Java embedded storage for time series and meta data in Smart Grids

Stephan Cejka
Siemens AG
Corporate Technology
stephan.cejka@siemens.com

Ralf Mosshammer
Siemens AG
Corporate Technology
ralf.mosshammer@siemens.com

Alfred Einfalt
Siemens AG
Corporate Technology
alfred.einfalt@siemens.com

***Abstract*—We present a Java-based embedded data store for edge-to-cloud storage optimized for Smart Grid time-series measurements. The key performance indicators expected of applications and operators of a Smart Grid monitoring and control system - frequent readouts, immutability, statistical indicators - are optimally supported. Furthermore, the data store is tailored for operation on platforms with limited storage and processing resources. We show that our implementation is superior to state of the art and off-the-shelf solutions in data retrieval time and needed storage size.**

I. Introduction

One of the key benefits of the Smart Grid vision for the low voltage grid is the availability of measurement data. While remote reading of billing information accelerates business processes and creates customer value, operative applications working on an unprecedented base of data from a previously "dark" grid area provide immense benefits for utilities: grids can be operated much closer to capacity; brownout and blackout situations can be predictively mitigated, and equipment maintenance scheduled in a timely manner. The performance of different architectures for data processing have been evaluated in [1] in comparing key cost indicators such as energy consumption, processing, communication and storage costs. Operative applications – e.g., voltage control algorithms[2], state estimators[3] or equipment monitoring software – can generally be hooked into the Smart Grid data stream according to one of two architectural philosophies:

1) In the sensor-to-cloud (2-tier) approach, all data from distributed sensor nodes are cleared to the cloud and processed remotely. A centralized architecture[1] concentrates data processing and storage on a central facility in the system. In a decentralized architecture[1] the load is distributed in the system to localized management platforms. State estimation and voltage control applications based on a 2-tier architecture were investigated in [4].
2) In the fog-/edge computing (3-tier) approach, an additional middleware layer pre-processes the data before transferring them to the cloud, thus creating a hybrid architecture approach[1] implemented in several demonstration projects like [2], [3] or [5].

While the 2-tier approach generally provides the benefits of unlimited storage and computing resources, its drawbacks are manifold:

- Reliable data connections have to exist for every sensor node
- Data transfer latency can be a prohibitive factor in operation-critical control applications
- The backend network could be spammed with data sampled at resolutions irrelevant for the actual applications
- High traffic on a multitude of connections increases the probability of malicious interference
- It might be prohibited to transfer certain data to a geographically non-locatable storage for legal or privacy reasons
- Utility network architectures might interfere with direct data transfer

The fog-/edge computing architecture addresses many of these problems. But aside from increased rollout costs, devices on the network edge are usually more or less constrained in terms of computing power and storage capacity.

The latter point becomes a critical factor if aside from live stream data, historical time-series data are relevant for operative applications. Many applications could benefit from access to time-series data: control algorithms could be bootstrapped much more efficiently; state estimators refined; pattern recognition software for equipment status seeded; and stream data could be interpolated in case of equipment outage.

In this paper, we perform a rigorous analysis of storage solutions and their applicability to fog-/edge computing problems in the Smart Low Voltage Grid. We propose our own storage solution, which is optimally suited for a set of use cases we encounter in actual field test projects.

The remainder of this paper is structured as follows: Section II presents use cases for data retrieval in Smart Grid environments. It shows why neither SQL nor NoSQL solutions are reasonable for the persistence of such data by presenting benchmarks of the considered options. We present related work and specify why a new implementation is necessary to fulfill the requirements of the use cases. In Section III we provide an evaluation of a time- and space-efficient persistence format to be used in our solution. Section IV describes our local implementation, the cloud functionality and additional components that this database system consists of. We show a comparison of our solution to the best solution of the

evaluation in Section V. Section VI finally concludes this paper with an outlook to future work.

