TUM School of Computation, Information and Technology
Technische Universität München

An Industrial Sensor Data Processing and Query System

Roman Johannes Karlstetter

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr.-Ing. Jörg Ott

Prüfer der Dissertation:
1. Prof. Dr. rer. nat. Martin Schulz
2. apl. Prof. Dr.-Ing. Walter Stechele

Die Dissertation wurde am 26.06.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 07.11.2023 angenommen.

Acknowledgments

I would like to use this space to express my heartfelt gratitude to all who have contributed to the completion of this dissertation.

First, I would like to sincerely thank Prof. Dr. Martin Schulz for supervising me, especially for helping me structure the various aspects of this work and our previous publications. His expertise and insightful feedback have been invaluable throughout this entire process. I want to thank Prof. Dr. Walter Stechele for agreeing to serve as the second examiner of this work. I also especially want to thank Prof. Dr. Carsten Trinitis for his continuous help. He has been active during the whole path (and actually even before that as advisor of my master thesis), including project application, project team recruiting, research discussions and reading the (almost) final version of this thesis. I am also grateful to Amir Raoofy for the endless technical discussions, in particular at the beginning of TurbO. I think our collaboration was very fruitful—this definitely can not be taken for granted. I also want to thank all other members of the CAPS chair. Even though I was at the chair only occasionally, I always felt very welcome.

I would like to express my deepest gratitude to my employer IfTA GmbH, particularly Dr. Jakob Hermann, for allowing me to research in such an exciting industrial environment. Without his entrepreneurial courage and the initial idea for this whole project, HAQSE and this dissertation would not exist. Working at the intersection of industrial application and academics has not always been straightforward, but I always had the freedom to work on the aspects I deemed essential. I am incredibly grateful to Dr. Robert Widhopf-Fenk for mentoring me during this journey. Our regular discussions about technical details and other not-so-technical aspects contributed significantly to keeping my motivation alive. A big thank you also goes to all my other colleagues at IfTA. The list of names exceeds the space on this page, but be assured that I am very grateful for all the discussions and delicious lunch breaks we had together.

I also want to thank SWM, particularly Julius Becker, for providing us access to all this sensor data and for taking care of regularly exchanging HDDs.

I am indebted to everyone else proofreading and providing valuable feedback on (parts of) earlier versions of this dissertation: Paul Rötzer, Fabian Legl, Alexander Aumann, Dr. Driek Rouwenhorst.

Finally, I'd like to thank my family and friends for their unwavering support.

I am grateful to the funding organization *Bayerische Forschungsstiftung*: they partly sponsored this work under the research projects *"Optimierung von Gasturbinen mit Hilfe von Big Data"* (AZ-1214-16) and *"Von der Edge zur Cloud und zurück: Skalierbare und Adaptive Sensordatenverarbeitung"* (AZ-1468-20).

Abstract

Complex systems represent the backbone of our modern society. To manage this complexity, operators of such systems are relying on monitoring their assets with sensors. As the requirements for these complex systems increase, the amount and sophistication of the employed sensor technology also increase. Consequently, the amount of generated sensor data is growing continuously, too. This creates several challenges for downstream systems to handle and process these sensor data streams. First, the *data rate* at which sensor data is generated keeps increasing for two main reasons: On the one hand, sampling rates for individual sensors increase, exceeding 100 kHz in some applications. On the other hand, the number of sensors per monitored machine is growing. This makes it possible to improve, e.g., the spatial fidelity of the monitored data. As a consequence of the increasing data rate, the amount of required *storage space* increases accordingly. In addition, the requirements with regard to retrieving such large amounts of sensor data keep growing. Users *retrieve* this data for a variety of different applications. They require large time spans of sensor data to be ready to use in interactive data exploration tasks like, e.g., interactive visualization. Moreover, sensor data is used for other analytic tasks like training machine learning models or other forms of algorithm development. Finally, users expect data from different, geographically distributed systems to be easy to integrate into their analyses. In essence, users of industrial sensor data require retrieval with low latencies and high throughput.

However, existing systems do not fulfill all these requirements. In this dissertation, we present HAQSE, a *H*ierarchical *A*ggregation and *Q*uery *S*tor*E*. HAQSE is a sensor data processing and storage system that solves three main challenges. First, HAQSE is optimized for consuming high data rate sensor streams. We show that HAQSE can consume sensor streams faster than any other system we include in our experiments. Second, HAQSE efficiently uses the available storage space for long-term sensor data storage. This is done by employing a lossless compression technique specifically targeted at noisy floating-point data. Third, sensor data stored in HAQSE can be retrieved with high throughput and low constant overhead. This is achieved by using an appropriate data storage layout and a binary protocol for query responses. For enabling interactive queries of large time spans, HAQSE relies on precomputed hierarchical aggregations. Lastly, HAQSE supports queries on data that is available on geo-distributed instances.

All these results make it possible for contemporary and future sensor processing applications to increase sensor stream rates further. Ultimately, all this contributes significantly to the stable operation of modern and complex infrastructures.

Zusammenfassung

Komplexe Systeme bilden das Rückgrat unserer modernen Gesellschaft. Um die Komplexität dieser Systeme beherrschbar zu halten, überwachen Betreiber ihre Anlagen mit einer Vielzahl an verschiedenen Sensoren. Mit den steigenden Anforderungen an diese komplexen Systeme steigt auch die Anzahl und Qualität der eingesetzten Sensorik. Folglich nimmt auch die Menge der generierten Sensordaten kontinuierlich zu. Die Verarbeitung dieser Sensordaten in nachgelagerten Systemen ist mit einigen Herausforderungen verbunden. Erstens steigt die Datenrate, mit der Sensordaten erzeugt werden. Dies hat zwei Gründe: Einerseits steigen die verwendeten Abtastraten; diese übersteigen schon heute in einigen Anwendungen die Marke von 100 kHz. Zum anderen steigt die Anzahl der Sensoren pro überwachter Anlage, um den Detailgrad im generierten Datensatz zu erhöhen, also zum Beispiel um die räumliche Auflösung der überwachten Anlage in den Sensordaten zu verbessern. Als Folge der immer größeren Datenraten steigt auch der benötigte Speicherplatz, um die Sensordaten langfristig abzuspeichern. Darüber hinaus steigen die Anforderungen an den Zugriff auf solch große Sensordatenmengen ständig. Verschiedene Nutzer benötigen die gespeicherten Sensordaten für unterschiedliche Anwendungen. Diese Nutzer benötigen oft große Zeitspannen an Sensordaten, zum Beispiel für die interaktive Visualisierung. Außerdem werden Sensordaten für analytische Aufgaben wie das Training von Machine-Learning-Modellen oder andere Formen der Algorithmenentwicklung verwendet. Des Weiteren erwarten Nutzer, dass Sensordaten aus geografisch verteilten Systemen einfach in Datenanalysen nutzbar sind. Im Wesentlichen fordern die Nutzer von industriellen Sensordaten, dass Abfragen mit geringen Latenzen und hohem Durchsatz durchgeführt werden können.

Bestehende Systeme erfüllen jedoch nicht alle diese Anforderungen. Diese Dissertation stellt HAQSE (Hierarchical Aggregation and Query StorE) vor, ein System zur Verarbeitung und Speicherung von Sensordaten, das drei wesentliche Herausforderungen löst. Erstens: HAQSE ist für die Verarbeitung von Sensordatenströmen mit hoher Datenrate optimiert. Diese Arbeit zeigt, dass HAQSE Sensordatenströme schneller als jedes andere von uns untersuchte System verarbeiten kann. Zweitens: HAQSE nutzt den verfügbaren Speicherplatz für die langfristige Speicherung von Sensordaten effizient aus. Dies wird durch die Entwicklung einer verlustfreien Kompressionstechnik erreicht, welche auf verrauschte Gleitkommazahlen zugeschnitten ist. Drittens: in HAQSE gespeicherte Sensordaten können mit hohem Durchsatz und geringem konstanten Overhead abgerufen werden. Dies wird durch die Verwendung eines geeigneten Speicherlayouts und einem binären Protokoll für die Beantwortung von Abfragen

ermöglicht. HAQSE nutzt vorberechnete hierarchische Aggregationen um Abfragen über große Zeitspannen von Sensordaten in interaktiver Geschwindigkeit ausführen zu können. Schließlich unterstützt HAQSE auch Abfragen von Sensordaten, die auf geografisch verteilten Instanzen vorliegen.

All diese Ergebnisse ermöglichen es, den aktuellen und zukünftigen Ansprüchen der Sensordatenverarbeitung sowie steigenden Sensorabtastfrequenzen und damit Datenraten gerecht zu werden. Diese Arbeit zeigt, dass HAQSE die Analyse von Sensordaten erheblich beschleunigen kann. Damit hilft HAQSE Anlagenbetreibern und Ingenieuren bei der Entscheidungsfindung und der Entwicklung neuartiger Methoden zur Analyse von Sensordaten. All dies trägt letztendlich auch maßgeblich zu einem stabilen Betrieb unserer modernen Infrastruktur bei.

