Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes

Thomas Goldschmidt, Anton Jansen, Heiko Koziolek, Jens Doppelhamer, Hongyu Pei Breivold
Industrial Software Systems
ABB Corporate Research
Ladenburg, Germany and Västerås, Sweden
Email: thomas.goldschmidt@de.abb.com

***Abstract*—Today's industrial control systems store large amounts of monitored sensor data in order to optimize industrial processes. In the last decades, architects have designed such systems mainly under the assumption that they operate in closed, plant-side IT infrastructures without horizontal scalability. Cloud technologies could be used in this context to save local IT costs and enable higher scalability, but their maturity for industrial applications with high requirements for responsiveness and robustness is not yet well understood. We propose a conceptual architecture as a basis to designing cloud-native monitoring systems. As a first step we benchmarked three open source time-series databases (OpenTSDB, KairosDB and Databus) on cloud infrastructures with up to 36 nodes with workloads from realistic industrial applications. We found that at least KairosDB fulfills our initial hypotheses concerning scalability and reliability.**

I. Introduction

To supervise industrial processes, such as power production, oil refineries, or chemical processes, today's industrial control systems store large amounts of sensor data into special databases. These databases then allow human operators to analyze trends in sensor data (e.g., temperature or flow pressure curves) and to optimize the industrial processes.

Industrial monitoring systems are prevalently designed without the ability for scaling with the number of nodes (horizontal scalability) as they are assumed to be statically deployed to a fixed number of servers located within an industrial plant. The advent of cloud computing technologies potentially provides an opportunity to save local IT costs and exploit highly scalable remote IT infrastructures that are not affordable for smaller industrial plants.

While cloud technologies have been successfully used for enterprise applications, their maturity for industrial applications with higher requirements for responsiveness and robustness is largely unknown. Sakr et al. [1] survey numerous large-scale data management approaches specifically for cloud environments, but did not discuss their trade-offs in industrial settings. Wlodarczyk [2] compared four time series database systems that exploit cloud environments. However, this comparison was only done on a conceptual level, no benchmarking with realistic workloads was performed.

The contribution of this paper is an evaluation of cloud-native time series databases regarding their scalability and robustness. To assess these qualities we defined two representative workload profiles from the smart grid domain. Furthermore, we observe how graceful the databases handle loads beyond their current capabilities. Based on these profiles we evaluate three different open-source time series databases (OpenTSDB, KairosDB and Databus). In particular, we aim to determine the scalability and reliability of the technologies.

The remainder of this paper is structured as follows: Section II explains the architectural context of our benchmark, i.e., how are the time series intended to be included in the overall architecture of an industrial monitoring system. Sections III and IV describe our benchmark setup and the experiment, while Section V presents and discusses the results of the benchmark. Section VI highlights related work and Section VII concludes the paper.

II. Architectural Context

A. Conceptual Architecture

Fig. 1 depicts a high-level overview on our conceptual architecture that defines the context for our benchmarks. It mainly targets level 3 (defines the activities of the work flow to produce the desired end-products) and selected areas of level 2 (defines the activities of monitoring and controlling the physical processes) of the ISA-95 standard [3], i.e., activities of manufacturing execution systems (MES) [4]. In the figure, a number of annotations highlight various technology decision point options. The main driver for the given architecture is the need for scalability as we want to offer a multi-tenant large scale cloud system for industrial monitoring. Thus, the employed components need to have scalability properties which adhere to modern cloud, horizontal scale properties. While there are many commercial offerings for the mentioned technologies, here we focus on Open Source alternatives to avoid bias and allow independent replication of our experiments.

Clients access the cloud-based monitoring system through **Web Browsers** using HTML5 via desktop or mobile devices. The architecture includes a **TimeSeries DB** to efficiently store

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.86

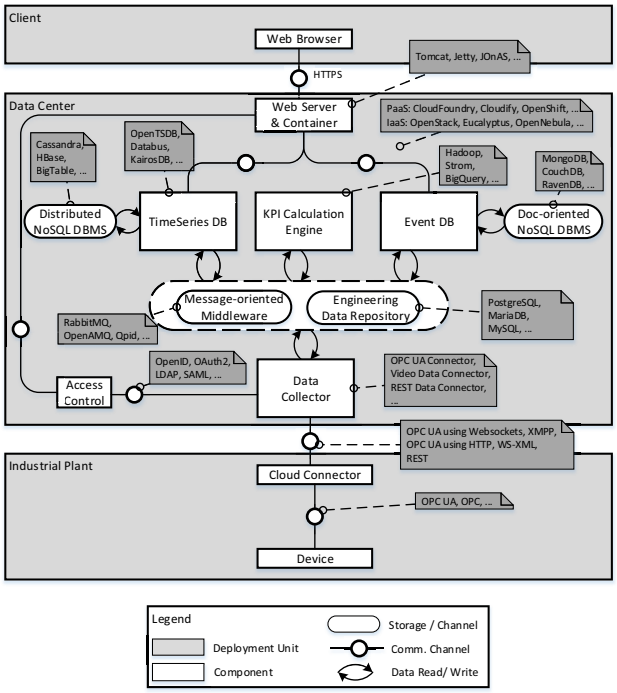

Fig. 1. Conceptual architecture for a cloud-native monitoring system.

arrays of numbers indexed by time. For example, a time series could be the flow speed of oil through a pipeline. The underlying industrial process continuously uploads this data based on various sensor readings. Due to the high data volume in many industrial processes, the storage solution must be scalable. Numerous OSS time series databases are available and under commercial use.

