Comparative Evaluation of Database Read and Write Performance in an Internet of Things Context

Denis Arnst*,
University of Passau, Passau, Germany
Email: *arnst@fim.uni-passau.de
Thomas Herpich†, Valentin Plenk‡, Adrian Wöltche§
Institute of Information Systems at Hof University, Hof, Germany
Email: †thomas.herpich@hof-university.de, ‡valentin.plenk@iisys.de, §adrian.woeltche@iisys.de

***Abstract*—In the context of the Internet of Things (IoT), there is the need to manage huge amounts of time series sensor data, if high frequency device monitoring and predictive analytics are targeted for improving the overall process quality in production or supervision of quality management. The key challenge here is to be able to collect, transport, store and retrieve such high frequency data from multiple sensors with minimum resource usage, as this allows to scale such systems with low costs. For evaluating the performance impact of such an IoT scenario, we produce 1000 datasets per second for five sensors. We send them to three different types of popular database management systems (i.e., MariaDB, MongoDB and InfluxDB) and measure the resource impacts of the writing and reading operations over the whole processing pipeline. These measurements are CPU usage, network usage, disk performance and usage, and memory usage results plus a comparison of the difficulty for the developers to engineer such a processing pipeline. In the end, we have a recommendation depending on the needs, which database management system is best suited for processing high frequency sensor data in an IoT context.**

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

I. INTRODUCTION

Internet of Things (IoT), Industry 4.0 (I4.0), ... these current buzzwords and many more refer to data-based management strategies, i.e., a new way of processing big and smart data. While many papers propose data-mining algorithms to extract commercial value from a database or a data lake (e.g. [2]–[4]), less address the computing requirements of such algorithms in combination with the systems writing or reading the data. The need for such an evaluation arises, because of the urge to become more and more precise in the technological advancement, the quality management has to keep up for being able to minimize defective products. We call this the industrial data analytics process. Without proper systems for managing high frequency data of dozens or hundreds of different sensors, it is for example nearly impossible to detect electrical distortions in the power supply of precision tools for producing highest quality engine parts. Having possible candidates of systems for managing such data without having to pay enormous sums of money, allows to incorporate a new level of quality management and even predictive analytics in new fields of technological systems, e.g., in production, surveillance, smart home, security business or economics that would otherwise be too expensive or too slow.

In this paper, we therefore evaluate the computing requirements on all parts of an IoT and I4.0 sensor system in a benchmark scenario for being able to recommend one or more systems depending on the needs of the industrial data analytics process [5]. The benchmark scenario is based on one of our research projects, where we collect and store $\approx 4\frac{\mathrm{GB}}{\mathrm{day}}$ of sensor data. This does not sound much, but within a year of measurement, this can grow to $\approx 1.5\frac{\mathrm{TB}}{\mathrm{year}}$, which is a lot for a traditional database system. This is why we need to focus on a small impact of resource usage for being able at all to accomplish the goal of high frequency data management.

In our scenario, for simulating this big picture, we first store generated time series sensor data to a database, which simulates the acquisition, and then we retrieve parts of the data for simulating the analytics part. We think that typical sensor acquisition computing resources have only low performance when sitting nearby the sensor (i.e., integrated circuits only made for reading and sending the sensor data). For the data to reach the database server, we believe that there might be cases where no cable connection is set up but wireless data transportation could be installed. Although the server itself is normally well suited concerning its computational capacity, the low resource impact on writing is an important goal. With reading the data, we think most work is still on the database server, which has to find and accumulate the needed data points. The client that reads the data might be a normal desktop computer or laptop, but also could be a smartphone or tablet, so the performance impact on the reading client side also is not to be left out. As the database server is the primary key in performance here, since all the writing and reading work is done there, our benchmark mainly focuses on the different database system servers.

Of course, the typical database servers can be tuned towards high performance reading or writing of data, but often not towards both at once. This is especially the case, when a fast retrieval is more important than a fast storage, for example with time series data in predictive analytics. When comparing different sensor readings at different points in time, relational databases rely on B-tree indexes that allow 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. [6] 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 exist (e.g. [7]–[10]).

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.

Moreover, our experience is that companies rely on systems they already know and that have been proven to work stably. Additionally, the developers often have a long experience in standard database systems such as relational databases but not in specialized databases like the mentioned time-series databases. We believe that there are many installations of traditional databases that are considered for the industrial data analytics process instead of choosing a specialized tool, because of the risk that comes with a software that has not been tested and validated by the company yet. Therefore, we also consider the implementation difficulty of specialized databases in comparison to traditional systems, and we also develop "sophisticated" algorithms for getting more performance out of these systems, which would not be available with rather "naive" implementations.

In Section IV, we describe different implementations we developed for writing to the databases and reading from them. We evaluated several ideas from [11], such as time series grouping, which is such a more sophisticated approach.

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. We believe that the infrastructural impact is most important for deciding which database is best suited for a specific IoT or I4.0 scenario. Section VI discusses and explains the findings. Section VII summarizes the paper and gives recommendations for different needs in an industrial data analytics process.

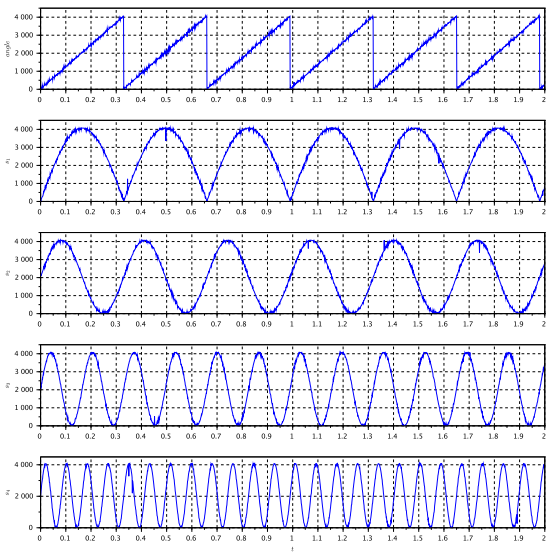

Figure 1. Simulated Test Data: Machine Angle (top) and four Data channels

II. Benchmark Application

One of our current research projects is using predictive maintenance for analyzing data stemming from a complex tool operating within an industrial machine tool. The tool is equipped with 13 analog and 37 digital sensors recording mechanical parameters during operation. The machine opens and closes the individual tool components $\approx$ 3 times per second, i.e., 3 working cycles per second.

Our data-gathering application records ≈ 300 samples per cycle from the sensors and stores them in a database for later analysis. Basically, it stores $1000\frac{\text{samples}}{\text{sec}}$. This keeps the software structure simple and universal and requires few computing resources on the system writing the data.

For our analysis on the client side we need to retrieve all samples in one cycle. This does not correspond to the structure of the database. Our client software maps the time-series data to machine cycles by using one of the analog input channels as abscissa. This channel, shown as top channel in Figure 1, represents the rotatory angle of the machine tool's main drive. One rotation corresponds to one machine cycle. The time-series data of this channel is a sawtooth wave. The period of this wave is equal to the cycle time.

