





DBMS-Benchmarker: Benchmark and Evaluate DBMS in Python

Patrick K. Erdelt ¹ and Jascha Jestel¹

¹ Berliner Hochschule für Technik (BHT)  Corresponding author

DOI: [10.21105/joss.04628](https://doi.org/10.21105/joss.04628)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [George K. Thiruvathukal](#)



Reviewers:

- [@simon-lewis](#)
- [@erik-whiting](#)

Submitted: 17 June 2022

Published: 02 November 2022

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Summary

DBMS-Benchmarker is a Python-based application-level blackbox benchmark tool for Database Management Systems (DBMS). It is intended for reproducible measurement and easy evaluation of the performance the user receives, even in complex benchmark situations. It connects to a given list of DBMS (via JDBC) and runs a given list of (SQL) benchmark queries. Queries can be parametrized and randomized. Results and evaluations are available via a Python interface and can be inspected with standard Python tools like pandas DataFrames. An interactive visual dashboard assists in multi-dimensional analysis of the results.

This module has been tested with Clickhouse, Exasol, Citus Data (Hyperscale), IBM DB2, MariaDB, MariaDB Columnstore, MemSQL (SingleStore), MonetDB, MySQL, OmniSci (HEAVY.AI), Oracle DB, PostgreSQL, SQL Server, SAP HANA, TimescaleDB, and Vertica.

See the [homepage](#) and the [documentation](#) for more details.

Statement of Need

Performance benchmarking of database management systems (DBMS) is an active research area and has a broad audience. It is used “*by DBMS developers to evaluate their work and to find out which algorithm works best in which situation. Benchmarks are used by (potential) customers to evaluate what system or hardware to buy or rent. Benchmarks are used by administrators to find bottlenecks and adjust configurations. Benchmarks are used by users to compare semantically equivalent queries and to find the best formulation alternative*”, Erdelt (2021). Approaches and their special implementations are also examined in benchmarks in academia. There is a large variety of DBMS types and products. For example, solid IT GmbH (2022) ranks 350 DBMS (150 Relational) and Carnegie Mellon Database Group (2022) lists 850 DBMS (280 Relational). We focus on Relational DBMS (RDBMS) in the following. Their types can be divided into, for example, row-wise, column-wise, in-memory, distributed, and GPU-enhanced. All of these products have unique characteristics, special use cases, advantages and disadvantages, and their own justification. In order to be able to verify and ensure performance measurements, we want to be able to create and repeat benchmarking scenarios. Repetition and thorough evaluation are crucial, in particular in the age of cloud-based systems with their diversity of hardware configurations (Kersten et al., 2018; Kounev et al., 2020; Raasveldt et al., 2018).

Thus there is widespread need for a tool to support the repetition and reproducibility of benchmarking situations, and that is capable of connecting to all these systems.

When we collect a lot of data during benchmarking processes, we also need a tool that will help with the statistical, visual, and interactive analysis of the results. The authors advocate using Python as a common Data Science language, since “*it is a mature language programming,*

easy for the newbies, and can be used as a specific platform for data scientists, thanks to its large ecosystem of scientific libraries and its high and vibrant community", Igual & Seguí (2017). This helps in implementing the tool within a pipeline, for example to make use of closed-loop benchmarking situations (He et al., 2019), or to closely inspect parts of queries (Kersten et al., 2018). It also allows the use of common and sophisticated tools to inspect and evaluate the results. To name a few: pandas (McKinney, 2010; The pandas development team, 2020) for statistical evaluation of tabular data, scipy (Virtanen et al., 2020) for scientific investigation of data, IPython and Jupyter notebooks (Kluyver et al., 2016) for interactive analysis and display of results, Matplotlib (Hunter, 2007) and Seaborn (Waskom, 2021) for visual analysis, or even machine learning tools. Moreover, Python is currently the most popular computer language (PYPL, 2022; TIOBE, 2022).

