Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing

Mark Raasveldt, Pedro Holanda, Tim Gubner & Hannes Mühleisen
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
[raasveld,holanda,tgubner,hannes]@cwi.nl

ABSTRACT

Performance benchmarking is one of the most commonly used methods for comparing different systems or algorithms, both in scientific literature and in industrial publications. While performance measurements might seem like an objective measurement on the surface, there are many different ways to influence benchmark results to favor one system over the other, either by accident or on purpose. In this paper, we perform an extensive study of the common pitfalls in database performance comparisons, and give tips on how they can be spotted and avoided so a fair performance comparison between systems can be made. We illustrate the common pitfalls with a series of mock benchmarks, which show large differences in performance where none should be present.

CCS CONCEPTS

• **Information systems → Database performance evaluation**;

KEYWORDS

Benchmarking, Performance Evaluation

ACM Reference Format:
Mark Raasveldt, Pedro Holanda, Tim Gubner & Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of 7th International Workshop on Testing Database Systems (DBTEST'18).* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

Reporting experimental results is highly popular in data management research. Good-looking experimental results lend credence to otherwise hard-to-judge proposals. For certain publication venues, submitting a paper without experimental results and a graph like Figure 1 is unwise. However, the great emphasis being put on experimental results regularly leads to questionable experimental setups and therefore questionable results.

Performance is one of the easiest to measure quality metrics of any system. It is an attractive measure of how good a given system or algorithm is, because it is an "objective" measurement that makes it easy to compare two different systems to one another. Performance metrics are frequently used in both scientific papers and by database vendors to show how newly proposed systems or algorithms perform.

DBTEST'18, June 2018, Houston, Texas USA

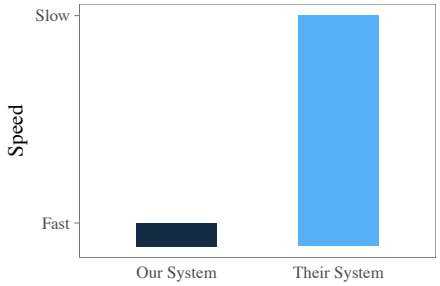

Figure 1: Generic benchmark results.

However, there is a fundamental conflict of interest when running performance benchmarks, especially when an evaluation is performed against previous work. Researchers are incentivized to make their work compare favorably with previous work. Whether intentionaly or inadvertently, this will lead to a disadvantage for previous work, for example through less careful configuration. The experimental results that are then published frequently lead to heated disputes between researchers. The conflict of interest is even more pronounced when companies publish benchmark results comparing their systems with the competition. Some of those results have been prominently denounced as "benchmarketing" [7], but still occur often.

Fair performance benchmarking is all but trivial, and it is very easy to misrepresent performance information to make one system or algorithm look better than another, either by accident or on purpose. In this paper, we will explore the common pitfalls in database performance benchmarking that are present in a large number of scientific works, and describe how to avoid them in order to make fair performance comparisons.

Contributions. The main contributions of this paper are as follows:

- We perform a literature study on different techniques for reproducible benchmarking of generic computer science programs.
- We explore how these techniques fit into the database benchmarking scene, and discuss commonly made database-specific benchmarking mistakes that result in misleading performance comparisons.
- We discuss how these mistakes can be spotted and avoided in order to perform a fair comparison between different database systems and algorithms.

Outline. This paper is organized as follows: Section 2 discusses related work in conducting fair performance comparisons. In Section 3, we discuss common pitfalls when performing fair benchmarking of database systems. We draw our conclusions in Section 4. Finally, in Appendix A, we provide a checklist that can be used to ensure that a performance comparison between systems is fair.

2 RELATED WORK

Performance analysis and comparison has long been recognized as a critical issue in practical computer science in general, and systems research in particular. Ideally, performance analysis is done according to the principles of controlled experiments, where the environment is tightly controlled and individual variables are varied to study their impact. Several collections of best practices and recommendations exist.