We use this scenario of writing and reading sensor data as a base idea for this paper to benchmark the industrial data analytics process. For the tests in this paper, we substitute the actual instrumentation and signal conversion with a small program that creates the sawtooth wave and four sine waves. For more realistic data, we add some random noise. This simulates the uncertainty of the sensor readings due to the sensor resolution and electrical distortion. This prevents the optimazation of the algorithm by hard-coding a machine cycle duration, which would be possible if the data was completely deterministic. Our reading algorithm, which searches for the measured machine cycle by comparing the noisy abscissa data, can be seen as very realistically usable this way.

Figure 1 shows the simulated data. In total, we simulate 5 analog channels with a resolution of 12 bit (represented using 2 bytes) and a sample rate of $1000\frac{\text{samples}}{\text{sec}}$. This corresponds to a data rate of $10000\frac{\text{bytes}}{\text{sec}}$ of simulated data at the sensor. Later, we add timestamps for each sample with millisecond resolution, which increases the amount of data sent to the database server.

Figure 2 shows the flow of the data through our setup. The reason for this data flow is that we think this exactly matches a typical industrial environment, where sensors gather measurements (Data-Source), a piece of small software transmits this information to a database server (Database Writer), and a monitoring service reads the data from the database (Database Reader).

In our setup, the Banana Pi single board computer is running two separate applications: the first simulates sensor data for replacement of real sensors. The second 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 ≈ 16 kByte, data is lost. This is similar to reading a real sensor which does not cache the data or waits for another application to read the data before it overwrites its internal memory with new measurements.

Figure 2 then shows the database specific applications with gray background. These writer and reader applications are implemented each for InfluxDB, MariaDB and MongoDB, because every database has it's own application programming interface. They use high-level libraries as far as possible to access the database.

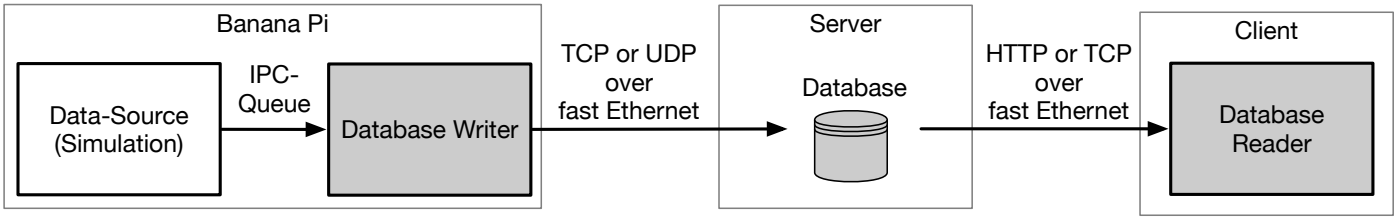

Figure 2. Block Diagram of the Test setup

For the transmission of the data, in the case of InfluxDB, the writing application uses a very fast and easy to access UDP interface. The advantage of this interface is that it is sufficient to send a simple concatenated string that is then interpreted by the InfluxDB server system for commitment of the data. For reading InfluxDB, we use the HTTP interface with an high-level library, which is based on TCP packets. This is because the UDP interface is solely for writing and the HTTP interface is the recommended way of querying data from InfluxDB. MariaDB and MongoDB both use specific TCP connections for the writing and reading of the data.

Concerning the hardware used in our setup, the single board-computer is a Banana Pi M3. This system uses an ARM Cortex A7 (8 x 1.8 GHz) with 2 GB DDR3-RAM and has Gigabit Ethernet on board.

The three databases are run on a dedicated server with an Intel Core i5-4670 CPU (4 x 3.4 GHz), 16 GB DDR3-RAM (4 x 4 GB) and a 256 GB SSD on SATA 3.1 (6.0 Gb/s), also with Gigabit Ethernet. This server is different from the server used for the measurements in [1], because, unfortunately, the old hardware broke.

The database reader is run on a dedicated desktop computer with an Intel Core i7-4785T (4 x 2.2 GHz with SMT), 16 GB DDR3-RAM (4 x 4 GB) and Gigabit Ethernet.

Concerning the software, the simulation application as well as the InfluxDB and MariaDB writing and reading applications are written in C, the MongoDB writing and reading applications are written in C++. We decided to use C-based languages for performance reasons, so that the application benchmark can be run with native platform speed.

All systems are running a Linux CentOS 7 without X.org server and with the same level of system updates. They are linked via a Gigabit Ethernet switch for non-blocking network IO.

III. Choice of databases

Various publications like [8] or [12] list a huge number of different databases. They distinguish three categories relevant for us: 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 focused on mature (stable releases available for at least three years) and free software with options for enterprise support. We mainly consulted the database ranking website [12] as a basis for selecting databases for our comparison.

As a representative RDBMS we selected the open source database MariaDB [13]. It is a fork of the popular MySQL database and widely used in web applications and relational scenarios. [14] lists MySQL and its more recent fork MariaDB combined as top RDBMS.

We selected MongoDB [15] 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 [16].

As TSDB we chose InfluxDB [17], which claims to be highly specialized in sensor data. This claim is confirmed by the score in [18].

We believe that our choice stands for all major and currently important and well-known types of database systems in the IoT and I4.0 industrial data analytics process.

IV. The different Implementations

As seen in Figure 2 the database writers commit the sensor data to the databases (see Section IV-A). The database readers (see Section IV-B) read the written sensor data and then calculate a sum for one machine cycle of sensor data, thus simulating light analytical processing.

Each writer and reader is written for each database system, so that we have at least three writing and three reading applications for comparison purposes. Moreover, because of the architecture of a database server (NoSQL vs. RDBMS vs. TSDB), we also had to implement different ways of storing the data. Additionally, during early development and because of recommendations at MongoDB, we decided to write an additional two variants in data storage and transport for both MariaDB and MongoDB to optimize a degradation behavior that occurs with what we call a ”naive” implementation. This sums up to five different writing and five different reading applications for our benchmark. Later, we will introduce a sixth and seventh reading variant of the InfluxDB application for being able to compare the reading better to the two variants of the other database systems.

The difference is that in theoretical database lecture, we are taught to normalize data for being able to freely filter and combine without having redundant entries. This typical (or ”naive”) way of implementing database architectures in our case leads to a performance degradation because of the high frequency of values. The trouble lies in the frequent updates of index structures and files on the hard disk. As a typical hard disk drive (HDD) is able to perform $\approx$ 100 input or output operations per second (IOPS), a traditional, normalized approach would not be able to handle such high frequency inserts of 1000 sensor reads per second. For still being able to measure and compare this traditional approach, we had to use solid state drives (SSD), which can reach about $\approx$ 100K of IOPS today, which in our scenario is enough for not being the bottleneck.

Nevertheless, by using what we call ”advanced” implementation strategies, optimized towards this special problem of high frequency inserts of data, we can reduce the number of necessary IOPS of the database server and improve the performance for having more capacity left to address increasing workloads.

The ”advanced” approach caches all the sensor data of one second (in our case, which could be more in reality, if needed) and then only writes our $5 * 1000$ data points in one operation to MariaDB and MongoDB. Interestingly, InfluxDB does the same on the server side and caches the data until a larger block is filled, before it writes the data down to the disk. Because InfluxDB does this itself (as it is specialized for time series data), we only had to write one version for this database server.