II. Evaluation of edge storage solutions

To conclusively assess the performance of existing solutions suited for time-series storage, we propose short use case scenarios relevant in an operative Smart Grid context. These use cases are framed by the assumption of a reasonably scaled edge/fog computing platform able to host a variety of applications and allowing near-instantaneous local communication between those applications.

Use case 1
A newly installed or rebooting control algorithm operating on Smart Grid measurement data (e.g. three-phase voltage or current measurements) requests historical data to obtain an impression of the current grid state.

Use case 2
A grid operator views historical time-series data on a user interface.

Use case 3
An algorithm uses historical data to interpolate temporarily missing measurement values.

Use case 4
A topology detection algorithm requests statistical data related to measurement frequency.

Use case 5
A preprocessor extracts relevant key performance indicators to report to a top-tier system.

Use case 6
An algorithm detects an anomaly based on the analysis of stored values.

Depending on the actual hardware installation of the edge/fog system, different storage types, storage sizes, computing resources and connections will be available. It can be generally assumed that the storage will be solid-state and has to conform to at least industrial specifications regarding temperature range, shock resistance, EMC compliance etc., for reliability reasons. Such a component will naturally be expensive and thus be severely limited in storage size and non-redundant, which posits additional constraints for the software storage solution.

A. Benchmarks

Smart Grids bring an enormous growth in the volume of data that needs to be processed. The data that are required to be persisted consist of the actual time series data (e.g., periodical measurements of Smart Meters) and meta data of the sources' data point which may include the data point's description, location and other meta fields as well as statistical values based on the time series data, e.g., for detection of abnormal values. Some of these use cases require a retrieval of time series data of a certain time span. Data that is no longer of relevance to any of the local use cases are no longer required to be saved locally, keeping in mind that the amount of memory may be limited. Permanent storage of time series data is assumed to be available in the cloud, which provides unlimited space and replication capabilities, and allows the use of these data for offline algorithms, e.g., related to grid planning or long-term equipment outage analysis. Reading back data from the cloud would be possible, but is assumed to incur much higher latency than local access, and is additionally subject to network quality of service issues. From these conditions framing the use cases presented earlier, the requirements for the local storage system can be summarized as:

1) Access to short-term time series for many data points (use case 1) as well as longer time series for single data points (use cases 2, 3) needs to be fast.
2) Statistical and meta-information needs to be readily available (use cases 4, 5, 6).
3) The size of the time-series data must be as small as possible to accommodate limited resources on the edge device.
4) A mechanism must exist for offloading data to the cloud to conserve local storage space.

We evaluated different off-the-shelf SQL and NoSQL database solutions in regards to these requirements.

Test data were generated for every second in one year resulting in about 31.5 million entries. Each entry contains an identifier, a timestamp, the value and an array of tags. For the evaluation, MongoDB was selected because it is the best known NoSQL database, H2 because authors claim that this database system has the highest performance in their benchmarks[6], and postgreSQL due to its high distribution. As the underlying application platform we used in evaluation is written in Java, H2 has the added benefit of being able to run embedded. This is also relevant for the size of the database management system and all libraries that are necessary to execute it, which is more than 400 MB for MongoDB but less than 5 MB for H2. The system used for the benchmarks contained an Octa-Core Intel Core i7-4800MQ CPU 2.7 GHz, 16 GB RAM, SSD; running Windows 7 and Java 8. We used H2 1.4.181; postgreSQL 9.3.5 with the JDBC driver version 9.3-1100 and MongoDB 2.6 with the Java driver version 2.12.3.

Local data retrieval is required by the use cases in two forms: either a range of time series data of one data point is requested or meta data including the most recent entry of one data point is required. The first benchmark in figure 1 shows the required size to store the test data. The second benchmark in figure 2 shows the retrieval time for a time slice of one month of time series data (≈2.6 million entries), while the third benchmark in figure 3 shows the retrieval time for the most recent entry of one data point.