2.1 General Guidelines

The general experimental process consists of experimental planning and setup, experiment execution, data collection, data analysis and presentation. Data analysis and presentation are generic over many fields of natural sciences. In statistical data analysis, common issues include "cherry-picking" data, hiding extremes in averages and presenting differences caused by limited accuracy or noise [11]. We note that these issues are also applicable to data management research, but do not focus on them in particular. Similarly, the visual presentation of experimental results through plots is prone to misleading usage. For example, poorly chosen scales make it difficult to draw correct conclusions from them [21].

Regarding practical Computer Science, Jain distinguishes between "mistakes" and "games" in benchmarking [12, p. 127ff]: *Mistakes* are considered to be ill-advised but inadvertent choices regarding benchmark planning and execution. For example, the test workload may be ill-chosen to represent the real-world workload by only focusing on average behavior. *Games* on the other hand are deliberate and purposeful manipulation of the experiment to elicit a specific outcome. For example, if the software or hardware configuration differs between experiments, results are not comparable.

The issue of misleading performance experimentation through benchmarks has been discussed extensively in the High-Performance Computing (HPC) community, for example concerning the impact of compilation flags [15]. Hoefler and Belli describe 12 "rules" for HPC benchmarking [10]: They derive common pitfalls and recommendations to avoid them from issues identified in 120 HPC papers. They stress "interpretability" of experimental results; such that they "provide enough information to allow scientists to understand the experiment, draw their own conclusions, assess their certainty, and possibly generalize results". They identify a host of issues, among them misleading uses of derived metrics such as speedup or geometric means, selective reporting of results without reason and incorrect use of statistical summaries. For example, they stress that averages should only be used if there is no variance in the result measurement, which is almost never the case in benchmarking.

Van der Kouwe et al. [20] describe a number of benchmarking "crimes" in the systems security context, where benchmarking is increasingly important. They note that these crimes threaten the validity of research results and thereby hamper progress. Building on the list of benchmarking crimes [9]. They describe the need for "complete, relevant, sound and reproducible" experimentation. From an extensive survey of 50 top-tier papers from the field, they find that easily preventable benchmarking crimes have been and are widespread, with various issues found in 96% of papers.

2.2 Database-Specific Guidelines

In the data management field, Gray cites widespread "Benchmarketing" as the motivation to devise formal benchmark specifications with precise descriptions of setup, allowed optimizations and metrics to be computed [7, ch. 1]. O'Neil's description of the Set Query Benchmark [7, ch. 5] contains a "checklist" of configuration and reporting requirements. This checklist stresses some important points for benchmarking database systems (summarized here): Data should be generated in the precise manner specified, and reports should include resource utilization for loading and disk space for the database, the exact hardware/software configuration, effective query plans, overhead of collecting statistics, memory usage over time, elapsed time, CPU time and I/O operation count.

Nelson expands on O'Neil's list with a similar checklist in a subsequent chapter [7, ch. 7]. For example, emphasis is put on the need for equivalent syntax of benchmark queries between system, to avoid giving a single system unfair advantages.

Similarly, the current edition of the well-known TPC-H benchmark lists various guidelines for benchmark implementation that are also applicable beyond TPC-H [2, ch, 0.2]. In particular, systems built solely to perform well on this particular benchmark are prohibited from the official competitors. There are several criteria to determine whether this is the case. For example, it is forbidden to take advantage of the limited nature of [functionality tested by the benchmark] and systems may not be significantly limited in their functionality beyond the benchmark. However, in scientific publications the benchmarks only very rarely run to this specification.

Manolescu and Manegold describe principles of performance evaluation extensively in databases [14]. They note that "absolutely fair comparisons are virtually impossible", but point out compiler flags, configuration, query plans and several other factors as pitfalls. They suggest a healthy dose of critical thinking when confronted with experimental results collected by others. Interestingly, they also note that solid craftsmanship is a required pre-condition for "good and repeatable performance evaluation".

Recently, Purohith et al. point out benchmarking "dangers" using SQLite as an example [17]. They show that transaction throughput performance varies by a factor of 28 depending on a single parameter setting and point out that none of 16 papers surveyed reported the parameters necessary to interpret benchmark results. They continue to discuss the impact of a number of influential SQLite parameters.