Contents

1 Introduction

Various aspects of our daily lives are driven by complex systems. In many cases, much of this complexity is completely hidden from general awareness. One such example is the energy sector, more specifically, generation of electrical energy. Even an excerpt from the list of challenges looks demanding: producers need to collaboratively fulfill the varying demand over the course of the day, and energy needs to be distributed from producers to consumers that may be geographically far apart, while keeping the power grid stable at the nominal frequency. All of this is especially challenging in the face of the inevitable transition to renewable but more volatile energy sources. One key technology that can help in this transition process are gas-fired power plants, employing heavy-duty gas turbines at their core. These gas turbines can flexibly be turned on and off to satisfy demand peaks or during times when energy generation from solar and wind is low. Furthermore, gas turbines help to stabilize the grid, as they contribute to the required rotational inertia [64, 106]. They also can be controlled precisely and quickly, providing primary frequency control that is used to, e.g., compensate rapid changes in demand [83]. Gas turbines can also be operated using more climate-friendly fuel mixes, e.g., adding green hydrogen or other synthetic fuels as an energy source.

However, the combustion process in gas turbines is prone to thermoacoustic instabilities, especially when changing the employed fuel mix [13, 67]. At the same time, economic pressure is constantly staying on a high level, and ecological and political regulations for operating such power plants are getting stricter [25]. Consequently, operators of such infrastructure are required to build measures to better understand and control the behavior of the various components of such power plants. One of the strategies pursued is to continuously monitor the combustion process, using dynamic pressure sensors. The number and quality of these sensors are growing steadily [107, 130], and so does the generated amount of sensor data.

The described trend towards increasing the amount of sensing technology installed for monitoring complex systems is not unique to gas-fired power plants. As another example from the energy generation sector, wind turbines require measures to prevent ice building up on blades in cold temperature environments [133]. Similar developments can be observed in many other industrial applications. In data centers or high-performance computing centers, operators face challenges regarding energy-efficient and thus economically viable operation. It is thus indispensable to monitor the energy consumption behavior of such computing centers [52, 78, 84]. Similarly, sensor data also plays a vital role in experimental research facilities like CERN. These build and

deploy sophisticated data acquisition systems to process and store data generated by experiments [132].

In short, there is a general trend in many industry sectors towards equipping complex systems with more sensor technology. In this dissertation, we address processing, storage, and retrieval of the increasing volume of sensor data that industrial applications continuously generate. We introduce a sensor data processing and storage system called *HAQSE* that improves over the state of the art in multiple dimensions and is suitable for many real-world industrial applications. While this research was motivated by the specific application of combustion monitoring in gas turbines, the developed approach and methods generally apply to a wide variety of scenarios. The presented approach is suitable for any application that produces large amounts of structured sensor data.

In the remainder of this first chapter, we start with a more detailed explanation of our motivating example of combustion monitoring (Section 1.1). After that, we generalize this use case and discuss the motivation and challenges of this work (Section 1.2). Based on this, we sketch our approach (Section 1.3). Lastly, we highlight our main contributions (Section 1.4), and outline the further structure of this thesis (Section 1.5).

1.1 Motivating Application Example: Combustion Monitoring in Gas-Fired Power Plants

As sketched at the beginning of this chapter, there are several applications that employ sensors for continuous monitoring. In this section, we demonstrate the particular challenges and requirements for a sensor processing and storage system using the specific example application of gas-fired power plants. Such gas-fired power plants primarily generate electricity. Additionally, if the location is suitable and permits it, the remaining heat is reused for district heating. At the core of such a power plant, a stationary, heavy-duty gas turbine (see Figure 1.1) burns some kind of fuel, in most cases natural gas. The energy when burning that fuel is released in form of compressed, hot gas. This is converted to electricity by streaming the compressed, hot gas through multiple stages of blades of a turbine. The turbine is connected to a generator that produces electrical energy.

The individual components of the gas turbine are protected by several monitoring systems. One of these systems monitors the combustion process. This is necessary as the combustion is prone to so-called thermoacoustic instabilities that can quickly damage the structure of the turbine. Monitoring uses up to dozens of high-temperature pressure sensors that are sampled at rates of up to 50 kHz[1]. One of the primary uses

[1]The specific configurations vary depending on turbine type and application requirements of the operators. Protection against thermoacoustic instabilities requires sampling rates of around 25 kHz. Other applications, e.g., detecting foreign objects in the turbine, require sampling rates of 50 kHz or sometimes even higher values.

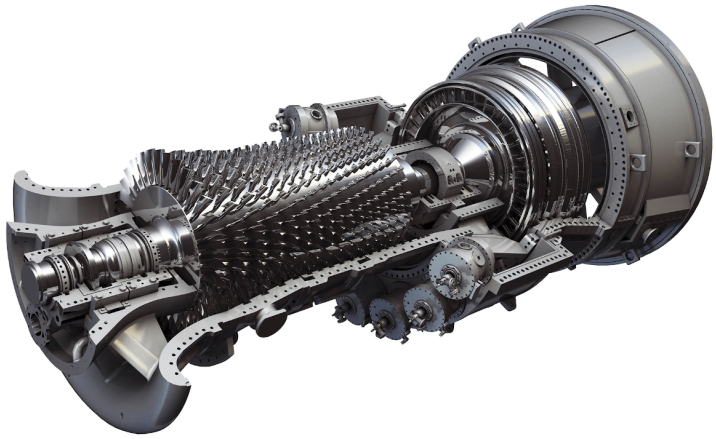


Figure 1.1 GE 9E gas turbine [35].

of pressure sensor data is real-time protection of the gas turbine. This is achieved by analyzing short tumbling or slightly overlapping time windows[2] and communicating the result to the gas turbine control system. This real-time analysis includes transformations to the frequency domain by calculating Fast Fourier Transforms (FFTs) on the analyzed time windows and other, gas-turbine-type specific metrics like amplitudes in certain frequency bands. To expand the context for the sensor data and analysis results, other data sources provide streams of operating and condition data. This additional data reflects the state of other components in the system and contains metrics such as the input fuel rate, the electric power output, valve positions or ambient pressure and temperature. Operating and condition data is sampled at frequencies of around 1 Hz, but there are possibly hundreds or even thousands of such signals. The raw sensor samples, the analysis results as well as operating and condition data are all used for further, historical data analysis. One particular example of such historical data analysis is root cause analysis in case of, e.g., a damage event. For that reason, all this sensor data needs to be stored permanently.

The collected data is used for several applications. It is visualized in interactive dashboards to provide operators with a quick overview of the current machine state. In addition to that, it is used in algorithm development, helping engineers to design new metrics that are used for real-time protection or long-term damage prevention. Furthermore, this data is used as a basis for training machine learning models, e.g., algorithms for detecting anomalies.

Since power plants are critical infrastructure, the monitoring and protection systems are generally isolated from the outside, i.e., they have no connection to the Internet. Consequently, processing and storage of the generated sensor data must happen on-

[2]Typical analysis window sizes range from 50 ms to 800 ms.

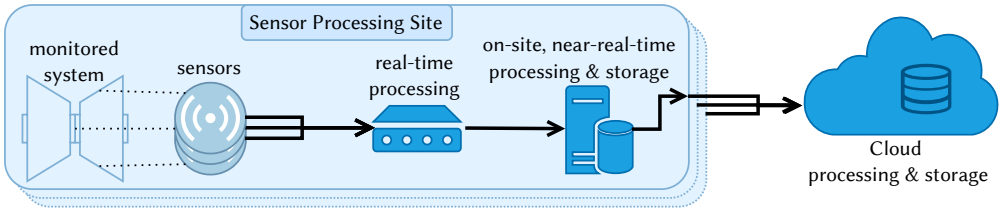


Figure 1.2 Simplified illustration of sensor data flow from sensor to cloud. Data is sampled synchronously from multiple sensors using a real-time processing system. This yields a multivariate sensor data stream, which is further processed and initially stored on-site. Parts of the sensor data are then sent to a processing and storage cloud system, where data from multiple, geo-distributed sensor processing sites is available.

site, inside the local monitoring network (as sketched in Figure 1.2). The installed monitoring systems are expected to work for more than 15 years with as little maintenance as possible. For that reason, operators prefer systems without error-prone parts like cooling fans. Consequently, these systems are cooled passively and thus, processing and initial storage happens on systems with limited processing resources.

In addition to local protection and monitoring, gas turbine manufacturers have an interest in comparing the behavior across a fleet of turbines. For that reason, some parts of the collected sensor data are possibly transferred to some kind of central storage system. Such a central storage system may be deployed in the cloud so that it can be accessed from anywhere. The amount of data sent to this central system depends on various aspects like the type of turbine, the available bandwidth as well as the policy of the respective operator with regard to connecting their system to the Internet.

The whole flow of sensor data is sketched in Figure 1.2: from sensors monitoring the machine, via local processing and storage, to cloud systems, where data from multiple sites is accumulated.

1.2 Challenges

While the example of combustion monitoring in gas turbines originally motivated this work, there are other applications that face similar challenges. Driven by the needs of these applications, the amount and quality of sensing technology in industrial applications increase steadily. The volume of generated sensor data follows this upward trend. This is caused by two main factors, both accelerated by trends such as increased system complexity or decreasing prices for sensing technology. On the one hand, the *number* of sensors used for monitoring a certain system is growing steadily, improving the general quality of monitoring. This leads to, e.g., a higher spatial fidelity or information about an increasing amount of individual components of the monitored system. On the other hand, higher *sampling frequencies* improve the temporal resolution of the resulting raw sensor data and derived analysis values. This is especially

important for monitoring and understanding fast oscillatory processes, often found in industrial settings [67, 133]. Both aspects lead to a growing accumulated data rate that needs to be processed. Since an increasing share of that sensor data is stored for later analysis, the associated storage requirements grow accordingly. In many cases, sensors measure real numeric values, commonly represented as floating-point values in digital systems.