To our knowledge, there is no other such tool, c.f. also the studies in Seybold & Domaschka (2017) and Brent & Fekete (2019). There are other tools like Apache JMeter (Java), HammerDB (Tcl), Sysbench (Lua/JIT), OLTPBench (Java), and BenchBase (Java) that provide very nice features. However they do not fit these needs, since they are not Python-based. Moreover some are limited in supported DBMS, in supporting repetition and (statistical) evaluation, or do not support randomized queries. The design decisions of this tool have been described in more detail in Erdelt (2021). DBMS-Benchmarker has been used as to support receiving scientific results about benchmarking DBMS performance in Cloud environments as in Erdelt (2021) and Erdelt (2022).

Summary of Solution

DBMS-Benchmarker is Python3-based and helps to **benchmark DBMS**. It

- connects to all DBMS having a JDBC interface
- requires *only* JDBC - no vendor specific supplements are used
- benchmarks arbitrary SQL queries
- supports planning of complex test scenarios
- allows easy repetition of benchmarks in varying settings
- allows randomized queries to avoid caching side effects
- investigates a number of timing aspects
- investigates a number of other aspects - received result sets, precision, number of clients
- collects hardware metrics from a Prometheus server (Rabenstein & Volz, 2015)

DBMS-Benchmarker helps to **evaluate results** - by providing

- metrics that can be analyzed by aggregation in multi-dimensions
- predefined evaluations like statistics
- in standard Python data structures
- in Jupyter notebooks - see [rendered example](#)
- in an interactive dashboard

Some features are inspired by [TPC-H](#) and [TPC-DS](#) - Decision Support Benchmarks, which are provided in part as predefined configs.

A Basic Example

The following simple use case runs the query `SELECT COUNT(*) FROM test` 10 times against one local (existing) MySQL installation.

Run `pip install dbmsbenchmarker` for installation. Make sure Java is set up correctly. We assume here we have downloaded the required JDBC driver, e.g., `mysql-connector-java-8.0.13.jar`.

Configuration

DBMS configuration file, e.g. in `./config/connections.config`

```
[
  {
    'name': "MySQL",
    'active': True,
    'JDBC': {
      'driver': "com.mysql.cj.jdbc.Driver",
      'url': "jdbc:mysql://localhost:3306/database",
      'auth': ["username", "password"],
      'jar': "mysql-connector-java-8.0.13.jar"
    }
  }
]
```

Queries configuration file, e.g. in `./config/queries.config`

```
{
  'name': 'Some simple queries',
  'connectionmanagement': {
    'timeout': 5 # in seconds
  },
  'queries':
  [
    {
      'title': "Count all rows in test",
      'query': "SELECT COUNT(*) FROM test",
      'numRun': 10
    }
  ]
}
```

Perform Benchmark and Evaluate Results

Run the CLI command: `dbmsbenchmarker run -e yes -b -f ./config`

After benchmarking has completed we will see a message like Experiment `<code>` has been finished. The script has created a result folder in the current directory containing the results. `<code>` is the name of the folder.

Run the CLI command: `dbmsdashboard`

This will start the evaluation dashboard at `localhost:8050`. Visit the address in a browser and select the experiment `<code>`. Alternatively you may use Python's pandas.

Description

Experiment

An **experiment** is organized in *queries*. A **query** is a statement that is understood by a Database Management System (DBMS).

Single Query

A **benchmark of a query** consists of these steps:

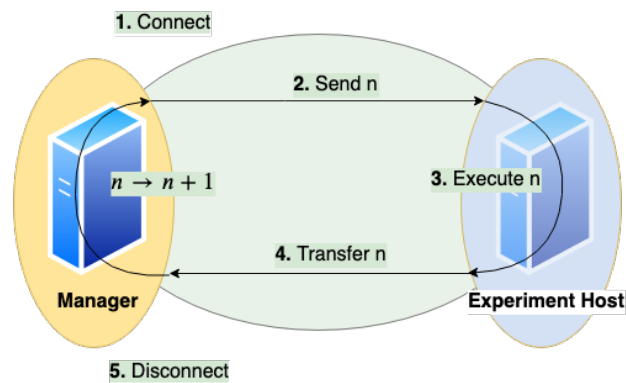


Figure 1: measured times of query processing parts.

