Comparative Evaluation of Database Performance in an Internet of Things Context

Denis Arnst*, Valentin Plenk†, Adrian Wöltche‡
Institute of Information Systems at Hof University, Hof, Germany
Email: *denis.arnst@iisys.de, †valentin.plenk@iisys.de, ‡adrian.woeltche@iisys.de

***Abstract*—We use an application scenario that collects, transports and stores sensor data in a database. The data is gathered with a high frequency of 1000 datasets per second. In the context of this scenario, we analyze the performance of multiple popular database systems. The benchmark results include the load on the system writing the data and the system running the database.**

Keywords–performance; benchmark; nosql; relational; database; industry 4.0; mariadb; mongodb; influxdb; internet of things; high frequency data acquisition; time series.

I. Introduction

Recently popular media are heralding the advent of a new age with buzzwords like ”Internet of Things” (IoT) or ”Industry 4.0” (I4.0). One of the popular mantras is ”data is the new oil”. This claim is surely true for applications like predictive maintenance where data gathered during operation of a production machine is mined for wear indicators. Many papers address the ”refining process” (e.g. [1]–[3]) and propose data-mining algorithms that extract said indicators from a database or a data lake.

In this paper, however, we focus on collecting and storing time series data as integral part of the industrial data analytics process [4]. This can be very challenging both in terms of engineering the instrumentation and in implementing fast data-acquisition and data-handling software. In one of our research projects, we collect and store $\approx 4\frac{\mathrm{GB}}{\mathrm{day}}$.

Standard databases can be tuned towards high performance reading or writing of data, but often not towards both at once. Especially when a fast retrieval of time series data is of interest, for example in predictive analytics, relational databases rely on B-tree indexes that permit a fast search for data. These indexes are a huge performance bottleneck if frequent updates are made. This stems from B-trees being optimized for random fills and not for updates only coming from one side of the tree. [5] propose structures like the B(x)-tree to overcome this problem. Nevertheless, standard databases do not implement specialized index structures in most cases. Instead, specialized ”time-series” databases for this use case exists (e.g. [6]–[9]).

To verify whether these databases are more suitable for our application, we use the benchmark scenario presented in Section II that generates a standard load on all subsystems of the setup, to compare relational, NoSQL and specialized time-series databases. Section III presents our test candidates.

In Section IV, we describe the different implementations we developed for writing to the databases. We evaluated several ideas from [10], such as time series grouping.

To evaluate the database performance we measure the load on the involved infrastructural components, i.e., CPU, memory, network and hard disk, and perform the benchmarking, as described in Section V. Section VI discusses our findings. Section VII summarizes the paper and gives a brief outlook on our future work.

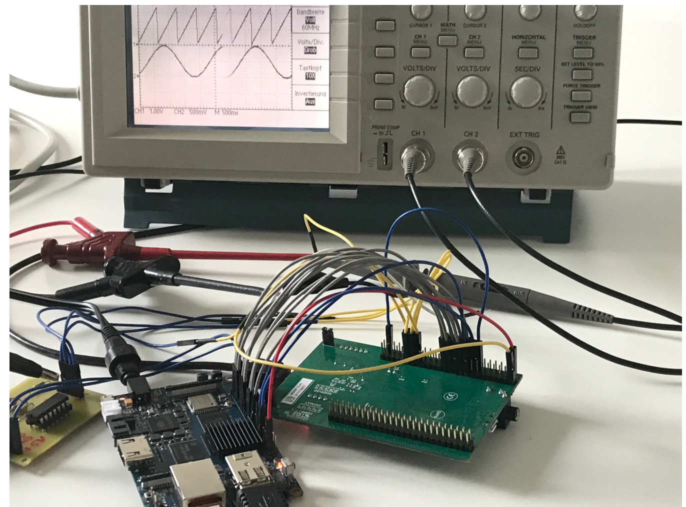

Figure 1. The test setup

II. Benchmark Application

One of our currrent projects is using predictive maintenance for analyzing data stemming from a complex tool operating within an industrial machine. The tool is equipped with 13 analog and 37 digital sensors recording mechanical parameters during operation of the tool. The machine tool opens and closes the tool ≈ 3 times per second, i.e., 3 working cycles per second. Our application records ≈ 300 samples per cycle from the sensors and stores them in a database for later analysis.