The TimeSeries DB relies on a **Distributed NoSQL DBMS** to achieve high scalability and elasticity. Such a DBMS must be able to store billions of values and seamlessly adapt to workload changes to efficiently use the given hardware resources. The DBMS is especially optimized for write operations and supports occasional read operations. The DBMS may support clusters spanning multiple datacenter to achieve high availability. OSS examples with commercial adoption are Cassandra and HBase.

To capture alarm and event data from industrial processes, the monitoring system additionally contains an **EventDB**. The EventDB must support simple queries like all alarms with a certain priority or all alarms from a certain area. **Document-oriented NoSQL databases**, such as MongoDB, CouchDB, or RavenDB, offer redundant storage with scalable performance and sufficient query capabilities at the same time. Thus, this database is distinct form the time-series database.

The **KPI Calculation Engine** computes key performance indicators from the raw monitored data, such as machine waiting times, production throughput, mean time between failures, etc.

The communication between the components in the datacenter is facilitated through a **Message-oriented Middleware** (MOM).

The **Data Collector** is the endpoint that receives, caches and enqueues all data points as well as events from customer's sites or other installations. To offer endpoints for different kinds of clients that can push data different technologies for the endpoint are possible. For a generic endpoint a REST-based implementation can be offered. For gathering data within a plant and transferring it to the data center the framework uses the **Cloud Connector**. The devices in the plant, such as different types of sensors and actuators provide the actual data values and events.

B. Focus of this Paper

Based on our conceptual architecture we implemented a prototype. We aim at subsequently evaluating the different characteristics of the used technologies in a systematic way. We started by analyzing one of the core technologies of the industrial monitoring system which is the *time-series database*. In this paper, we focus on evaluating the scalability and partially also robustness of this core technology. Future efforts will aim at extending the evaluation to other parts of the architecture as well as other architectural concerns such as availability and security.

III. Benchmark Design

Our benchmark aims at evaluating the scalability characteristics of the time series database under test, namely the scaling coefficient for different workloads. The following hypotheses we aim to either prove or disprove:

1) **Linear scalability:** Time series databases on top of cloud infrastructures scale linearly with the number of nodes employed in the cluster.
2) **Industrial workloads:** Cloud-based time series databases are able to handle industrial workloads.
3) **Workload independence:** The scaling is independent of the type of data being stored. Only the amount of data accounts for scalability.
4) **Resiliency:** Cloud-based time series databases can tolerate crashes of up to two instances.
5) **Read/write independence:** Cloud-based time series databases show an independent read and write performance (i.e., no "noisy neighbor" problem).

Additionally, we wanted to identify bottlenecks for specific workloads in order to determine the most effective way for scaling out the cluster. For example, we wanted to know if provisioned I/O (pre-defined and guaranteed I/O operations[1]) would be needed in case of hard disks being the bottleneck.

A. Industrial Workloads

For performance and scalability benchmarking, we have acquired realistic workload profiles from representative industrial processes. The following paragraph including Table I will give an overview on the different load profiles consisting of

[1]http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PIOPS.html

combinations from two workloads and different read and write scenarios.

1) Phase-Measurement Workload: This workload comes from electrical power engineering in the context of Wide Area Measurements Systems (WAMS), which employs Phasor Measurement Units (PMU) to measure the electrical waves of a power grid. The use of GPS receivers allows for time synchronization of individual PMUs, thereby offering synchronized real-time measurements of multiple remote measurement points on the grid. Each PMU has fourteen analog and eight digital signals, which we modelled as 14 32-bit floats[2] following a 12.5 Hz Sinus curve.

In the first load profile called PMU_Write, a PMU performs a HTTP POST for the data collected since the last time (max. 2000ms ago) the PMU uploaded data with a resolution of 20ms. This accounts for 50 samples of each signal / sec, and a total of 750 values per second per PMU. The PMU will sleep for 1000ms after a successful post request. We define the system to have reached maximum capacity, when the average post latency is greater than 1000ms.

Besides the PMU_Write profile, we also have two read profiles related to PMU. The read tests are performed on a time series database that holds 15 hours of data for the identified maximum sustainable number of PMUs. In the first read workload profile, called PMU_ReadSingle, 1 minute of data of a random selected signal is requested starting from a random moment in the collected 15 hours of data. This query represents the typical amount of data an engineer will analyze when working with a single PMU's data. We define the system to have reached its maximum read capacity when the average request time is greater than 1 second. The maximum number of requests/sec before reaching this stop criterion is the value we are after.

