Athens Institute for Education and Research
ATINER

ATINER's Conference Paper Series
COM2018-2554

# A Concept of an In-Memory Database for IoT Sensor Data

**Marina Burdack**
**Research Fellow**
**Aalen University of Applied Sciences**
**Germany**

**Manfred Rössle**
**Professor**
**Aalen University of Applied Sciences**
**Germany**

**René Kübler**
**Software Developer**
**Oskar Frech GmbH + Co. KG**
**Germany**

# An Introduction to ATINER's Conference Paper Series

Conference papers are research/policy papers written and presented by academics at one of ATINER's academic events. ATINER's association started to publish this conference paper series in 2012. All published conference papers go through an initial peer review aiming at disseminating and improving the ideas expressed in each work. Authors welcome comments.

Dr. Gregory T. Papanikos
President
Athens Institute for Education and Research

This paper should be cited as follows:

**Burdack, M., Rössle, M. and Kübler, R.** (2018). **"A Concept of an In-Memory Database for IoT Sensor Data",** Athens: ATINER'S Conference Paper Series, No: **COM2018-2554.**

Athens Institute for Education and Research
8 Valaoritou Street, Kolonaki, 10671 Athens, Greece
Tel: + 30 210 3634210 Fax: + 30 210 3634209 Email: info@atiner.gr URL: www.atiner.gr
URL Conference Papers Series: www.atiner.gr/papers.htm
Printed in Athens, Greece by the Athens Institute for Education and Research. 
ISSN: 2241-2891
04/10/2018

# A Concept of an In-Memory Database for IoT Sensor Data

**Marina Burdack**
**Manfred Rössle**
**René Kübler**

## Abstract

In the context of digital transformation and use of Industry 4.0 technology in companies, machines and other objects are increasingly being equipped with sensors. Normally, these machines are monitored 24/7, so that data streams are continuously generated by sensors. These data has to be stored in a database. In order to facilitate a fast data mining process and the use of machine learning algorithms, a performant and robust data store for the vast amount of sensor data is necessary. These raw time series sensor data has typical structures that are difficult to model with traditional database management systems. Here, column-oriented In-Memory databases like SAP HANA or Gorilla are better suited. However, SAP HANA have not been developed to store relational data, so that it contains components like transaction and concurrency control, which are unnecessary for the named range of application, because machine learning algorithms only need reading access. By reducing this concept to the essentials, a specialized, lightweight In-Memory database management system can be developed, which perfectly fits to the characteristics of time series sensor data. For that concept the benefits of the In-Memory data structure of SAP HANA and Facebook Gorilla are merged and combined with additional meta information like limits for minimum and maximum warning for each sensor, special user specified column fields or rules for sampling and replenishment values. The evaluation of the implemented prototype shows on the one hand that the time series sensor data can be stored efficiently using a new table structure and an intelligent combination of the ZFP compression method with a block orientated data structure, which results in a good insert performance. On the other hand, this storage logic leads to an efficient data access of the compressed in-memory data structure, thus every reporting or analyzing tasks access the data efficiently and fast.

**Keywords:** In-Memory Database, Internet of Things, Machine Learning, Time Series, Sensor Data.

## Introduction

Caused by the digital transformation and the new technologies in kind of Industry 4.0, more and more companies begin to monitor their machines by the use of internet of things technology. For this purpose each physical machine is equipped with different sensors and an application programming interface (API), which allows to communicate with the internet (Zhonglin and Yuhua, 2011). Thanks to the API it is possible to use the internet of things network protocol mqtt (message queue telemetry transport) – a publish subscribe protocol (Hunkeler, Truong, & Stanford-Clark, 2008) - to send the sensor data to another machine like a server which contains a database for storage the data in real-time. For this sending process, the machine normally publishes the data itself by sending it to a defined MQTT-broker like mosquitto ("Eclipse Mosquitto," 2018b) and the server subscribes this data from the same broker and storage it continuous in the database. (Hunkeler et al., 2008) Through this process the data is generated continuously (Gama and Rodrigues, 2007, p. 25) and send in defined time intervals – nearly in real-time - from the machine, so that the data is called streaming data (Gama et al., 2006, p. 1) - especially by the monitoring of sensors. Every data point is combined with a timestamp (Xu et al., 2012, p. 1)–they called time series sensor data. “A data stream is an ordered sequence of instances that can be read only once or a small number of times using limited computing and storage capabilities. These sources of data are characterized by being open-ended, flowing at high-speed, and generated by non-stationary distributions in dynamic environments” (Gama and Rodrigues, 2007, p. 25). Usually many sensors are being used to monitor a single machine. The information from different sensors can be combined to generate new knowledge. For example, a sound, a speed and a vibration sensor can be used to monitor the flow speed and noise of the conveyor. Normally the variables would be dependent, and the combined information leverages the anomaly detection. So, there are many different sensor possibilities to monitor a machine. Normally not only one machine is monitored, still a whole process line with different machines. Thus, a dynamic and complex sensor network exists (Zhonglin and Yuhua, 2011, p. 137). This network creates a huge amount of streaming data (Zhonglin and Yuhua, 2011, p. 1) and allows to generate knowledge (Kanawaday and Sane, 2017). The streaming data has to be stored durable and in way to enable performant data selection for live analyses and predictions (Kanawaday and Sane, 2017).

The objective of real-time monitoring of machines is that in cause of an error or unexpected mistake, the cause can be immediately detected and solved. By the use of machine learning algorithm (Kanawaday and Sane, 2017), it is possible to make predictions on base of the actual sensor data and predict the future error. It is also possible to predict the time when a component has to be changed. Then it is called predictive maintenance.