We identify two general processing and usage patterns for sensor data generated in an industrial system.

On the one hand, sensor data is used for *real-time* monitoring, control and protection. In this case, short tumbling or partly overlapping time windows of sensor data are analyzed in real time, providing analysis results to control and protection systems that react in a timely manner to the observed behavior. Additionally, recently measured values are displayed in dashboards or other, more sophisticated types of visualization that show the recent history of values.

On the other hand, the generated sensor data is stored for later use, creating a database of *historical* sensor data. This historical sensor data is stored for several purposes. First, it is used for interactive data exploration, often accomplished through visualization of data for one or multiple sensors of a certain, interactively chosen time span. Furthermore, historical data is used in after-the-fact analysis with the overarching goal of improving current and future operation of the monitored system. Such historical sensor data analysis often covers large time spans of up to several years. Sensor data is used to derive and predict trends, analyze past behavior and avoid future problems. In addition to that, historical data may serve as a database for training machine learning models. Such models can be used to, e.g., detect anomalies during operation. More generally, machine learning models can help to classify the operating mode of the monitored system. This, in turn, enables implementation of strategies like predictive maintenance. It is essential to recognize that not all uses of historical data require the full temporal resolution initially available. On the contrary, using full-resolution data might be prohibitive in interactive settings when large time spans (with respect to the original sampling frequency) are explored [52].

To summarize, there exist both a growing amount of sensor data that is available for analysis and various applications that make use of this data. This has several implications:

Storage space First, the amount of storage space required to permanently store this data is growing proportionally to the amount of generated sensor data. It is thus necessary to use the available storage space efficiently. This essentially means that the produced data must be compressed appropriately. There are two fundamentally different ways of compressing data: lossless compression and lossy compression. On the one hand, for lossless compression, the full precision of the acquired data is preserved, and no information is lost. This lossless compression is particularly challenging for floating-point data, as different noise sources make

it hard to compress such data without losing information. The storage space savings for lossless compression are limited by the data's entropy. On the other hand, lossy compression schemes achieve much higher compression ratios. This is possible by reducing the precision of the stored data. How much information loss is acceptable is heavily application-dependent and can be configured in most lossy compression algorithms.

Data ingestion Second, caused by higher temporal resolution and number of sensors, the data rate of the sensor stream to process increases. This necessitates that the processing and storage system is able to cope with the growing data rates of the sensor stream. Many existing systems cannot consume data at such high rates because they only provide strong durability guarantees that are prohibitive with respect to ingestion performance. Moreover, to support the high sampling rates, timestamps must be represented with suitable precision.

Data retrieval Third, retrieving data for near real-time or historical data analysis must be almost instantaneous (interactive usage), but also deliver high throughput for larger amounts of queried data (e.g., when used for machine learning). Queries to such time series data are expected to be mostly time-range-based, i.e., access a contiguous sequence of samples between two points in time. Furthermore, we assume that only a small subset of the available sensors is accessed in each query. Hence, the following requirements for querying data can be derived: the data latency (duration from the point in time data is sampled by the sensor until it can be queried from the system) must be small, queries must be processed with low constant overheads, and it must be possible to deliver query responses with high throughput. Since it should also be possible to interactively explore large time spans, there must be a way to quickly retrieve (approximate versions of) sensor data for such large time spans.

Industrial applications are special regarding the deployment context of sensor processing and storage systems. In particular, it is often desirable—and in some cases necessary—for such a system to be deployed "on-site", "close" to the monitored system. More specifically, this means that measured sensor data is accessible via a local network since the system is expected to work without a (potentially unreliable) connection to the Internet. The reasons for this are manyfold. When monitoring and processing sensor data for critical infrastructure, such as power plants, operators want data to be physically present on-site as this enables self-sufficient operation. This is required in case external infrastructure like data centers fail or the connection to such infrastructure is cut off. In addition to that, especially for critical infrastructure, some operators completely isolate the internal network from the outside due to cyber-security concerns. In other cases, operators do not want to entrust their sensor data to third parties like cloud providers because of data privacy issues. This is particularly important to operators in sectors with a high risk of industrial espionage. In addition to that,

some installations lack the required bandwidth to an external compute and storage infrastructure. In these cases, it is infeasible to transfer the continuously produced data stream out of the local network. This is especially relevant for geographically remote installations. Even if the necessary bandwidth is available, transferring data outside the local network might not be desirable because of latency requirements of certain applications. Depending on the specific circumstances, having data in external systems may increase the access latency by orders of magnitude.

While locally processing data avoids the problems just discussed, it implies several challenges that need to be dealt with. Most importantly, processing and storage of the generated sensor data happen on computer systems with limited processing resources like central processing unit (CPU) and random-access memory (RAM). This is due to several factors. First, in many cases, the physical size of the system may be limited, as the available space in compute racks is shared with other systems. Furthermore, as explained in the specific example before, systems may be required to be cooled passively, drastically restricting power consumption. There is also no easy way of scaling compute and storage systems like it would be possible in a cloud-based scenario.

As just discussed, processing data locally is of great importance. However, there are still cases where applications require data from multiple, geographically distributed sites. As one example, operators sometimes wish to compare a fleet of machines of identical type to identify differences in operation and further optimize design. For that reason, some parts of the sensor data—e.g., overview data or certain interesting time spans—may be sent to other processing facilities on-site or in the cloud. These systems often have different properties in terms of storage capacity or processing resources. In case of cloud systems, these aspects can more easily be scaled according to growing application needs. This has implications for data queries: when querying this data, it is important to consider the different parts of sensor data distributed along the edge-to-cloud continuum. Data from the different sources should be seamlessly integrated when queried.

Existing systems (discussed in more detail in Section 2.4) cannot fulfill all these requirements and lack in at least one of the mentioned aspects.

1.3 Approach

In this section, we present our approach. We design a system for processing and permanently storing streams of industrial sensor data. The general idea is to create a time series storage system, similar to the storage component of existing systems such as InfluxDB [54], QuestDB [93] or TimescaleDB [126]. There are, however, several design decisions that make our system different and that help to fulfill the requirements and address the challenges listed above.

Efficient Storage The first aspect we consider is storage efficiency. We aim to develop a method to store large quantities of floating-point sensor data. Ideally, that method can be integrated into an existing file format such that it can also be used independently of the rest of the system. While an efficient compressed representation is important, it is thus also crucial for our approach to be easy to implement. This ensures that our approach can be quickly adopted in different implementations. Furthermore, both compression and decompression should be able to deliver high throughput. Since our system stores time series data, it is also required to store series of timestamp values. Thus, in addition to floating-point sensor data, our system must handle storing timestamp values in a storage-efficient way. We rely on existing methods [69, 90] to achieve that.

Efficient Stream Ingestion Second, our envisioned applications require that our solution is able to ingest data faster than state-of-the-art systems. We deliberately relax durability guarantees to achieve that goal. Since our primary use case is fast ingestion of *streams* of data, it is not necessary to acknowledge reception and storage of every single sample. Instead, it is sufficient that data is persisted on durable media eventually. Our targeted use cases also make it possible to limit the number of input stream sources for a certain *storage unit* to a single source at any given time. By employing a binary streaming protocol, we avoid decoding and processing overheads when receiving data.

Efficient Queries Third, we specialize our system to range-based queries. These queries are expected to retrieve data for only few sensors. There are several techniques we use to realize this. First, also related to the storage aspect, we rely on a sensor-ordered or columnar data format. This is preferable for the targeted range-based access patterns. Furthermore, we design an index structure that exploits the contiguous, time-ordered nature of time series data. For delivering the actual query results to the client, we use a batched, binary streaming format. This enables high-throughput responses and delivers data in a format that is easy and efficient to process for clients. For approximate query responses, we rely on hierarchies of windowed aggregations that are efficiently calculated already during data ingestion. Distributed querying logic is located on the client side, avoiding shared state and the need to manage the potentially complex network topologies.

We implement our ideas in one integrated, holistic system. This makes it possible to run our system on small-scale, low-power systems, which is a typical use case in an industrial context. The individual parts—in particular, the storage format—are expected to be usable independently of the rest of the system. Integrating everything in a single application allows for reducing communication overheads between the components of our system, enabling efficient sensor data processing.

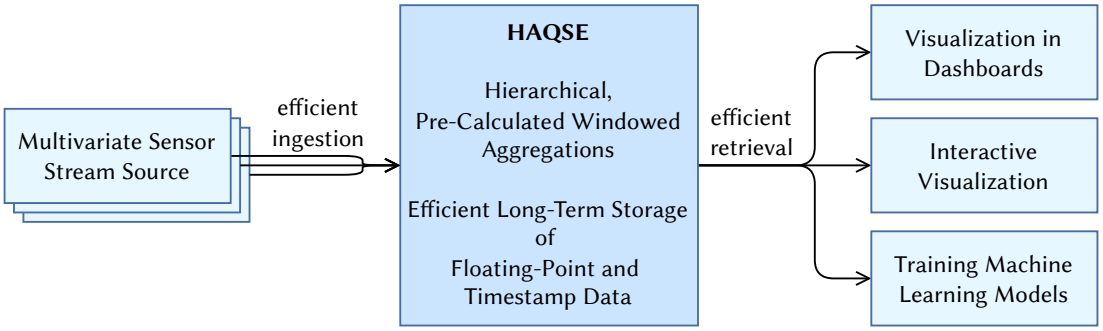


Figure 1.3 HAQSE is a sensor data processing and storage system. It can efficiently ingest and process streams containing multivariate sensor data. It offers a range-based query interface that can be used for retrieving data for a variety of use cases.