The second read workload profile, called PMU_ReadAll, reads the value of 50 stored signals for a random selected 1s time window in the collected 15 hours of data. This query represents the typical amount of data an engineer will analyze when searching for deviations in PMU synchronization. A similar stopping criteria as for the first read test is used, but now the capacity is reached when the average request time is greater than 20 seconds, as we are requesting large amounts of data.

Finally, we also measure combined read and write workloads. We assume a possible ratio for the PMU_WRS (write & read single) which combines 90% PMU_Write requests and 10% PMU_ReadSingle requests. For PMU_WRA (write & read all) we define the ratio as 95% PMU_Write and 5% PMU_ReadAll. The The stopping criterion for the combined tests is reached when the average response time is above 1 second.

2) Smart Meter Workload: The second workload is taken from the advanced metering infrastructure (AMI) for smart grids. Such systems collect energy usage from smart meters installed in customer homes. The system stores and analyzes the data in order to optimize energy production and distribution and to prevent outtakes. The concrete scenario requires to store many meter readings (in excess of 1,000,000) in 15 minute intervals that arrive in a 2-minute time window. Each meter reading includes a 32-bit float representing the energy consumed thus far. The connections are typically not made by each meter individually but rather by intermediate aggregators which typically aggregate several hundreds to thousands of meters. For our benchmark we aggregated 1,000 meter readings per request. Whereas the PMU workload tests the ability to support continuously running uploads, this workload tests the ability of the time series databases to handle peak demands.

[2]The digital signals have been merged into a single value.

Name	OpenTSDB	KairosDB	Databus
Version	2.0 RC1	0.9.1	1.1.0
License	LGPLv2.1+	Apache 2.0	Mozilla Public 2.0
Storage	HBase 0.94	Cassandra 1.2.8	Cassandra 1.2.8
LOC	34,305	31,474	95,418
Activity[3]	21	53	113
Committers[4]	2	2	2

TABLE II
DETAILS ON THE TIME SERIES DATABASE PROJECTS.

B. Profile Definitions

Table I gives an overview on the profile definitions and the defined number of experiments. For the write scenario we executed both workloads (PMU and Smart Meters) starting with an empty database. We then executed the experiment for varying cluster sizes, i.e., 3, 6, 12, 24, and 36 nodes.

IV. EXPERIMENT

A. Evaluated Time Series Databases

Cloud-based time series databases originated often from IT resource monitoring use cases. For these use cases, time series databases gather information from servers, applications, and other IT resources. Two of the three databases we analyzed come from this domain (OpenTSDB and KairosDB). Databus instead was developed for energy monitoring. There are many time series databases available [2]. However, most of them did not fit our requirements to be open-source, self-deployable and having a REST API (removing TempoDB and Squwk from the list). Other projects seem to be abandoned or in an immature state (like Apache Chukwa, Sensor DB, or Rhombus). The next sections and Table II give further details on the evaluated technologies.

1) OpenTSDB: OpenTSDB [5] collects and stores time series data. It is based on HBase [6] database and runs on Linux platforms. To store data into OpenTSDB provides a REST API (also Telnet and batch import of data are supported), which accepts post messages to containing JSON formatted data. The latest 2.0 release candidate of OpenTSDB allows a millisecond resolution to store time series data. OpenTSDB also offers a comprehensive web interface for querying, displaying and analyzing the time series data.

[3]Total number of commits in the months October-December 2013.

[4]Active committers committing to dev. branch / master, excluding merge commits for the same period as Activity.

Profile	Scenario	Workload	DB content	# of nodes	Max resp. time	# Experiments per DB
PMU_Write	Write	PMU	empty	3,6,12,24,36	1s	5
SmartMeter_Write	Write	Smart Meter	empty	3,6,12,24,36	60s	5
PMU_ReadSingle	Read Single	PMU	15h PMU	12	1s	1
PMU_ReadAll	Read All	PMU	15h PMU	12	20s	1
PMU_WRS	90% Write + 10% Read Single	PMU	15h PMU	12	1s	1
PMU_WRA	95% Write + 5% Read All	PMU	15h PMU	12	1s	1

TABLE I
OVERVIEW OF THE DEFINED BENCHMARK PROFILES.

2) KairosDB: KairosDB [7] is a rewrite of OpenTSDB that more clearly separates between data retrieval and its representation. Therefore, KairosDB also includes more query functionality such as more advanced time series aggregators. KairosDB uses Cassandra [8] as storage layer targeting at improved speed and scalability. However, an abstraction at the storage layer allows using also other databases. As OpenTSDB, KairosDB supports submitting data points via both Telnet as well as a REST API. The REST API accepts POST payloads in JSON format. Furthermore, KairosDB also has native support for millisecond time series granularity, supports compressed data upload of batch data.

3) Databus: The National Renewable Energy Laboratory (NREL) developed Databus [9] for collecting and storing time series data. A key feature of Databus is its ability to store large amounts of data collected at a one second granularity. Databus runs on top of the Cassandra database [8]. Databus requires the POST payload of a request to be in the JSON format. In Databus there are two types of tables, i.e., Stream tables for time-series data, and Relational table for relational data. To translate data into different forms when requesting data Databus supports different computational modules, such as an SQL module, a module to compute splines.