The best way to store data durable and get a direct transaction based access to the stored data is the use of OLTP (online transaction processing) (Harizopoulos et al., 2008, p. 1) databases. A decade ago, databases were stored on the hard discs, because the cost of the main memory was much more expensive, and the capacity was too limited. Thanks, the new in-memory databases with compressed column stores and the rapid development of main memory in terms of lower cost and higher capacity (Harizopoulos et al., 2008, p. 3), is it possible to fit these OLTP databases in the main memory. Thus the "most OLTP transactions can be processed in milliseconds or less" (Harizopoulos et al., 2008, p. 1). These real-time transactions are a main advantage of in-memory databases like SAP HANA (Färber et al., 2012) to generate real-time knowledge, but they contains components which are not necessary for storing time series sensor data.

The special characteristics of the time series sensor data and the advantages of in memory databases leads to that, the new database must meet certain requirements for real time analyzing and storage the data.

This paper describes a theoretical concept of a new in memory database, that are based on the advantages of the SAP HANA (Färber et al., 2012) and the open source Gorilla (Pelkonen et al., 2015) database architecture. To understand the used technologies different data compression especially lossy (Goyal et al., 2008) and lossless (Kokovin et al., 2018) compression techniques, have to be discuss and the characteristic of time series data are defined.

## Literature Review

In the literature different approaches can be found to store sensor data.

### *Decentral Databases*

Tsiftes and Dunkels (2011) developed Antelope, a database management system, which allows to store a database in every sensor. To store the data, they use flash memory. "Antelope provides a dynamic database system that enables run-time creation and deletion of databases and indexes. Antelope uses an energy-efficient indexing techniques that significantly improve the performance of queries" (Tsiftes and Dunkels, 2011, p. 316). Unlike to our database management system Antelope uses the classical relational database terminology from Codd (1970). To store the sensor data itself, they create a relation which consists of columns for the primary key (an integer value), a sensor ID, the time and the sensor value. In addition to the relation, the architecture contains also an index abstraction and indexer process component (Tsiftes and Dunkels, 2011). But the architecture of Antelope does not allow the storage of user specified meta data for the sensor like minimum and maximum limits or rules for data sampling. They do not describe clearly if the use any compression to store the data.

### *Central Storage Solution*

The decentral database solutions (Tsiftes and Dunkels, 2011) are restricted by the capacity of the storage (Li et al., 2012). Therefore, a long-time storage of a huge period of sensor data is not possible. For machine learning or reports also historical data is needed, so that a central storage solution is better suited.

Li et al. (2012) developed IOTMDB "a storage solution for massive IoT data based on NoSQL" (Li et al., 2012). The system is divided in four main parts: the master node, which manages the slave nodes clusters, the standby node as a backup note of the master node, the data reception node for receiving the data from the sensors and make the preprocessing work of the data and at last the slave node, which contains all sensor data. For the preprocessing they divided the raw data "into two categories: light-weight data like numerical data and characters; and multi-media data like videos, audios and signals. For these two kinds of data, the processes of preprocessing are different" (Li et al., 2012, p. 52). The preprocessing contains the following steps: extracting specific information, cleaning the data, included filling missing values, de-duplicating the data and finally a customer processing step, but the user need programming knowledge, because this step can only done by code. To storage the data, the IOTMDB use sample records which contains different sample elements which represents key values pairs and static and dynamic information (Li et al., 2012). But this data model doesn't contain user specified meta data for each sensor like minimum or maximum limits. It allows user specified preprocessing but only with programming effort. In addition, it is not only used for sensor data, so it contains components which are not necessary. In the actual version they use no data compression technique.

Pramukantoro et al. (2017) describe another central storage solution. They present "a framework consists of the Internet Gateway Device (IGD) function, a Web service, NoSQL database, and IoT Application. The framework efficiently handles the structured and unstructured of sensor data" (Pramukantoro et al., 2017, p. 1). For storing data, they use MongoDB as a NoSQL-database. Each kind of sensor gets an own topic, which belongs to the sensor values. With these topics the values are stored in the database. If they use data compression methods are not described. Finally, it does not allow the saving of meta data to the sensors.

### *Time Series Sensor Database*

To storage only sensor data, so called times series sensor databases are developed. Bader et al. (2017) compared twelve Open-Source time series sensor databases with 27 criteria. They researched if the databases support the following important function for data analytics: insert, read and delete sensor data, scan data over a defined time range, the use of aggregations like AVG, SUM, MIN, Max and the downsampling as a specific function for time series sensor data. Only the database management systems KairosDB (2018), InfluxDB (2018b) and MonentDB (Stratos Idreos et al., 2012) meet all requirements.

KairosDB use the distributed NoSQL database management system (Lakshman and Malik, 2010) apache Cassandra to storage data. To store time series sensor data, the three different "column families" are used: "Data Points Column Family", "Row Key Index Column Family" and the "String Index Column Family" ("Cassandra Schema — KairosDB 1.0.1 documentation," 2018). These database use a data compression method during the writing process ("Cassandra - Documentation - Compression," 2017).

InfluxDB stores time series sensor data in so called logical grouped “measurements”. Each measurement contains to each time value one data point with one or more sensor values (“fields”). To identify each time series in a measurement a unique “tag” is used. The database supports a SQL-like query language ("InfluxData Documentation," 2018b) “The timestamps and values are compressed and stored separately using encodings dependent on the data type and its shape. Storing them independently allows timestamp encoding to be used for all timestamps, while allowing different encodings for different field types. For example, some points may be able to use run-length encoding whereas other may not.” ("InfluxData Documentation - In-memory indexing and the Time-Structured Merge Tree (TSM)," 2018c).

MonetDB (Stratos Idreos et al., 2012) is an OLAP optimized, column based database management system which stores the data in fixed schema in tables. It supports the SQL-Query language and use dictionary encoding for columns which contains string.