3 COMMON PITFALLS

In this section, we will discuss different pitfalls that are commonly made when attempting to perform a performance comparison between different database management systems. We will describe how to recognize them, and provide artificial examples of many of the mistakes made and the impact they can have on the experimental results.

All experiments in this section were run on a desktop-class computer with an Intel i7-2600K CPU clocked at 3.40GHz and 16 GB of main memory running Fedora 26 Linux with Kernel version 4.14. We used GCC version 7.3.1 to compile systems. All systems were configured to only use one of the eight available hardware threads for fairness. Furthermore, unless indicated otherwise, we have attempted to configure the systems to take full advantage of available memory. Versions used were MariaDB 10.2.13, MonetDB 11.27.13, SQLite 3.20.1, and PostgreSQL 9.6.1. More information about the configuration (e.g., compilation flags) can be found in the source code.

We report the median value together with non-parametric, quantile-based 95% confidence intervals as recommended by [4, 10]. Our experimental scripts, results, configuration parameters and plotting code are available[1].

We would like to stress that inclusion of a particular system in the experiments below is purely illustrational and in no way implying misleading benchmark results in the corresponding publications.

3.1 Non-Reproducibility

The possibility of reproducing experiments is one of the foundations of scientific research. It allows other researchers to verify results and spot any mistakes made while conducting the original experiment. When providing experimental results that are not reproducible, it is possible to claim anything and report any numbers, as nobody can verify that they are correct.

> An article about computational result is advertising, not scholarship. The actual scholarship is the full software environment, code and data, that produced the result. [5, 6]

In data management it is relatively easy to make reproducible experiments, as performing a performance benchmark is much easier and cheaper than, for example, conducting a large scientific survey. Unfortunately, many authors of database papers do not provide for ways to easily reproduce their experiments. In many papers the code used by the authors to run the benchmark is kept as closed-source or "on request" from a non-responsive e-mail address. In other cases, the data or queries used are not disclosed, or proprietary hardware or back-end systems are used to run the experiments. Often, "intellectual property" and related reasons are cited as reasons for the inability to allow reproduction.

Astonishingly, it is still acceptable to publish an explicitly non-reproducible paper at major data management conferences. There are some attempts to improve the situation, for example the "SIGMOD Reproducibility" project or the preceding SIGMOD repeatability experiment [13], but participation is/was not mandatory. The consequence is that reviewers and readers have to simply trust authors of a non-reproducible paper that their results have been obtained properly even though they theoretically could claim to have measured whatever results they wanted. Calling systems "DBMS-X" to avoid incurring the wrath of their legal departments ("DeWitt clause") is also counter-productive for reproducibility. Even if someone had access to a particular system, how would one know which one was used [22]?

[1] https://github.com/pholanda/FairBenchmarking

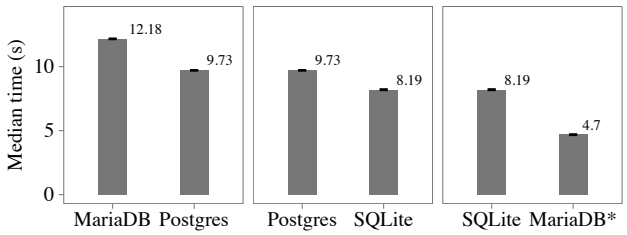

Figure 2: TPC-H Scale Factor 1 Query 1 Results.

Figure 2 shows an exaggerated example of the possible consequences of this situation. Here, we present experiments comparing the performance of TPC-H Query 1 on SF1 between MariaDB, PostgreSQL and SQLite. The leftmost panel shows PostgreSQL being faster than MariaDB. The middle panel shows SQLite being faster than PostgreSQL. The right panel finally shows MariaDB* being faster than SQLite. So MariaDB is both the slowest and the fastest system. This is obviously a contradictory situation that is provoked here for illustratory purposes, with $M < P$, $P < S$ and $S < M$, creating a contradiction similar to the famous paintings by M.C. Escher.