B. Experiment Environment

The server clusters for the experiments were set up on infrastructure provided by Amazon Web Services (AWS). We used Priam by Netflix [10] and Apache Whirr [11] to be able to quickly deploy the Cassandra and HBase clusters for our time series databases. These tools on top of Amazon's compute service EC2 created and configured a fresh cluster for each experiment quickly and teared it down afterwards. From the set of instance types available at AWS, we chose m1.large [5] instances with 8GB Memory, 2 cores and 2 420GB ephemeral disks. We used the local machine disks for persistent storage. For Cassandra we combined the disks into a single RAID0. Our choice of instance type and storage for AWS deployments of Cassandra was based on recommendations from [12] and [13]. Based on these recommendations we also decided to use triple replication for the underlying databases.

The time series database server components ran on the same nodes as the NoSQL database services. Amazons' Elastic Load Balancing service distributed performance test traffic across all cluster nodes. AWS CloudWatch was used to monitor the machine instances, e.g. CPU utilization, data transfer and disk usage. As test driver infrastructure we used the Visual Studio Ultimate Web Load Test tools. The setup consisted of a client for developing, triggering and monitoring the load tests, a controller node that distributed load tasks and gathered the metrics and finally a set of worker machines that create the actual load. As we were not interested in measuring network effects, we decided to put all servers and load driver machines in the same AWS region (EU-West (Ireland)). The overall infrastructure cost (i.e. the AWS bill) for setting up and running the experiments was around 2000 US$.

C. Experiment Execution

To identify the maximum amount of PMUs/smart meters that the systems can handle we gradually increased the load by adding more and more virtual users (VU), during the run of the experiment. A virtual user, in the PMU case produces and submits the data of one PMU. Further VUs were added until either the stopping criterion defined for the respective scenario was reached or errors such as timeouts occurred. The latter manifested in the load balancer producing a timeout as its underlying service did not respond in a timely manner. The default timeout for an AWS elastic load balancer was at 60s.

For the read profiles we filled the database with 15 hours of data based on the highest stable input load the respective database could handle. The read time frames had been randomly selected from the available data to reduce the effect of caching. Furthermore, the amount of data in the database (e.g., 15GB per node for KairosDB[6]) was designed to be larger than the buffers of the underlying database (e.g., 4GB for Cassandra).

V. RESULTS

A. OpenTSDB

Performing load tests against OpenTSDB proved to be difficult, as results could not be replicated between runs. Some of the underlying HBase instances ran out of their allocated memory during our load tests, which seemed to correlate with the amount of data being pushed, but not the actual data rate.

This behavior is usually not problematic, but does become a problem as OpenTSDB continues to accept data from clients even though the underlying storage cluster is no longer working. Even on light loads, given enough time, some of

[5]http://aws.amazon.com/en/ec2/instance-types/

[6]In total this accounts for 60GB of raw data: 15Gb per node * 12 nodes = 180GB / 3 times replication = 60GB raw data.

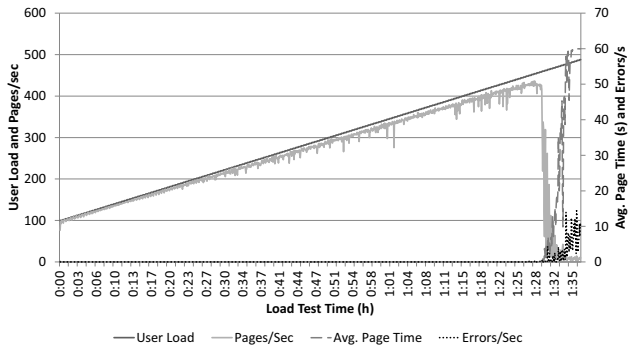

Fig. 2. Load test run of KairosDB in the PMU write scenario for a 24 node cluster. The maximum capacity appears to be around 420 PMUs. As the load increases there is a sudden drop in thoughput as the system gets overloaded.

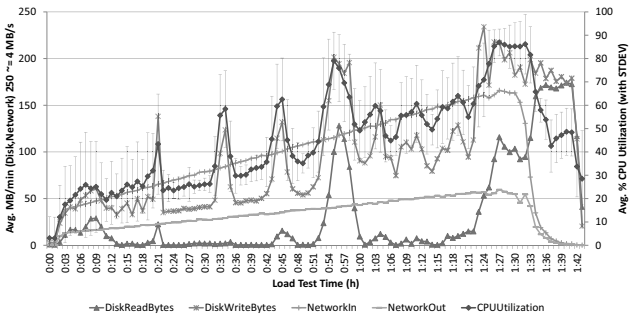

Fig. 3. Resource utilization for the load test run of KairosDB in the PMU_Write scenario with 24 nodes. The standard deviation of the CPU utilization for the different nodes is denoted as error bars on the respective series.

the HBase nodes run out of memory. Increasing the memory limits allocated to the HBase JVMs did not solve the problem. Additionally, OpenTSDB does not throttle the IPC requests to the HBase instance. Together, these two aspects complicate recovering an out-of-memory HBase instance, prevent data loss, and to determine the load OpenTSDB can handle.

