Technical Report CS 01-19

Database support for automotive analysis

Dennis Marten, Holger Meyer and Andreas Heuer

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik
Lehrstuhl für Datenbank- und Informationssysteme

Long version of a 5-page conference paper for the conference "Lernen, Wissen, Daten, Analysen" (LWDA) 2019,

erhältlich unter:
www.ls-dbis.de/digbib/dbis-tr-cs-01-19.pdf

Database support for automotive analysis

Dennis Marten[1], Holger Meyer[1], and Andreas Heuer[1]

Institute of Computer Science, Rostock University, Albert-Einstein-Strasse 22,
18059 Rostock, Germany { dm, hme, ah }@informatik.uni-rostock.de

Abstract. Based on an analysis of typical automotive measurements data as found in ASAM MDF files, we derive requirements managing these data in a database system and create a mapping to a relational database structure. The performance of a parallel relational storage gets compared to direct access and querying time-series from Python in different scenarios. A hybrid approach using some object-relational features of PostgreSQL performs best in most cases.

Keywords: automotive data · time-series data · time merge · relational database systems · ASAM MDF files

1 Introduction and Motivation

In the automotive industry the amount of collected sensor data is continuously growing. This imposes new requirements on the IT infrastructure regarding acquisition, storage, preprocessing and analysis of data. Nowadays, big data frameworks mostly rely on database software systems. Therefore, our investigations will consider these primarily. We compare several hardware and software systems and show to what extent they meet the requirements. In this paper we explain how database oriented solutions for storage and analytics can be used to effectively handle time series sensor data.

The paper is structured as follows. In the second section a summary on research projects and established systems that combines time series management and database techniques is given. The following sections gives a brief overview of classic automotive analysis methodology, including the standardized measurement file format MDF and a set of basic operations. This overview is used as a motivation for the database supported solution presented in section 4, which is evaluated in section 5. Lastly, a conclusion is given in section 6 with a description of future research opportunities we consider to investigate.

2 State of the Art

To the best of our knowledge, there are no research projects that combine the standardized automotive data format ASAM MDF and database technology, but many efforts have been taken for managing efficiently time-series data in databases. At least, Johanson [10] calls for information and communication support for automotive testing and validation.

Seminal work of managing time-series data was already done starting in the 1990s, e.g. Chandra [4] describes how to manage temporal financial data in an extensible relational database. In his PhD thesis [3], Castillejos focusses on models for mapping time-series data to relational databases. Modelling univariate, multivariate and irregular time-series is lined out, compared and validated. The approach didn't take advantage of nested (object-relational) structures and aims at stock price scenarios. The sequential character of time-series and their large scale volatile data sets led to specialized systems for data streams. Issues in data stream management are discussed by Golab and Öszu[7]. The importance of efficiently manage time-series data is demonstrated by e.g. several Google patents [9] among others.

Intensive research has already be done on indexing, querying and mining time series data at large. Ding et al.[6] give an experimental comparison of representations and distance measures for querying and mining of time-series data, where Zoumpatianos [20] focusses on indexing very large data sets. Keogh et al. [11] summarizes the state-of-the-art in indexing and mining large time-series databases.

Aside research done on managing and querying time-series data, there are several systems capable of managing sequential data. At the time of writing, Wikipedia lists fifteen time-series database systems, like InfluxDB, Informix TimeSeries, and TimescaleDB. These systems are relational, object-relational or use more flexible storage models like NoSQL systems and cloud data management does. An overview of time-series storage and processing in a cloud environment is presented in [18]. [19] describes how to manage very large sensor-network data using bigtable, Google's variant of a column family data storage in a distributed file system (GFS). Call [16] for a comparison of NoSQL time-series databases.

3 Specification of Data Formats and Analysis

In this section, we give a brief overview on traditional automotive data analysis, including a short description of the MDF file format, as well as examples of basic automotive data analysis. All these aspects directly influence the design of the database support approach discussed in section 4.