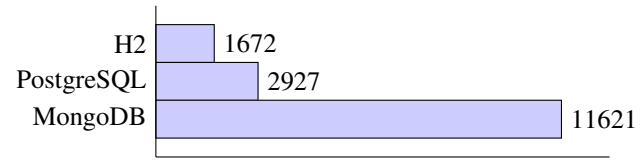

Fig. 1. Size of data (MB)

B. Results

The benchmarks show that regular SQL/NoSQL solutions are not suitable for the stated requirements. Retrieval times in both SQL databases are prohibitive for the near-realtime requirements of operative applications. MongoDB shows acceptable timing properties, but due to its nature as a documents

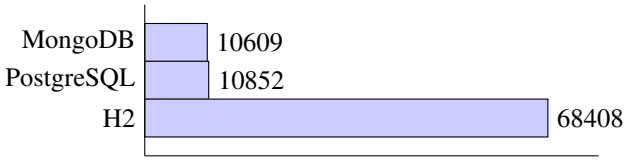

Fig. 2. Time to retrieve one month of entries (ms)

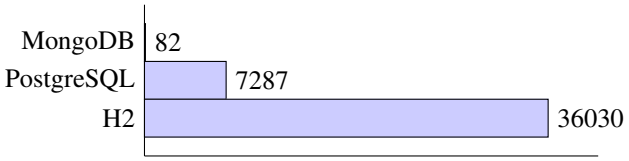

Fig. 3. Time to retrieve most recent entry (ms)

database, the storage size is very large. In all candidates, statistical information and meta-data must be added as separate layer on top of the actual database.

C. Existing Solutions

There are no widely available solutions for local Low Voltage Grid control and monitoring or general sensor measurements in an industrial environment. A lot of products exist that claim to be suited for time series storage. However none of them able to run embedded in Java, which is preferable in terms of system manageability, installation size and complexity.

The existing solutions are either standalone (time series) database systems or HBase/Hadoop-based. OpenTSDB as one of these examples builds on-top of Hadoop was shown to be reasonable for Smart Meter use cases in [7]; however in this solution all data are stored centrally. HBase/Hadoop-based solutions are not reasonable for our use cases to be used on small machines due to the high overhead and complexity. A selection of other viable solution we considered includes:

- **Cube** can be used for time series in JavaScript and is built on top of MongoDB.
- **RRD4J** is developed in Java. While it is reasonable for small time series, due to its round-robin fashion outdated data will be discarded. Therefore, this solution is not reasonable for our use case.
- **Cassandra** is a decentralized solution that handles scalability and high availability. The overall complexity of this solution is generally too high for our application.
- **InfluxDB** is a standalone solution written in Google Go. A library to access the database from Java via a REST interface however is available. It is expected that solutions using REST cannot fulfill the time constraints.
- **neo4j** is a graph database, where relations between nodes can easily be expressed as relationships. It should therefore be simple to handle time series, however the query language is not self-explanatory and too complex for this use case. Furthermore, in a base evaluation, we could not achieve comparable values to our own solution.

III. Evaluation of persistence formats

Based on the requirements, an efficient persistence format for data is required under the assumption that

1) it is not possible to hold all collected data in the main storage,
2) the gathered data need to be persisted on disk for safety reasons, as data would get lost on failure or restart and
3) data need to be persisted permanently on a cloud host.

The sensor data are generally small in size, but measurements are occurring very frequently. Data are never updated, but potentially read multiple times. The choice of an adequate data persistence format significantly influences performance. Text-based, human-readable formats are not considered for this evaluation, as the storage size is bigger and the serialization/deserialization procedure usually consumes significant time. Currently, it is also not required that the format is readable by a program written in a different programming language. The evaluated formats represent the most popular fraction of all available. Using a persistence format that is already available in favor of an own requirement-driven format yields the added benefits of being industry-tested and widely known.

A. Java Serialization

Java Serialization is the built-in ability of Java to persist its objects, reload and reuse them again. Not much effort is required by the programmer, making this method the most convenient. To define Java objects as being serializable, they need to implement the `Serializable` interface, requiring that all (non-transient) fields of the class in turn can be serialized.