1. Establish a **connection** between client and server
This uses `jaydebeapi.connect()` (and also creates a cursor - time not measured)
2. Send the query from client to server and
3. **Execute** the query on server
These two steps use `execute()` on a cursor of the JDBC connection
4. **Transfer** the result back to client
This uses `fetchall()` on a cursor of the JDBC connection
5. Close the connection
This uses `close()` on the cursor and the connection

The times needed for the connection (1), execution (2 and 3), and transfer (4) steps are measured on the client side. A unit of connect, send, execute, and transfer of a single query is called a **run**. Connection time will be zero if an existing connection is reused. A sequence of units of sending, executing, and transmitting between establishing and discarding a connection is called a **session**. This is the same as a run, if we always reconnect prior to sending a query, but if we choose to reuse a connection this will cover multiple runs.

A basic parameter of a query is the **number of runs**. To configure sessions it is also possible to adjust

- the **number of runs per connection** (session length) and
- the **number of parallel connections** (to simulate several simultaneous clients)
- a **timeout** (maximum lifespan of a connection)
- a **delay** for throttling (waiting time before each connection or execution)

for the same query. Parallel clients are simulated using the `pool.apply_async()` method of a `Pool` object of the module `multiprocessing`. Runs and their benchmark times are ordered by numbering. Moreover we can **randomize** a query, such that each run will look slightly different. This means we exchange a part of the query for a random value.

Basic Metrics

We have several **timers** to collect timing information in milliseconds and per run, corresponding to the parts of query processing: **timerConnection**, **timerExecution**, and **timerTransfer**. The tool also computes **timerRun** (the sum of *timerConnection*, *timerExecution*, and *timerTransfer*) and **timerSession**.

We also measure and store the **total time** of the benchmark of the query, since for parallel execution this differs from the **sum of times** based on *timerRun*. Total time means measurement starts before the first benchmark run and it stops after the last benchmark run has finished. Thus total time also includes some overhead (for spawning a pool of subprocesses, computing the size of result sets, and joining results of subprocesses.) We also compute **latency** (measured time) and **throughput** (number of parallel clients per measured time) for each query and DBMS.

Additionally error messages and timestamps of the begin and end of benchmarking a query are stored.

Comparison

We can specify a dict of DBMS. Each query will be sent to every DBMS in the same number of runs. This also respects randomization, i.e., every DBMS receives exactly the same versions of the query in the same order. We assume all DBMS will give us the same result sets. Without randomization, each run should yield the same result set. The tool can automatically check these assumptions by **comparison** of sorted result tables (small data sets) or their hash value or size (bigger data sets). In order to do so, result sets (or their hash value or size) are stored as lists of lists and additionally can be saved as csv files or pickled pandas DataFrames.

Monitoring Hardware Metrics

To make hardware metrics available, we must provide the API URL of a Prometheus Server. The tool collects metrics from the Prometheus server with a step size of 1 second. We may define the metrics in terms of Prometheus's **promql**. Metrics can be defined per connection.

Results

As a result, we obtain measured times in milliseconds for the query processing parts: connection, execution, and data transfer.

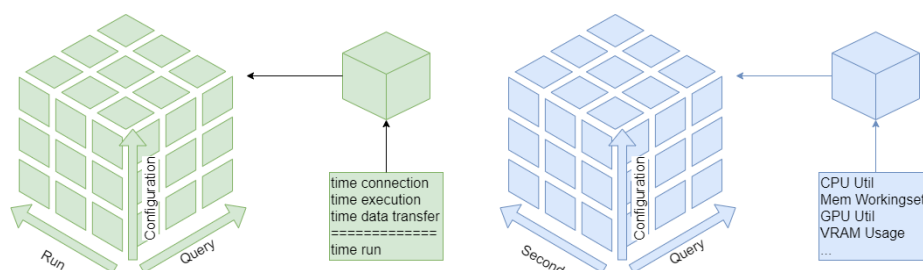


Figure 2: evaluation cubes for time and hardware metrics.

These are described in three dimensions: number of run, of query, and of configuration. The configuration dimension can consist of various nominal attributes like DBMS, selected processor, assigned cluster node, number of clients, and execution order. We also can have various hardware metrics like CPU and GPU utilization, CPU throttling, memory caching, and working set. These are also described in three dimensions: Second of query execution time, number of query, and of configuration.