In addition to these three databases, the time series data base from Facebook Gorilla (Pelkonen et al., 2015) is also from high interest, because it is “a fast, scalable, In-Memory Time Series Database” (Pelkonen et al., 2015). Facebook uses this database to storage monitoring data from Facebooks-infrastructure for the past 26 hours and allows the automatic warning in case of anomalies and allows rapid searching of errors. The database itself uses for persistent storage the data the Apache HBase database. (Pelkonen et al., 2015) To storage the time serious sensor data, each time series are identified by a unique key. This key is a string which is the only meta data. The time series itself consists only of a pair of columns, one column for the timestamps and one column for the sensor values. New values are inserted with three elements. The first element is the text of the time series as string, the second the timestamps and the third elements are the sensor values (Pelkonen et al., 2015, p. 1818). To select these stored data the Beringei Client can be used. It allows to select one or more time series in a defined time solution, with the same time range. The client decompresses the data blockwise, data outside the requested timeframe is discarded and the time resolution gets downsampled if it is too precise. The Gorilla database stores the different times series in block sizes of two hours. Each time series has exactly one open block where new data can be appended. During the insertion the database uses different compression techniques for the timestamps and the sensor values. If a block has gain the limit, this block will be closed. Closed blocks are only readable and cannot be changed (Pelkonen et al., 2015).

## Methodology

To develop a new in memory database especially for IoT sensor data it is necessary to discuss the characteristic of time series sensor data and how a new database can be developed on the base of existing in-memory databases.

*The Characteristic of Time Series Sensor Data*

The main characteristic of time series sensor data is that they have a defined start and end time (normally endless) and defined range of values. Thus, the values always inner these range and can be used for example to transform and visualize the data in real-time.

But these data have special characteristics. Each sensor generates in regular time data points (Xu et al., 2012, p. 1) which consists a key/value pair. The key is a timestamp which will be generated automatically. The value is the special measurement value from each sensor. Furthermore, they can contain further components like a value unit or other values. In case of the continuously generation of sensor data a data stream of each sensor will be generated which is growing up monotonously by increasing the timestamps. This leads to that new data points never have an older timestamp than the data point before and are only append to the existing data. So, a continuous data stream is generated. If values have to be deleted, only the oldest values (with the lowest timestamp) are removed. Each value of these data stream is not able to be updated. But it is also possible that a sensor records values only then if the value changes. Then unregular values are recorded.

In spite of these main components, it is necessary that before starting monitoring a whole machine, the data structure from each sensor is defined. If the sensor data are send in a sensor network by mqtt (Hunkeler et al., 2008), this structure is used to define for example a json-string structure. So each data point are send via a json string to the mqtt-broker and there it can be subscribed by an client (Hunkeler et al., 2008). The client is able to read this json string.

*Keep it Simple / the Needed Functions of the New Database*

On base of these defined characteristic of time series sensor data, it is shown that the new database needs only a few functions, because time series sensor data is read-only. The values of the times series cannot be updated or deleted. This leads to define, that the time series sensor data are stored in a column store. This allows OLAP operations.

Therefore, that the database is used in context of Industry 4.0 and by use of the mqtt network protocol the database access is controlled, because only the mqtt broker writes into the database. To store the data efficiently, an optimized data structure is needed, because machine learning algorithm only need rapid access for real-time prediction.

In the fact, that the different sensor of a machine generates a huge amount of time series data in few seconds, a light-weight compression method is needed, which allows to store these data in the main memory and allows a direct access to these compressed values. For the purpose of machine learning, special reading commands: scan, average, count and maximum are needed. For sensor values a function of downsampling is appropriate, which allows to summarize values to a lower frequency. (For example, time series values from 0.1s frequency to 1s frequency).

But what about the other components of a typical databases, which are normally used for example for an enterprise relationship management? By monitoring machines, only key/value sensor data is stored, so that no concurrency control and only a light user management system is required. Only the mqtt-broker has writing access to the database and the application for machine learning for example has read-only access.

By the characteristic of time series sensor data and the fact that this data is not updated, no transaction and no complex indexes are necessary. The data never needs to be reorganized, so that the predefined indexes are enough. Additionally, that the database is stored in the main memory no buffer is needed, because the main memory allows fast data access.

The last point is that no data models with definitions of primary and foreign keys are needed, because only sensor values will be stored. And therefore, it is enough to create one table for the sensor values. For more sensor specific information like the maximum and minimum value limit, meta data can be used.

*The Different Data Compression Techniques*

Data compression is a very important aspect by the use and the select of databases especially in the context of the Internet of Things. In only a few minutes, only one sensor can generate a large data volume (Hänisch et al., 2016, p. 4). But if a whole machine is monitored and not only one component, mostly more than one sensor will be used. For example in the die casting project "DataCast" are "three pressure-, five temperature and one form-fill-control-sensor" are used to monitor only the die-casting-mold" (Rössle and Kübler, 2017, p. 7). This leads to an enormous data volume of multiple gigabytes in few minutes (Hänisch et al., 2016, p. 4). To store this data in a fewer size in a database, there are different kinds of data compression methods. The effect is a higher capacity, without expansion of the physical storage. "The aim of data compression is to reduce redundancy in stored or communicated data, thus increasing effective data density" (Lelewer and Hirschberg, 1987, p. 261).

But what is data compression? Data compression is a method, which consists of two paired algorithms. The first algorithm is the coder, which transform the data by the use of different mapping rules into a coded message. The second algorithm is the decoder, which use the same mapping rules as the coder and decode the coded message into the original data format (Lelewer and Hirschberg, 1987, p. 264).

In the literature different data compression methods can be found. These techniques can be split in two fundamental concepts; the lossless compression and the lossy compression concepts. The goal of the lossless compression concept is the remove of redundant data, whereas the original data are full reconstructable. In contrast to the first concept, lossy compression removes irrelevant data, where the original data cannot be completely reconstructed, because information was deleted. The implement the following concept of the in-memory database especially for IoT sensor data, the ZFP lossy compression for floating point data (Lindstrom, 2014) are used.

The ZFP compression is a lossy compression for 32 und 64-bit floating point data (float/double). It is possible to compress one, two or three-dimensional arrays. For higher data dimensionality, a higher compression ratio is possible.

*The Requirements for the Lightweight Database Management System for IoT Sensor Data*

By the upper discussion of the characteristic of time series sensor data the following requirements are defined for a lightweight database management system for IoT sensor data. The goal of this new in-memory database is the faster analyzing of the stored time series sensor data than other database management systems. For that reason, the principals of the primary data storage in the main storage and the compression of data structures with direct access should be used. The main functionality of this database management system is the section insert, select and storage of sensor data.