B. Protocol Buffers

Protocol Buffers were initially developed by Google to define messages for communication between programs, offering a language-neutral mechanism for serializing structured data. A simple language independent schema file is written in a data description language (DDL) and compiled by use of the `protoc` compiler which generates executable code for the target platform[8]. Protocol Buffer is a data-interchange format alone, allowing RPC (remote procedure calls) only through third-party libraries. Generated classes can be used in the application for population, serialization (marshaling and unmarshaling), and access of Protocol Buffer messages. Data that are serialized using this generated code can be written to a file or transmitted over the network, and received at or read in by a program potentially written in another language. While text-based formats are self-describing, Protocol Buffer messages stay meaningful only using the same schema for deserialization.

C. Apache Thrift

Apache Thrift, in comparison to other formats, has the unique advantage that it includes an RPC framework[9]. The format itself is similar to Protocol Buffers' format, also having the demand to efficient cross-language data serialization. Thrift was initially developed at Facebook in 2006, as they required

to integrate functionalities of programs written in different languages. It was always intended to be open-source, the project being first hosted directly at Facebook, but later moved to Apache. The Thrift grammar is much richer than Protocol Buffers in terms of supported constructs.

D. Apache Avro

Apache Avro is another option that provides a binary format. In contrast to Protocol Buffers and Thrift, the schema can be integrated in the stream which makes sense for a big number of elements following that schema. Avro supports reflective schema generation from a Java object. It is mainly used in Apache Hadoop.

E. Benchmarks

Plenty of evaluations between different formats are available online, many of them being use-case driven. Objective evaluations usually do not put one option in favour of another, concluding that the best format depends on the requirements.

The system used for the benchmarks was described in Section II-A. Following versions were used: Protocol Buffers 2.6.1, Apache Thrift 0.9.2 and Apache Avro 1.7.7. For evaluation, a file containing one data point with 2592000 measurement values was written to the disk and read in again. The number of entries represents one month of measurements from a typical Smart Low Voltage Grid installation, with a data occurrence of about 1 Hz, which was empirically proven to be representative. Relevant benchmarks are:

- the time that is required to serialize the Java object to a file on the disk, which is shown in Figure 4
- the time that is required to de-serialize this file to a Java object again, which is shown in Figure 5
- the size of the persisted file, which is shown in Figure 6

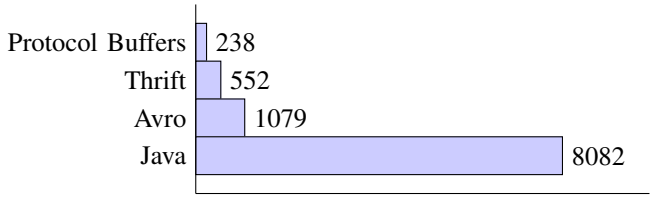

Fig. 4. Write Time (ms)

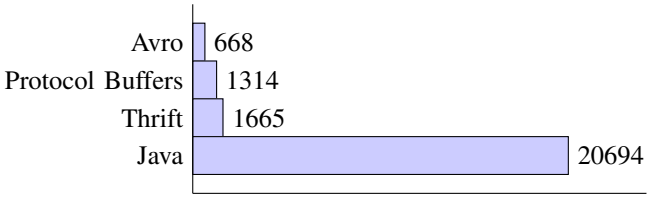

Fig. 5. Read Time (ms)

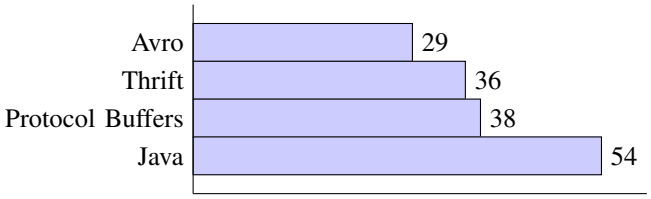

Fig. 6. File Size (MB)