Evaluation

Python - Pandas

The cubes of measurements can be sliced or diced, rolled-up, or drilled-down into the various dimensions and several aggregation functions for evaluation of the metrics can be applied: first, last, minimum, maximum, arithmetic, and geometric mean, range, and interquartile range, standard deviation, median, some quantiles, coefficient of variation, and quartile coefficient of dispersion. This helps in univariate analysis of center and dispersion of the metrics to evaluate measures and stability.

The package includes tools to convert the three-dimensional results into pandas DataFrames, like covering errors and warnings that have occurred, and timing and hardware metrics that

have been collected or derived. For example the latency of execution, aggregated in the query dimension by computing the mean value, can be obtained as:

```
df = evaluate.get_aggregated_query_statistics(
    type='latency', name='execution', query_aggregate='Mean')
```

Latency of Timer Execution [ms]

DBMS	MariaDB-1-1	MariaDB-2-1	MonetDB-1-1	MonetDB-2-1	MySQL-1-1	MySQL-2-1	PostgreSQL-1-1	PostgreSQL-2-1
Q1	115984.029097	113477.396985	5559.138363	4515.038655	139963.454816	139644.360665	20034.833936	19861.297945
Q2	13189.844139	13157.687652	413.282013	340.652305	1782.370459	1560.817159	7211.191122	17486.172169
Q3	58800.386259	55994.774677	1118.389753	283.285230	60650.473856	60324.399922	15162.580179	19747.151356
Q4	10906.366797	10700.690025	167.731983	156.817029	10353.680159	10227.460235	63275.764168	3659.762128
Q5	43172.769596	41958.683499	369.404377	382.882849	44848.791658	40186.573100	5782.834073	6885.320396
Q6	19219.304236	19288.558926	1785.574434	129.571968	24070.314904	24026.849064	2917.765952	2885.997845
Q7	35226.790069	33578.401355	2198.102969	3050.271067	27260.520071	24198.575423	11244.196273	13544.882668
Q8	74788.269761	73308.950132	1091.897862	3154.334292	117483.286765	106573.598974	12783.235100	13338.417255
Q9	141087.924653	139178.495053	4908.692492	3954.063788	124482.944844	112482.017125	44059.743012	42577.751900
Q10	141401.555155	142088.174021	906.461463	801.192185	20991.290802	20745.993865	20828.577831	16729.723187
Q11	3943.081469	4038.932427	260.222192	114.384230	5474.876384	5460.062068	3688.202220	1934.406897
Q12	132080.785154	128806.873278	3773.600698	191.276890	34563.874573	34269.322594	15897.675939	7168.201096
Q13	114271.550681	115453.225490	3589.015730	3179.584549	161979.926707	160748.794775	18783.155397	7769.695470
Q14	511364.784783	497102.376689	181.431396	136.607647	29699.769745	29359.402171	8135.799106	3260.492730
Q15	38717.798192	38992.169477	270.773724	282.713636	53015.246627	52652.230879	10828.831254	7963.955605
Q16	4027.831425	4120.540270	686.746840	691.220624	5078.042286	5088.938285	6058.316758	5587.771454
Q17	1276.899524	1252.970511	1891.844391	2360.095346	13071.609235	12453.614161	11934.943291	13523.710171
Q18	126002.332541	129191.025399	465.953219	477.032062	30781.519091	30425.720738	33115.222733	29473.692810
Q19	2384.642124	2344.654581	251.829252	248.328930	3420.048188	3349.401808	568.557070	492.506636
Q20	20484.981706	20624.654693	230.338829	208.398596	10160.403924	9924.714682	7587.395355	9418.309655
Q21	246127.850226	245983.909848	71511.727077	70921.302693	81236.928218	78550.780588	6734.338372	6795.589572
Q22	1825.728476	1837.013443	600.653826	967.767178	2372.459627	2337.835912	761.021456	744.030803

Figure 3: example DataFrame: latency of execution times aggregated.