Insert New Time Series Sensor Data

The database management system has to secure, that by the insert of new data the technology of data streams can be used. This means, that the different data points of a sensor are being send in this order as they are generated from the sensor. Nevertheless times series sensor data, which is not generated in constant sequence has be stored without adaption on the client. Furthermore, it should be able to save metadata to each time series as for example name, type, measurement accuracy, unit and place. Additionally, it should be possible to define border values for each sensor, which allows an extern program to show a direct warning if any input value is out of range. The border values are necessary to find quick mistakes in each sensor based on wrong time series data values.

Select Time Series Sensor Data

The new in memory database supports the classical aggregation function: sum, avg, min and max for each column to select stored time serious sensor data. All queries are limited to one time series and time area. As an option, each column can be limited. As a special operation it supports downsampling. Downsampling is being used to group the data to a lower time resolution.

These operations allow the analysis of the data direct, without sending the whole data for further analyzing to a third program.

Storage

The new in memory database must ensure that if the volatile memory is erased, for example through loss of current, the data can be restored.

*The Concept of the New in Memory Database*

Based on the defined requirements and the goal to store the time series sensor data efficient and performant for explorative data analyzing purposes, the following concept of an in-memory database is suggested.

This concept is based on the special characteristics of the two in-memory databases SAP Hana (Plattner, 2013) and Gorilla (Pelkonen et al., 2015). SAP Hana stores the primary data in the main memory, which allows a rapid access to the data. It is also stored in write optimized form on the persistent storage. If the volatile memory is lost, the database can be restored. Thanks to special characteristics of the time series sensor data – add new values with append only – an in-memory data structure like the data structure from the database Gorilla can be used. This data structure allows – in contrast to the data structure from SAP HANA - to add new time series sensor data to the database without any difficult merge-process.

Logical Database Level

On the logical level of the concept the data is organized in a table structure. This structure has to be defined before the first time series values are added to the new table. This data structure contains the following information:

- **Table name**
- User specified fields table of (Key-Value)
- Time resolution of the table (definition of the step size)
- Start time
- **columns**
    - **column name**
    - User specified column field (Key-Value)
    - Limit warning minimum
    - Limit warning maximum
    - **rule for sampling values**
    - **rule for replenishment values**
    - **column group**
        - **ZFP type (float or double value)**
        - **Fixed rate**

To create a new table, the table name and minimum one column with the column name, the rule for sampling the input values, the rule for replenishment the missing values and the definition of the column group consisting of the ZFP type and a fixed rate, have to be defined. The start time is set by the system itself after adding the first time series data point to this defined table. Therefore, every table represents an equidistant timeline that means between every data point the same time difference exists and therefore the table contains an implicit time index. This time index can be defined by the start time and the time resolution of the table. By editing the rule for sampling values, it can define how the input values should transformed to the defined table time resolution. Thus, it is possible to store different kind of sensor in one table with different frequency. About the restriction, that by inserting new data in the table structure, all columns have to be filled, it is possible to define a rule for replenishment missing values. A conceivable rule is that the last value is taken to replenishment the missing value. In contrast to the missing values, it is possible by higher time resolution by one sensor, that to one timestamp are more than one sensor value are available. To solve this input problem, a rule for sampling values can be defined. For instance, these values are dropped or summarized. Another rule can be that the first value or the arithmetic mean of these values is stored. In addition to these data values user specified meta data can be stored on the base of the table definition and on the base of each column definition. To ensure that a third part program is able to show warning by the insert process of the time series sensor data, the border values of each sensor have to be defined. This will be done by define limit warning minimum and limit warning maximum for each column definition.

Insert from Time Series Sensor Data

Inserting time series sensor data, always one data point will be added to one table. For this purpose, all columns of this table structure have to be filled. Therefore, it is necessary that by creation the table structure rules for replenishment and sampling sensor data values for each sensor are defined. A big advantage of time series sensor data is that they are streaming data and new data points always have a higher timestamp than the last data point. This ensure the rule, that only time series sensor data are allowed to be added, which contains a higher timestamp as the last input values of the table.

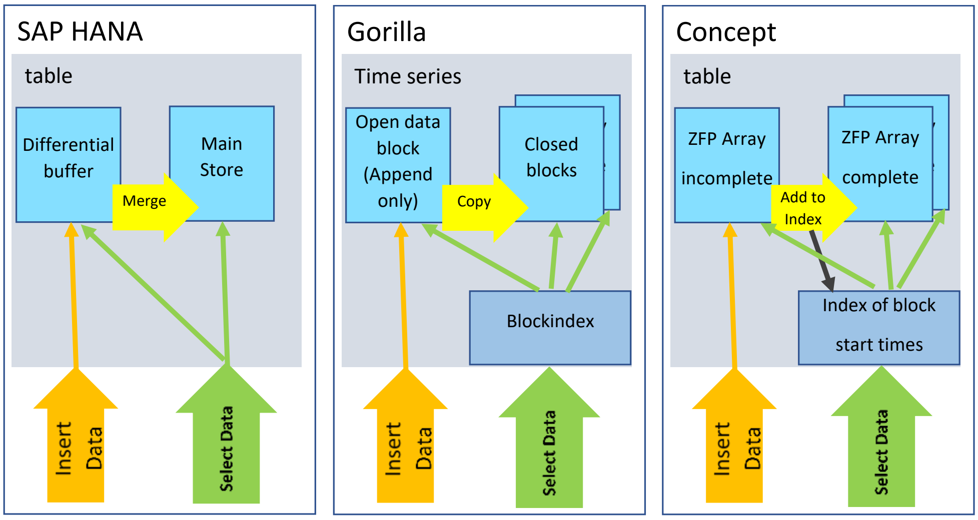
Select Time Series Sensor Data from Table Structure

To select time series sensor data from the stored table, at least the table name is necessary. But normally queries will be done where the timeline and the columns are limited. To select data in this in-memory database, the SQL like language from the time series database InfluxDB (2018a) works well.