3.1 The ASAM MDF File Format

The Measurement Data Format (MDF) is a binary file format that has been originally developed for automotive measurement and analysis in the 1990's. It has been officially standardized with version 4.0 by the Association for Standardization of Automation and Measuring Systems (ASAM) in 2009 and is up to this point one of, if not the standard for storing and reading automotive time series data in industrial usage. [2]

The MDF file format stores time series of channels clustered in channel groups. Every channel of a group shares the same time axis. Internally, MDF

files are organized in a complex system of structured blocks which are combined by pointers in order to allow fast navigation. [17]

3.2 Basic Automotive Data Analysis

One of the basic approaches for automotive analysis is to read and process MDF files via an high-level programming language like Python or Matlab as analysts usually have comparatively little time for a wide range of operations and varying data. Naturally, there are many different and practically relevant forms of analysis that can be done with structured automotive data. Her, we focus on two basic scenarios we have investigated that do overlap with a wide range of typical automotive analysis and are evaluated in section 5. The first scenario revolves around the investigation of a set of channels over several test runs, which are all stored in individual files. Such a comparison requires multiple basic operations, like the conversion of sensor data from its raw fixed-point form to meaningful physical values, usually stored in double precision. Depending on the problem and the corresponding channelgroups, the converted time series need to be interpolated, so that all channels share the same time axis. This is fundamental for further inter-channel processing, like pointwise comparisons and operations, for instance finding intervals under constraints [e.g. (channel A)^2 > channel B] or a mapping of multiple channels to a new time series using basic arithmetic operations. Interpolation of channels also called "time merge" can be done in many different forms. This is, next to the selection and conversion of channels, one of the fundamental operations that is worth pushing near the data source, as it can heavily reduce network traffic. Contrary to this, a last set of operations does frequently include the visualization of interpolated and composed channels; a step that is best processed locally via data visualization libraries.

The second scenario we have investigated can rather be described as a meta-information query as its purpose is to find files or measurements that satisfy a set of requirements. As will be seen in subsection 5.2, these types of preprocessing steps can be done very efficiently in database systems as these are designed for these kind of restricted inter-file selections.

These two scenarios have functioned as a starting point for our investigations, while more sophisticated applications like peek, outlier or overall pattern detection might be included in future research.

3.3 Limitations of an MDF-based Analysis

Naturally, any approach on measuring and processing data that can vary extremely in size and structure cannot come without its limitations. As the MDF file format can store (and compress) streaming data of multiple local devices, while offering a structure that allows fast and selective data access, its complex pointer-based design is somewhat based on the premise of local availability. This assumption is also based on the observation that packages we have tested for reading MDF files cannot run simultaneously on a single file, i.e., they are

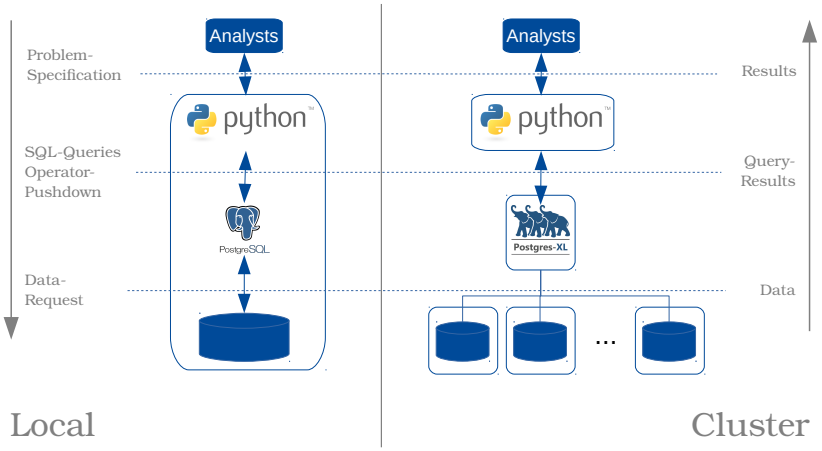

Fig. 1. Vertical architecture for database-supported automotive analysis. Blue borders represent individual nodes.

implemented for single threaded use and are not re-entrant.[1] The latter is no constraint if data is available on local storage.

