, 100\text{K}\}$) disjunction of `o_orderdate = ` *date* predicates:

```
SELECT o_orderkey FROM orders WHERE pred LIMIT 10;
```

As shown in Table 1, MonetDB fails to execute the filter predicates with 10K and 100K components. By contrast, PostgreSQL executes all three queries, albeit with latencies that increase super-linearly regarding the number of components. We speculate that Postgres in contrast to MonetDB uses lazy evaluation of predicates and thus incrementally parses and executes the predicate, which allows Postgres to abort predicate evaluation early if the outcome is clear.

4.4 Discussion

Overall, our case study showcases the need for surprise benchmarking. In all three surprise categories, we observe that going beyond standardized benchmark queries drives well-tested and evaluated databases such as Postgres and MonetDB on much more challenging performance terrains even on the small scale factors (i.e., TPC-H scale factor 0.1 and 1) used in our case study.

5 A COMMUNITY EFFORT

We propose *surprise benchmarking* as a new form of evaluating data management systems beyond static benchmarks. For such a concept to be successful over a longer period of time, it has to become a community effort. Below, we outline how such a surprise benchmark concept could become community-driven.

Periodic benchmarks. Instead of a one-time benchmark effort, we envision periodic benchmarks to establish the practice of "surprising" systems. Similar to other annual contests, e.g., the SIGMOD programming contest or the DEBS Grand Challenge [4], a surprise benchmark could be run at regular intervals, e.g., quarterly or bi-annually. While we aim to automate as much as possible, the preparation still entails a non-trivial overhead for each benchmark. Thus, the time frame in which the benchmarks are run should give both the organizers and participants enough time to prepare.

Types of workloads. A regular and frequent benchmarking interval, i.e., multiple times per year, allows us to cover a wide range of workloads and systems in a shorter period of time. While one benchmark focuses on, e.g., OLAP workloads, the next can focus on, e.g., OLTP workloads. By covering a range of workloads, we also encourage new ideas and less commonly used workloads to surprise systems. For example, at DBTest 2018, Vogelsgesang et al. [17] shared that 50% of the data that they see at Tableau are strings. This insight could be turned into a string-only database benchmark, which strongly contrasts academic mostly-integer benchmarks. As shown in Section 4.2, the choice of strings for joins can impact the performance significantly. However, running a benchmark for a certain workload requires expertise in the area, as the surprise queries may otherwise have low quality. Especially the wrong- and off-road queries require insights into how systems are actually implemented and used but also if such a query is actually realistic.

Release previous surprises. To further strengthen the community, we propose to share the surprises from benchmarks once they are completed. This gives the database community a larger pool of benchmarks to choose from and also highlights inefficiencies in current systems. Such a surprise may encourage researchers to investigate problems that are less widely studied, as they are not covered by standard benchmarks.

Share insights with the community. Aside from the competition of such benchmarks, we believe that the insights gained from the workload surprises as well as system behavior are valuable to an audience beyond only the organizers and participants. To increase the visibility of these findings, we propose to establish a format to share them at their respective annual community events or at cross-community events such as DBTest or TPCTC.

6 DISCUSSION AND OPEN CHALLENGES

To streamline the surprise benchmarking process, we aim to automate as much of the setup and procedure as possible. However, this is not always trivial due to the heterogeneous nature of target systems, setups, and workloads. In this section, we outline open challenges that we plan to investigate throughout the course of future surprise benchmarks.

6.1 Infrastructure

Our initial setup targets single-node databases that can run in a single Docker container. While this covers a wide range of systems developed in academia, many commercial or production-grade systems have more elaborate setups.

Distributed systems. Evaluating a multi-node database in a single container is not sensible, neither is squeezing multiple services of a distributed system into a single node, as this hides inherent network-based communication. Thus, an open challenge remains how to orchestrate such setups in an automated and identical (or at least comparable) setup across distributed systems. A possible solution is to add more advanced tooling, e.g., Kubernetes, for distributed systems.

On-prem vs. cloud. Our current setup requires us to run the system under test in our environment, which categorically disqualifies all cloud-based systems. However, as these systems are increasingly relevant, it is important to benchmark them as well. As we do not control these systems, we need to find a way to integrate them into our automated setup. One way is to run benchmarks "blindly" using user accounts that the cloud provider that are not disclosed to the cloud provider as benchmark users. That way, cloud providers can not prepare for the benchmark user. Such a setup would already be possible today by using the available interfaces of cloud databases to query these systems and control the resources used. Clearly, agreements with cloud database providers are needed so that we are allowed to publish the benchmark results.