1.4 Contributions

In this dissertation, we present HAQSE, the **H**ierarchical **A**ggregation and **Q**uery **S**tor**E**. HAQSE is a sensor data processing and storage system that addresses all points discussed in Section 1.2 and implements the approach sketched in Section 1.3. This section outlines the contributions of this thesis. As mentioned above, there is a great need for efficient, large-scale sensor data processing systems in industrial environments. However, existing systems (discussed in more detail in Section 2.4) reach their limits and cannot satisfy the data throughput and query needs of contemporary and future sensor processing applications. In particular, all existing systems lack at least in one of the following aspects:

- *Stream rate*: Coping with stream rates of hundreds of kHz for a multivariate stream of dozens of sensors or ten thousands of sensors sampled at several Hz.
- *Efficient storage*: Storing noisy floating-point sensor data in a form that uses the available physical storage space efficiently.
- *Efficient retrieval*: Quickly retrieving large time windows of sensor data for interactive exploration (with low latency) as well as analysis and machine learning tasks (with high throughput).
- *Geo-distributed retrieval*: Retrieving data from multiple, geo-distributed sites.
- *Efficient use of hardware resources*: Capable of being executed on low-power, passively cooled, small-scale systems.

With HAQSE, we holistically cover the complete pipeline (see Figure 1.3): receiving a stream of sensor data, generating hierarchical aggregates for efficient queries, encoding and compressing it for efficient storage, and enabling low latency and high-throughput data retrieval through language-independent Application Programming

Interfaces (APIs). While the individual contributions are largely orthogonal to each other and can be used independently, our implementation in one single system yields synergies that would otherwise not have been easy to generate.

In particular, our contributions are the following:

- We **improve the storage efficiency for floating-point sensor data** by **extending the Apache Parquet file format** [116] with the *Byte Stream Split* encoding. When combined with an off-the-shelf compression scheme, Byte Stream Split is suitable for compressing floating-point sensor data. We evaluate our two-stage compression approach and compare it to several existing floating-point compression methods for a collection of publicly available floating-point datasets. This comparison shows that our approach works well across *all* tested datasets and is the best method for *most* of them. We also compare our approach against the alternatives available in Apache Parquet. We do this on a gas turbine monitoring dataset (see Section 1.1). This shows that our approach outperforms all existing methods in all three important metrics: compression ratio, compression throughput and decompression throughput. Byte Stream Split is available as part of the official, open source Apache Parquet specification and implementation projects. The encoding can be efficiently mapped to Single Instruction, Multiple Data (SIMD) vector instructions of modern CPU architectures.

- We design, implement and evaluate a scheme for **ingesting high-rate sensor streams**, limited only by network or storage device throughput. It involves a transformation from a time-ordered to a sensor-ordered layout. This enables further improvements in all steps of the downstream sensor processing pipeline. Based on the sensor-ordered layout, we design and implement a computationally efficient way to compute **hierarchies of windowed aggregations** on sensor data. The input interface to our system uses a language-independent, binary protocol that also supports input in arbitrarily sized batches, enabling high ingestion throughput rates.

- We design our system to support **efficient retrieval** of **historical** as well as **live sensor data**. We propose Stream Segment Index, a tree-based index data structure that enables efficient range-based queries. We show that exploiting an additional persisted index structure at the level below is important for scaling our approach to larger sizes. Additionally, we design a novel approach for querying sensor data residing on geo-distributed nodes that does not require shared state.

We thoroughly evaluate every component of HAQSE in isolated experiments, helping us to gain detailed insights into the individual components. Additionally, we compare HAQSE to state-of-the-art time series management systems, showing that it outperforms these state-of-the-art solutions in many aspects for our targeted use cases. Furthermore, we deploy our system in a real-world environment, showing that it is capable

of coping with the workload generated by gas turbines used for municipal electricity generation as one specific example.

This work was carried out in the context of a joint project of Technical University of Munich and IfTA GmbH[3]. This connection made it possible to have access to expert knowledge in real-time sensor data acquisition and analysis as well as detailed domain knowledge about gas turbine operation. Furthermore, through the close contacts of IfTA GmbH to industry, the developed system could be evaluated in a production-grade industrial environment. While HAQSE is designed with a specific industrial application in mind, the focus of this dissertation, however, lies in showing the generality of the approach and methods.

1.5 Thesis Organization

This thesis is organized as sketched in Figure 1.4. In Chapter 2, we define concepts used in this thesis, explain use cases, analyze requirements for our system and give an overview of the state of the art. Based on the derived requirements, in Chapter 3, we present a high-level overview of our system. This comprises the interfaces of our system, the individual components, and how they interact with each other. This chapter strives to provide enough information to understand the general architecture of our system, enabling implementors to develop data input and query clients. The details of the individual components, in-depth evaluations, as well as comparison with state-of-the-art systems follow in the next three chapters. In Chapter 4, we discuss all aspects related to efficient long-term sensor data storage. In particular, we discuss data layout alternatives as well as lossless and lossy compression techniques to make efficient use of the available storage space. In Chapter 5, we explain all aspects of our system that are required for efficient sensor data stream consumption, from the input interface to internal processing. This also includes calculation of hierarchical windowed aggregations. In Chapter 6, we describe our approach for ensuring efficient sensor data retrieval and evaluate its performance. The research for this thesis was performed in the context of a real industrial problem. In Chapter 7, we describe the integration of our system into a real-world sensor monitoring installation for a gas-fired power plant, operated by a customer of IfTA GmbH. For this, we describe the integration of HAQSE into an existing sensor data generation and analysis infrastructure and show that our system improves many application scenarios. In Chapter 8, we outline future work and conclude this dissertation.

[3]Basic information about IfTA GmbH can be found in the glossary.

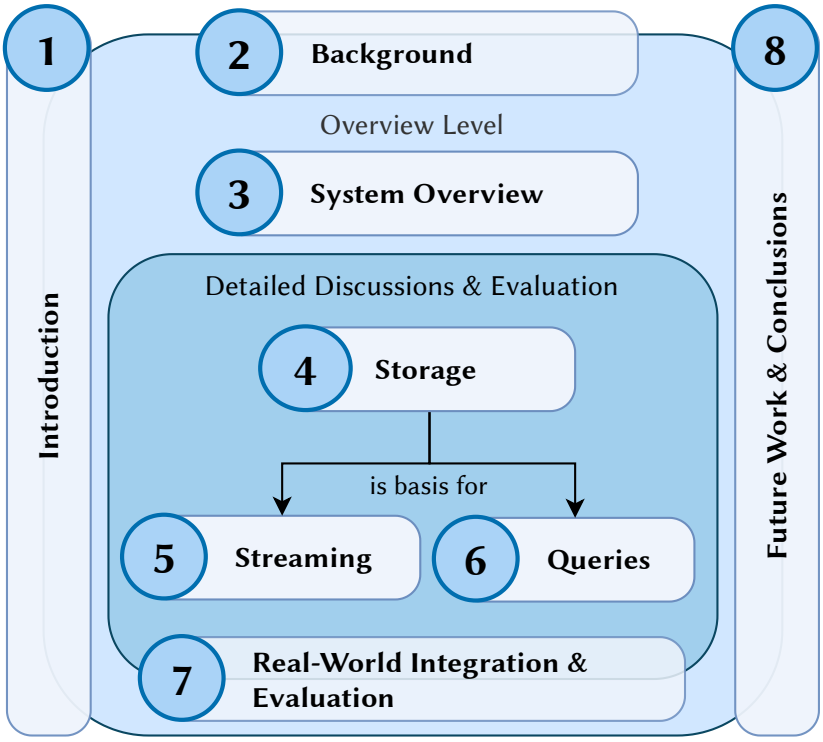


Figure 1.4 Organization of this dissertation (boxes are linked to the respective chapters).

2 Context, Requirements and Related Work

In this chapter, we discuss the background and context of this thesis. In Section 2.1, we start with defining a set of core concepts used in our approach. These concepts help to derive application requirements. They also show that our approach is not specific to a certain application scenario, but is generally applicable to all cases that fit the presented abstractions. In Section 2.2, we list example applications from various industrial fields that represent exemplary use cases and have a common set of challenges that motivate our work. In Section 2.3, based on these use cases, we derive important processing and storage system requirements that we will use as the foundation of the rest of this dissertation. Finally, in Section 2.4, we discuss the state of the art and review related work.

2.1 Definitions

In this section, we define concepts and abstractions that we use later on to specify the interfaces of our system. This helps to understand the applicability of the system we will describe in the following chapters. We start by describing common concepts and build on and specialize from these.

Definition 1 (Sensor and Sensor Measurements)**.** A *sensor* σ is a source of quantifiable information. At any instant (also called timestamp) ts, it can produce a scalar *sensor measurement value* v. The process of producing such a sensor measurement value v is called sampling. Values v for a certain sensor σ are either real-valued or integral numbers, i.e., $v \in \mathbb{R}$ or $v \in \mathbb{Z}$. To process this information in a digital system, the measurement values v must be discretized and encoded in some binary format. We expect this format to be of a fixed type using a defined number of bits per value and adhering to a certain interpretation of these bits.