While MDF4 files do not have a restriction on its actual size, analyzing a set of large or even many moderate size files can become hardware demanding for local setups. Neglecting possible main memory shortages on long measurements, the disk storage problem can be worked around using network drives. However, as MDF files cannot be accessed from multiple instances at once, use cases where several analysts work on the same set of files need to implement additional scheduling strategies. Despite these restrictions on computability, the network drive approach is used as a reference for the performance of the database approach described in section 4 and evaluated in section 5.

4 Database Support

In order to overcome imminent data size limitations we reviewed solutions that are based on database support with Python front ends using a conventional vertical architecture as depicted in Figure 1. Hereby, we took two groups of systems into considerations:

1. relational database systems, and
2. NoSQL database systems combined with Apache Spark.

[1] Two python packages were evaluated and `asammdf` choosen for better performance. Actually, the documentation for `mdfreader` 3.2 claimes to be optionally multi-threading usable while the favored `asammdf` produces non-repeatable read in multi-threading application.

These system does not only offer scalability, but many more advantages that can benefit automotive designs, such as:

- central and consistent data storage
- isolated, simultaneous access on data
- transparent physical and logical optimization
- parallel query processing
- fast selective queries via index structures
- data security

We investigated either storing converted time series or raw fixed-point data with its corresponding conversion rules and online conversion into its floating-point representation. Furthermore, we differentiated between local and cluster computations and took a set of three basic automotive analysis (including the two scenarios described in subsection 3.2) for possible experimental evaluations.

In the following subsections we describe aspects that fundamentally influenced the efficiency and choice of the developed database models. Hereby, we focus on relational database systems, especially PostgreSQL and its parallel branch Postgres-XL [1] as both offer a wide range of functionality, a BSD-like license and has shown promising results for the described automotive scenario.

4.1 Choice of Database System and Compression

As channels that are stored in channel groups share the same time axis and therefore hold the same length, one can suspect that MDF data is suited for relational storage schemas. However, the efficiency of read and write operations does heavily depend on the internal storage of data and additional functionalities of the database system. Especially the aspect of sequential and ordered storage of time series is fundamental for at least two reasons. First, if combined with index structures, time series can be accessed using as few database page requests as possible and thus optimizing data access. As a second point one needs to consider that automotive data is heavily compressible, especially in the context of run length encoding(RLE). To put this into perspective: storing a MDF3 file with approximately 800 MB of data led, depending on the systems and schemas we tested, to database sizes between 140 MB and 25 GB. As shown in Figures 2 and 3, sequential storage and therefore the potential of RLE can be achieved by either using array data types in row stores or strictly relational schemas with column stores (as array data types have not been supported by the column stores we have evaluated). While conceptually similar, the degree of compression has been shown to be more effective in row stores with arrays (Postgres) than in column stores using strictly relational schema. With these findings we could already neglect a popular row store that does not support array data types and has proven to be unsuited for this kinds of analysis in initial tests.

4.2 Choice of Schema

With the insights on the necessity of data compression, we developed 4 different database schemas in order to satisfy the requirement of either storing raw data

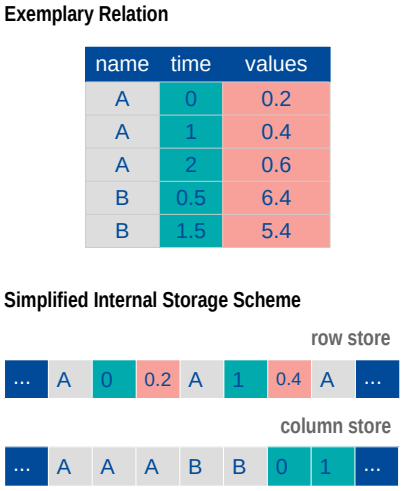

Fig. 2. Simplified representation of internal storage for strictly relational schemas of time series. Contrary to column stores, row stores do not store time axis and samples sequentially for the corresponding schema.

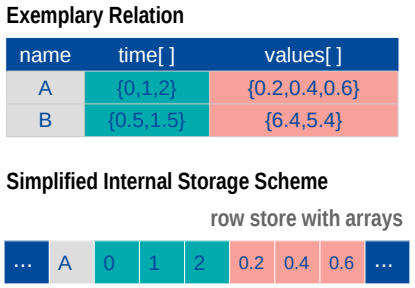

Fig. 3. Simplified representation of internal storage for object-relational schemas of time series. Row stores that support array data types are able to store time series sequentially.