The Physical Database Level

On the base of the in-memory data structure the concept is similar to the gorilla database, because both concepts structure their data in frames. But in contrast to the concept of the gorilla database the kind of data compression and the show of the time values have to be adjusted. In the Figure 1 the data storage of each database are shown. SAP Hana use for storage data the differential buffer ("Delta Store") and the main store. The differential buffer is used to insert-, delete- und update values (Plattner, 2013, p. 167). The changed values are transferred with a merge process to the main store (Plattner, 2013, p. 181). It is possible to query values from both components. In contrast to SAP Hana, Gorilla organizes the sensor data in blocks. Each time series contains own data blocks. Each block stores values of two hours. To each time series an open data block exists. This new data is added only to this open block as append only. If a block is closed, it will be copied to another storage part and is added to a block index. At the closed blocks only reading access is allowed. It is also possible to query data from open blocks (Pelkonen et al., 2015). The last component in the Figure 1 on the right shows the new developed database storage concept, which will be following explained.

**Figure 1.** *Abstract Overview about the In-Memory Data Storage Concepts of SAP Hana, Gorilla and Own Concept*



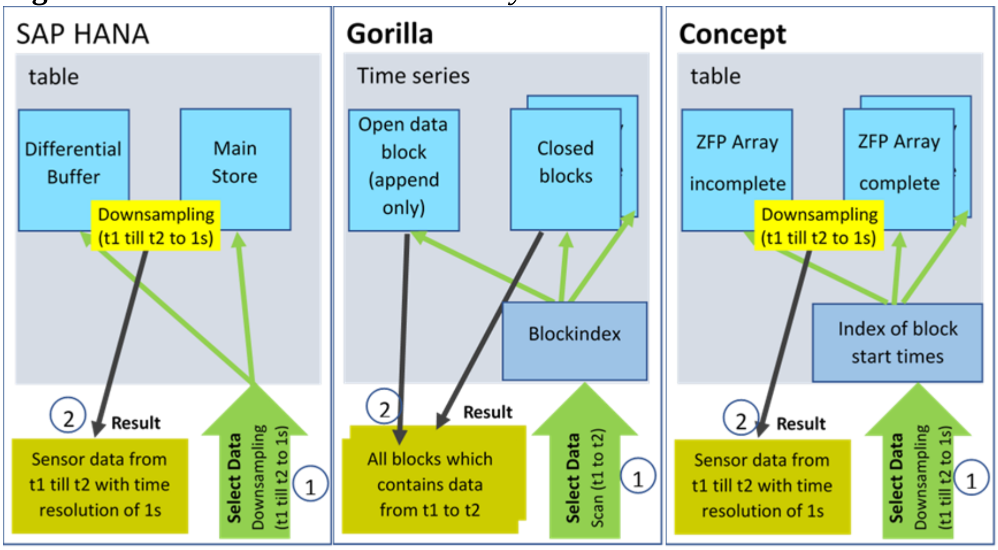

Insert New Time Series Sensor Data

The gorilla database uses a compression method, which is optimized for monitoring data. It compresses data frames in a two hour time interval and a lowest temporal resolution from one second (Pelkonen et al., 2015). By inserting new data in an existing open data block the new values are being compressed during import and are added to the existing data block by append only. The existing open data block is flexible sized, so that it sizes up, by each added value until the limit of two hours is reached and the block is closed. At this moment the block only contains compressed values. An additional compression of the full block is not necessary. After closing the block it will be copied to another storage area (Pelkonen et al., 2015). For the concept of the new in memory database the base logic from the gorilla database is taken, but the size of one block is not flexible, it will be fixed in the beginning. Storage for a full ZFP-Array is reserved for the used ZFP compression method. On the base of the logic of the Gorilla database each data block has a time limit of two hours. This leads to the fact that with the declared resolution time of a sensor a limited count of values can be inserted in an open data block. If the limit is arrived the data block will be closed, and a new empty data block will be created. The old data block gets an index based on the timestamp of the first input values and is added to the index (see Figure 1).

Select Time Series Sensor Data

By query the time series sensor data with a select statement (number 1 in Figure 2), Gorilla return (number 2 in Figure 2) the whole compressed block to the client. In contrast by the use of ZFP arrays in the new in memory database architecture (right figure in Figure 2) is it possible to get a direct access to the stored values, caused by the transparent compression method. As a result, the sensor data are displayed in the needed downsampling rate und time resolution. The same theory is used by the dictionary encoding (Plattner, 2013, p. 37) in the SAP Hana database (left figure in Figure 2). This allows the direct performance of aggregations functions on the compressed in-memory data structures, as illustrated in the following **Error! Reference source not found.**. As a result of a query, also the needed sensor data are displayed. Figure 2 also shows that the three databases contain three different query interfaces. SAP HANA allows the downsampling of sensor data in queries at the access to the differential buffer and the main store. On the contrary, the Gorilla database allows only queries on the block index, which linked with different kind of blocks. Each block is stored with a special sampling rate of data. Downsampling of stored data are here not possible. Through the combination of the block logic from the gorilla database and the downsampling possibility from SAP Hana the big advantage of the new in-memory database is generated. It is the possibility to use downsampling in queries to access data over the block index structure with the stored start time of each compressed ZFP-array. So, it is possible to execute a query with downsampling function direct on the compressed in-memory-data structure.