The aggregation of the sensor data in case of MariaDB und MongoDB, which theoretically should be a lot faster than the single insertion of data, unfortunately, has one big flaw that has to be considered before following this approach: As the data has to be cached in the writing application, it is not available on the reading side, because it is not stored in the database, yet. So when the monitoring application needs live view of data, an additional pipeline from the sensor read to the live monitoring system has to be established. As we write the data to a database first, which allows advanced database querying techniques not available in client software without additional libraries, but does not allow live monitoring, we decided that this flaw has no impact in our case. Moreover, as our reading algorithm needs complete machine cycles for analysis, our scenario is not in the need of having live access to the sensor data. As we also believe that many analytical applications have no advantage of live data and still work fine when the data is available only one or more seconds later, this flaw is not further discussed in this paper.

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 achieve optimal database performance results. This is also a reason for the usage of C and C++ as underlying languages, because this allowed us to tune our algorithms towards optimal performance, which would not have been possible, for example, with a language that has no pointer arithmetic or that uses a garbage collector. It could, of course, also be possible, to compare the optimized implementations in C and C++ against implementations without optimizations like prepared statements, and others, but as this paper is about different database processing pipelines and wants to max out the performance, not lay out all possible code optimization techniques, this is not covered here.

A. Database Writer and Database Structure

Now, every millisecond, the simulator (i.e., sensor) writes a new measurement value into its local memory, overriding the previous measurement. The sensor uses an internal clock for this, which wakes up every millisecond. Our simulator uses the `clock_nanosleep` function for simulating this behavior of updating a measurement each millisecond, like a real sensor could do.

The database writer application running on the single-board computer now reads that new measurement from the simulator (i.e., sensor) over the IPC queue, five values per millisecond, so that we have received 1000 measurements per second with five sensor measurements each, at the end of a second. As the database writer is running on a computer that has a system clock being synchronized to the real time (via network time protocol), compared to a real sensor that might not have a synchronized system time or no time at all, the timestamp for the datapoint is added by the database writer. Listing 1 shows the structure of the datapoint that is then sent to the database: It contains the added timestamp and the set of the five digital values read from the sensor (i.e., simulator). The timestamp added has a resolution of one nanosecond (for further adjustments to even higher frequencies than 1000 values per second) and uses $8 + 4 = 12$ bytes of memory for representing the second as `long` and the relative nanosecond part of the corresponding second as `int`. The digital values are represented as 16-bit (2 byte) integers. Thus one datapoint uses $12 + 5 * 2 = 22$ bytes of memory in sum.

Listing 1. One datapoint

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

1) MariaDB – Individual datapoints: This is a straightforward implementation of the data structure (we called it ”naive” earlier). We sequentially store each datapoint as five rows in the database table, so we have a normalized table structure (i.e., each measurement gets its own row). This results in a high rate of operations on the database ($1000 \frac{\text{writes}}{\text{second}}$). The impact could be even higher, if we had written each measurement in a single commit, but we aggregated each sensor readout (with five values each) in one commit, so that five rows are inserted per commit. This means that we have 5000 $\frac{\text{rows}}{\text{second}}$ of sensor data. Moreover, it means that the index also is updated $1000 \frac{\text{times}}{\text{second}}$ and the disk probably has a four to five times higher load, because it has to write the database log, the data itself, the index update, file system table updates and maybe also file access and or modification times. For this reason, this normalized approach is not suited for a traditional hard disk drive.

Table I shows the structure of the data, which is based on the `data_point` structure. A compound index is set on *second* and *nanosecond* for later retrieval of single measurements in our reading benchmark. *number* describes the index of the sensor, so in our test, a value from 0 to 4 for the five different sensor values, *measurement* is 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 C implementation of the algorithm based on `libmariadb` uses prepared statements, struct data binding and a single commit for five rows per sensor read for higher

performance. These performance optimizations, the explicit transaction preparation and commitment and the manual creation of tables necessary for a relational database (not needed in the other database systems) make the MariaDB code the largest and most complicated of all our implementations.

2) MariaDB – Bulk Datapoints: As mentioned in Section IV and in the preceding paragraph, this implementation collects all datapoints for one second in memory (i.e., same `int64_t s` value), creates one JSON document per second and writes this document out as one row per second. Thus, we can store the data in bigger units, which reduces the load dramatically. Instead of $1000\frac{\text{writes}}{\text{second}}$ with 5000 rows per second, we now only have $1\frac{\text{write}}{\text{second}}$ with 1 row per second and only 1 index update per second.

For the storage of the JSON document, the table structure is a little bit different. In MariaDB, the JSON field is an alias for longtext field. Yet, the specialized JSON query commands in MariaDB work for such fields, which could allow to later query within the JSON data directly, though we did not use this approach in our reading benchmark, because of other reasons discussed later. Table II shows the used structure with this approach. *second* has an index, again for faster retrieval of the data later in our reading, *size* contains the length of the text in the JSON field, and *measurements* is the mentioned JSON document, built in linear time according to the example in Listing 2. The JSON document now contains the nanoseconds with the related measurements according to the `data_point` structure. We did not save the amount of measurements (five) within the JSON document, because we have a defined `data_point.measurements` size of five. This means that we can save this space in our scenario, as we will never have fewer or more than five measurements per sensor read per millisecond in our datapoints.

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

```
{
 "measurements": [
   {"ns":346851124,"m":[389,792,602,315,552]},
   {"ns":346933204,"m":[516,794,634,317,559]}
   ...
]}
```

The difficulty of this adaption is similar to the original, individual approach, but in one detail is quite 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 in a real-world scenario), 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. As dynamic arrays copy their memory contents multiple times while growing, this theoretically reduces performance. But as we use a global string variable and as the length of the datapoints as string is always very similar, the dynamic string normally only grows during the first iteration and then is not reallocated anymore in subsequent calls. This is why we can state that the JSON document is built in linear time during the benchmark.

3) MongoDB – Individual Datapoints: As a document-orientated database, MongoDB allows for flexible schemata, which allows us to leave out any schema creation. Data is internally organized in BSON (Binary JSON) documents, which are in turn grouped into 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. This is like a mixture of the individual MariaDB and the bulk MariaDB approach, just with a JSON document per sensor read, so with 1000 JSON documents per second.

The database supports setting an index on a field of a document, so to support further searching of measurements, we set an index on time as we have in MariaDB. With such a structure, similar to the individual MariaDB approach, numerous documents are created per second. After each document has been added, the index also needs to be updated, which results in a similarly high computational effort as with the individual MariaDB approach.

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, as there is no schema for clear orientation. Also the need to link two libraries creates additional effort.

4) MongoDB – Bulk Datapoints: As already stated in Section IV-A2, we can store a collection of datapoints at once. In MongoDB, we can implement this with the structure shown in Listing 3.