(bit varying) and its conversion rules, or converted floating point data (double precision). Besides the conversion aspect, the schemas differ in whether they include array data types or only atomic data types as discussed in the previous subsection. The entity relationship model of the schema that incorporates arrays and stores floating point data is depicted in Figure 4.2. This is the most compact model, as it does not need to include conversion rules and can neglect extra attributes necessary to enumerate time series entries. Nevertheless, the core of the representations is the same. Meta-informations for files, devices and channels are stored in individual tables and are replicated in cluster scenarios. All time axis are stored separately from the time series with defining file and group identifiers. Time series are defined by the identifiers of the file, the channelgroup and the channel (from the channel dictionary) and both, timestamps and samples have been distributed by file on the cluster, in order to maximize performance for intra-file queries. Converted data samples are stored in one big table, while in raw data scenarios sample data is stored clustered with samples that use the same conversion rule (we have faced 4 different conversion rules in our test data). Initially, we have created a view `timeseries` that merged all the converted data into the same schema the table of the floating point scenario has, hoping that we could reuse all procedures written for the corresponding case. Unfortunately, we have found that selections in Postgres could not be con-

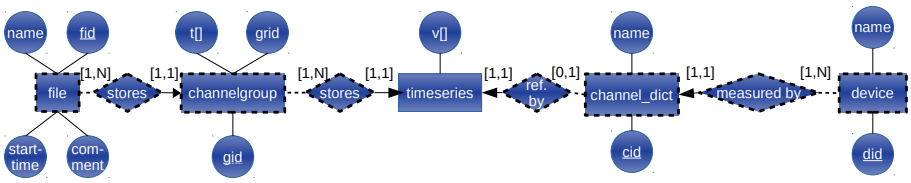

Fig. 4. Entity relationship model of the floating point schema for database systems that support arrays. Dashed borders in combination with arrows describe weak entity relationships.

sistently pushed onto the original tables, leading to unnecessary conversions of numerous channels. Therefore we needed to rewrite queries, manually adjusting selections and unions. Due to the large amount of channel data that is stored in the `timeseries` table(s), it is essential to use index structures. In fact, one of the parallel relational systems that have been tested does not support those to this point and needed to be neglected, due to weak performance. Currently, we are using mostly multidimensional b-trees on the aforementioned identifiers to allow for fast selections on either, files, groups, channels or even combinations of them. As a provisional conclusion on schemas we can state that using arrays have shown to be very effectiv, even though data has to be unnested for further operations and other database systems tested that do not support arrays are usually considered faster. Superior compression and faster channel selection capabilities, because of fewer tuples have shown to be essential for this kind of scenarios.

4.3 Data Import

Compression does not only influence the database size itself, but should also be considered when data is written into or fetched from the database. Currently we are reading MDF files via the Python package *asammdf* [5], reorganizing the data according to the respective database schema and either sending data to the database system via Python-database-connector or writing CSV files in order to enable the use of bulk load functionalities. Hereby, the latter method usually outperforms the direct Python-DB-connection, but is still comparatively slow, due to costly disk operations. We have found, that the connections between Python and databases cannot benefit from internal compression of data, meaning that arrays in python are sent in a decompressed form to the database system, where data will be compressed again. In order to speed up the import, we consider the implementation of a combination of the following techniques. First and foremost, the most effective approach is to avoid unnecessary data communication at all by pushing the import functionality into the database system. Unfortunately, this step is heavily dependent on the possibility of writing complex UDFs in a

concrete database system and therefore compromises the system independency of a database solution. Nonetheless, this approach would most likely heavily speed up the import as communication and file writing costs have shown to be the defining performance factor. The second possibility is to communicate time series data in compressed form and adjust its form in the database via internal queries or UDFs. This solution does ensure lower communication costs and is independent of the database system used, but will most likely be not as efficient as the import pushdown. Lastly, an easy way to improve the throughput of the import is to process multiple files at once. However, this is not sufficient for scenarios with big or few files.