For the tests in this paper, we substitute tool and machine tool with electronic function generators as shown in Figure 1. One function generator is set to make a sinus wave. It is wired to a divider circuit, which accepts one input and divides it into four outputs of different amplitudes. The other generator creates a sawtooth wave. The resulting five analog outputs are wired to GPIO-Inputs of a STM32F4-Discovery board.

In total, we sample 5 analog channels with a resolution of 12 Bit (represented using 2 bytes) and a sample rate of $1000\frac{\mathrm{samples}}{\mathrm{sec}}$. This corresponds to a data rate of $10,000\frac{\mathrm{bytes}}{\mathrm{sec}}$.

Figure 2 shows the flow of the data through our setup. The sensor data is gathered by a microcontroller which sends it to a single board computer via a parallel interface. The single board computer is running two separate applications: one reads from the parallel interface and adds a timestamp to the sensor data. The second application receives the data and writes it to the database on our server. These applications are linked via a Linux message queue. If the second application is not reading fast enough to keep the buffered data in the queue below $\approx 16\mathrm{kByte}$ data is lost.

We use a STM32F407 on a STM32F4Discovery evaluation board to convert the sensor data from analog to digital. The

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-669-9

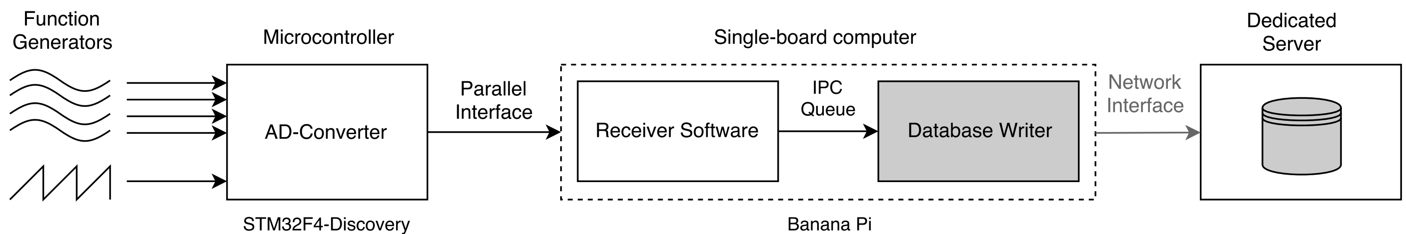

Figure 2. Block diagram of the test setup

embedded application is written in C and does not use any operating system.

The Single-board-Computer is a Banana Pi M3 running the Linux distribution CentOS 7 without an X.Org-Server. This system uses an ARM Cortex A7 Octa-Core with 2 GB RAM and has GigaBit Ethernet on board. The two applications running on this system are written in C and C++.

The database is run on a dedicated server running Linux with an AMD Phenom(tm) II X6 1055T Processor, 16GB RAM (4 x 4GB, DDR3, 1333 MHz) and a 128GB SSD running on a ASRock 880G Extreme 3 mainboard. It also runs CentOS 7 as distribution.

Banana Pi and server are linked via fast ethernet.

The parts of Figure 2 shown with gray background are database specific. We use high-level libraries to access the database and provide three different implementations and server installations.

III. Choice of Databases

Various publications like [7] or [11] list an huge number of different databases. They distinguish three categories: Relational Database Management Systems (RDBMS), NoSQL Database Management Systems (DBMS), and the more specialized Time Series Databases (TSDB). For our benchmark, we chose one system for each category. For the selection we focus on mature (stable releases available for at least 3 years) and free software with options for enterprise support. We mainly consulted the database ranking website [11] as basis for selecting databases for our comparison.

As a representative RDBMS we selected the open source database MariaDB [12]. It is a fork of the popular MySQL database and widely used in Web-Applications and relational scenarios. [13] lists MySQL and its more recent fork MariaDB combined as top RDBMS.

We selected MongoDB [14] as a DBMS advertised expressly for its usefulness in an IoT context with a lot of sensor data. It is also the most promising document store [15].

As TSDB we chose InfluxDB [16] which claims to be highly specialized in sensor data. This claim is confirmed by the score in [17].