**Figure 2.** *Data Access to the In-Memory Data Structure*




Storage of the Time Series Sensor Data

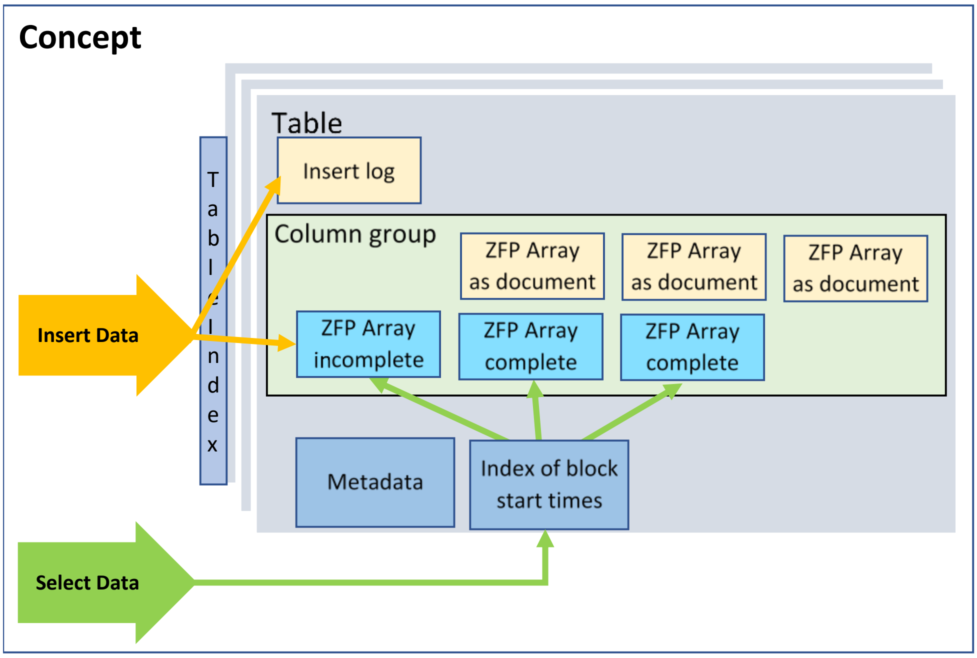
The important second point is the storage of time series values. The Gorilla database stores time values explicitly (Pelkonen et al., 2015). The new in-memory database is able to optimize the storage of the time values through an implicit equidistant time index. This equidistant time series have not to be stored, because it is possible to calculate them with the following function:

$$t = ax + s$$

where s = start time, x = row number and a = distance between the different time values. In this way a constant access of O(1) to the elements is possible. Another possibility is to store the explicit time index. Because the time values of a time series are strength monotonously increasing. Through the rule that only values with a higher timestamp can be added to the table, the time index is automatically sort.

The logical separation of the table in table groups – as in the following Figure 3 shows - makes it possible that data with similar data value range can be stored in high compressed two-dimension ZFP-Arrays.

**Figure 3.** *Design for Optimized Storage of the Time Series Sensor Data in the Physical Database Level*




Each ZPF-Array is split into blocks. If one block is complete, a new block will be created and the completed block is saved in the background as a document. If a table contains more than one column group, all column groups have the same size. The new block start time is added to the index and the insert operations write into the new ZFP-array. To avoid data loss by restarting the database server or during a power off, the data in the uncompleted arrays is stored in a write optimized insert log. An insert log contains same count of values, like the uncompleted block and it is only necessary until the block is not completed and saved. After a loss of the data from the main memory, the insert log and the saved blocks can be used to recovery the in-memory structure in the main memory.

Handling Missing Values

ZFP arrays are not able to store NaN (“not a number”, undefined or unrepresentable value) and infinite values. But by the work with time series sensor data and the connection via mqtt it is possible that the connection is lost for a few seconds, and no new time series sensor data values are send. Perhaps only the timestamps can be located but not the value. This situation has also been displayed. For this problem some possible solutions exist. A first solution is the possibility to define a very high default value as a mistake marker that the mistake can be shown for example in a real-time monitoring dashboard. The disadvantage at this solution is that this value is disrupting the compression block where it belongs to. A second solution is to define a rule to fill automatically missing values in the table structure. Through this, it is possible to close little spaces in the time series and the values are compressed very well.

Another solution is a one-bit flag, which indicates the value as original sensor value or as a replaced missing value.

## Findings

To evaluate the concept of the new in-memory database, two kinds of in-memory databases are developed in C++. The first prototype includes the ZFP compressed in-memory data structure by use of the ZFP library (Lindstrom, 2014), ("LLNL/zfp") . The second prototype uses the uncompressed in-memory data structures std::vector ("std::vector – cppreference.com," 2018g) from the C++ standard-template-library. With the second prototype the disadvantages of the compression method will be shown. To evaluate the new in-memory database a benchmark with different kind of databases and the uncompressed in-memory database are conducted. For the benchmark the duration time in seconds of the operations: SCAN (queries on time ranges), AVG (calculation of the arithmetic mean), MAX (highest sensor value), downsampling (aggregate values to a smaller time resolution) and the compression rate are detected.

As test data the time series sensor data of the research project “DataCast” (Rössle and Kübler, 2017) is used. This data set contains 25.303.888 values and needs as uncompressed double value 193 MB of storage. The following databases are used: the KairosDB ("KairosDB," 2018) a shared on time series data optimized database on the base of Apache Cassandra, the InfluxDB (2018b) a time series sensor data optimized database and the MonetDB (Stratos Idreos et al., 2012) as a column stored relational database.

*Evaluation of Different Configuration Option of the ZFP-Library*

The compression tests are split in different areas. First the different "DataCast" (Rössle and Kübler, 2017) sensor are single compressed with ZFP-1d arrays. In the following test, sensor data of more sensors are together compressed in ZFP-2d arrays. The last test contains the determination of the duration time of the different operations. For this we use the ZFP-2d array compression for the own in-memory database and the other selected databases.

Data Compression with ZFP 1d-Arrays

The “DataCast” sensor values are compressed once with a compression ratio of 16 bit per value and once with a compression ratio of 21 bit per value. By use the first compression ratio it takes 878ms to compress the whole “DataCast” data set. In this case the storage size will size down to 48mb with the accuracy of two till three decimal places. In contrast to the first compression ratio the second takes 1355ms and needs 96mb storage, but with an accuracy of seven till eight decimal places.