Without extensive details on the benchmark execution, however, it would be impossible to realize that MariaDB* used floating-point columns (`DOUBLE` instead of `DECIMAL`) in the `lineitem` table, which made a very significant performance difference due to an inefficient decimal implementation in MariaDB. Note that both variants are allowed according to the TPC-H benchmark specitication [2, sec. 1.3].

How To Avoid. In order to allow for reproducible experiments, it is important that all configuration parameters are known so the experiment can be exactly reproduced. This includes seemingly minor factors such as the operating system the machine is running on, how the server is installed, the server version, how the server is setup and the server configuration flags. In addition, when comparing against the authors' own algorithm or implementation, the source code should always be available to allow people to reproduce the experiments.

3.2 Failure To Optimize

Benchmarks are often used as a way of comparing two systems, or as a measure for how effective a newly proposed algorithm or technique is. As a result, the general setup of experiments is to compare the authors' newly proposed system or algorithm against an existing system, and show that the authors' system performs better than the current state of the art.

However, this experimental setup gives the author very little incentive to properly optimize the current system. The worse the state of the art system does, the better the authors' system looks. This can be problematic, especially for certain systems that rely heavily on proper configuration, as the performance of an improperly configured system can be significantly worse than the performance of a properly configured system.

Differing Compilation Flags. When compiling a database system from source, the proper compilation flags can result in a significant performance difference. Sanity checking code that is enabled

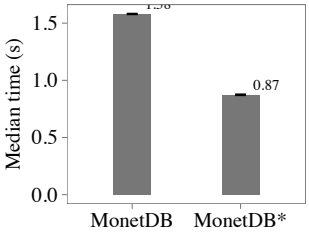

Figure 3: TPC-H Scale Factor 1 Query 1 Results.

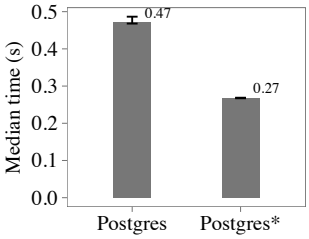

Figure 4: TPC-H Scale Factor 1 Query 9 Results.

only for debug builds can result in significant performance slowdowns. Figure 3 shows the significant performance difference between two systems, DBMS A and B. They are actually simply either an unoptimized (a) and an optimized (b) build of MonetDB. The unoptimized build automatically includes sanity checking code that scans entire columns to ensure certain properties hold. However, these scans are only included to detect bugs, and are not necessary for the operation of the database system. However, compiling MonetDB from the source repository without explicitly enabling an optimized build results in these checks being performed, thus it is easily overlooked by accident.

Sub-Optimal Configuration. Many database systems include a long list of configuration flags that change various different aspects of how the database operates. Certain databases, such as MonetDB [3] or Peloton [16] are designed to include only a few configuration parameters. Others, such as PostgreSQL [19] and MariaDB ship with a large configuration file that allows you to modify anything ranging from the maximum amount of connections to the entire underlying storage engine of the database. These settings have a profound effect on the performance of the system. Consider, for example, the benchmark shown in Figure 4, where we measure the performance of PostgreSQL executing Q9 of the TPC-H benchmark. By using different configuration flags the performance of the database improves drastically.

How To Avoid. Unfortunately, optimizing a DBMS for a particular workload is far from a trivial task. It is so complex that there is an entire profession dedicated to the purpose. However, even performing a small amount of optimizations goes a long way towards making a more fair performance comparison, including reading the guidelines written on how to optimize a specific system for a benchmark. Another option is to involve representatives from the systems that are compared, or to use configuration options used in previous publications from those representatives.

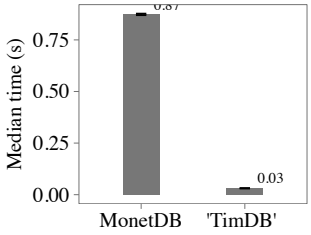

Figure 5: TPC-H Scale Factor 1 Query 1 Results.

3.3 Apples vs Oranges

A performance comparison between two systems can only be fair if both systems perform the exact same functionality. This might seem obvious, however, there are many cases in which small differences are ignored that could explain a large part of the observed performance difference.