F. Format Decision and Comments

Java Serialization requires a factor 10 more time than the other options and can also not convince in terms of file size. Avro's results show that it is much slower in write speed, however much faster in read speed than Protocol Buffers and Thrift. A winner between these three options is hard to determine and is subject to a weighting of the results. In our use cases it is required that files are small of size and efficient in time that is required to write and read this file. Considering that the schema is fixed and not subject of changes (which Avro would support best), neither requiring RPC functionality nor support in other languages (which Thrift would provide) and specifying that superior timing outweighs space considerations (all formats are very compact), the evaluation leads to the result that Protocol Buffers is the best option for these use cases. File compression in Apache Avro is best, however tests show that the size of created files of data in both Protocol Buffer and Thrift format can still be decreased by one third using compression. This may be taken into account for very old data in a further step.

IV. Implementation

The architecture of Storacle, our data base solution consists of three layers:

1) measurements are initially saved in RAM
2) data are continuously persisted to the local mass storage, where the data are temporarily held available for local application's usage
3) data are uploaded periodically to the cloud for permanent storage

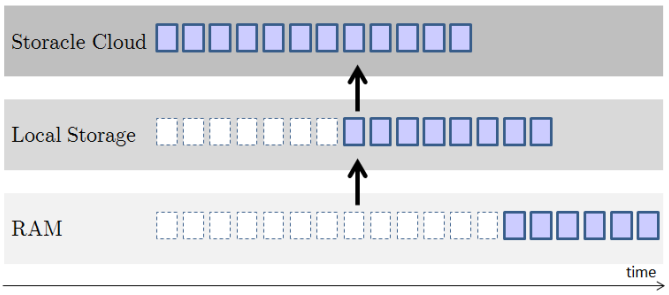

Fig. 7. Three-layer architecture of storage tiers

Figure 7 shows currently available data in each layer as filled boxes. White dashed boxes represent data, which are not required in the RAM respectively at local mass storage anymore, but were present at this layer in the past and are now available at a higher layer. Data present at a lower layer, but not yet locally persisted nor copied to the cloud are not

shown at higher layers. The rightmost box at RAM layer is the most recent entry. Arrows represent the data flow to higher layers.

A. Local storage

The utilization of Protocol Buffers results in very fast serialization. In test cases, no delays occurred when 300 individual sensor nodes, each generating one measurement entry per second, were stored. The data were transmitted over a middleware based on vert.x[1], temporarily saved at the local storage and eventually uploaded to the cloud.

Time series entries, in our use case, not only contain the time stamp and the measurement value, but also a frequency value, describing the time span that elapsed since the reception of the previous measurement, and a delta value, which is the deviation between the time of the original measurement and the time of the reception of the entry at the storage module.

Meta data are saved separate from time series data and include the most recent entry, tags and additional meta-fields for each data point. These tags and meta-fields are both lists of strings, with the only difference that tags can be part of queries. Methods are available to retrieve a list of data points which contain either all or any of the requested tags. Furthermore, meta data can be queried to evaluate whether a data point is already known, and requested to return this data point if available or requested to add this data point to meta data if not. Tags and meta data of a data point in turn can be retrieved, added or removed. Besides that, meta data also include statistics of measurement, frequency and delta values. Statistical values are limited to those that can be calculated without having all (previous) values available. Therefore, each received entry is used only once to update the statistics and is no longer required afterwards. Recorded statistics include the current count of entries (n) that influenced the statistics, the minimum and the maximum value, as well as the mean and the variance value. A histogram is saved for measurement values. The current meta data is periodically persisted to the local mass storage, where in turn it is pulled periodically by the cloud.