After 12 tests (each taking more than 1 hour) for a 6 node cluster using the PMU_write scenario, we determined that for the maximum capacity lies around 50 PMUs at 48,4 requests/sec with at least one HBase instance running out of memory at around 43 minutes when using a constant load.

Concluding, both HBase and OpenTSDB fail to protect themselves properly against massive uploads. HBase *assumes* that it is configured correctly for the workload it gets. Hence, expert tuning and monitoring of the HBase cluster is required to make it operational sustainable. We did not have the time to perform this for this benchmark. On the other hand, OpenTSDB fails to detect problems with HBase and to alert its clients when it no longer can handle the load.

B. KairosDB

The results of a typical PMU_Write load test for a 24 node KairosDB cluster are depicted in Figure 2. The graph illustrates the slowly increasing amount of virtual users and beneath that curve the amount of pages per second. Ideally the curves would overlap as then the complete data for the PMU over the last second can be transferred as one. As the maximum capacity of the system, in this case around 420 PMUs, which corresponds to the amount of PMUs installed in small sized country, is reached there is a drop in the responses and response times for individual requests go up. As the load balancer times out at 60s longer request times show up as timeout errors in the graph.

Figure 3 presents the server-side metrics for the same test run. It shows the average values of various metrics for the server nodes in the cluster over the time of the test run. Disk write does not go beyond 4 MB/s on average which is way beyond the maximum throughput of the provisioned disks. Based on this data, which we also gathered for the other benchmark profiles we can conclude that KairosDB under the given workloads is CPU bound. Even in the PMU_ReadAll profile which is the most I/O intensive profile wrt. the disk utilization, CPU is saturated before the disk does. More specifically, in this profile the disk read utilization is on average across all nodes only around 4 MB/s (max. 10MB/s) when the maximum throughput is reached.

Having a deeper look at the CPU utilization reveals that the load is mostly evenly distributed between the KairosDB process (˜50%) and the Cassandra process (˜50%). Together with the CPU boundedness of the system this allows for a good scaling behavior. If more performance is required additional nodes can be added increasing the capacity of both KairosDB and Cassandra. We also found this in the distribution of the load among the nodes. A simple load balancer configuration is enough to evenly distribute the load.

Regarding robustness we found that the self-protection of KairosDB behaves well. KairosDB monitors how well Cassandra handles the current workload and tries to adapt its queues accordingly so that they don't overrun. This results in the clients being throttled by response times when the back-end can't handle more load. This behavior can be seen in the increasing response times towards the end of the load test run depicted in Figure 2. Still, if load goes up more and queues still fill more, nodes might fail. However, due to triple replication this is not fatal.

We then could also observe a graceful degradation if a node fails as the system remains functional. The load balancer can then, based on configured health checks, just remove the failing instance and distribute the load to the remaining nodes.

For the PMU_write profile shown in Figure 4 KairosDB scales almost linearly with the amount of nodes. The largest cluster with 36 nodes could handle a maximum of 598 PMUs with a maximum of 538 requests per second and a write throughput of 403,500 values written per second.

The SmartMeter_Write profile reached as much as ˜6,000,000 smart meters in the 36 node cluster as a maximum throughput which corresponds to the amount of meters of a large city. As depicted in Figure 5 the scalability is not fully linear with the number of nodes in the cluster. When the amount of smart meters goes beyond 2,000,000 the scalability slightly decreases but then remains stable. A reason for this

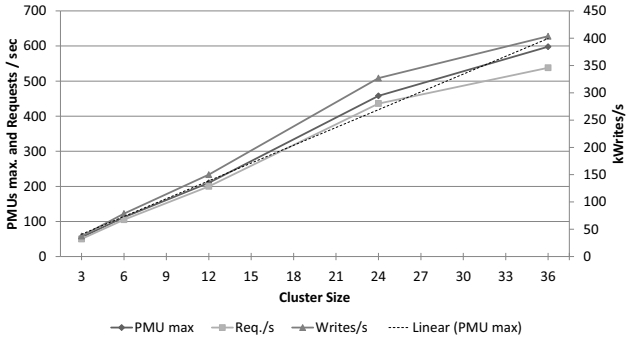

Fig. 4. Scalability results for KairosDB in the PMU write scenario showing a nearly linear scaling from 3 to 36 nodes.