The most common way in which this occurs is when a small, standalone program is compared against a full-fledged DBMS. The full-fledged DBMS has to perform many different tasks. It has to handle arbitrary queries, maintain transaction isolation, handle updates to the data and deal with multiple clients issuing queries in parallel. These can all lead to restrictions in what the DBMS is capable of when it comes to a specific task. Meanwhile, the standalone program only has to perform that one specific task. This lack of restrictions will lead to the stand-alone program performing better than the database at that specific task, however, in many cases the performance of the algorithm will go down when it is properly integrated into the DBMS as suddenly error checking code has to be introduced and changes to the code have to be made to properly consider all these factors. Figure 5 shows how a hand optimized stand-alone program compares against the fairly feature-complete MonetDB system [3] when executing Query 1 of the TPC-H benchmark, clearly an unfair and misleading comparison.

Subtle differences in functionality can also lead to different performance characteristics. For example, overflow handling is an often overlooked part of query execution. However handling overflows is necessary to guarantee correct query results regardless of which data is stored in the database. To safely handle overflows one can either check for overflows, so called overflow checking, or ensure that no overflows occur on the data set, so called overflow prevention [8]. Comparing an implementation with overflow handling to an implementation without overflow handling is not a fair comparison.

How To Avoid. When comparing two systems, it is important that the same functionality is measured. When benchmarking a new algorithm, it should ideally be correctly integrated into a complete system, in which case it can be compared against other full systems. It should also always be verified that the new algorithm provides the exact same results as the old algorithm in all cases.

3.4 Over-specific Tuning

Standardized benchmarks, such as the TPC-H [2] and TPC-C [1], are a great step towards making experiments reproducible, as everyone running the benchmark is executing the same queries on the same

dataset. However, they introduce a new issue: heavily tuning a system or algorithm towards a specific benchmark.

Since everything about the standardized benchmark is known up front (e.g. the specific workload, cardinalities of intermediates, distributions of data, selectivities of predicates and amount of groups created) a large amount of benchmark-specific optimizations can be performed. Heuristics can be tuned so the correct join orders are always chosen on the benchmark, and the data can be sharded in such a way that the work is exactly evenly split among different nodes for the benchmark. These can all lead to a system having better performance on a specific benchmark than another system, while performing worse when dealing with a set of similar queries.

How To Avoid. These issues can be prevented by running more experiments rather than only the standardized benchmarks. While the standardized benchmarks are a good baseline comparison, certain systems optimize against a benchmark so heavily that it is no longer a good tool for comparing them to other systems. In addition to running the standard benchmark, a set of different queries should be run and measured as well.

3.5 Cold/Warm/Hot Runs

It is important that a differentiation between so-called "hot" and "cold" runs is made. For certain systems and workloads, the initial (cold) run will take significantly longer than subsequent "hot" runs as relevant data needs to be loaded from persistent storage and the query is parsed/compiled into native code. Subsequent (hot) runs are often faster, as either the buffer pool or the operating system would have already cached the required data, and it is a common feature for databases to include a plan cache in case the same query is run again soon. For this reason, results from hot and cold runs should be reported separately.

In addition, care should be taken when measuring a cold run, as a "warm" run might be measured by accident [14]. It is not trivial to properly measure cold runs. A typical way to collect cold run data is to restart the database server, run a query, and repeat. However, this process ignores the fact that operating systems will use available main memory to cache hard disk blocks for added I/O efficiency. The proper way to collect cold run data is thus to stop the database server, drop all OS caches[2], start the server again, run and time a single query, and repeat. Moreover, in a cloud environment, clearing the caches might be downright impossible because caching might also occur on the virtualization host. There, the only choice might be to start another virtual machine. Both processes makes collecting proper cold run times very time-consuming and inconvenient.

3.6 Ignoring Preprocessing Time

Often in performance benchmarks the setup time (including loading the data into the database and performing preprocessing) is ignored. Hence the creation time of indices is also disregarded. It should be noted that this can also unfairly give certain systems advantages over other systems, as often spending more time on index creation will lead to a faster index. However, if the creation time is discarded as part of the preprocessing time, DBMSs that have expensive-to-construct but efficient indices are given an unfair advantage over databases that have cheap-to-construct indices that might perform less efficiently.