Time series data are persisted to one file for each data point. Assuming that one value per second is added per data point (i.e. per file) and one entry shows a size of about 18 bytes in Protocol Buffer representation, about 1 kB of size is required per minute. Once the file size exceeds a defined threshold, it is split in half resulting in an old file and a current file. In tests, we used a maximum file size of 16 kB, which means that in result each (old) file has a size of about 8 kB. The file is split into an old file and a current file approximately each 8 minutes, thus leaving the last 8 minutes in the current file. However, for real use cases it is assumed that the maximum file size will be higher. All data that is part of the current file also resides in the Java heap, thus it can be queried without the need to read from the local mass storage medium. Data is periodically persisted to the local mass storage to provide safety in case of a failure. Methods exist for retrieving an entry for a certain time stamp or a list of entries in a certain time span. It is not possible to retrieve any data that is older than the defined hold time and therefore no longer available at the local storage host. After the expiration of that time, data are available at the cloud only.

The physical location of the data is opaque to the user. Requests to persist data are usually not necessary as data are persisted periodically as well as on shutdown.

B. Cloud

The cloud host is the permanent storage of data that were collected and temporarily saved locally. Data is periodically pulled by the cloud. Files are pushed to the cloud by the local cloud manager only if they were not already pulled by the cloud, therefore the push-method is only a backup method. As already described, files are split into an old file and a current file, if the file size exceeds some defined threshold value. Files get removed on the local storage not before the hold time of the entry is expired and the file was pushed to or pulled by the cloud. In the implementation, a file gets removed on the next run of the local cloud manager if the hold time of the last entry of an old file has been expired. The cloud itself never removes any file from the local storage.

For the connection between the local instance host and the cloud, an SSH connection is used. Files are transferred over the SSH File Transfer Protocol (SFTP).

C. Additional components

An export of time series data in a time span to a CSV (Comma Separated Value) file is supported. The exporter uses files exclusively from the cloud, with the consequence of a delay of availability.

Information regarding data points and meta data can be monitored from both local and cloud hosts by a monitoring program. For each data point, the local and remote file size, the corresponding earliest and latest available entry on both machines and the statistics can be retrieved. As the pull thread runs periodically, the time stamp of the last run as well as the version (file time stamp) of the meta data that is currently available on the cloud can be retrieved. Furthermore, a list of all known data points and the required file size sum on both machines are retrievable.

V. Benchmarks

To show the advantages of our solution we will face it off with the respective winner of each benchmark category described earlier in this paper. As this is an embedded solution, its installation size is small in contrast to standalone solutions.

A. Benchmarks

1) Data size: This benchmark evaluates the size of the data. The importance of this benchmark stems from the limitation of local storage. Results are shown in Figure 8.

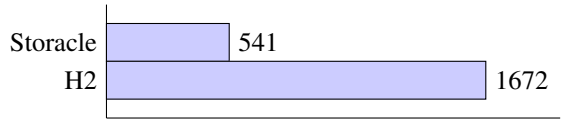

Fig. 8. Size of data (MB)

[1] http://vertx.io

2) Retrieve time slice: This benchmark evaluates the time of the retrieval of all entries of one measurement point in one month (≈2.6 million entries). The retrieval of events in a time slice is one of two queries that occur in the use cases, hence the high relevance. Results are shown in Figure 9.

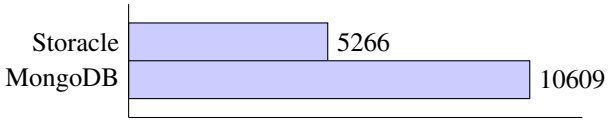

Fig. 9. Time to retrieve one month of entries (ms)

3) Retrieve most recent entry: This benchmark evaluates the time for the retrieval of the most recent entry of one measurement point, which is the second query that occurs in the use cases. Results are shown in Figure 10.

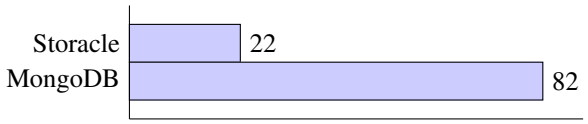

Fig. 10. Time to retrieve most recent entry (ms)

B. Comments and Results

Storacle is best suited for the requirements as due to the use of Google's Protocol Buffers by far requires the least amount of space, performs extremely fast serialization/de-serialization, is fully embedded and thus has low installation size and maintenance complexity, and is optimally tailored to provide an interface for time-series retrieval. As described the solution is limited to time series data and meta data of data points. There was no other solution that came near the values achieved by our solution and could be used as replacement. We inserted 10 years of data for one data point into the data base and measured the time for this operation. The data size increased linear as expected, while influences on retrieval time were minimal. It shows that our solution is scalable.