An actual sensor can be of different types. On the one hand, a sensor can be a physical device measuring an observed quantity in the real world. On the other hand, it can be an abstract, virtual source that produces v from another source of information, potentially deriving it from one or multiple physical sensors. However, for the system discussed in this work, the exact sensor type producing measurement values is inconsequential. In practice, the digital format is something like a `float32` or `int64`, em-

ploying, e.g., a standardized floating-point number encoding like defined in the norm IEEE 754 [50] or the two's complement representation [85] for integers.

In real applications, a *physical* sensors measures continuous quantities like a pressure or temperature. Such a physical sensor is sampled by a measurement system that converts analog sensor readings to digital values. There is also a wide range of examples for *virtual* sensors. This can, e.g., be the load of a processor or the number of occupied bytes of memory in a computer system. It can, however, also be an analysis result on the measurements of another sensor, such as the result of a frequency transformation (e.g., a FFT). In this case, the associated timestamp ts could be the timestamp of the first sample of the analyzed time window. In every case, the value of a sensor is represented by a scalar numeric type of fixed size. More complex analysis results like spectral data or values $\in \mathbb{C}$ must first be decomposed into their individual components so that they can be represented by multiple, "virtual" sensors.

Definition 2 (Sensor Schema)**.** A list of sensors $\mathcal{S} = (\sigma_1, \sigma_2, ..., \sigma_n)$ is called a *sensor schema* or simply *schema*. Each sensor σ in the schema $\mathcal{S}$ has a name that allows to uniquely identify the sensor σ in $\mathcal{S}$. We refer to the number of sensors $|\mathcal{S}|$ in a schema as the *schema cardinality* or *schema size*.

Sensors in a schema are not required to have identical types. Furthermore, while having a unique name is technically not necessary (sensors could also be identified by their position in the schema), it makes interaction with the system (especially for humans) a lot easier.

Definition 3 (Sensor Sample)**.** A sensor sample ψ_i is a list of sensor measurement values $v_i^{\sigma_j}$ generated by the sensors σ_j in a sensor schema $\mathcal{S}$ at a particular instant ts_i. Since all $v_i^{\sigma_j}$ are measured for the same instant ts_i, the sensors are said to be sampled synchronously.

The sensor schema $\mathcal{S}$ from above also defines how sensor samples ψ can be stored for later retrieval. When combined with a description for the timestamp, $\mathcal{S}$ can be used to define, e.g., the schema of a table in a database management system where sensor names represent column names and sensor types represent column types. The timestamp column requires a separate, reserved column name and a type that is suitable for representing the respective timestamp values. A sensor sample ψ can then be thought of as a row in such a database table (e.g., one row in Table 2.1). Or, when looking at the plots in Figure 2.1, one sample is a vertical section across all three plots for one particular timestamp.

Definition 4 (Sensor Source Stream)**.** A *Sensor source stream* is a (possibly infinite) sequence of sensor samples $\ldots, \psi_{i-1}, \psi_i, \psi_{i+1}, \ldots$, generated by the sensors σ of a sensor schema $\mathcal{S}$. If $|\mathcal{S}| = 1$, the sensor source stream is *univariate*, if $|\mathcal{S}| > 1$ it is *multivariate*.

In this work, we focus on multivariate sensor streams. Multivariate sensor streams are special since they exhibit properties that allow and require certain optimizations

in order to make processing and storage more efficient or even viable at all. We exploit some of these aspects in the design of our system. Our approach is also applicable to univariate sensor streams as these are just a special case of multivariate sensor streams.

We assume that the timestamps ts_i for the sequence of sensor samples $\psi_i, \psi_{i+1}, \ldots$ are strictly increasing for a certain sensor stream, i.e., $ts_i < ts_{i+1} \forall i$. In Figure 2.1, we show an excerpt of a sensor source stream for three sensors over a duration of a bit more than two seconds.

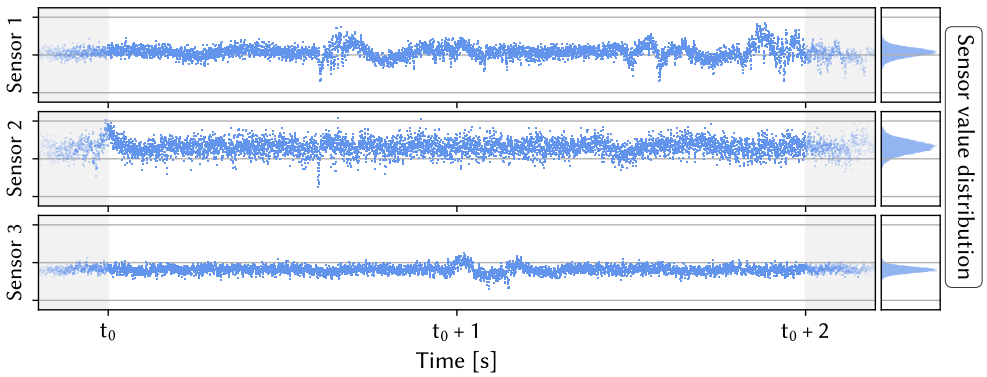


Figure 2.1 Example of raw sensor data streams for three different sensors, sampled synchronously. These three plots show an excerpt of slightly more than two seconds, spanning more than 50 000 raw sensor readings in each plot. The right part of the plot shows a histogram of the value distribution for each of the sensors.

Definition 5 (Stream Resolution and Sampling Rate). The difference $ts_{i+1} - ts_i$ between the timestamps of two consecutive samples ψ_i and ψ_{i+1} is called the *stream resolution* Δt. We expected the stream resolution Δt of a certain sensor source stream to be more or less constant. This means that $ts_{i+1} = ts_i + \Delta t + \epsilon, \epsilon \ll \Delta t$ for almost all i. In other words, over a long enough period of time, the error ϵ is expected to cancel out such that the average $\overline{\Delta t}$ converges against the nominal value of Δt. A sensor source stream is expected to produce samples ψ at a constant average *sampling rate* of $f_s = \frac{1}{\Delta t}$.

The example data from Figure 2.1 is an excerpt from a real system. The Δt in that application is 39.0625 µs, i.e., the sampling rate is 25.6 kHz. The samples ψ of a sensor source stream usually appear ordered in time, i.e., sample after sample. We thus call the samples on this stream to be *time-ordered*.

Definition 6 (Sample Batch). In addition to this sample-wise generation, systems generating a sensor source stream might buffer and collect multiple samples $\psi_i..\psi_{i+bs-1}$ and send these as one *sample batch*, containing a batch of contiguous samples (see Table 2.1). The number of samples bs in a sample batch is called *sample batch size* or simply *batch size*.

When the sample batch is used as input to a system, we refer to it as *input sample batch*.

timestamp *ts*	sens1	sens2
594796279880	4.246	8998.3
594796279882	4.312	8999.7
594796279884	4.356	8999.7
594796279886	4.416	9000.0
594796279888	4.426	9000.3
594796279890	4.411	9000.2
...		

Table 2.1 Example sensor data based on a schema with two sensors. Highlighted in red: a single sample (ψ_i). Highlighted in blue: one sample batch with a batch size of four.

Definition 7 (Stored Sensor Stream)**.** The sensor samples $\Psi = (\psi_0, \psi_1, ..., \psi_m)$ of a sensor source stream can be stored in some kind of database so that data can be retrieved later on from that database. When stored in such a database, we call the set of sensor samples Ψ a *stored sensor stream*. Such a stored sensor stream Ψ requires a unique stream name in the database to identify it when retrieving sensor samples from a collection of multiple stored sensor streams $\Psi_0, \Psi_1, \ldots, \Psi_N$.

There are a few conceptual differences between a sensor source stream and a stored sensor stream. For one, a sample ψ in a sensor source stream needs to be processed in-situ, i.e., when it is generated. In contrast to this, samples $\psi \in \Psi$ that are available in a stored sensor stream can be retrieved for later use. As a consequence, though, a stored sensor stream is never infinite, as it is always limited by the amount of available storage space.

An example of a stored sensor stream is sketched in Table 2.1. In this example, the schema $\mathcal{S}$ would consist of two sensors producing real-valued measurement values, with the sensor names *sens1* and *sens2*. One sample ψ_i corresponds to one row (as highlighted in red) in the table. A sample batch would correspond to a contiguous range of rows in the table, as highlighted in blue in Table 2.1. As the dashed boxes around the set of values of one column in that table indicate, we expect a sample batch to be organized column-by-column, i.e., sensor-ordered.

Definition 8 (Range-Based Query)**.** A contiguous excerpt of a stored sensor stream Ψ can be retrieved using a *range-based query*. Such a query consists of the following parts:

- The name of the stored sensor stream Ψ
- A query range consisting of start and end timestamps ts_s and ts_e
- A list of sensor names for which to retrieve measurement values v

Only samples ψ_i that have a timestamp $ts_s \leq ts_i \leq ts_e$ match the query. The result set only contains data for the specified sensor names. This list of sensor names may only contain names that are part of the schema $\mathcal{S}$ of Ψ.

The amount of data in the result of range-based queries scales linearly with the size of the specified time range (assuming the complete time range contains samples). Consequently, very large time ranges imply very large result sets, which makes retrieval of such results very slow. This is unnecessary if only an overview of the time range is required. Hence, it is useful to retrieve approximate versions of Ψ, as Ilsche et al. have shown [52, 53]. We thus define a few additional constructs.