Even if indices are not explicitly created, certain databases might perform indexing or other steps while loading the data. For example, MonetDB will automatically create imprints on a column when a range filter is applied, leading to subsequent range queries on that column performing significantly better. When the timing of the initial query is ignored (as part of a "cold" run), this can lead to misleading performance results when compared with a RDBMS which does not use automatic indices.

Another automatic preprocessing step to be aware of is automatic dictionary encoding of string values. When a string column is loaded into MonetDB, for example, the strings are stored as integer offsets into a heap and duplicate strings are eliminated. In this system, an equality comparison between strings in the query can be answered using only integer comparisons, which is much faster than performing complete string comparisons.

How To Avoid. Either create indexes for both systems that are being benchmarked, or do not create indexes for both systems. Be wary of systems that include automatic index creation or preprocessing techniques, to ensure that they do not have an unfair advantage when preprocessing time is ignored.

3.7 Incorrect Code

When measuring the performance of a newly designed algorithm, it is possible that bugs in the implementation are overlooked by accident. A problem with the implementation might result in incorrect results being produced. If these results are not checked, but only the performance is measured, the benchmark could be measuring an incorrect implementation of the algorithm that might be faster because the bug results in e.g. less data being touched. This problem is exasperated when the benchmark is not reproducible, as the problem may not be discovered and the performance results presented in the paper may be taken as fact, even though the implementation that is measured is incorrect.

Another more subtle way in which a program can be incorrect is in that it works for a specific set of data, but does not answer the query correctly in the general case. This can happen when e.g. overflow handling is neglected, or when the program takes advantage of specific properties of the current dataset such as hardcoding the amount of groups in an aggregation.

How To Avoid. Always check whether your program provides the correct output for a specific query by comparing the output against reference answers, either obtained either from the benchmark specification or from running the same query with the same dataset in a well-known and well-tested RDBMS such as SQLite or PostgreSQL. In addition, make sure to check if the program provides the correct results for the query even when the data is changed.

4 CONCLUSIONS AND OUTLOOK

We are optimistic that issues we have discussed in this paper will be addressed in due time by the data management community. Besides raising awareness of pitfalls we also believe that enforcing reproducibility will result in a self-reinforcing effect where more care is taken to ensure results hold up when experiments are repeated by others. Not ensuring reproducibility is not an alternative, there

[2] `echo 3 > /proc/sys/vm/drop_caches` with root privileges (!) on recent Linux systems

are numerous examples of other fields of research where decades of research where put into question when repeated attempts at replication failed.

In our own experience, we found it useful to provide a virtual machine image to support reproducibility [18]. In this image, all compared systems are installed with the data loaded. That way, all the relevant configuration, data, queries etc. is contained in the image. Minor issues with this system are that a) the image is rather large and needs to be hosted somewhere and b) the pseudo-anonymity of "DBMS X" can be broken by looking at the image.

We could have cited numerous examples for each of the issues we discussed in this paper, but refrained from doing so since research is hardly advanced by pointing fingers. We would like to note that the authors are in no way immune from these issues, and one will probably find examples in our previous work. It is somewhat amusing that the pitfalls described more than 25 years ago (e.g. in [7]) are still a very real problem today.

Acknowledgments This work was funded by the Netherlands Organisation for Scientific Research (NWO), projects "Process Mining for Multi-Objective Online Control" (Raasveldt), "Data Mining on High-Volume Simulation Output" (Holanda) and "Capturing the Laws of Data Nature" (Mühleisen). We would like to thank Stefan Manegold for his valuable input on this work.