4.4 Push-Down of Operations into the DBMS

As displayed in Figure 1 one of the main concepts is the pushdown of operations from an high level language (Python) into the database (Postgres). In previous research, we have shown that, depending on the scenario, many operations can benefit from SQL-based processing for a multitude of reasons [12,13,14,15]. These include performance, interchangeability of underlying database systems, due to the standardization of SQL, as well as data security or data privacy via manipulation of queries [8].

Based on the overview in subsection 3.2 we have implemented the pushdown of several operations, in order to minimize communication costs. As selections and projections are conceptually predestined for in-database processing, we investigated more advanced methods like the time merge, the conversion of raw data or the composition of new time series via pointwise arithmetic operations of channels. While all of these can be computed via ANSI:SQL statements, we have found that only our implementation of the zero-order hold time merge is inefficient and therefore tested an UDF implementation in Postgres. This step has been proven effective (especially for local setups) as can be seen in subsection 5.1.

5 Evaluation

In this section we evaluate the performance of the two basic scenarios from subsection 3.2, comparing a Python implementation with the most promising approaches in PostgreSQL 11.3 and Postgres-XL 10r1. Both test cases are evaluated on a local setup with 5 MDF files with an overall size of 3.7 GB, as well as on a cluster with 43 MDF files with an overall size of 50 GB.

For the local setup a notebook with a 4×2.1GHz Processor, 12 GB DDR-3 RAM, 250 GB SSD disk space and Ubuntu 18.04.2 has been used. Here, PostgreSQL runs on the same notebook and the MDF files are stored on the solid state disk. In the cluster setup, Python does also all the processing on the same notebook, but the MDF files are stored on a cluster and are accesed via network drive. Postgres-XL runs on the same cluster, using 5 homogeneous processing nodes with 2×1.9 GHz, 16 GB DDR3 RAM, 20 GB SSD disk space for the

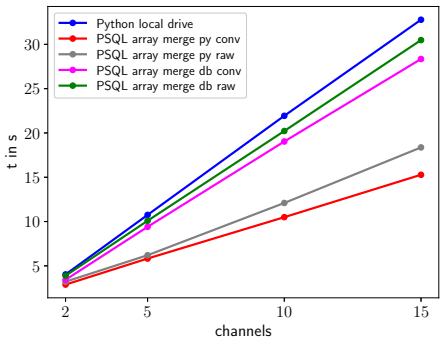

Fig. 5. Experimental evaluation of the local channel selection scenario over 5 measurements, including a time merge and plot as described in subsection 5.1.

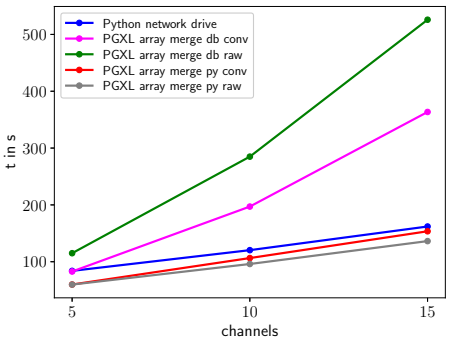

Fig. 6. Experimental evaluation of the cluster channel selection scenario over 5 measurements, including a time merge and plot as described in subsection 5.1.

operation system and software, as well as 1 TB SAS 7.2K disk space for data. All nodes are connected via a 2×10 Gigabit network. On each processing node run 2 data nodes and 1 coordinator. As the default configurations of Postgres are usually leading to underwhelming performances all instances (including the local setup) have been slightly adjusted, offering more main memory for buffers and garbage collection. Every experimental data point presented is the average out of 10 runs in order to compensate for variances, due to non-connected influences.

5.1 Channel Selection over Multiple Measurements

In Figures 5 and 6 the experimental evaluation of the first scenario from subsection 3.2, the selection and plotting of different channels over numerous files, is depicted. Here, we have tested array-based schemas in Postgres, differentiating between in-database channel conversion or floating point storage and a pushdown of time merge (via UDF) into Postgres or interpolating time series in Python. Naturally, online channel conversion should be slower than storing already converted data as it takes additional operations. Anyhow, the purpose for the inclusion of conversion steps is on the hand to show that it is possible to convert raw data online in database systems and on the other that it should take comparatively little time to do so.