Data Compression with ZFP 2d-Arrays

First all “DataCast” values are compressed in a 2d-array by the use of different kind of bit per values. As of 12 bit per values a compression ratio of 3.4 is achieved. Here 64 million values are inserted in one second. Afterwards the sensor data is compressed with different bit per values in similar sensors namely pressure and temperature sensor. Even by the lowest bit per value rate of four bit, there is no mistake in the accuracy. Thereby the storage size of the raw sensor data is reduced up to a sixth by a compression time of 262 million values per second.

*SCAN, AVG, MAX and Downsampling*

To show the effort of the ZPF-compression a second prototype of the new in-memory database is implemented, which uses the uncompressed C++ vector instead of ZFP-arrays. In addition to this database, the first prototype and the above selected databases are compared in the following benchmark. The duration times of the scan, avg, count, max and downsampling operations on basis of the “DataCast” values are evaluated and the compression ratio for each database is calculated.

Figure **4** illustrates the result of the benchmark in an overview.

**Figure 4.** *Benchmark with DataCast Sensor Data (Rössle and Kübler, 2017)*

| | **Scan** | **AVG** | **COUNT** | **MAX** | **Downs.** | **Compr.-rate** |
|---|---|---|---|---|---|---|
| KairosDB | 3.3 s | 78.4 s | 57.0 s | 0.6 s | 16.2 s | 1.33 |
| InfluxDB | 1.7 s | 1.9 s | 1.5 s | 0.04 s | 0.19 s | 1.8 |
| MonetDB | 8.2 s | 0.09 s | 0.06 s | 0.007 s | 0.11 s | 1.0 |
| ZFP | 0.012 s | 0.65 s | 0 s | 0.004 s | 0.29 s | 3.7 |
| C++ Vector | 0.0008 s | 0.035 s | 0 s | 0.0004 s | 0.004 s | 1.0 |

For evaluation of operations on the different databases the compression configuration for the ZFP-array has to be defined. For the ZPF-array the "DataCast" sensor values are compressed with a fixed rate from 16 bit per value and need 965 MB of main memory. This configuration in addition with the use of the implicit time index provides a compression ratio of 3.7 and shows in contrast to the other databases the best value. But through this compression the "DataCast" values lose the accuracy after the third decimal.

The evaluation of the scan operations by a duration time of one second shows, that the ZFP-arrays are in contrast to another database with 0.012s faster, but the uncompressed C++ vector takes only the 10,000 time from the MonetDB.

The first mathematical reading operations average (AVG) is executed over all temperature sensor values with timeline of two hours. The evaluation shows that the KairosDB is the slowest database with over 70 seconds and the ZPF-array takes for the operation less than one second but is slower as the MonetDB. The fastest database is the C++ vector.

In order to get the maximum of the temperature sensors of the "DataCast" values the count operation is used on a time range of ten seconds. All databases need less than one second. Although the C++ vector without any compression method is 100 times faster than the Influx DB and the fastest database in this case in the test.

The test of the count operation shows that both prototypes count the values extremely fast, because they use the implicit time index, so that only the highest row index has to be read. This is a big advantage in contrast to the other databases.

For evaluation of the downsampling method the values of one hour are used to calculate the average of the values in time slots of 15 minutes. The result of this test shows, that KairosDB needs with 16 seconds the longest. But the ZFP-array not perform and takes 0.29 seconds. The fastest database in the test is also the C++ vector with 0.004 seconds.

In summary both self-implemented prototypes of in-memory databases show that the concept of the ZPF compression performs. The ZFP version has a very good compression ratio and still can perform with the scan, count and max operation on the "DataCast" values. For the average and downsampling operation the MonetDB is the quickest database. But the benchmark also shows that the in-memory database with the uncompressed C++ vector is the fastest database in all categories.

## Conclusions

In conclusion the first prototype of the new in memory database merges the benefits of the in-memory data structure of SAP HANA and Facebook Gorilla databases. Through the block orientation of Gorilla, the disadvantages of the complex merge process, performed by SAP Hana, can be eliminated.

Additionally, it shows on the one hand, that time series senor data can be stored efficiently using a new table definition and an intelligent combination of the

ZFP compression method with the block orientated data structure. In this kind the idea of the transparent compression method from SAP HANA is used, where transparent compression methods are based on the dictionary encoding. Thus, a high compression rate of the time series sensor data can be achieved. On the other hand, this storage logic leads to an efficient data access of the compressed in-memory data structure, thus every reporting or analyzing for example by the use of machine learning algorithm can work rapid and efficiently. Furthermore, the database allows every reading operation, like for example mathematical operations, on the stored data.

Finally, thanks the storage on main memory and the use of in-memory technology, the new in-memory database can be perfectly use for storing IoT time series sensor data. To gain knowledge of the stored data only real-time reading accesses to the data are necessary. The database also works with streaming data, so that a real-time monitoring of machines is possible and in case of errors an immediate reaction is guaranteed.

## Further Work

On the results of the first prototype a new prototype will be implemented. By monitoring a machine over 24 hours a day and seven days a week a huge amount of data has to be stored, but for real-time prediction and real-time monitoring in the most cases only the last two or four hours of data are interesting. What about the old data, which must be stored for documentation and reporting purposes? Now, these data has also stored in the main memory. But the main memory is fewer limited as the hard disk, because it is more expensive. So, a new component and technology has to be added to the first prototype. SAP HANA in this case uses a nearline storage to move old data onto a hard disk, where the reporting tools are able to access to these data. Because our database was developed especially for rapid data access of time series sensor data, we have to be aware, that the use of this technology will split the data in two different storages, so the data access may lose performance. Therefore, we have to research and evaluate different storage solutions for old data and add the best to our in-memory database solution.

## References

Bader, A., Kopp, O., Falkenthal, M. (2017). Survey and comparison of open source time series databases. *Datenbanksysteme Für Business, Technologie Und Web (BTW 2017)-Workshopband*.

Cassandra - Documentation - Compression. (2017). Retrieved from http://cassandra.apache.org/doc/latest/operating/compression.html.