GUI - Dashboard

The package includes a dashboard that helps in interactive evaluation of experiment results. It shows predefined plots of various types, which can be customized and filtered by DBMS configuration and query.



Figure 4: screenshot of dashboard.

Acknowledgements

We acknowledge contributions from Andre Bubbel to include TPC-DS queries.

References

- Brent, L., & Fekete, A. (2019). A versatile framework for painless benchmarking of database management systems. In L. Chang, J. Gan, & X. Cao (Eds.), *Databases theory and applications* (pp. 45–56). Springer International Publishing. https://doi.org/10.1007/978-3-030-12079-5_4
- Carnegie Mellon Database Group. (2022). Database of Databases. In *Database of Databases*. <https://dbdb.io>
- Erdelt, P. K. (2021). A framework for supporting repetition and evaluation in the process of cloud-based DBMS performance benchmarking. In R. Nambiar & M. Poess (Eds.), *Performance evaluation and benchmarking* (pp. 75–92). Springer International Publishing. https://doi.org/10.1007/978-3-030-84924-5_6
- Erdelt, P. K. (2022). Orchestrating DBMS benchmarking in the cloud with kubernetes. In R. Nambiar & M. Poess (Eds.), *Performance evaluation and benchmarking* (pp. 81–97). Springer International Publishing. https://doi.org/10.1007/978-3-030-94437-7_6
- He, S., Manns, G., Saunders, J., Wang, W., Pollock, L., & Soffa, M. L. (2019). A statistics-based performance testing methodology for cloud applications. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 188–199. <https://doi.org/10.1145/3338906.3338912>
- Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90–95. <https://doi.org/10.1109/MCSE.2007.55>
- Igual, L., & Seguí, S. (2017). *Introduction to data science - a Python approach to concepts, techniques and applications* (pp. 1–215). Springer. <https://doi.org/10.1007/978-3-319-50017-1>
- Kersten, M. L., Koutsourakis, P., & Zhang, Y. (2018). Finding the pitfalls in query performance. In A. Böhm & T. Rabl (Eds.), *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018*, (pp. 3:1–3:6). ACM. <https://doi.org/10.1145/3209950.3209951>
- Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., & Willing, C. (2016). *Jupyter notebooks – a publishing format for reproducible computational workflows* (F. Loizides & B. Schmidt, Eds.; pp. 87–90). IOS Press. <https://doi.org/10.3233/978-1-61499-649-1-87>
- Kounev, S., Lange, K.-D., & Kistowski, J. von. (2020). *Systems benchmarking - for scientists and engineers*. Springer. <https://doi.org/10.1007/978-3-030-41705-5>
- McKinney, W. (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt & Jarrod Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61). <https://doi.org/10.25080/Majora-92bf1922-00a>
- PYPL. (2022). *PYPL PopularitY of Programming Language index*. <https://pypl.github.io/PYPL.html>
- Raasveldt, M., Holanda, P., Gubner, T., & Mühleisen, H. (2018). Fair benchmarking considered difficult: Common pitfalls in database performance testing. *Proceedings of the Workshop on Testing Database Systems*, 2:1–2:6. <https://doi.org/10.1145/3209950.3209955>

- Rabenstein, B., & Volz, J. (2015). *Prometheus: A next-generation monitoring system (talk)*. USENIX Association.
- Seybold, D., & Domaschka, J. (2017). Is distributed database evaluation cloud-ready? *New Trends in Databases and Information Systems*, 100–108. https://doi.org/10.1007/978-3-319-67162-8_12
- solid IT GmbH. (2022). DB-Engines Ranking. In *DB-Engines*. <https://db-engines.com/en/ranking>
- The pandas development team. (2020). *Pandas-dev/pandas: pandas (latest)* [Computer software]. Zenodo. <https://doi.org/10.5281/zenodo.3509134>
- TIOBE. (2022). TIOBE Index - TIOBE. In *TIOBE*. <https://www.tiobe.com/tiobe-index>
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17, 261–272. <https://doi.org/10.1038/s41592-019-0686-2>
- Waskom, M. L. (2021). Seaborn: Statistical data visualization. *Journal of Open Source Software*, 6(60), 3021. <https://doi.org/10.21105/joss.03021>