IV. The different Implementations

Every millisecond the Database Writer application running on the single-board computer receives a new datapoint. Listing 1 shows the structure of the datapoint: It contains a timestamp and a set of five analog values. The timestamp has a resolution of one nanosecond and uses 12 bytes of memory. The analog values are represented as 16-bit integers. Thus one datapoint uses 22 bytes of memory.

Depending on the architecture of the database, we implemented different ways of storing the data detailed in the following sections. Each implementation itself is optimized concerning runtime complexity for reduced influence on the benchmarks by using memory usage techniques (i.e. stack memory allocation), database specific techniques (i.e. prepared statements), and general algorithmic design principles. This way, we are able to receive optimal database performance results. It is, however, possible, that non-optimized client implementations negatively impact the throughput. This is not covered by this paper for now.

A. MariaDB – Individual datapoints

This is a straightforward maybe even naive implementation of the data structure. We sequentially store each datapoint in the database. This results in a high rate of operations on the database ($5000 \frac{\text{writes}}{\text{second}}$). Table I shows the structure of the data. A compound index is set on *second* and *nanosecond*. *number* describes the index of the sensor, *measurement* the corresponding sensor value.

TABLE I
MariaDB - Table structure of individual datapoints

Field	Type
second	bigint(20)
nanosecond	int(11)
number	smallint(5) unsigned
measurement	smallint(5) unsigned

Our implementation of the algorithm based on `libmariadb` uses prepared statements and struct data binding for higher performance. Our performance optimizations because of the creation of tables and the explicit transaction preparation and commitment make the MariaDB code the largest and most complicated of all our implementations.

B. MariaDB – Bulk Datapoints

This implementation collects all data from one machine cycle at once (in our test scenario: one cycle per second)

Listing 1. One datapoint

```
1 struct data_point
2 {
3   int64_t s;
4   int32_t ns;
5   uint16_t measurements[5];
6 };
```

and writes out one row per cycle. Therefore, we can store the data in bigger units, which reduces the load dramatically. In MariaDB, the JSON field is an alias for longtext field. Yet, the specialized JSON query commands in MariaDB work for such fields, which allows to later query the denormalized data saved. Table II shows the used structure. *second* is an index, *measurements* contains a JSON document built according to the example in Listing 2. The document contains the measurements and its time in nanoseconds in relation to the second of the table. Thus, the rate of index updates is reduced to 1 per second.

TABLE II
MARIADB - TABLE STRUCTURE OF DATAPOINTS IN BULK

Field	Type
second	bigint(20)
size	int(10) unsigned
measurements	json

Listing 2. MariaDB - JSON Documents

```
1 {
2  "measurements": [
3    {"ns":346851124,"m":[389,792,1202,315,552]}
4    ,{"ns":346933204,"m":[516,794,634,317,559]}
5    ...
6 ]}
```

The difficulty of this adaption is similar to the original ”naive” approach , but in one detail even more complicated: As it is theoretically impossible to know how many measurements one cycle will have (most of the time the stated 5000 measurements per second in our case, but this is not guaranteed), we needed to implement a dynamically growing character field for the JSON data. We also needed to change the struct binding in the transaction commitment for honoring the dynamical length of the JSON data.

C. MongoDB – Individual Datapoints

As a document-orientated database, MongoDB allows for flexible schemata. Data is organized internally in BSON (Binary JSON) documents, which are in turn grouped in collections.

Saving the individual datapoints according to Listing 1 each measurement would be a document with the time of measurement and the values organized as a JSON-array.

The database supports setting an index on a field of a document. To support further searching of measurements, an index is set on time. With such a structure, numerous documents are created per second. After each document, the index needs to be updated, which results in high computational effort.

The software for the MongoDB Database Writer is written in C++ and uses `mongocxx` in conjunction with the `bsoncxx` library. The document orientated approach of MongoDB makes designing data structures very flexible. However, the freedom leads to more work on the initial programming approach. Also the need to link two libraries creates additional effort.

D. MongoDB – Bulk Datapoints

As stated in Section IV-B we can store a bigger number of datapoints at once. In MongoDB, we can implement this with the structure shown in Listing 3.