Hardware. An overarching issue across all types of systems is a comparable hardware setup. Different systems may target different hardware, e.g., modern SSDs or GPUs, while others need fast RDMA networks or programmable network cards. If the benchmarks are run on our infrastructure, an open challenge is how to address these varying hardware demands. A solution could be to move everything to the cloud. However, even there, it is not always possible to get all types of hardware reliably and the surprise aspect can be disclosed to the provider.

No one-size-fits-all solution. The above challenges show that there is most likely no one size fits all solution for surprise benchmarking, as requirements vary too much across systems and workloads. While this may initially seem like a major issue for automated benchmarking as we propose, we argue that this can actually be beneficial. Instead of focusing on integrating all systems in one large benchmark, we can focus on different groups of systems in addition to different workloads, at the cost of more configuration overhead.

6.2 Workloads

Enriching well-known benchmarks is a promising way to bootstrap the surprise benchmarking effort, but it is not likely to generate excitement in the long term. We envision multiple ways to solicit ideas for new surprises ranging from anecdotal sources of unpredictable performance from industry practitioners to challenging scenarios encountered by application developers to complex scientific computing use cases and domains that often require specialized data stores such as high frequency trading and real time systems.

Metrics. At first glance, performance metrics such as latency and throughput are obvious choices for comparing systems under test. However, many applications have additional requirements such as low tail latency, indicating predictability of performance, and elasticity defined as an ability to serve increasing number of (short) requests without an increase in latency. Multi-tenancy is an orthogonal challenge that stresses systems' resource management and its ability to provide consistent performance irrespective of other tenants' behavior. Ensuring that evaluation metrics align with the nature of the surprise is an open challenge.

6.3 Target Audience

One of the key requirements for a successful community effort we envision is attractiveness to many participants, ideally from all parts of the database community. However, designing the benchmark that would enable a fair comparison of academic prototypes, open source community projects, and commercial cloud native systems is challenging. Instead, we aim to offer different targeted challenges that appeal to parts of the community focusing on different user scenarios. To make participation attractive to commercial vendors and academics alike, we could also consider the option of private benchmarking that would increase attractiveness to vendors whose systems have known limitations in the targeted area.

Inspiring further research. If all participants perform well in a surprise benchmark contest, it clearly is not surprising enough. Since our goal is to increase the spectrum of well understood workloads, we hope that old surprises will inspire analysis of observed behavior and novel ideas that would improve the system designs.

Leaderboards. Automated testing infrastructure is essential for lowering barriers to entry and making participation more appealing. It will also enable easier evaluation of future research proposal that aim to solve challenges posed by prior surprises. Having a common leaderboard that would display both initial participants' scores and later entrants would provide a quick overview of the positive impact of the surprise benchmark.

7 CONCLUSIONS

We have proposed a *surprise benchmarking* methodology as an antidote to overtuning of database systems to well-known static benchmarks. Our initial case study based on TPC-H has illustrated the approach and provided some initial surprising results, already offering promising avenues for improvements for Postgres and MonetDB, the first two systems under test. We are planning to organize the first public contest in the near future.

ACKNOWLEDGMENTS

We thank Schloss Dagstuhl and the organizers and participants of the Dagstuhl Seminar 23441 on *Ensuring the Reliability and Robustness of Database Management Systems.*

REFERENCES

[1] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[2] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. 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 (Lecture Notes in Computer Science, Vol. 10661)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer, 103–119. https://doi.org/10.1007/978-3-319-72401-0_8

[3] Christoph Brücke, Philipp Härtling, Rodrigo Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proc. VLDB Endow.* 16, 12 (2023), 3649–3661. https://doi.org/10.14778/3611540.3611554

[4] DEBS Grand Challenge. 2024. https://2024.debs.org/call-for-grand-challenge-solutions

[5] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, Goetz Graefe and Kenneth Salem (Eds.). ACM, 8. https://doi.org/10.1145/1988842.1988850

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[7] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[8] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[9] Martin L Kersten, Alfons Kemper, Volker Markl, Anisoara Nica, Meikel Poess, and Kai-Uwe Sattler. 2011. Tractor pulling on data warehouses. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[10] Alberto Lerner, Matthias Jasny, Theo Jepsen, Carsten Binnig, and Philippe Cudré-Mauroux. 2022. DBMS Annihilator: A High-Performance Database Workload Generator in Action. *Proc. VLDB Endow.* 15, 12 (2022), 3682–3685. https://doi.org/10.14778/3554821.3554874

[11] Tapti Palit, Yongming Shen, and Michael Ferdman. 2016. Demystifying Cloud Benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 122–132.