Listing 3. Datapoints in bulk

```
{
  "time" : ISODate("2018-02-12T19:56:49Z"),
  "measurements" : [
    { "time" : ISODate("2018-02-12T19:56:49.135Z"), "sensors" : [ 0, 0, 0, 9, 347 ] },
    { "time" : ISODate("2018-02-12T19:56:49.136Z"), "sensors" : [ 0, 2, 4, 10, 351 ] },
    ...
  ]
}
```

The time value of the top-level document has again a precision of one second. This document holds all datapoints sampled during this second in an array, similar to the bulk MariaDB variant. Every nested document contains the exact time of its measurement and the actual sensor-values. This is similar to the MariaDB JSON document, though the time has no extra field and the nanoseconds have a defined format (ISODate) that bloats the document. Of course, we thought about using the nanosecond solely, like with MariaDB, but MongoDB recommended the general use of ISODate, so we followed this recommendation.

With this approach, similar to MariaDB, 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. Although MongoDB can retrieve values directly within a JSON document, similar to MariaDB, the document first has to be loaded and parsed, before the database server can find the queried value. However, because of the high increase in write throughput, we accept this drawback.

Similar to the MariaDB approach, the application creates a document for a whole second and fills it until the second has passed. Then it sends the document to the database server once per second.

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

5) InfluxDB: As a time-series database InfluxDB has a strict schema design we have to follow. 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-value pairs called fields. Timestamps have an accuracy of up to one nanosecond and are indexed by default. 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). This is how the protocol is built up: The first string is the name of the measurement, here simply *measurement*. Subsequently follow the key-value pairs 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 chose the UDP protocol, which is also supported. We believe that the overhead is smaller when using the UDP protocol, because HTTP is a very verbose protocol, especially when sending small requests very frequently like in our case. So in this case, the data is composed into the 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 straightforward 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, when small overhead is desired.

Of course, the choice of UDP has the probability of data loss, which is acceptable in our use case, because we have very high frequency data and the loss of single measurements could be compensated, for example by interpolation, or just be ignored, when reading. For enabling the UDP service of InfluxDB, the OS was configured corresponding to the information provided by InfluxData [19]. Because we use dedicated computers linked together with a nonblocking switch, we had no measurable UDP loss in our tests. When using a wireless link between database writer and database server, maybe the HTTP protocol should be preferred despite the large overhead.

With InfluxDB, additional implementation variants were not needed, because the scheme is fixed and InfluxDB itself caches, accumulates in bigger units and even compresses the data, before it is written to disk. This is why InfluxDB is the only database system that works out of the box for time series data without additional effort in optimizing the datapoint storage.

B. Database Reader

In our reading part of the benchmark, we want to simulate an interactive monitoring application. We use the database reader application variants to retrieve the data for three randomly selected machine cycles per second. The rate of $3\frac{\text{reads}}{\text{sec}}$ is quite high for an interactive application, where a user selects individual machine cycles for further analysis, but we assume the user clicks very fast and often. A non-interactive application, e.g., condition monitoring for predictive analytics, will process the consecutive cycles at the end of the data set, not randomly selected cycles. As the most recent data might still be cached inside the database or page cache of the operating system, and could also be selected by just jumping to the end of the data set minus a selected range, our random selection is an adequate usage scenario between worst and best case. As we select by time and always have set indexes on the time, the selection should not trigger sequential scans of the data but make usage of our selected database structures.

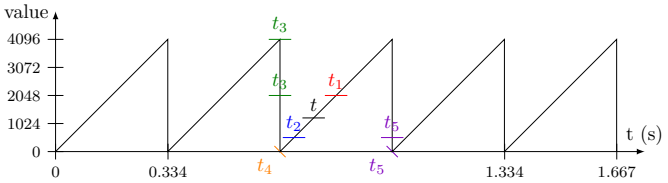

Figure 3. Strategy to find start and end time for a machine cycle

Figure 3 explains our implementation of the reading algorithm. We start at t, which is randomly selected in our benchmark. From there we search forward to t_1, the first point after t where the data of the abscissa-track is bigger than half the amplitude. This value is fixed, as we know the range of the abscissa amplitude. From t_1 we search backward to t_2, the first point before t_1 where the data of the abscissa-track is smaller than an eighth of the amplitude. From t_2 we search backward to t_3, the first point before t_2 where the data of the abscissa-track is (again) bigger than half the amplitude. The start point of our cycle is at t_4 the minimum value of the abscissa-track in the range $t_3 \leq t_4 \leq t_2$. The end point of the cycle is at t_5. We find t_5 by searching forward from t_1 for the first point where the data of the abscissa-track is (again) smaller than an eighth of the amplitude.

This strategy can be implemented by issuing several `SELECT` statements to the databases. In the following we refer to this strategy as "individual".

Alternatively we can simply retrieve data for a longer time span around the time t in question, e.g., $\lfloor t - 1\,\text{sec} \ldots \lfloor t$ for

$t - \lfloor t < 0.5$ and $\lfloor t \ldots \lfloor t + 1\,\text{sec}$ for $t - \lfloor t \geq 0.5$, and then perform the same search operation on the retrieved data in memory. This alternative way is necessary for the write strategies in sections IV-A2 and IV-A4 where the database structure does not allow to retrieve individual samples with high performance. We said that the database allows individual selection with JSON query commands, but as the data has to be loaded and parsed in either way, we could just send it to the client application and work with the aggregated second on the reading side, again, just as when we aggregated on the writing side. This leads to the five (or seven with the InfluxDB alternatives) different reading applications, so that we have a direct comparison to the five writing applications. Moreover, we believe that the direct queries are slower, for example in MariaDB (see Section IV-A2), because for the selection of several time points in one document, the data has to be loaded several times in sequence for each select statement. When we copy the whole JSON block to the client side, parse it in linear time O(n) and search solely in-memory, we can guarantee a single read on the database side and reduce the possible load on the server, as well as guarantee the linear searching time on the client side. In the following we refer to this strategy as "bulk".

For InfluxDB, while we can retrieve individual datapoints, we also had a bulk variant that read the whole block with all measurements as with the MariaDB und MongoDB bulk variants, and we implemented a "bulk-1" variant of this strategy, which reads only the data in the column corresponding to the abscissa track, i.e, the machine angle, as a block, and reads the data for the other four columns in a second read operation spanning $t_1 \ldots t_5$. The reason for this is that a machine cycle in our benchmark is ≈ 0.33 sec and when we read a block of a whole second, many measurements (of ≈ 3 machine cycles) are transmitted, though only around one third of the data (one machine cycle) is really needed. With our MariaDB and MongoDB bulk variants, we had to transmit the whole block to prevent the database from parsing the data multiple times, but InfluxDB has the flexibility to load blocks of different datapoints. We were able to investigate the transmission bandwidth with this behavior, at least for InfluxDB, too. This is why we have seven reading applications.

V. TESTING

Most applications in our context face limitations in terms of computing power and network bandwidth. For example, IoT sensor devices in the field of smart home are often battery powered and wirelessly connected. In I4.0 context, sensors have no computing power for analytical data processing and they often are connected via proprietary protocols and interfaces. Although servers are often better equipped, the client systems like tablets, smartphones or notebooks have limited computing capacity in comparison. Consequently, when writing data from the sensor to the server, in our scenario, we measure the load on the single board computer, the load on the server, and the network load. When reading data, we measure the load on the server, the load on the client, and the network load as well. With these measurement parameters, we have a good overview of all critical components of the industrial data analytics process.