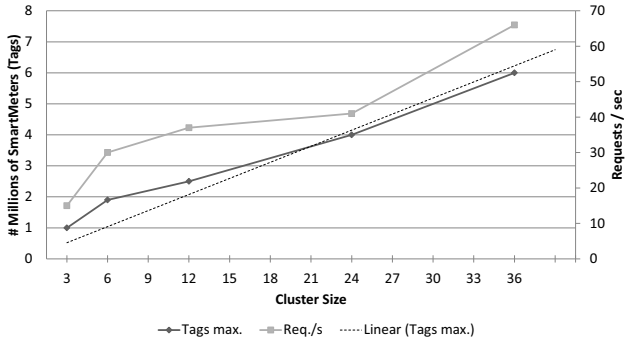

Fig. 5. Scalability results for KairosDB in the smart meter write scenario show an almost linear scaling. The bump at the smaller cluster sizes, might be related to caching of table mappings which do not fit in the cache at at a larger number of tables.

could be the massive amount of different tables that need to be handled in this case. The mapping table which identifies which metric belongs to which table could most probably be completely cached for lower numbers but not anymore for a number metrics beyond 2,000,000.

The read test profiles results shown in Figure 6 reveal another interesting behavior. As a baseline we defined the maximum throughput for the write case in a given cluster, i.e., around 200 PMUs for the 12 node run. The results for the PMU_ReadSingle profile show a maximum throughput of 225 read PMUs with a maximum of 166 requests per second. The throughput for the PMU_ReadAll profile is significantly lower. Only 40 virtual users could be handled with a maximum of 9.8 requests per second. This lower throughput results from the data structuring. In this profile data from 50 different PMUs and therefore also 50 different tables has to be accessed. This causes significantly more effort on the database side to load the appropriate chunks of data.

Finally, we analyzed the combined write and read profiles. Compared to the write only baseline, the PMU_WRS and even the PMU_WRA profile show a slightly increased throughput. This indicates a good separation between the read and write pipelines in the system. Even heavy queries, such as those in the PMU_WRA profile do not negatively influence the write performance of the system.

In conclusion, KairosDB, handled our industrial workloads

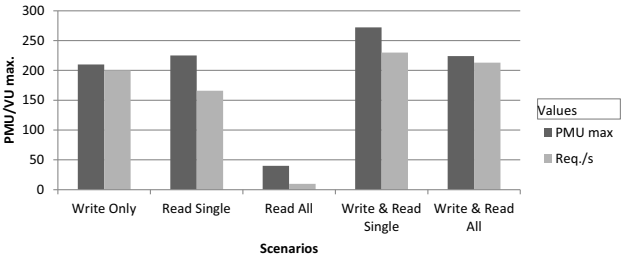

Fig. 6. Results for KairosDB in the PMU_Read profile.

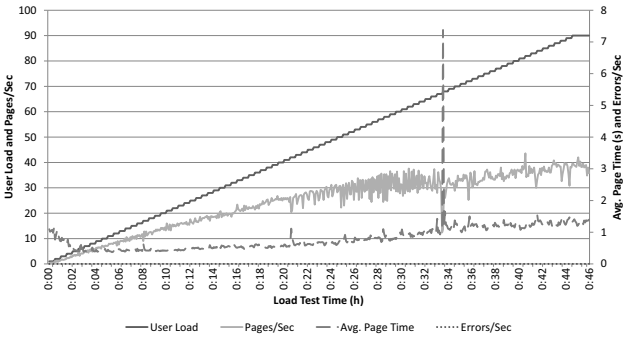

Fig. 7. Typical load test run behavior of Databus in the PMU write scenario from a 24 node test showing a maximum capacity of around 65 PMUs.

well. Especially with the graceful degradation and linear scaling behavior it fulfills the requirements for the implementation of our architecture.

C. Databus

For Databus, the behavior for the PMU_Write scenario was similar to that of KairosDB (see Figure 2 and 7), i.e. once the number of requests/sec started to decrease the number of errors and request response time increased. However, there are two significant differences. Firstly, the requests/sec first stabilizes for a while in Databus before the request response time starts to increase. Secondly, request (timout) errors do not occur so suddenly when the system gets overloaded as for KairosDB.

Figure 8 presents the results for Databus for the PMU write scenario. There is a widening gap between the requests/sec and the write/sec because of Databus' mechanism for handling HTTP connections. When a client establishes a HTTP

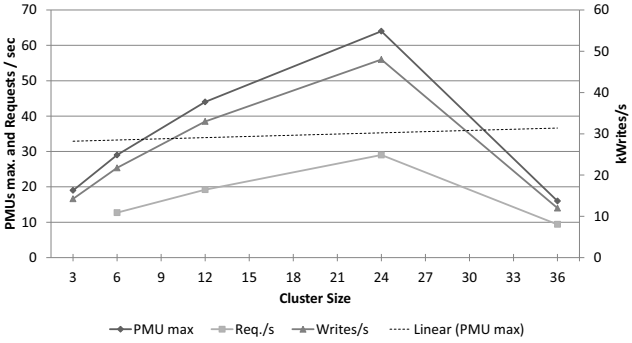

Fig. 8. Results for Databus in the PMU write scenario.