Cassandra Schema — KairosDB 1.0.1 documentation. (2018). Retrieved from http://kairosdb.github.io/docs/build/html/CassandraSchema.html.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, *13*(6), 377-387. https://doi.org/10.1145/362384.362685.

Eclipse Mosquitto. (2018b). Retrieved from https://mosquitto.org/.

Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W. (2012). SAP HANA database. *ACM SIGMOD Record*, *40*(4), 45. https://doi.org/10.1145/2094114.2094126.

Gama, J. and Rodrigues, P. P. (2007). Data Stream Processing. In J. Gama & M. M. Gaber (Eds.), *SpringerLink: Springer e-Books. Learning from Data Streams: Processing Techniques in Sensor Networks* (pp. 25-39). Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg. https://doi.org/10.1007/3-540-73679-4_3.

Gama, J., Rodrigues, P., Aguilar-Ruiz, J. (2006). An Overview on Learning from Data Streams. *New Generation Computing*, *25*(1), 1-4. https://doi.org/10.1007/s00354-006-0001-5.

Goyal, V. K., Fletcher, A. K., Rangan, S. (2008). Compressive Sampling and Lossy Compression. *IEEE Signal Processing Magazine*, *25*(2), 48-56. https://doi.org/10.1109/MSP.2007.915001.

Hänisch, T., Rössle, M., Kübler, R. (2016). Storing Sensor Data in Different Database Architectures. *Athens: ATINER'S Conference Paper Series, No: COM2016-1955*.

Harizopoulos, S., Abadi, D. J., Madden, S., Stonebraker, M. (2008). OLTP through the looking glass, and what we found there. In L. V. S. Lakshmanan, R. T. Ng, & D. Shasha (Eds.), *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (p. 981). New York, New York, USA: ACM Press. https://doi.org/10.1145/1376616.1376713.

Hunkeler, U., Truong, H. L., Stanford-Clark, A. (2008). MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks. In *COMSWARE 2008: The Third International Conference on Communication System Software and Middleware and Workshops: Bangalore, India, 5-10 January* (pp. 791-798). [Piscataway, N.J.]: IEEE. https://doi.org/10.1109/COMSWA.2008.4554519.

InfluxData Documentation. (2018a). Retrieved from https://docs.influxdata.com/influxdb/v1.5/.

InfluxData Documentation. (2018b). Retrieved from https://docs.influxdata.com/influxdb/v1.5/.

InfluxData Documentation - In-memory indexing and the Time-Structured Merge Tree (TSM). (2018c). Retrieved from https://bit.ly/2zMHg0b.

KairosDB (2018). Retrieved from http://kairosdb.github.io/.

Kanawaday, A. and Sane, A. (2017, November - 2017, November). Machine learning for predictive maintenance of industrial machines using IoT sensor data. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)* (pp. 87-90). IEEE. https://doi.org/10.1109/ICSESS.2017.8342870.

Kokovin, V. A., Uvaysov, S. U., Uvaysova, S. S. (2018, March - 2018, March). Real-time sorting and lossless compression of data on FPGA. In *2018 Moscow Workshop on Electronic and Networking Technologies (MWENT)* (pp. 1-5). IEEE. https://doi.org/10.1109/MWENT.2018.8337187.

Lakshman, A. and Malik, P. (2010). Cassandra. *ACM SIGOPS Operating Systems Review*, *44*(2), 35. https://doi.org/10.1145/1773912.1773922.

Lelewer, D. A. and Hirschberg, D. S. (1987). Data compression. *ACM Computing Surveys*, *19*(3), 261–296. https://doi.org/10.1145/45072.45074.

Li, T., Liu, Y., Tian, Y., Shen, S., Mao, W. (2012, November - 2012, November). A Storage Solution for Massive IoT Data Based on NoSQL. In *2012 IEEE International Conference on Green Computing and Communications* (pp. 50-57). IEEE. https://doi.org/10.1109/GreenCom.2012.18.

Lindstrom, P. (2014). Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, *20*(12), 2674-2683. https://doi.org/10.1109/TVCG.2014.2346458.

LLNL/zfp. Retrieved from https://github.com/LLNL/zfp.

Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J., Veeraraghavan, K. (2015). Gorilla. *Proceedings of the VLDB Endowment*, *8*(12), 1816-1827. https://doi.org/10.14778/2824032.2824078.

Plattner, H. (2013). *Lehrbuch In-Memory Data Management: Grundlagen der In-Memory-Technologie*. Wiesbaden: Springer Fachmedien Wiesbaden GmbH.

Pramukantoro, E. S., Yahya, W., Arganata, G., Bhawiyuga, A., Basuki, A. (2017, October - 2017, October). Topic based IoT data storage framework for heterogeneous sensor data. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)* (pp. 1-4). IEEE. https://doi.org/10.1109/TSSA.2017.8272895.

Rössle, M. and Kübler, R. (2017). Quality Prediction on Die Cast Sensor Data. *Athens: ATINER'S Conference Paper Series, No: COM2017-2272*.

Std::vector – cppreference.com. (2018g). Retrieved from https://bit.ly/2QqKUSW.

Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, Martin L. Kersten. (2012). MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, *35*(1), 40-45.

Tsiftes, N. and Dunkels, A. (2011). A database in every sensor. In J. Liu, P. Levis, & K. Römer (Eds.), *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems* (p. 316). New York, NY: ACM. https://doi.org/10.1145/2070942.2070974.

Xu, Z., Zhang, R., Kotagiri, R., Parampalli, U. (2012). An adaptive algorithm for online time series segmentation with error bound guarantee. In E. Rundensteiner (Ed.): *ACM Digital Library, Proceedings of the 15th International Conference on Extending Database Technology* (p. 192). New York, NY: ACM. https://doi.org/10.1145/2247596.2247620.

Zhonglin, H. and Yuhua, H. (2011, August - 2011, August). Preliminary Study on Data Management Technologies of Internet of Things. In *2011 International Conference on Intelligence Science and Information Engineering* (pp. 137-140). IEEE. https://doi.org/10.1109/ISIE.2011.63.