For the concrete measurement process, we defined the following: The system load on the computers is measured in terms of CPU and memory usage. For this, we created a script, which runs the specified application for 15 minutes, after a warm-up phase of 5 minutes for filtering out cold-start phenomenona like caching data in the operating system page cache, or CPU clock changes due to heat or power usage (especially on the single board computer, which reduces its CPU clock under heavy load).

Before the test run stops 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 returns 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%. Therefore, on our 8-core single-board computer, the absolute maximum value would be $L_{CPU} = 800\%$. On the server with 4 cores, the absolute maximum value is 400%. The client has simultaneous multithreading enabled, so its 4 cores are doubled to 8 threads in the operating system, for an absolute maximum value of 800% instead of 400%.

L_{mem} indicates the 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 server. To determine this parameter, we first empty the respective database completely by removing its data folder. Then we start the database and measure the disk space of the folder before we test. After the test we measure again the used disk space. L_{disk} is then calculated as the difference between the folder size after the test and the folder size before the test with an empty database folder. `du -sh <path>` is used to get the disk consumption of the respective data folder.

To put the results in perspective: Our benchmark application gathers and transmits ≈ 26.4 MByte of raw data during the 20 minutes of our test. This is calculated as the following: We have $5*2$ bytes of sensor data plus 12 bytes of timestamp data per sensor read. We read each sensor 1000 times per second. This sums up to $(5*2+12)*1000 = 22000$ bytes per second, or 22 kBytes per second. We measure 20 minutes (of which 15 minutes are used for the CPU, memory and network measurement). So we have $22\ \text{kBytes} * 60\ \text{s} * 20\ \text{min}/1000 = 26.4$ MBytes, if no sensor value is omitted (which can happen in the UDP InfluxDB writing test).

L_{IO} then shows the average disk input-/output in $\frac{\text{kBytes}}{\text{s}}$ caused by the database writing operation being measured using the `pidstat` command. As we use a SSD on the server, we have raised the hardware limit of the IOPS a lot, compared to a traditional HDD.

L_{net} finally shows the average bandwidth used on the network. We obtain that value with the command `nload`. We run our test in the university network and therefore have additional external network load (for example DHCP packets, ARP requests, discovery services, ...). However before each test, we observe the additional network load for some time, and as it was always smaller than $1\frac{\text{kBytes}}{\text{sec}}$, we neglect it.

To put L_{IO} and L_{net} in perspective: In our benchmark we transfer $22\frac{\text{kBytes}}{\text{sec}}$ from the simulator to the database writers,

but as the raw data has to be packed into UDP or TCP packets, which itself are packed into ethernet frames, the network load has to be larger than the raw data. Especially in case of the RDBMS and NoSQL database, we have an additional communication protocol overhead (like SQL in case of MariaDB), which adds more data to transmit than, for example, the InfluxDB Line Protocol does. Our network bandwidth results show the data plus the overhead for being able to see what the individual applications really need as underlying network throughput.

As our network connection between the tested systems is always $1\frac{\text{Gbit}}{\text{sec}}$, our hardware network limit is high enough for our benchmark. Nevertheless, when using slower or weaker connections (i.e., wireless network), the network bandwidth, that may also differ in time, has to be considered as a hard limit.

Before each test, we reboot the operating systems used in the test. In case of a write test, we then erase the database folder before starting the database. Then we turn on the simulator of the sensor data (in case of a write test), the database server (always), log in to the single-board computer (in case of a write test) or the client (in case of a read test) and start the database writer or reader software for the currently active database. The actual benchmark begins with restarting the database reader or writer after the warm up phase. The database folder of course is not deleted after the warm up phase and is measured over the whole 20 minutes, because this could interfere with the other measurement parameters like CPU, memory and disk IOPS. The reason for this is that index structures like B-trees are never completely filled, but at least 50%. This is for faster insertion in consecutive insert operations, as the tree does not have to be updated for each insert operation. When the database is empty though, the tree is small and has to expand fast during the first minutes, as there are, in absolute numbers, not enough empty buckets for holding all the high frequency values. After the warm-up phase, we believe that the index tree is big enough (as $\frac{1}{4}$ of the data is already written and the tree has enough space left for approximately another fourth, when it is filled between 50% to 75%), so that tree rebalancings do not occur often, and have no noticeable impact on the other measurement parameters anymore. Of course, this example calculation of the tree structure is not exactly precise, but explains very well why we do not clear the database folder again after the warm-up phase.

So in the end, we let the system gather the CPU, network, memory, and disk IOPS data only for the 15 minutes phase and the results from the 5 minutes warm-up phase are thrown away, except for the disk usage, as stated. The performance data detailed in Section VI is gathered by two scripts running on the computers used for the test, which start and stop the applications and measure the resource usages. We test each database server and each writing-reading application bundles sequentially, as testing everything in parallel would interfere with each other's measurements.

VI. Results

Tables IV and V show our results for writing, respectively reading. Figures 4 and 5 then visualize the data in relation to the maximum values for each criterion.

To directly compare all our candidates, we calculate a combined score by weighing the parameters. We think that there are parameters that are more important in an IoT and I4.0 context, than others. We distinguish between critical (weight of 3), important (weight of 2) and normal (weight of 1).

Since we find that the CPU is the most critical and limiting parameter, we give it a weight of 3 on the server. On the client it might even be more limiting due to the Banana Pi's low power design, which also justifies a weight of 3.

In absolute terms, the RAM usage on server and client was very small compared to the available RAM, and therefore we give it a normal weight (with 1).

As already stated, we used an SSD for our benchmarks, which would not be a limiting factor in our tests. Nevertheless, as it is possible to have a server with an HDD, which in that case would be critical, we value the L_{IO} parameter with a mixture of both scenarios as important (so 2).

As the disk usage already correlates with L_{IO} (i.e., both disk parameters), we weight it with 1, so that the impact of the disk results is in a decent relation to the other component's results. Additionally, the disk usage is not critical, as disk space is easy to expand, but for example a high-performance CPU can not easily be doubled to increase processing power for non-parallel algorithms.

In wide areas, network-bandwidth could be a limiting factor, especially when we have wireless connections, for example in smart home scenarios. As in our main context of IoT and I4.0, where we believe it is easier to connect the machines with cables (as they need a large power supply, too), we give the network an important weight (of 2). Wireless scenarios would require an even bigger weight.

Finally, we take the subjective difficulty of our implementations into account. We grade on a scale from 5 (i.e., most difficult), to 1 (i.e., easy), and weight this parameter with a normal weight. The individual rating is determined by the experience with the client implementation described in Section IV. We know that this parameter is not objective, but as we explained in Section I, the developer experience is an important argument in deciding which database is selected. Nevertheless, because we do not want to give a subjective parameter an important weight, we only weight it with 1, as stated.