The scenario investigates 5 MDF files and fetches a randomized choice of channels (the same for all systems in one run) with varying numbers.

In the local setup (Figure 5) only 5 files have been stored on the notebook, due to space limitations. Anyhow, it can be seen that all Postgres solutions clearly outperform the pure Python implementation. Furthermore, the time merge pushdown has proven very benefical, as these solutions do scale better in respect to the number of channels as all the other approaches.

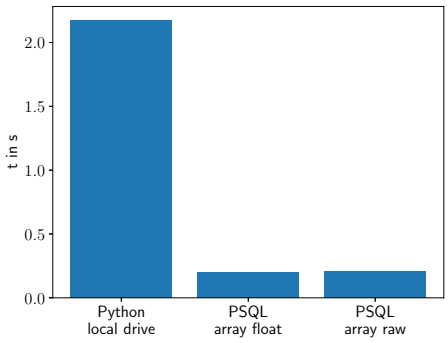

Fig. 7. Experimental evaluation of a local selection of measurements under constraints as described in subsection 5.2. Three out of five MDF files match the given requirements.

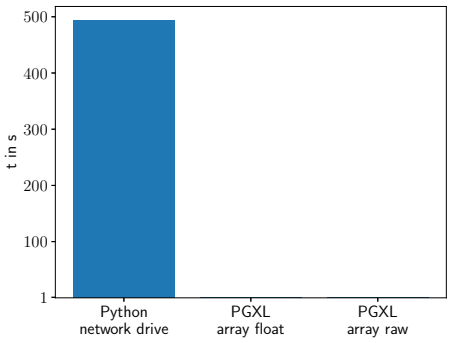

Fig. 8. Experimental evaluation of a selection of measurements under constraints as described in subsection 5.2 on a cluster. Here, 19 out of 43 MDF files match the given requirements.

Counter-intuitively, this does not hold for the cluster scenario as can be seen in Figure 6. Here, 5 out of 43 MDF files are randomly chosen for each run.

It can be seen that Python is generally faster when time merges are not pushed down, but slower when non-merged channel data is queried from Postgres-XL. Anyhow, it can be seen that with increasing number of channels the pure Python solutions might scale better than the Postgres approach storing converted data.

We suspect a number of different reasons for this findings. First and foremost, as a relatively young project with a comparatively small developer team, we expect some unintended and not yet optimized behavior. As an example one can see that the online conversion with a time merge in Python is faster than the version that stores the already converted data, although it takes lesser operation to process.

Additionally, we have found that UDFs are inexplicably slow in Postgres-XL, especially when accessing array data. Therefore we are considering to test similar implementations on commercially established parallel database systems that also support array data types in order to verify our claims.

A second factor to consider is, that our current test data can not be considered big data (at least in compressed form). This might weaken potential benefits of communication costs and in-database processing.

5.2 Finding Measurements under Constraints

The experimental results of the second scenario from subsection 5.2 can be seen in Figures 7 and 8. In both cases (local and cluster) all available measurements have been checked, whether they fulfill a given set of requirements. These include:

- a set of given channels need to be measured,
- channels had to be recorded with a given sample rate,
- the measurement had to be done in a certain period of time, and
- the measurement length had to surpass a given threshold.

Here, the time series itselves only need to be checked for existence, but the time axis of every channelgroup have to be evaluated for minimum and maximum values. In the local setup 3 out of 5 files fulfill the requirements, whereas 19 out of 43 do on the cluster. Both, Figures show that these highly selective inter-file problems can be done very efficiently in database systems, outperforming pure Python solutions by quite a margin. These results do clearly motivate including database technology in automotive analysis, especially if analysts need to prefilter measurements or channels out of bigger data sets.

6 Conclusion and Discussion

In this paper we have discussed a database support for automotive analysis. Key aspects, like choice of database systems and schemas, pushdown of operations into the database, as well as data import have been discussed. It could be shown that database systems does not only provide needed scalability but also accelerate basic methods for automotive analysis.