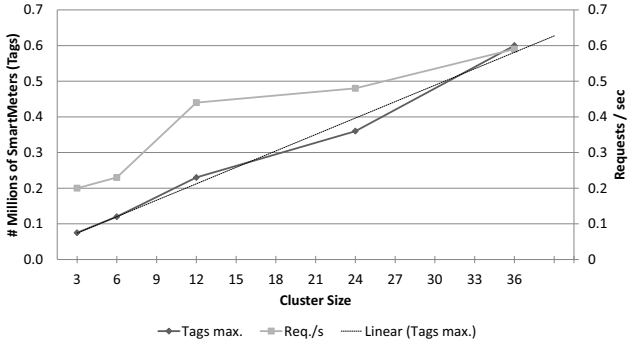

Fig. 9. Results for Databus in the smart meter write scenario.

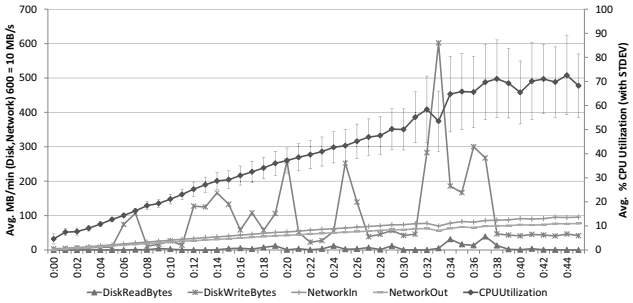

Fig. 10. Databus resource utilization of 24 nodes in the PMU write scenario. The standard deviation of the CPU utilization for the different nodes is denoted as error bars on the respective series.

connection, Databus accepts and acknowledges the first packet of the request. However, it only starts to request for content of the HTTP request once the previous batch has been processed and written to the Cassandra cluster. Consequently, Databus actively throttles its client HTTP connections. Databus therefore does not asynchronously decouple writing to its Cassandra back-end from reading information from its clients.

Although initially the performance of Databus seems to scale linearly, at 36 nodes this trend is broken. Even corresponding with the authors of Databus, we were not able to identify the source of this problem. Furthermore, we also could not conduct the read load tests as we constantly ran into problems during the 15h input phase for Databus.

The results for Databus in the smart meter write scenario are presented in Figure 9. The results show a linear relationship between the number of smart meters Databus can handle and the number of nodes in the cluster. However, the number of smart meters Databus can handle is an order of magnitude lower than KairosDB. Because Databus throttles HTTP connections, thereby causing an inflexibility in the system to deal with peaks in work loads, as is the case with the smart meter scenario.

Similar to KairosDB, Databus is CPU-bound. Databus achieves a uniform distribution of CPU utilization across the cluster (e.g. see standard deviation in Figure 10). Compared to KairosDB, the CPU utilization of Databus is much smoother, with no spikes happening at individual nodes until a single core is fully utilized, i.e. at the 50% utilization point, as the virtual machines have 2 cores available. However, Databus is significant less efficient than KairosDB, a typical ratio between the CPU usage of Cassandra and Databus on a node is 30% versus 70%.

The self-protection of Databus is limited, as it is mainly based on throttling its HTTP connections. Databus has an issue with dealing with quickly ramping up traffic. In such cases, nodes can stale out at 100% CPU utilization from Databus (not Cassandra) without recovering when the connections are terminated and the cluster is given several hours to recover. Most likely this behavior is due to a concurrency connection handling bug. As another issue, heavily used nodes can run out of memory. To resolve this issue, we increased the initial memory size of the JVMs from 1GB to 4GB each, which resolved this issue.

D. Summary

The results for the databases differed heavily. For example, KairosDB could handle almost an order of magnitude more PMUs/smart meters than Databus. Furthermore, we failed to execute the load tests for OpenTSDB in a reproducable way. Therefore, the checks against most of our initial hypotheses only hold for KairosDB:

1) **Linear scalability:** The evaluated databases, especially KairosDB, show a good near-linear scaling behaviour.
2) **Industrial workloads:** KairosDB was able to handle both workloads to an extent which would result in realistic cluster sizes, i.e., a 24 node cluster could handle the smart meters of a large city.
3) **Workload independence:** For both types of industrial workloads KairosDB as well as Databus could be scaled in a similar, linear way.
4) **Resiliency:** Even with one or two instances down KairosDB and to a certain extent also Databus could continue working. Even though, response times partly went beyond the specified timeouts.
5) **Read/write independence:** As the combined read/write troughput of KairosDB is even above the write only performance and even complex queries (e.g., in the PMU_WRA profile) did not lead to a degredation of the write thoughput we can assume a good separation between read and write tasks.

We calculated the costs of the used time series database clusters based on AWS prices. For example, the 24 node cluster capable of handling 6,000,000 smart meters would cost 4147.20 US$ with on-demand machines or 1468.80 US$ on 3-year reserved instances per months.

E. Limitations and Threats to validity

Threats to the validity of our results mainly stem from three different areas. First, we did not have expert knowledge in tuning the underlying databases (HBase and Cassandra). Secondly, the time series databases (OpenTSDB, KairosDB and Databus) are still immature. Future, more stable versions of these technologies might perform differently. Finally, as we run our experiments in a virtualized environment we had no control of hardware underneath provided test VMs. However,

as AWS does not do over-provisioning of CPUs and we used local (ephemeral storage) the effects of this uncertainty might be not that significant.