REFERENCES

[1] 2010. *TPC Benchmark C Standard Specification.* Technical Report. Transaction Processing Performance Council. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf
[2] 2013. *TPC Benchmark H (Decision Support) Standard Specification.* Technical Report. Transaction Processing Performance Council. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf
[3] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (Dec. 2008), 77–85. https://doi.org/10.1145/1409360.1409380
[4] Jean-Yves Le Boudec. 2010. *Performance Evaluation of Computer and Communication Systems.* EPFL Press.
[5] Jonathan B. Buckheit and David L. Donoho. 1995. *WaveLab and Reproducible Research.* Springer New York, New York, NY, 55–81. https://doi.org/10.1007/978-1-4612-2544-7_5
[6] Jon F. Claerbou and Martin Karrenfach. 1992. Electronic Documents Give Reproducible Research a New Meaning. (1992).
[7] Jim Gray. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[8] Tim Gubner and Peter Boncz. 2017. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. In *ADMS 2017.*
[9] Gernot Heiser. 2010. Systems Benchmarking Crimes. (2010). https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html
[10] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15).* ACM, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644
[11] Darrell Huff and Irving Geis. 1993. *How to Lie With Statistics.* W. W. Norton & Company.
[12] Raj Jain. 1991. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling (Book Review). *SIGMETRICS Performance Evaluation Review* 19, 2 (1991), 5–11. http://doi.acm.org/10.1145/122564.1045495
[13] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. Shasha. 2008. The Repeatability Experiment of SIGMOD 2008. *SIGMOD Rec.* 37, 1 (March 2008), 39–45. https://doi.org/10.1145/1374780.1374791
[14] I. Manolescu and Stefan Manegold. 2008. Performance Evaluation in Database Research: Principles and Experience.
[15] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Not.* 44, 3 (March 2009), 265–276. https://doi.org/10.1145/1508284.1508275
[16] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR 2017, Conference on Innovative Data Systems Research.* http://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf
[17] Dhathri Purohith, Jayashree Mohan, and Vijay Chidambaram. 2017. The Dangers and Complexities of SQLite Benchmarking. In *APSys.* ACM, 3:1–3:6. http://dl.acm.org/citation.cfm?id=3124680
[18] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage - A Case For Client Protocol Redesign. *PVLDB* 10, 10 (2017), 1022–1033. http://www.vldb.org/pvldb/vol10/p1022-muehleisen.pdf
[19] Michael Stonebraker and Greg Kemnitz. 1991. The POSTGRES Next Generation Database Management System. *Commun. ACM* 34, 10 (Oct. 1991), 78–92. https://doi.org/10.1145/125223.125262
[20] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. 2018. Benchmarking Crimes: An Emerging Threat in Systems Security. *CoRR* abs/1801.02381 (2018). http://arxiv.org/abs/1801.02381
[21] Howard Wainer. 1984. How to Display Data Badly. *The American Statistician* 38, 2 (1984), 137–147. https://doi.org/10.1080/00031305.1984.10483186
[22] David Wheeler. 2017. The DeWitt clause's censorship should be illegal. (2017). https://www.dwheeler.com/essays/dewitt-clause.html

A FAIR BENCHMARK CHECKLIST

In order to avoid the common pitfalls described throughout the paper you can use the following checklist as a data management systems oriented guide:

- **Choosing your Benchmarks**.
  - ☐ Benchmark covers whole evaluation space
  - ☐ Justify picking benchmark subset
  - ☐ Benchmark stresses functionality in the evaluation space
- **Reproducible**. Available shall be:
  - ☐ Hardware configuration
  - ☐ DBMS parameters and version
  - ☐ Source code or binary files
  - ☐ Data, schema & queries
- **Optimization**.
  - ☐ Compilation flags
  - ☐ System parameters
- **Apples vs Apples**.
  - ☐ Similar functionality
  - ☐ Equivalent workload
- **Comparable tuning**.
  - ☐ Different data
  - ☐ Various workloads
- **Cold/warm/hot runs**.
  - ☐ Differentiate between cold, warm and hot runs
  - ☐ *Cold runs*: Flush OS and CPU caches
  - ☐ *Warm runs*: Describe what and is measured, and how?
  - ☐ *Hot runs*: Ignore initial runs
- **Preprocessing**.
  - ☐ Ensure preprocessing is the same between systems
  - ☐ Be aware of automatic index creation
- **Ensure correctness**.
  - ☐ Verify results
  - ☐ Test different data sets
  - ☐ Corner cases work
- **Collecting Results**.
  - ☐ Do several runs to reduce interference
  - ☐ Check standard deviation for multiple runs