As this project started as an investigation of several systems and storage types, there are plenty of opportunities for further, more detailed investigations in the near future. One of these: the pushdown of the import of MDF files into the database has already been discussed in subsection 4.3. Additionally, we would like to investigate more inter-measurement analysis on big data, as we expect this kind of operations to be very beneficial with parallel relational database solutions, as well as solutions using apache spark and NoSQL systems. Furthermore, we would like to include more operations for time series analysis. This includes in-database pattern recognition, for example peak-detection, or operations we have previously investigated in SQL, like machine learning methods [13] or fourier analysis [15]. These operations could be further combined with index structures in order to offer analysts possibilities to search for special curve progressions over multiple files.

References

1. 2ndQuadrant: Postgres-XL official website (2019), `https://www.postgres-xl.org`
2. Association for Standardization of Automation and Measuring Systems: ASAM MDF (2019), `https://www.asam.net/standards/detail/mdf`
3. Castillejos, A.M.: Management of time series data. Ph.D. thesis, University of Canberra (2006)
4. Chandra, R., Segev, A.: Managing temporal financial data in an extensible database. In: VLDB. pp. 302–313. Citeseer (1993)
5. Daniel Hrisca: asammdf documentation (2019), `https://asammdf.readthedocs.io/en/latest/#`

6. Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., Keogh, E.: Querying and mining of time series data: experimental comparison of representations and distance measures. Proceedings of the VLDB Endowment **1**(2), 1542–1552 (2008)
7. Golab, L., Özsu, M.T.: Issues in data stream management. ACM Sigmod Record **32**(2), 5–14 (2003)
8. Grunert, H., Heuer, A.: Query rewriting by contract under privacy constraints. OJIOT **4**(1), 54–69 (2018), https://www.ronpub.com/ojiot/OJIOT_2018v4i1n05_Grunert.html
9. Ikawa, N., Kawamoto, S.: Time-series data management device, system, method, and program (Mar 2016), uS Patent 9,298,854
10. Johanson, M.: Information and communication support for automotive testing and validation. In: New Trends and Developments in Automotive System Engineering. IntechOpen (2011)
11. Keogh, E.: A decade of progress in indexing and mining large time series databases. In: Proceedings of the 32nd international conference on Very large data bases. pp. 1268–1268. VLDB Endowment (2006)
12. Marten, D., Heuer, A.: A framework for self-managing database support and parallel computing for assistive systems. In: Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments, PETRA 2015, Corfu, Greece, July 1-3, 2015. pp. 25:1–25:4 (2015). https://doi.org/10.1145/2769493.2769526, https://doi.org/10.1145/2769493.2769526
13. Marten, D., Heuer, A.: Machine learning on large databases: Transforming hidden markov models to sql statements. Open Journal of Databases (OJDB) **4**(1), 22–42 (2017), https://www.ronpub.com/ojdb/OJDB_2017v4i1n02_Marten.html
14. Marten, D., Meyer, H., Dietrich, D., Heuer, A.: Sparse and dense linear algebra for machine learning on parallel-rdbms using SQL. OJBD **5**(1), 1–34 (2019), https://www.ronpub.com/ojbd/OJBD_2019v5i1n01_Marten.html
15. Marten, D., Meyer, H., Heuer, A.: Calculating fourier transforms in sql. In: Advances in Databases and Information Systems - 23nd European Conference, ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings (2019)
16. Rudolph, K.: A Comparison of NoSQL Time Series Databases, vol. 1282. GRIN Verlag (2015)
17. Vector Informatik GmbH: Format Specification MDF Format Version 3.3.1 (2014)
18. Wlodarczyk, T.W.: Overview of time series storage and processing in a cloud environment. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings. pp. 625–628. IEEE (2012)
19. Yu, B., Cuzzocrea, A., Jeong, D., Maydebura, S.: On managing very large sensor-network data using bigtable. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012). pp. 918–922. IEEE Computer Society (2012)
20. Zoumpatianos, K.: Indexing for Very Large Data Series Collections. Ph.D. thesis, University of Trento (2016)