VI. Conclusion and Future Work

We presented a Java-based embedded data store for storage of Smart Grid time-series measurement data, superior in data retrieval time and needed storage size than off-the-shelf solutions. It was shown that neither SQL nor NoSQL solutions are reasonable for the requirements that stem from the use cases related to Smart Low Voltage Grid operations. There are various standalone solutions for time series databases already available; however for our use cases, data need to be temporarily stored on the local site and later persisted permanently to the cloud. Current solutions addressing time series data are designed to be used at a central server or a distributed cloud, which does not fulfill these requirements. Furthermore, the limited size of storage and the preference for an embedded Java-based solution were not met by any competitors.

As it stands, our solution is both flexible and scalable. With the evolution of Smart Grid use cases, it is expected that further adaptions will be required. A number of parameters govern background processes: file sizes, persistence times, etc. In a future iteration, these parameters could intelligently adapt to local conditions, such as network quality and computational load. For scenarios where local storage space is extremely limited or not available at all, our solution could operate in a RAM-to-cloud fashion. For local operative applications, read-back from the cloud needs to be enabled.

Acknowledgements

This scientific work is part of the research project SCDA - Smart Cities Demo Aspern, which is funded by the program SMART CITIES DEMO of the Austrian "Klima- und Energiefonds" (KLIEN) managed by the Oesterreichische Forschungsförderungsgesellschaft (FFG).

References

[1] S. Akshay Uttama Nambi, M. Vasirani, R. Prasad, and K. Aberer, "Performance analysis of data processing architectures for the Smart Grid," in *Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2014 IEEE PES*, Oct 2014, pp. 1–6.

[2] A. Einfalt, F. Zeilinger, R. Schwalbe, B. Bletterie, and S. Kadam, "Controlling active low voltage distribution grids with minimum efforts on costs and engineering," in *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, Nov 2013, pp. 7456–7461.

[3] C. Oerter, N. Neusel-Lange, P. Sahm, M. Zdrallek, W. Friedrich, and M. Stiegler, "Experience with first smart, autonomous LV-grids in Germany," *IET Conference Proceedings*, pp. 0717–0717(1), January 2013. [Online]. Available: http://digital-library.theiet.org/content/conferences/10.1049/cp.2013.0891

[4] R. Kberle, M. Fiedeldey, B. Brennauer, E.-P. Meyer, M. Metzger, A. Szabo, J. Bamberger, S. Krengel, and T. Wippenbeck, "Messungen und Analysen für aktive Verteilnetze mit hohem Anteil regenerativer Energien und Elektromobilität," in *VDE-Kongress 2012 - Intelligente Energieversorgung der Zukunft*. VDE, 2012.

[5] D. Geibel, T. Degner, A. Seibel, T. Bolo, C. Tschendel, M. Pfalzgraf, K. Boldt, P. Muller, F. Sutter, and T. Hug, "Active, intelligent low voltage networks - Concept, realisation and field test results," in *Electricity Distribution (CIRED 2013), 22nd International Conference and Exhibition on*, June 2013, pp. 1–4.

[6] "H2 benchmarks," http://www.h2database.com/html/performance.html, [Online; accessed 02-October-2014].

[7] S. Prasad and S. Avinash, "Smart meter data analytics using OpenTSDB and Hadoop," in *Innovative Smart Grid Technologies - Asia (ISGT Asia), 2013 IEEE*, Nov 2013, pp. 1–6.

[8] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, Oct. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1239655.1239658

[9] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation," Facebook, Tech. Rep., 4 2007. [Online]. Available: http://thrift.apache.org/static/files/thrift-20070401.pdf