Listing 3. Datapoints in bulk

```
1 {
2   "time" : ISODate("2018-02-12T19:56:49Z"),
3   "measurements" : [
4     { "time" : ISODate("2018-02-12T19:56:49.13
        5Z"), "sensors" : [ 0, 0, 0, 9, 347 ] }
        ,
5     { "time" : ISODate("2018-02-12T19:56:49.13
        6Z"), "sensors" : [ 0, 2, 4, 10, 351 ]
        },
6     ...
7   ]
8 }
```

The time value of the top-level document has a precision of a second. This document holds all datapoints sampled during this second in an array. Every nested document contains the exact time of its measurement and the actual sensor-values. With this approach, the index has to be updated only once per second resulting in optimized write performance. Nevertheless, it must be considered that in this case only a whole second but no parts of it can be retrieved efficiently. However, because of the high increase in write throughput, we accept this drawback.

The application creates a document for a whole second and fills it until the second has passed. Accordingly one such document is inserted per second.

The documentation for MongoDB provides examples for the use of streams and basic builders consisting of function calls. Yet the use of nested structures and the nature of C++-streams is poorly documented in the doxygen-based manuals, increasing the implementation effort.

E. InfluxDB

As a time-series database InfluxDB has a strict schema design. Every series of data consists of points. Each point has a timestamp, the name of the measurement, an optional tag, and one or more key-values fields. Timestamps have an accuracy of up to one nanosecond and are indexed. The name of the measurement should describe the data stored. The optional tags are also indexed and used for grouping data. Data is retrieved with InfluxQL, a SQL-like query language. Data is written using the InfluxDB line-protocol (Listing 4). The first string is the name of the measurement, here simply *measurement*. Subsequently following the key-values with five measurements and finally a timestamp in nanosecond precision.

Listing 4. InfluxDB Line-Protocol example

```
1 measurement m0=0, m1=0, m2=0, m3=9, m4=347
    1518465409001000000
```

The Database Writer for InfluxDB is written in C. The default API for InfluxDB is HTTP. For our high-frequency write access however, we haven chosen the UDP protocol which is also supported. In this case, the data is composed into a line-protocol with simple C-String functions and sent with

the Unix function `sendto`. Since no external code is required and a custom design of the data structure is not possible, using the database is straight-forward and fast to implement.

Additionally, InfluxDB also offers built-in functions to process data statistically and a client library is not necessary, which is a benefit for software developers using it.

The choice of UDP has the probability of data loss, which is acceptable in our use case. For enabling the UDP service of InfluxDB, the OS was configured correspondingly to the information provided by InfluxData [18].

V. TESTING

Most applications in our context face limitations in terms of computing power and network bandwidth. Consequently we measure the load on the single board computer, the load on the server and the network load.

The system load on both computers is measured in terms of CPU and memory usage. We created a script, which runs the specified application for one hour. Before it ends the application, it uses two Linux-System commands to gather the following parameters.

L_{CPU} indicates the processor usage. We obtain this value with the Linux command `ps -p <pid> -o %cpu` which will return a measure for the percentage of time the process <pid> spent running over the measurement time.

The maximum value for one core is always 100%. On our 8 core single-board computer the absolute maximum value would be $L_{CPU} = 800\%$. On the server the absolute maximum value is 600%.

L_{mem} indicates memory usage in kByte. We use the amount of memory used by the process <pid> as the sum of active and paged memory as returned by the command `ps aux -y | awk '{if ($2 == <pid> ) print $6}'`. It outputs the *resident set size* (RSS) memory, the actual memory used which is held in RAM.

L_{disk} shows the the amount of disk used by a database. To determine this parameter we first empty the respective database completely by removing its data folder. Also, we start the database and measure the disk space of the folder before we test. After the test we measure the used disk space again and use the difference as result. `du -sh <foldername>` is used to get the disk consumption of the respective data folder. To put the results in perspective: Our benchmark application gathers and transmits $\approx 53\mathrm{MByte}$ of raw data during the one hour of our test.

L_{IO} shows the average disk input output in $\frac{\mathrm{kb}}{\mathrm{s}}$ caused by the database writing operation. This was measured via `pidstat` command.

L_{net} shows the average bandwidth used. We obtain that value with the command `nload`. We run our test in the university network and therefore have additional external network load. However before each test, we observed the additional network load and as it was always smaller than $1\frac{\mathrm{kbytes}}{\mathrm{sec}}$, we neglected it.