Based on the gathered result data, we calculate a score according to the following formula, where
$i = \{net, CPU, mem, IO, \ldots\}$ and
$\text{imp} = \{MongoDB_{individual}, MongoDB_{bulk}, \ldots\}$:

$$Score_{\text{imp}} = \left[1 - \frac{\sum_i \left(\frac{L_i}{max_i(L_i)} \cdot weight_i\right)}{\sum_i weight_i}\right] \cdot 100\%$$

In this formula, we first normalize the resource usage to the maximum value for each column in Tables IV and V. Then we sum up the weighted normalized values, and normalize again to the sum of all weights. Lastly we "invert" the value by subtracting it from 1. Thus, the best score is 100%.

Table III shows the aggregated scores for writing and reading. The total score is the average between write- and read-score. For writing, we only have one InfluxDB application, which is the reason for the same score in all three InfluxDB-Writing cells. The ranking and data differs from our

TABLE III. Scored Ranking

	Writing		Reading		Total	
Implementation	Score	Rank	Score	Rank	Score	Rank
MariaDB Individual	15%	7	71%	5	43%	6
MariaDB Bulk	59%	1	83%	2	71%	1
MongoDB Individual	35%	6	50%	7	43%	7
MongoDB Bulk	54%	2	60%	6	57%	5
InfluxDB Individual	39%	3	93%	1	66%	2
InfluxDB Bulk	39%	3	80%	4	60%	4
InfluxDB Bulk-1	39%	3	82%	3	61%	3

previous benchmark in [1], as we repeated all benchmarks in a completely new environment, due to the fact that the previous setup and hardware was not available anymore. In the previous paper, we also used analog circuitry and a signal-generator for generating our sensor data, but as this device was no longer available, for this benchmark, we use a simulator to generate our data. Moreover, we adapt the weights to be more objective, which also has an impact on the ranking. Nevertheless, the order of the ranks is the same as last time, so we think our new setup is quite comparable.

A. Write Benchmark Results

In the write benchmarks, all five implementations show little CPU usage with $L_{CPU_{Server}} \leq 5\%$ on the server and with $L_{CPU_{Client}} \leq 14\%$ a significant but not critical CPU usage on the Banana Pi. MongoDB individual, MongoDB bulk and MariaDB bulk are the least demanding implementations with respect to the server's CPU. MariaDB bulk also is least demanding on the client's CPU. We can see that the server CPU usage is in all cases far from critical, with the client CPU usage being much higher.In cases of InfluxDB and MariaDB Individual, as the CPU usage is a lot higher, this can limit the amount of sensor information or the frequency, that the database is able to process, earlier, than in the other scenarios.

The parameters $L_{mem_{Server}}$ and $L_{mem_{Client}}$ are fairly uniform and non critical. We can see that the server's memory usage is a lot higher in all scenarios, compared to the client's memory usage. This is because of the database servers being complex software products, requiring some memory to operate. Nevertheless, with typical servers having often more than 16 GBytes of memory nowadays, the memory usage is always uncritical. As the client's memory usage in the write benchmarks is only a few megabytes per test run, we can say that this also has no real impact. It is interesting that the InfluxDB UDP Line Protocol writing application uses by far the least amount of memory on the client. This does make sense, because we had no additional libraries involved in this writing application.

L_{net} is also similar for all systems. With $L_{net} \approx 400$ kBit/s MariaDB bulk needs less network bandwidth. We think, this is because the data that is sent to MariaDB in the bulk variant, is transferred as binary data (i.e., bulk data binding) attached to the insert SQL statement. All other variants have either more individual transport operations, or transport the data with more describing variables. For example, InfluxDB always sends the full measurement name and the timestamp as full timestamp with nanosecond precision, while MariaDB bulk splits the timestamp into the second (transmitted once) and the nanoseconds since the last full second. MongoDB also is more verbose because of the JSON-format with the ISODate. We believe that this explains, why MariaDB bulk has the least network demand in writing.

In terms of disk usage, the compressing databases (i.e., InfluxDB) have a clear advantage. With $L_{disk} = 5$ MByte InfluxDB is the best in the test. Nevertheless, the bulk variants also need significantly less disk space than the individual variants. This is easy to explain, because in the individual data point rows, informations like the second are redundantly stored for each row. Moreover, the index structures are larger, as they have to reference more rows than in the bulk variants.

Concerning the implementation difficulty, InfluxDB was the easiest, needed the fewest lines of code, no additional libraries, and no schema to define. MongoDB Individual and bulk was much more difficult, because it needed two additional libraries (with BSON instead of JSON not being a well known data structure) and more effort in creating a schema for the document storage. MariaDB was most difficult in both cases, as in the individual case, the schema and index structure was more complicated (i.e., combined index, which needs attention in the definition because it is sensitive to the order of the definition of the columns), while in the bulk case, the schema and index were easier to define, but additional effort was needed for the buffering of the data up to one second, before the commit. Both MariaDB variants needed the most lines of code and had advanced techniques like prepared statements and bulk data binding applied for optimal performance, which increased the difficulty for the developer.

The way we wrote our data was relatively simple concerning its structure and insertion since we made no preprocessing for our analytical reading algorithm. Especially for InfluxDB this made the implementation very easy and straightforward because we could used the given schema. This led to a more difficult implementation on the reading side.

With our weighting, however, MariaDB bulk is the best ranked database. It needed the least resources concerning the critical CPU usage and the important network usage. Its disk usage was average, though the disk IOPS were worse than MongoDB and InfluxDB. Although its implementation was more difficult, it scored a little bit higher than MongoDB bulk, especially because of the much higher CPU usage of MongoDB. The InfluxDB writer only performed third, because it needed a lot more CPU resources on the server side, due to its buffering and compression mechanism.

B. Read Benchmark Results

In the read benchmarks, MongoDB individual could not keep up with 3 reads per second, as we defined in our scenario. This means that MongoDB individual has a noticeable latency in the interactive application use case. With $L_{CPU_{Server}} \approx 87\%$ (single thread, so maximum is 100% here) it almost blocked our database server, which would further delay parallel queries from other reading clients.

MongoDB bulk was better concerning the CPU usage on the server side, but caused a high load on the client with $L_{CPU_{Client}} \approx 16\%$. We attribute this to the JSON-formatted entries that the client had to parse. As we used a library for parsing, we could not optimize the parsing process towards this special JSON format, like we could do in MariaDB bulk.

This is why MariaDB bulk has the least CPU impact, as it only refers to loading blocks of data and sending them

over without further processing, because the client application parses the JSON data. This might have been different if we had used the server side JSON query mechanisms that we wanted to avoid because of the impact on the CPU usage. MariaDB Individual obviously causes a higher CPU load as it has to look up more individual table entries, and InfluxDB also uses more CPU power than MariaDB, as it has to decompress the data it previously compressed while saving.

For all approaches but MariaDB Individual, $L_{IO_{Server}}$ is below the data rate of $22\ \mathrm{kByte/s}$ that is theoretically required for reading 3 machine cycles per second (with each being $\approx 0.33\ \mathrm{sec}$), i.e., the same data rate as in the write benchmarks. We attribute this to the databases or operating system caching data in memory and the relatively small amount of data stored during our test ($\approx\ 26.4\ \mathrm{MBytes}$). With several hundred gigabytes or even terabytes of data, we might have overcome this problem when randomly selecting machine cycles. In a real-world scenario, where sensor information for several hundreds of sensors is logged over many years, we believe the outcome for the disk IOPS in reading will be different to our benchmark.