Definition 9 (Aggregated Stored Sensor Stream)**.** It is possible to aggregate a set of consecutive samples $\psi_{i..j}$ of a stored sensor stream Ψ based on a time window of a specific length Δt^{ω}. For this window, all sensor measurement values $v^{\sigma}_{i..j}$ for a certain sensor σ are combined into a single value v^{σ}_{A}, based on an aggregation function f_A. There might be multiple such aggregation functions; we call the set of aggregation functions $\mathcal{A}$. Aggregations are calculated for all sensors σ in the schema $\mathcal{S}$ and all aggregation functions $f_A \in \mathcal{A}$. The resulting columns are identified by the tuple (σ, f_A). Consequently, the schema of the aggregated stream is the Cartesian product $\mathcal{S} \times \mathcal{A}$ of the original sensor schema $\mathcal{S}$ and aggregation functions $\mathcal{A}$. The resulting windowed aggregation samples ψ^A use the start of the aggregation window as timestamp ts. These samples can again be persisted, resulting in an *aggregated stored sensor stream* Ψ^A that is derived from and associated with Ψ. The stream resolution Δt^{ω} of Ψ^A is identical to the window size Δt^{ω} of the respective windowed aggregation calculation. This Δt^{ω} is larger than Δt of the original source stream[1].

Aggregations are used to create summaries of stored sensor streams. Typical aggregation functions are *max*, *min* and *mean*. It is useful to define the window size Δt^{ω} a multiple of the stream resolution Δt of the sensor source stream.

Definition 10 (Sensor Stream Hierarchy)**.** For a certain Ψ, multiple aggregated stored sensor streams Ψ^A with different Δt^{ω} may exist. The set of such Ψ^A, ordered by Δt^{ω}, together with the original stored sensor stream, which we call Ψ^O in that context, is called a *sensor stream hierarchy* Ψ^H.

Figure 2.2 shows an example of such a sensor stream hierarchy with min and max aggregation functions. The aggregated versions provide approximations of the original data.

Definition 11 (Hierarchical Range-Based Query)**.** In contrast to above, a *hierarchical range-based query* is specified against a sensor stream hierarchy Ψ^H. Such a query is similar to a range-based query, but it specifies the name of the sensor stream hierarchy Ψ^H instead of a specific Ψ. By specifying a maximum sample resolution $\Delta t^{\max}$, the

[1] Or, equivalent to that, the f_s of Ψ^A is smaller than in the source stream Ψ.

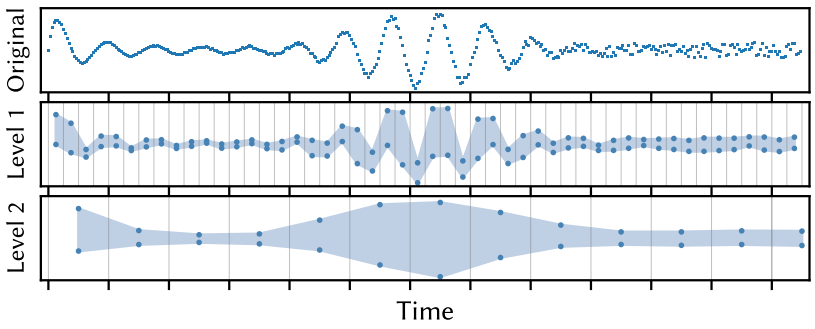


Figure 2.2 A multi-resolution sensor stream hierarchy stores original values and windowed aggregates. The top plot shows original sensor samples, the middle and bottom plots show aggregated values (visualized using min and max windowed aggregation values in this example).

system answering the query can select a matching level in the sensor stream hierarchy. This is done by selecting the Ψ^A with the largest $\Delta t < \Delta t^{\max}$. In addition to that, a list of aggregation functions $f_A \in \mathcal{A}$ needs to be specified as part of the query. The result set then contains the pre-computed aggregation samples ψ from the columns (σ, f_A) from the chosen Ψ^A or the original samples if Ψ^O is selected as the level to query.

Selecting a *matching level* from the hierarchy is actually not restricted to the logic just presented. This can be application specific: there are also cases in which it might be better to select the level that is closest to (and not necessarily smaller than) the requested resolution. Similarly, it does not really matter how the query specifies how a level should be selected: specifying a *maximum number of points* is equivalent to specifying a maximum resolution; together with the query range, both representations contain identical information. Both level selection and how queries specify which level to select can be easily adapted to specific use cases.

The system we present in this work uses a sensor source stream as input. Retrieving data from it is done via hierarchical range-based queries. We introduce an overview of the system in Chapter 3.

2.2 Example Application Domains & Scenarios

There are several application domains and scenarios that fit the abstractions presented in the previous section. We first discuss various application domains and the specific challenges in these fields. After that, we present usage scenarios, i.e., how sensor data is valuable for its users.

2.2.1 Application Domains

Sensors play a crucial role in monitoring and controlling operations in countless application domains. Exemplary application domains include—but are not limited to—

monitoring high-performance computing systems and power plants with rotating machinery such as gas or wind turbines. These example usage domains are all monitored by sensor technology and generate sensor streams with a high data rate. Additionally, in all these cases, users of sensor stream data analyze long time spans in different usage scenarios (also see Section 2.2.2).

Monitoring in High-Performance Computing

Sensor data is collected in High Performance Computing (HPC) systems for a variety of reasons. Two prominent examples are monitoring and operational data analytics [84]. Monitoring is challenging with respect to storage, scalability and performance: several terabytes of data may accumulate per day, these large volumes require frameworks that scale accordingly and the monitored system must not be affected by the monitoring system [84]. Operational data analytics uses and analyzes the collected monitoring data with the goal of improving the operation of an HPC system or data center [15]. Collecting sensor data is also essential for ensuring energy-efficient operation and improving future procurement of HPC systems [52, 78]. Typical aspects monitored in such HPC systems include the power consumption of the individual components of the system as well as metrics such as CPU and memory load values. As systems contain thousands of compute nodes, each one able to measure data from thousands of sensors, it becomes clear that the main challenge in such systems is the number of sensors that needs to be dealt with [84]. Sampling rates are typically in the order of few Hertz [52, 84].

Monitoring Power Plants

As discussed in Chapter 1, reliable and efficient power generation is challenging for a variety of reasons. The transition to renewable energy sources makes this even more challenging for operators of all kinds of power plants. Consequently, they invest in various monitoring solutions to control and protect the operation of their systems. Furthermore, they also use the generated monitoring data for implementing predictive maintenance and optimizing the future operation of these systems. In the following, we discuss two specific examples and why monitoring data is important and highlight particular challenges.

Gas-Fired Power Plants Gas-fired power plants employ gas turbines for electricity generation. These are very flexible in their operation, which makes them perfect for compensating peak demand scenarios or during times when renewable sources of energy cannot produce power. Under certain operating conditions, the combustion process in these gas turbines is prone to thermoacoustic instabilities. These instabilities can damage the turbine's structure, so it is crucial to detect and prevent these instabilities as early as possible. This is done by monitoring the combustion process using

high-temperature pressure or acceleration sensors. These sensors monitor the oscillating pressure or vibrations in the combustion chamber of a gas turbine with sampling rates of up to 50 kHz. In addition to that, high-frequency acceleration and distance sensors measure rotor dynamics. Finally, there are thousands of other sensors measuring operating conditions of the whole power plant such as power output or ambient temperatures. These operating conditions are sampled at few Hertz.

In addition to the raw data of a sensor, especially for oscillation phenomena, it is common to also calculate and store transformations into the frequency domain. This is often done through short-time Fourier transforms (STFTs) that are applied to equally-sized, moving or tumbling windows on the raw sensor stream[2]. These transformations into the frequency domain could be calculated very quickly on raw data when needed. However, storing the calculation result is still useful for several reasons. Sometimes, specific parameters such as the window size, the applied window function or the exact window alignment are not known in post-processing environments. Additionally, in some cases, it is sufficient to store only certain parts of the spectral data, as this can save a considerable amount of storage space. Finally, further aggregating spectral data over time can help to quickly identify problematic ranges when analyzing large time windows. This is done by calculating, e.g., the maximum value for each frequency over a certain time window.

Wind Turbines Wind turbines play an important role in the generation of renewable power. In cold operating conditions, ice may build up on the turbine blades of such wind turbines [133]. Operators need to detect this condition to counteract or stop the turbine. One way to detect ice on blades is by using fiber-optic acceleration transducers. Frequency analysis of the generated data helps to identify ice building up on the turbine blades since the mass of the turbine blade increases because of the ice attached to it. This change in mass can be observed in the frequency analysis of the acceleration data.

The data is also stored and used for historical data analysis.

Deployment Context

Industrial sensor data processing systems are often deployed in remote, fully automated environments. This has several consequences.

In contrast to other, similar work [52, 84], we do not target our system only at high-end server-grade systems. Instead, the operating environment on-site is restricted with respect to several dimensions. First, systems deployed on the edge are restricted in physical size since rack space is often very limited. Second, the processing hardware may consume only a limited amount of power. This is partly due to the small form factor since less heat can be dissipated on such small devices. Additionally, low power

[2] In most cases, these STFTs are implemented using the FFT.

consumption is also needed as fan-less systems are preferred in industrial settings and since fans required for cooling are a common point of failure. Replacement may be costly or not be possible at all in fully automated environments, strictly limiting the thermal design power (TDP) of employed systems.