To put L_{IO} and L_{net} in perspective: In our benchmark we transfer $10.000\frac{\mathrm{bytes}}{\mathrm{sec}}$ from the microcontroller to the single-board computer.

Before each test, we restart both the Banana Pi and the server. We then erase the database folder on the server and give both systems $\approx 5\mathrm{min}$ to settle. Then we turn on the function generators, log in to the single-board computer and start the Database Writer software for the currently active database. The actual benchmark begins with starting the Receiver Software.

We let the system gather data from the function generators for 60 minutes. The performance data detailed in Section V is gathered by two scripts running on the single-board computer and the server during the test.

VI. RESULTS

Table IV shows our results. Figure 3 visualizes the data in relation to the maximum values in respective to each criterion.

The Bulk implementations of MariaDB and MongoDB are able to surpass all other databases in regard to server processor usage. InfluxDB required the least CPU usage when only regarding individual implementations. All implementations could handle the high data rate, however the rate of the MariaDB individual implementation was fluctuating in tests. RAM usage of the InfluxDB components were the lowest. Nonetheless, even the utilization of MariaDB - the database with the highest memory usage - was absolutely seen so low that it may not be relevant. The usage and activity of the disk was significantly higher when using MariaDB compared to the others. InfluxDB and the bulk implementation of MongoDB got by with the least amount of disk usage.

To directly compare all our candidates we calculate a combined score by weighing the parameters. In a first step we set the values of each column in Table IV in relation to the columns maximum, so that we compare the relative performance. In the next step, before we add them up, we assign each parameter a weighting.

Since we find that the CPU is the most important parameter, we give it a weight of 2 on server and as resources on client are limited it is weighted with 2.5 there. In absolute terms, the RAM usage on server and client was very little and therefore we weight it with 0.25. For IO we used a SSD, when using a HDD, IO usage could pose a larger problem and therefore it is weighted with 2.5. As the disk usage is already correlating with IO, we weight it with 0.5 so that the impact of the disk results is in a decent relation to the other component results. On difficult places, network-bandwidth could be limited, potentially a data logging application could be connected wirelessly, so we weight it with 1.5.

Lastly we take the subjective difficulty of our implementations into account. We grade on a scale from 5, most difficult to 1 easy and weigh this parameter with 0.2. The individual rating is determined by the explained experience with the client implementation described in Section IV.

The weights are multiplied with each criterion and aggregated, resulting in points. High point values indicate high resource usage according to weighting. For scoring we "invert" the points with the formula

$$Score = max(Points) - Points$$

and normalize the scores relative to the maximum score.

Figure 4 shows the scores without aggregation, where the components forming the final results are outlined. For the final ranking shown in Table III we aggregated all scores by adding the non normalized values.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-669-9

TABLE III
SCORED RANKING

Implementation	Rank	Score
MariaDB – bulk	1	73
MongoDB – bulk	2	70
InfluxDB	3	64
MongoDB – individual	4	54
MariaDB – individual	5	3

VII. CONCLUSION AND FUTURE WORK

Generally speaking, MongoDB is a good choice. Due to the open structure, additional information can also be stored if required and it performs quite well on both implementations.

However, the optimized MariaDB implementation that saves data in bulks ranks first, as it consumed the least amount of CPU and network.

On the contrary, if a saving of individual values is desired, MariaDB is the last one and InfluxDB is the best in this case.

Our ranking is weighted after the use case described in Section II. When IO is much more important than CPU, MariaDB is potentially lesser ranked, as it had the most IO usage in both implementations.

The paper only covered the writing of databases. Later on, we want to measure the reading and querying performance in another paper. By ensuring that each database uses an index for time, we have already established a good basis for it. Nevertheless, we expect different winners in each test category for the readings.

REFERENCES

[1] D. Wang, J. Liu, and R. Srinivasan, "Data-driven soft sensor approach for quality prediction in a refining process," IEEE Transactions on Industrial Informatics, vol. 6, no. 1, Feb 2010, pp. 11–17, URL: https://dx.doi.org/10.1109/TII.2009.2025124 [retrieved: 2018-08-14].