The memory footprint $L_{mem_{Server}}$ is comparable to the writing benchmark, which is obvious in case of MariaDB and MongoDB being complex database systems. Interestingly, InfluxDB had much lower memory usage, which we believe lies in the fact that when writing, InfluxDB buffered and compressed the incoming data, while when reading, does not have to buffer anything and uses decompression methods normally being less memory intensive, as no code book has to be built up and held in memory.

Concerning the client memory, InfluxDB individual and bulk loading with all data were best in the memory usage.What surprised us was the memory footprint of InfluxDB, when bulk loading only the data of the abscissa track and then loading the machine-cycle in a second run (bulk-1) was tested. Although the CPU usage was lower, as less data had to be processed on the client side, the memory usage was clearly higher than in the other scenario. Unfortunately, we detected a memory leak in the code that was used for this benchmark, after the testing environment was already shut down and disassembled. We believe that the memory footprint should have been similar to the other two InfluxDB reading benchmarks.

The network bandwidth was much higher when reading than while writing. We think that this lies in the fact that in case of the individual queries, we sent six queries to the databases that each had to run three times per second over the network, and in case of the bulk queries, we always had to load two seconds of data (six machine cycles) three times per second for being able to select one machine cycle in the client application then. InfluxDB performed best in its individual case, having less impact than when writing, which does make sense, as the protocol needed to transmit only the filtered data.

The difficulty rating is that MongoDB was the easiest in this part, because the defined scheme used on writing as well as the ability of MongoDB to help us with the JSON (or BSON) parsing by its libraries, made it adequately difficult to implement. InfluxDB in its individual variant also was comparably easy to implement due to the given scheme and simple InfluxQL language. The bulk variants were more difficult in InfluxDB, because InfluxDB itself did not bulk-store the data, so we had to manually select more data, though InfluxDB could have filtered it for us server-side. We wanted to compare the resource impact of bulk variant reading in InfluxDB, but to sum up, this is not advisable, as the benefit compared to individual loading is small in the resource usage, but the implementation is a lot more difficult. MariaDB was by far the most difficult part, because even sending out the individual queries needed prepared statements and data binding mechanisms, which bloated the code a lot. The MariaDB bulk variant was the most difficult to implement, because we had to write a complete JSON parser ourselves (to be fair, only a highly optimized JSON parser for the underlying data structure was built, no general use parser) for guaranteeing linear parsing time for six in-memory queries, because we believe the that the database itself would have parsed the data multiple times.

With our weighting, InfluxDB individual is the best database for reading. This is because even though the overall CPU usage was higher than in case of MariaDB bulk, the other resource usages (like memory, disk IOPS and network bandwidth) were all lower. Moreover, it was relatively easy to implement, compared to the other variants.

We cannot recommend MongoDB individual, but MongoDB itself already recommends to bulk-write and -load the data. So at least, we can confirm this recommendation. MongoDB bulk, however, was still not very fast. We believe that this lies in the high CPU usage on the client, which is caused by the BSON library we used, which parses the document data in a more general and thus not optimized for this specific use case way.

MariaDB Individual on the hand end is normalized, like theory says is a good practice, but for this specific use case, the memory, CPU and disk usage on the server are quite high in comparison. MariaDB bulk is definitely the recommended way, despite its difficult implementation, as it causes nearly no load on the server and client, especially because we also implemented a highly optimized JSON parser for our own document structure.

C. Summary

In total we find MariaDB Bulk as the best implementations altogether with a score of 75%. However, the MariaDB implementations are quite complex and in our case, they were written by an experienced developer, who knew how to optimize the workload. This means that RDBMS are still capable of working with the types of data we investigated.

With 66% the InfluxDB individual implementations are not far behind, while being much easier to implement. We believe that if the developer is willing to learn a new database system for this use case of time series data management in the context of IoT and I4.0, InfluxDB might be the right choice.

VII. Conclusion

We introduced a complete benchmark set that was inspired by real world scenarios from IoT and I4.0 scenarios and benchmarked a set of three different types of database systems with one or more variants of writing and reading applications to simulate the behavior of high frequency monitoring and predictive analytics in the context of the industrial data analytics process.

While we presented MongoDB as a good candidate in [1], we had to observe a lack of performance in the read

benchmarks. This might be caused by our high-frequency, low data volume read application. MongoDB obviously is better for larger documents with different schemata than for relational structured time series data.

MariaDB bulk scored best in the write benchmarks with a low resource requirement, and it performed well in the read benchmarks, especially because we were able to optimize a lot, like custom JSON parsing. This demonstrates that 'classic' RDBMS are able to keep up with more modern architectures like the one we benchmarked, although they put some strain on L_{IO}, L_{disk} and the developer.

The TSDB InfluxDB individual showed reasonably good write and read performance while requiring the least amount of database know-how. The naive read and write implementations scored quite well, but we think that in very large installations, the compression and server-side buffering mechanisms can lead to an earlier exhaust of resources than when using a RDBMS. If the server is more than capable enough and if the developer wants an easier implementation, then InfluxDB can be recommended.

With our results, individual developers and companies have a base for deciding which database system, and how much effort and resources are needed to implement high frequency monitoring and analytical processing techniques for improving their overall product.

ACKNOWLEDGMENT

This work was supported by the European Union from the European Regional Development Fund (ERDF) and the German state of Bavaria.

TABLE IV. Test Results: Write Benchmarks

	Server			Banana Pi		Infrastructure		Difficulty
	$L_{CPU_{Server}}$	$L_{mem_{Server}}$	$L_{IO_{Server}}$	$L_{CPU_{Client}}$	$L_{mem_{Client}}$	L_{net}	L_{disk}	
MariaDB Individual	5%	170.835 kByte	1840 kByte/sec	9,4%	2.096 kByte	715 kBit/s	88 MByte	5
MariaDB Bulk	1%	160.834 kByte	417 kByte/sec	6,5%	2.232 kByte	395 kBit/s	32 MByte	5
MongoDB Individual	1%	257.166 kByte	71 kByte/sec	13,9%	3.561 kByte	710 kBit/s	58 MByte	4
MongoDB Bulk	1%	137.701 kByte	56 kByte/sec	8,9%	3.607 kByte	617 kBit/s	15 MByte	4
InfluxDB	5%	147.444 kByte	37 kByte/sec	11,7%	284 kByte	785 kBit/s	5 MByte	1
Weight	3	1	2	3	1	2	1	1