The monitored systems are often deployed in geographically remote areas. Thus, there is an additional set of challenges related to network connectivity. First, bandwidth to these systems is often limited and may not be sufficient to transmit the complete sensor data stream to a cloud system on the Internet. Second, latency to other systems may be relatively high, requiring that any real-time control logic needs to be performed on-site. Third, connection reliability may be limited in certain scenarios. In such cases, continuously sending a sensor data stream is also intractable.

In cases when the monitored system is considered critical infrastructure—as is the case for power plants—the complete sensor processing, analysis and storage system might even be installed in an air-gapped environment, i.e., it might not be reachable via the Internet. As a consequence, the complete sensor processing and storage system has to be self-sufficient, i.e., all required functionality needs to be present on-site.

2.2.2 Application Scenarios

Industrial sensor data is used in different ways. We classify the various application scenarios along two dimensions. First, applications differ depending on the time since data has been generated. Second, the interactivity requirements of the respective use case vary to a large extent. This is visualized in Figure 2.3.

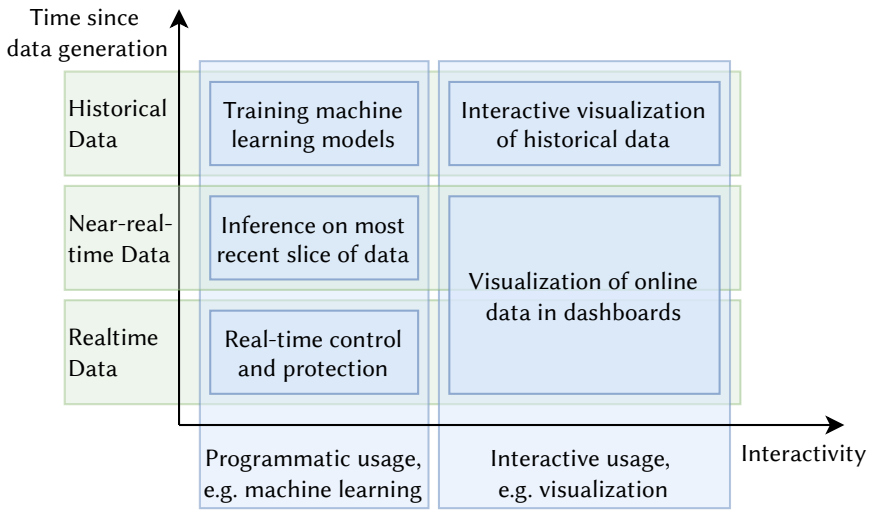


Figure 2.3 Classification of different uses for sensor data, with a specific example for each category.

Regarding time since data generation, the time range of interest ranges from ten to hundreds of milliseconds for real-time applications and goes up to several years of sensor data. Real-time applications typically process the most recent time slices of a sensor source stream, generating analysis results or visualizations for dashboard applications. After the data slice is processed and the respective result has been generated, the processed samples are dropped and the next slice of recent sensor samples is analyzed. For these (near) real-time applications, data storage and compression are not very important, as the processing happens on data present in volatile memory and no data accumulates over time. On the contrary, use cases that require historical data rely on some kind of long-term storage that provides samples for data retrieval. In these cases, efficient storage—including appropriate data compression—and retrieval are crucial for a satisfactory user experience.

Regarding interactivity, the main difference often lies in the amount of data and the required level of detail necessary for a particular application. As one example, in interactive exploration setups, users expect short response delays. The amount of data and level of detail is often more limited in such cases. On the contrary, for programmatic usage, the amount of data and level of detail is higher. This requires high query throughput so that the programs and algorithms using this data spend as little time as possible waiting for their input.

Especially regarding the interactive use of sensor data, the boundaries between historical and real-time data are blurry. Ideally, users monitoring most recent (real-time) data can seamlessly interact with the data and change the visualization to larger, historical time spans. In that sense, real-time data can also be considered historical since it has already been generated.

2.2.3 General Properties of Industrial Sensor Data

In the context of this thesis, the term *industrial sensor data* includes all sensor data that can be matched to the Definitions 4 and 7 (Sensor Source Stream and Stored Sensor Stream) from Section 2.1. In particular, this includes data generated by the applications presented in Section 2.2.1. As discussed in Section 2.1, we assume sensors are sampled more or less regularly. Thus, our focus is not on single, independent sensor measurements but on a stream of consecutive sensor samples. Alternatively, this stream can be thought of as a multivariate time series of sensor measurements. Several aspects make handling this data special compared to other types of data.

Time Series Data Sensor streams are generated by reading values from one (or multiple) sensors as they evolve over time. As a consequence, there is a natural, inherent ordering in these sensor readings. Since we assume that timestamps are strictly in-

creasing[3] (cf. Section 2.1), they are perfect for indexing the produced sensor data for time-range-based retrieval.

Numeric Noisy Data Sensor data is often subject to different sources of noise. Thus, the stored sensor streams contain noisy data. There are various sources of this noise. The noise may come from the actual physical process noise (where the measured physical phenomenon is inherently noisy), noise generated by the measurement chain (the physical sensor and all parts of the signal chain until the data is converted from an analog to a digital signal) as well as quantization noise induced by the (necessary) conversion to some kind of binary representation (floating-point, fixed-point or any lossy conversions between representations) in a digital processing system. The noise in the sensor data, especially in floating-points sensor data, makes it hard to compress efficiently using standard compression methods. These methods often rely on techniques like entropy encoding or dictionary encoding, which work better on data with less entropy and repeating structures like, e.g., natural language texts.

In some situations, it can be beneficial to reduce the precision of the stored data and store it using lossy compression. This is especially useful when the noise level of the measurement chain exceeds that of the actual physical process the sensors monitor. In such cases, when the signal entropy is increased by the measurement process, it is not useful to store the full resolution a certain data type (e.g., single-precision floating-point) is able to represent. In other cases, e.g., training a machine learning model, it also may not be necessary to have data available in full resolution.

Access Patterns Regarding data ingestion, sensor source streams, as defined in Section 2.1, represent an append-only workload. This means that existing data never needs to be updated with new values. When data is deleted, this is typically done in large contiguous chunks of the stored data. One common implementation of such a strategy is to delete the oldest chunk of data, creating a ring-buffer-like data structure for stored sensor streams.

While sensor streams are generated in a time-ordered fashion, access patterns for data retrieval differ for most analytic or visualization applications. Being time series data, users are often interested in the behavior of one or few sensors over a certain time range. This is the case when visualizing sensor data, which often happens for few signals over a certain time-range (as shown for three sensors in Figure 2.1), or when performing other analytic tasks.

[3] On a small scale, the timestamps should be evenly spaced, while on large scale, any time drift should be avoided. In order to achieve that without high-precision clocks on the sampling hardware, the sampling rate must be controlled and regulated by a reference clock source, e.g., an NTP-server or a GPS-clock. This is a non-trivial, but solvable task, as demonstrated by other works [49, 60, 82].

2.3 Requirements Analysis

Based on the definitions and application scenarios presented so far, we design a sensor data processing and storage system. In this section, we derive important requirements for this system. We partition these requirements into three different aspects: sensor stream storage, sensor stream ingestion and range-based queries. This base structure is also used to organize all other parts of this thesis.

2.3.1 Long-Term Storage Requirements

As discussed in the previous section, the amount of sensor data generated and required to be stored for historical data analysis is huge. We thus derive the following requirements for a long-term sensor stream storage solution:

Requirement 1 (Storage Efficiency for Floating-Point Data)**.** We want our system to have the ability to store large volumes of sensor samples efficiently, i.e., consume as little storage space as possible. Specifically, one of the main challenges is storing noisy floating-point measurement values. Thus, our system must include a way to efficiently represent floating-point as well as timestamp data for persistent, long-term storage.

Requirement 2 (Data Type Support)**.** There are several requirements regarding supported data types in our system. First, since we want to support high sampling frequencies up to at least 200 kHz, it is important that the data type used for storing the timestamp provides enough temporal resolution. Thus, the system must support timestamps with at least nanosecond precision. Furthermore, it must be possible to store real-valued data in both single-precision and double-precision floating-point format. In addition to that, our system should support storing integral as well as boolean values.

Requirement 3 (Optimized for Analytics of Time-Series)**.** The target use cases are mainly analytical, i.e., quickly reading a specified time-range for a small set of sensors. It is important that such data retrieval workloads can be executed efficiently on the chosen long-term storage format.

Requirement 4 (Interoperability)**.** The stored data format should be independent of a certain application software. Instead, the chosen data format should have proven library support in a variety of programming languages. This ensures that stored data is usable even without a certain application, which is often desirable in industrial contexts. This secures technology investments for users.

The listed storage requirements build the foundation of the system. Requirements 1 and 3 have a direct impact on the other aspects of our system like data ingestion or retrieving sensor data for later use.

2.3.2 Sensor Stream Consumption Requirements

Consuming sensor streams with high data rates is challenging. Many existing systems do not come close to the rate what industrial applications require. We mainly aim to support high sampling rates, but also scenarios with lower sampling rates and high schema cardinalities. We thus derive the following requirements:

Requirement 5 (High-Bandwidth Stream Consumption)**.** Our system should be able to consume sensor streams with high stream rates. Stream rates are dependent on two factors: the schema cardinality and the stream's sampling frequency. We want to support sampling frequencies of at least 200 kHz, but our system should work well regardless of which of the two factors leads to a high stream rate. In contrast to existing systems, for our system, sensor source streams originate from a single stream source. Existing systems are often built for consuming and integrating data from multiple sources into a single stream, which is not a main goal of our work.