[2] G. Köksal, İ. Batmaz, and M. C. Testik, "A review of data mining applications for quality improvement in manufacturing industry," Expert Systems with Applications, vol. 38, no. 10, 2011, pp. 13 448 – 13 467, URL: http://www.sciencedirect.com/science/article/pii/S0957417411005793 [retrieved: 2018-08-14].

[3] F. Chen, P. Deng, J. Wan, D. Zhang, A. V. Vasilakos, and X. Rong, "Data mining for the internet of things: Literature review and challenges," International Journal of Distributed Sensor Networks, vol. 11, no. 8, 2015, p. 431047, URL: https://doi.org/10.1155/2015/431047 [retrieved: 2018-08-14].

[4] J. Lee, H. D. Ardakani, S. Yang, and B. Bagheri, "Industrial big data analytics and cyber-physical systems for future maintenance & service innovation," Procedia CIRP, vol. 38, 2015, pp. 3 – 7, proceedings of the 4th International Conference on Through-life Engineering Services.

[5] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient b+-tree based indexing of moving objects," in Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, ser. VLDB '04. VLDB Endowment, 2004, pp. 768–779, URL: http://dl.acm.org/citation.cfm?id=1316689.1316756 [retrieved: 2018-08-14].

[6] S. Acreman, "Top 10 time series databases," URL: https://blog.outlyer.com/top10-open-source-time-series-databases [retrieved: 2018-08-14].

[7] A. Bader, O. Kopp, and M. Falkenthal, "Survey and Comparison of Open Source Time Series Databases," Datenbanksysteme für Business, Technologie und Web - Workshopband, 2017, pp. 249 – 268, URL: http://btw2017.informatik.uni-stuttgart.de/slidesandpapers/E4-14-109/paper_web.pdf [retrieved: 2018-08-14].

[8] D. Namiot, "Time series databases," in DAMDID/RCDL, 2015, URL: https://www.semanticscholar.org/paper/Time-Series-Databases-Namiot/bf265b6ee45d814b3acb29fb52b57fd8dbf94ab6 [retrieved: 2018-08-14].

[9] S. Y. Syeda Noor Zehra Naqvi, "Time series databases and influxdb," Studienarbeit, Université Libre de Bruxelles, 2017, URL: http://cs.ulb.ac.be/public/_media/teaching/influxdb_2017.pdf [retrieved: 2018-08-14].

[10] A. M. Castillejos, "Management of time series data," Dissertation, School of Information Sciences and Engineering, 2006, URL: http://www.canberra.edu.au/researchrepository/file/82315cf7-7446-fcf2-6115-b94fbd7599c6/1/full_text.pdf [retrieved: 2018-08-14].

[11] solidIT consulting & software development gmbh, "DB-Engines Ranking," URL: https://db-engines.com/en/ranking [retrieved: 2018-08-14].

[12] "MariaDB homepage," URL: https://mariadb.org/ [retrieved: 2018-08-14].

[13] solidIT consulting & software development gmbh, "DB-Engines Ranking of Relational DBMS," URL: https://db-engines.com/en/ranking/relational+dbms [retrieved: 2018-08-14].

[14] "MongoDB homepage," URL: https://www.mongodb.com/what-is-mongodb [retrieved: 2018-08-14].

[15] solidIT consulting & software development gmbh, "DB-Engines Ranking of Document Stores," URL: https://db-engines.com/en/ranking/document+store [retrieved: 2018-08-14].

[16] "InfluxDB homepage," URL: https://www.influxdata.com/time-series-platform/influxdb/ [retrieved: 2018-08-14].

[17] solidIT consulting & software development gmbh, "DB-Engines Ranking of Time Series DBMS," URL: https://db-engines.com/en/ranking/time+series+dbms [retrieved: 2018-08-14].

[18] "UDP Configuration of InfluxDB," URL: https://github.com/influxdata/influxdb/tree/master/services/udp [retrieved: 2018-08-14].