Concrete data from OpenTSDB could not be measured due to technical issues. Furthermore, as we could perform the read tests only for KairosDB we cannot relate the read performance to other technologies.

VI. Related Work

There is limited work in the literature on supporting the design of cloud-native monitoring systems for industrial processes. Most of the available literature on cloud computing technologies is either for enterprise applications or generic without a particular domain focus. Sakr et al. [1] provide a comprehensive survey of large scale data management approaches in cloud environments including MapReduce programming models and NoSQL databases.

There are also several benchmarks and performance comparisions for NoSQL databases available. Yet none of these involve time-series databases or industrial workloads. For example, Cooper et al. [14] introduced the Yahoo Cloud Serving Benchmark (YCSB) that issues differently distributed request against databases, and was applied on Cassandra, HBase, Yahoo's PNUTS and MySQL. Bushik [15] built on their results, but instead used commodity hardware for measurements and also analyzed MongoDB and Riak. Nelubin et al. [16] altered YCSB for ultra-high performance scenarios and compared Cassandra, Couchbase, MongoDB, and Aerospike. Datastax [17] benchmarked Cassandra, HBase, and MongoDB on Amazon AWS using modified YCSB workloads. The results of all these benchmarks are helpful, but cannot be directly transferred to the domain of cloud-native monitoring systems.

In the general area of cloud benchmarks Folkerts et al. [18] provide an introduction on the specific challenges in creating benchmarks for cloud applications, e.g., using meaningful metrics and creating an appropriate workload design.

Wlodarczyk [2] compared four time series database systems that exploit cloud environments including OpenTSDB, TempoDB and Squwk. OpenTSDB was used by Andreolini et al. [19] to monitor IT resources in cloud environments. None of these approaches focuses on industrial applications and their specific requirements.

In conclusion, related work does not address domain specific problems, like domain-specific workloads or types of databases (e.g., time-series databases). Thus, our paper fills the gap regarding available benchmark design as well as result data.

VII. Conclusion

We have proposed a conceptual architecture for a cloud-native monitoring system for industrial processes. We presented a benchmark for evaluating its included time-series database scalability and robustness based on realistic, industrial workloads. The results indicate that our hypotheses (linear scalability, support for industrial workloads, workload independence, resiliency and read/write independence) are mostly supported at least for KairosDB. The results can be used as a baseline for the evaluation of other time series databases.

In future work, we plan to conduct our benchmark in extended setups, e.g., spanning multiple availability zones on the server side as well as the load driver side. Furthermore, we will extend our evaluations and provide generic test scenarios as well as benchmark results for other components of the conceptual architecture.

References

[1] S. Sakr, A. Liu, D. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *Communications Surveys Tutorials, IEEE*, vol. 13, no. 3, pp. 311–336, 2011.

[2] T. W. Wlodarczyk, "Overview of time series storage and processing in a cloud environment," in *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, ser. CLOUDCOM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 625–628.

[3] ISA, "International standard for the integration of enterprise and control systems," http://www.isa-95.com/.

[4] M. Hollender, *Collaborative Process Automation Systems*, 1st ed. Isa, September 2009.

[5] The OpenTSDB Authors, "Apache HBase website," http://hbase.apache.org/, 2013, last visited: 2013-12-13.

[6] The Apache Software Foundation, "Opentsdb website," http://opentsdb.net/, 2013, last visited: 2013-12-13.

[7] ——, "Kairosdb website," https://code.google.com/p/kairosdb/, 2013, last visited: 2013-12-16.

[8] ——, "Cassandra website," http://cassandra.apache.org/, 2013, last visited: 2013-12-13.

[9] National Renewable Energy Laboratory, "Databus website," http://www.nrel.gov/analysis/databus/, 2013, last visited: 2013-12-13.

[10] Netflix, "Netflix priam website," https://github.com/Netflix/Priam, 2013, last visited: 2013-12-18.

[11] The Apache Software Foundation, "Apache whirr website," http://whirr.apache.org/, 2013, last visited: 2013-12-18.

[12] Wikipedia, "Cassandra wikipedia website," http://wiki.apache.org/cassandra/CassandraHardware, 2013, last visited: 2013-12-18.

[13] A. Cockcroft and D. Sheahan, "Benchmarking cassandra scalability on AWS - over a million writes per second," http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html, 2013, last visited: 2013-12-18.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[15] S. Bushik, "A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak," http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html, October 2012.

[16] D. Nelubin and B. Engber, "Ultra-high performance nosql benchmarking: Analyzing durability and performance tradeoffs," http://odbms.org/download/NoSQLBenchmarking.pdf, 2013.

[17] Datastax Corproration, "Benchmarking top nosql databases: A performance comparison for architects and it managers," http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf, Feburary 2013.

[18] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the cloud: what it should, can, and cannot be," in *4th TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC), VLDB*, 2012.

[19] M. Andreolini, M. Colajanni, and M. Pietri, "A scalable architecture for real-time monitoring of large information systems," in *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, 2012, pp. 143–150.