[12] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. 2017. Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 573–585. https://doi.org/10.1145/3127479.3128603

[13] Transaction Processing and Performance Council. [n. d.]. Transaction Processing and Performance Council. https://tpc.org/.

[14] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3209950.3209955

[15] Rathijit Sen and Yuanyuan Tian. 2023. Microarchitectural Analysis of Graph BI Queries on RDBMS. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 102–106. https://doi.org/10.1145/3592980.3595321

[16] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, Giovanna Guerrini and Norman W. Paton (Eds.). ACM, 17–28. https://doi.org/10.1145/2452376.2452380

[17] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3209950.3209952

A APPENDIX

A.1 On-road Surprise Queries

We consider the following generated queries for the on-road surprise category.

Listing 1: On-road query $G1$.

```
WITH ordered_parts AS (SELECT l.l_partkey, COUNT(*) AS order_count
                       FROM lineitem l
                       GROUP BY l.l_partkey),
     popular_parts AS (SELECT op.l_partkey
                       FROM ordered_parts op
                       WHERE op.order_count > (SELECT AVG(order_count)
                                               FROM ordered_parts))
SELECT p.p_name, p.p_type, p.p_brand
FROM part p
WHERE p.p_partkey IN (SELECT pp.l_partkey FROM popular_parts pp)
ORDER BY p.p_name;
```

This query identifies which parts are more popular than average based on their number of orders and lists these parts along with their type and brand.

Listing 2: On-road query $G2$.

```
WITH supplier_totals AS (SELECT s.s_suppkey,
                                SUM(ps.ps_supplycost * l.l_quantity) as total_cost
                         FROM supplier s
                            JOIN partsupp ps ON s.s_suppkey = ps.ps_suppkey
                            JOIN lineitem l ON ps.ps_partkey = l.l_partkey
                         GROUP BY s.s_suppkey)
SELECT r.r_name, AVG(st.total_cost) as avg_cost
FROM region r
    JOIN nation n ON r.r_regionkey = n.n_regionkey
    JOIN supplier s ON n.n_nationkey = s.s_nationkey
    JOIN supplier_totals st ON s.s_suppkey = st.s_suppkey
GROUP BY r.r_name
ORDER BY avg_cost DESC;
```

This query returns the average total supply cost per supplier within each region, ordered from highest to lowest.

Listing 3: On-road query $G3$.

```
WITH ordered_parts AS (SELECT l.l_partkey,
                              COUNT(DISTINCT l.l_orderkey) AS order_count
                       FROM lineitem l
                       GROUP BY l.l_partkey)
SELECT p.p_name, op.order_count
FROM part p
    JOIN ordered_parts op ON p.p_partkey = op.l_partkey
WHERE p.p_name LIKE '%bolt%'
ORDER BY op.order_count DESC;
```

This query returns the name and number of distinct orders for parts containing the word 'bolt' in their names, ordered from most to least frequently ordered.

Listing 4: On-road query $G4$.

```
WITH customer_orders AS (SELECT c.c_custkey,
                                COUNT(DISTINCT o.o_orderkey) as order_count
                         FROM customer c
                            JOIN orders o ON c.c_custkey = o.o_custkey
                         GROUP BY c.c_custkey),
     nation_orders AS (SELECT n.n_name, SUM(co.order_count) as total_orders
                       FROM nation n
                           JOIN customer c ON n.n_nationkey = c.c_nationkey
                           JOIN customer_orders co ON c.c_custkey = co.c_custkey
                       GROUP BY n.n_name)
SELECT r.r_name, SUM(no.total_orders) as region_orders
FROM region r
    JOIN nation n ON r.r_regionkey = n.n_regionkey
    JOIN nation_orders no
ON n.n_name = no.n_name
GROUP BY r.r_name
ORDER BY region_orders DESC;
```

This query returns the total number of distinct orders made by customers in each region, ordered from highest to lowest.

Listing 5: On-road query $G5$.

```
WITH ranked_orders AS (SELECT o_orderkey,
                              o_orderdate,
                              o_totalprice,
                              RANK() OVER (
                                  PARTITION BY o_custkey
                                  ORDER BY o_totalprice DESC, o_orderdate
                              ) AS price_rank
                       FROM orders
                       WHERE o_orderstatus = 'F')
SELECT o_orderdate, SUM(o_totalprice) AS total_sales
FROM ranked_orders
WHERE price_rank <= 3
GROUP BY o_orderdate
ORDER BY total_sales DESC, o_orderdate;
```

This query returns the total sales for each date, considering only the top three highest-priced completed orders per customer.