TABLE IV
TEST RESULTS

	L_{CPU_S}	L_{mem_S}	L_{CPU_C}	L_{mem_C}	L_{io}	L_{disk}	L_{net}	Difficulty
MariaDB – individual	36.6 %	201 kB	7 %	7.8 kB	7396 kB/s	1.27 GB	1.41 Mbit/s	5 (Very high)
MariaDB – bulk	0.6 %	172 kB	2.2 %	8 kB	378 kB/s	240 MB	0.37 Mbit/s	
MongoDB – individual	3 %	758 kB	6.6 %	12 kB	117 kB/s	204 MB	0.78 Mbit/s	4 (high)
MongoDB – bulk	1 %	209 kB	3 %	12 kB	56 kB/s	86 MB	0.63 Mbit/s	
InfluxDB	15.4 %	178 kB	4.9 %	2 kB	81 kB/s	89 MB	0.87 Mbit/s	1 (Very low)
Weight	2	0.25	2.5	0.25	2.5	0.5	1.5	0.2

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-669-9

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
$L_{CPU_{Server}}$ $L_{mem_{Server}}$ L_{IO} $L_{CPU_{Client}}$ $L_{mem_{Client}}$ L_{net} L_{disk}
MariaDB_Single MariaDB_Bulk MongoDB_Single MongoDB_Bulk InfluxDB

Figure 3. Overview of all Benchmark Values (normalized to respective Maximum)

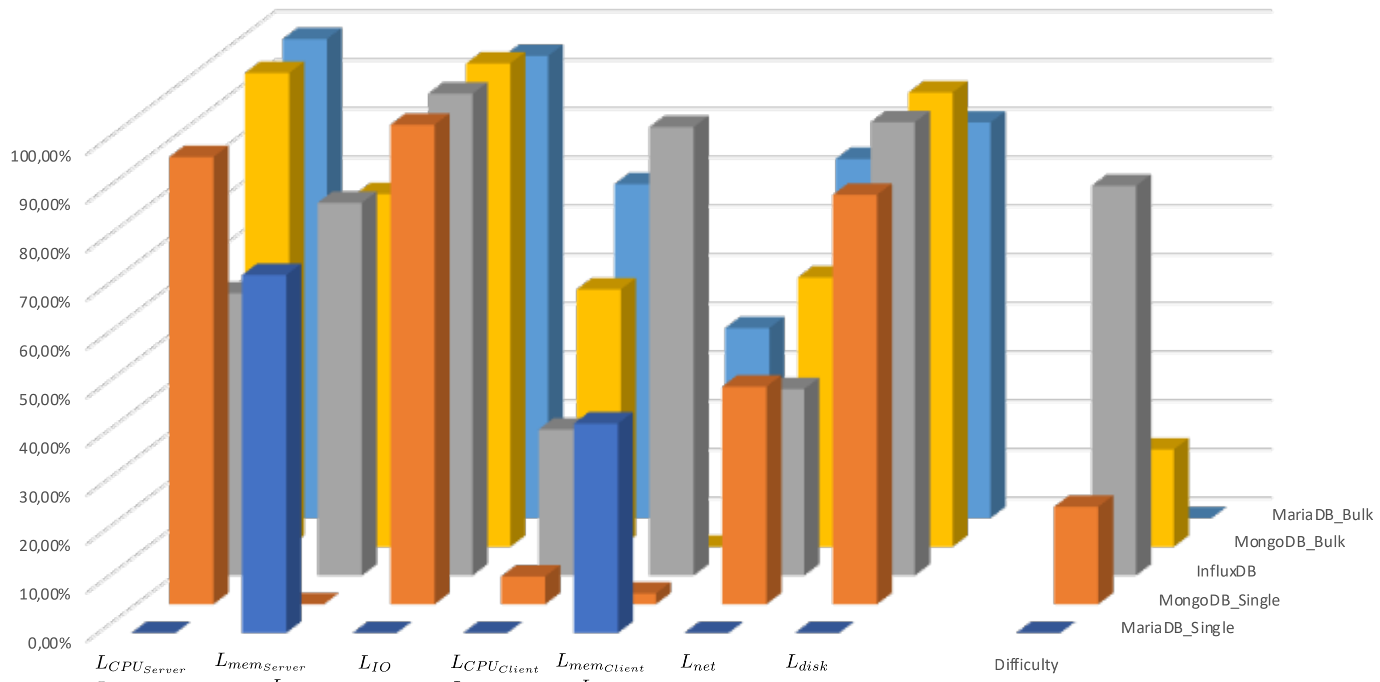

Figure 4. Weighted scores (normalized to maximum score)

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-669-9