TABLE V. Test Results: Read Benchmarks

	Server			Client		Infrastructure	Difficulty
	$L_{CPU_{Server}}$	$L_{mem_{Server}}$	$L_{IO_{Server}}$	$L_{CPU_{Client}}$	$L_{mem_{Client}}$	L_{net}	
MariaDB Individual	7%	208.896 kByte	46,9 kByte/s	0,4%	4.427 kByte	1025 kBit/s	4
MariaDB Bulk	0%	154.544 kByte	5,2 kByte/s	0,6%	3.979 kByte	3520 kBit/s	5
MongoDB Individual	87%	188.737 kByte	3,5 kByte/s	1,4%	5.487 kByte	6750 kBit/s	3
MongoDB Bulk	11%	121.252 kByte	1,6 kByte/s	15,9%	2.564 kByte	3520 kBit/s	3
InfluxDB Individual	8%	40.605 kByte	0,9 kByte/s	1,3%	1.533 kByte	530 kBit/s	3
InfluxDB Bulk	12%	37.272 kByte	1,1 kByte/s	7,0%	1.556 kByte	1898 kBit/s	4
InfluxDB Bulk-1	7%	26.970 kByte	1,2 kByte/s	3,4%	28.979 kByte	1025 kBit/s	4
Weight	3	1	2	3	1	2	1

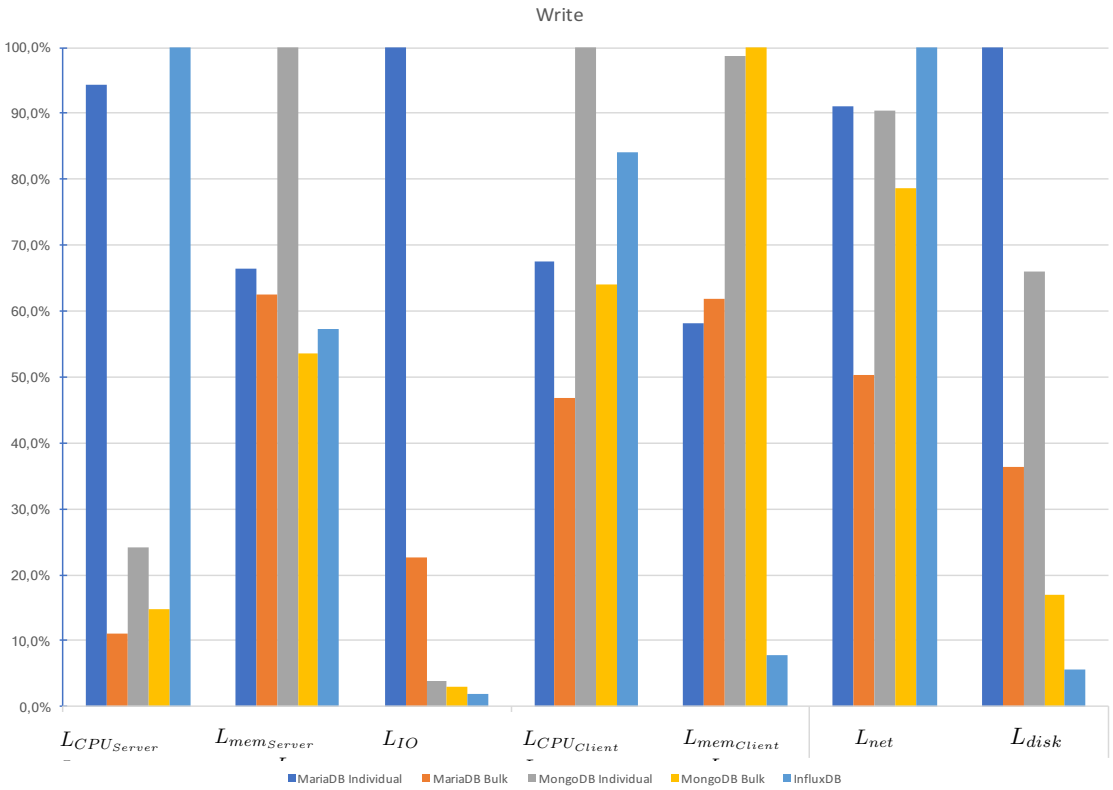

Figure 4. Overview of all Write Benchmark Values (normalized to respective Maximum)

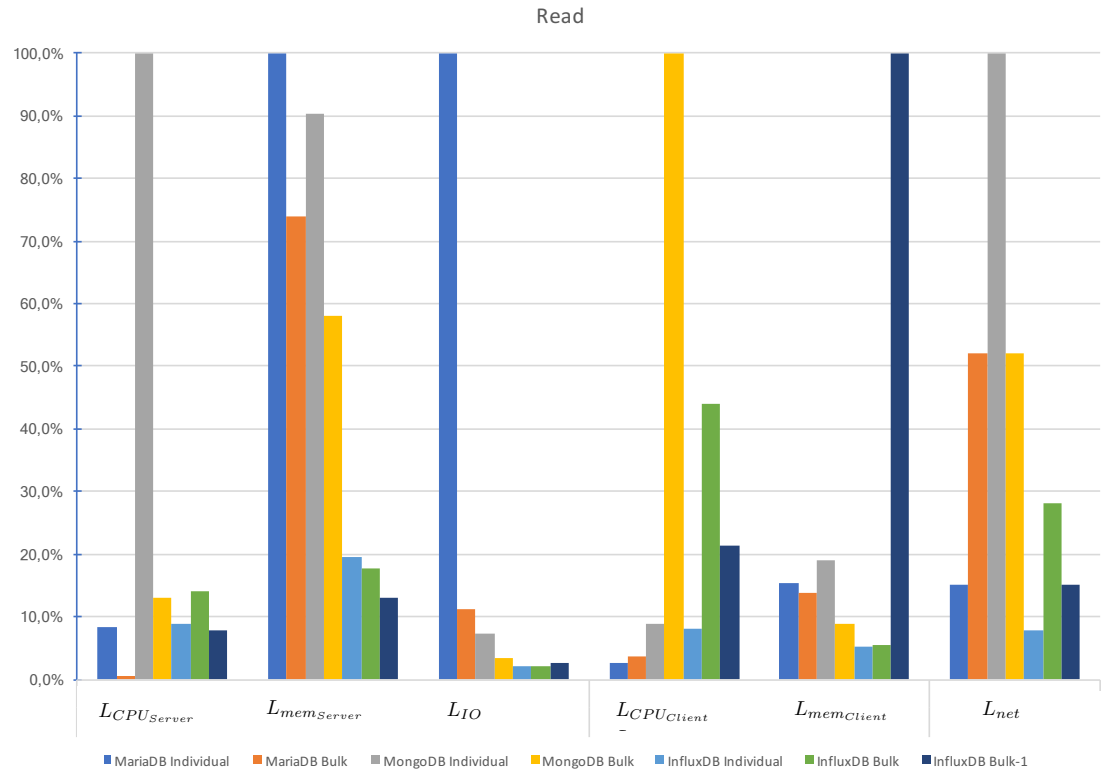

Figure 5. Overview of all Read Benchmark Values (normalized to respective Maximum)

REFERENCES

[1] D. Arnst, V. Plenk, and A. Wöltche, "Comparative Evaluation of Database Performance in an Internet of Things Context," in Proceedings of ICSNC 2018 : The Thirteenth International Conference on Systems and Networks Communications, Nizza, October 2018, pp. 45 – 50.

[2] 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].

[3] 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].

[4] 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].

[5] 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, URL: http://www.sciencedirect.com/science/article/pii/S2212827115008744 [retrieved: 2018-08-14].

[6] 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].

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

[8] 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].

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

[10] 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].

[11] 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].

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

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

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

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

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

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

[18] 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].

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