Requirement 6 (Durability Guarantees)**.** In order to provide high stream rates, we relax durability guarantees. Sensor streams are eventually persisted to disk. However, we do not require that durability is ensured and acknowledged after every received sample.

Requirement 7 (Online Hierarchical Aggregation)**.** Some application scenarios, e.g., interactive exploration settings, require responses to data queries to be answered within less than 100 ms. At the same time, the results of such queries need not always be exact: often, approximate results are sufficient. To be able to answer such approximate queries in a timely manner, we require our system to pre-calculate aggregations on data ingestion. This enables our system to serve these pre-calculated results for queries where approximate results are sufficient.

Requirement 8 (Footprint & Isolated Operation)**.** Since resources in industrial environments are often limited, we require the system to be able to run on a single node and perform all processing there. The system should use available system resources, especially the CPU, efficiently. Furthermore, since no constant connection to an Internet-connected cloud service or similar can be expected, the system should be able to operate self-sufficiently.

Requirement 9 (Interoperability)**.** It should be easy to create sensor stream sources that can provide data to our system. In particular, this means that it should be possible to develop sensor stream sources in a programming language or operating system-independent way.

Our ingestion pipeline must be able to prepare data such that it can be stored using a storage format that fulfills the requirements listed in Section 2.3.1. While we do not list this as a separate requirement here, combining the two sets of requirements is one of the primary challenges in designing and building a viable system. Some requirements listed here, in particular Requirement 7, already prepare data retrieval aspects.

2.3.3 Query Requirements

Retrieving data from a storage system is the main reason why it is created and used. Ultimately, without reading data from a database, there would be no need to store any of it in the first place. Based on the application scenarios mentioned previously, retrieving data from the system to be developed should satisfy the following requirements:

Requirement 10 (Query Speed)**.** Our system must be able to answer time-range-based sensor data queries quickly and efficiently. This includes two aspects. First, we require the *constant* query overhead to be small. This is most important for queries with a small result set, such as simple data queries for interactive visualization. Second, we also require *throughput* for larger result sets to be high, near what the hardware can support. This has been shown to be a bottleneck in many existing systems due to result set serialization [96]. High read throughput is important for applications such as machine learning, where large quantities of sensor data are used to train machine learning models. Our system should be especially efficient for queries that retrieve data for few sensors and a contiguous time-range that contains few to billions of raw data points.

Requirement 11 (Online/Offline Data Queries)**.** As soon as sensor samples are received by our system, it should be possible to query these samples. This is important for online data queries that query the most recently received samples. There should be no functional difference for users of our system whether they query most recent data or data that was received months ago. Performance differences in that regard should be minimal.

Requirement 12 (Querying Data in Different Time Resolutions)**.** It should be possible to quickly retrieve time-windowed approximations/aggregations of data. This can speed up retrieval of large time ranges considerably, when only an approximate overview of the queried time range is required. This is especially important for interactive scenarios like explorative data analysis. As mentioned before, such time-windowed aggregations may be computed already during ingestion (see Requirement 7).

Requirement 13 (Transparently Querying Multiple Sites)**.** If data is present on multiple sites, there should be a way to retrieve data from multiple such sites. Different sites cannot be expected to be interconnected at all times.

Requirement 14 (Footprint)**.** The system should use available system resources, especially CPU and RAM, efficiently, as deployment may happen in resource-constrained environments.

Requirement 15 (Interoperability)**.** It should be easy and efficient for other software systems to retrieve data from our system. This implies that the query interface should implement open standards. Ideally, the query interface is independent of specific programming languages, making adoption by existing tools easier.

All these requirements motivate design decisions that are distributed across the various components of our software architecture. We structure the remainder of this work according to the structure of these three sets of requirements. While we present a high-level overview of the whole design in Chapter 3, the details of the respective components will be discussed in Chapters 4 to 6.

2.4 Related Work

There is a considerable amount of related work on the various topics for which this work proposes improvements. Specifically, we review related work in the areas of time series management systems (Section 2.4.1), compression approaches for sensor measurements (Section 2.4.2), as well as methods for other related aspects (Section 2.4.3).

2.4.1 Time Series Management Systems

Previous research has shown that there is no data management system design that fits all use cases [105]. In particular, time series data requires different access patterns when compared to what general-purpose database management systems offer. For that reason, there is a large category of systems specialized for processing and storing time series: time series management systems. There are several existing solutions, both open source and commercial systems, that focus on storage of time series data [101]. The survey of Jensen et al. [61] gives a good picture of the state of the art.

InfluxDB

InfluxDB [54] is a widely used open-source time series database. In InfluxDB, data is stored in a schemaless design, organized in *buckets*. *Logically*, these buckets contain several *measurements*. Measurements, in turn, contain actually measured metrics in so-called *fields*, organized as *field keys* and the associated *field values*. In addition, such measurements may have *tags* to identify other measurement properties. Each entry in a measurement is uniquely identified by its timestamp, which has nanosecond precision. *Physically*, data on disk is stored in *series*, identified by a measurement, a specific value for each tag (a *tag set*) and a single field key. The series then stores the timestamps and values together as one Time-Structured Merge Tree [56] (TSM) file on disk, encoding blocks of timestamps and values independently. The schemaless design (and the implementation with storage in individual TSM files) requires that timestamps are stored redundantly for field values at equal timestamps. Floating-point data always uses double precision in InfluxDB.

Data retention is managed per bucket. *Shard groups*, which contain the actual TSM files, have a certain shard group duration associated with them. If all data of a certain

shard group is older than the specified data retention period, the complete shard group is deleted.

As tag sets and the number of measurements increase (its cardinality), the number of series stored in files increases quickly. To tackle this, InfluxData works on a new database core called *IOx*, which uses Apache Parquet for persisting data [88]. First results on using Apache Parquet also show that it is well possible to query Apache Parquet files with a latency of only several milliseconds [110].

Influx uses a text-based input protocol, the Influx Line Protocol (ILP). Data received via this protocol is first collected in a write-ahead log (WAL), and cached for enabling faster reads on the data. The contents of the WAL are regularly compacted in TSM files mentioned above. Additionally, TSM files are regularly combined in larger TSM files.

TimescaleDB

TimescaleDB [126], being an extension on top of PostgreSQL, takes a different approach. It utilizes so-called *Hypertables* to manage one consistent view over multiple child tables, so-called *chunks*, of data. This allows increasing throughput since costly index operations are performed on smaller contiguous time segments. It is exploited for appending data (requiring frequent reorganizations of the index) as well as for deleting old data, which simply happens in full chunks. Data can be inserted using Structured Query Language (SQL) statements or the PostgreSQL specific stream-protocol. TimescaleDB comes with all the features of PostgreSQL, making it a feature-rich and proven alternative for storing time series data. However, as shown in previous research [1], the performance of analytic workloads can be increased significantly by using columnar data layouts. TimescaleDB also allows compressing older parts of the data, which groups together values of individual columns of a contiguous set of rows. This results in a ”hybrid row-columnar format” [125] that also includes data hints in order to optimize querying data [80].

QuestDB

QuestDB [93] is a relatively new competitor in the field of time series databases. It is developed by the equally named company and claims to be *”the fastest open source time series database”* [93]. It features different input protocols (SQL, ILP, comma separated values (CSV)) and data retrieval options. QuestDB employs columnar storage [95], where each column is stored in a separate file. Ingesting incoming samples is achieved by writing to a memory-mapped data page for each such file. This data page is written back to persistent storage once it is filled completely. However, it does not support compression on the data model level. The only option is to use compression on the file system level. Furthermore, it has no support for precomputing windowed aggregations. Additionally, there is a serialization/deserialization overhead for input/output into/out of the system since all input/output is text-based.

MetricQ

MetricQ [52] originates as an academic project for managing energy measurements in an HPC environment. Its scalable design in based on a message broker (`rabbitmq`). This message broker processes all data that is going in to and out of the system. Ilsche [52] also presents the concept of Hierarchical Timeline Aggregation, enabling efficient aggregate queries over large datasets. In contrast to other systems, it employs a binary encoding (protocol buffers [40]) for data input and output, avoiding serialization/deserialization overheads to/from text. Furthermore, input can happen in sample batches. It uses a storage layout similar to InfluxDB, where each metric is stored together with its timestamp. There are a few limitations, however, that make it impractical for general sensor monitoring applications. First, all samples are stored as double-precision floating-point types. It is not possible to store single precision or integer data types. Furthermore, compression for long-term storage is not considered.

Others

There is also a list of other time series management systems. Building on top of HBase, OpenTSDB provides a solution targeting distributed setups [123]. BTrDB [6] introduces a copy-on-write data structure for managing multi-resolution statistical aggregates on time-series data. Data in BTrDB is stored in a Ceph [134] storage pool and compressed in a block-wise fashion. RRDTool [127] is a high-performance logging and graphing system for time series data. It only supports timestamps in a resolution of seconds. Gorilla [90] is an in-memory time series management system development by Facebook. It tries to reduce storage overhead by employing a lightweight floating-point compression approach. However, timestamps in Gorilla are stored with a granularity of only seconds.

Summary: Time Series Management Systems

Most of these solutions require some data serialization to text and parsing from text to ingest data (the notable exception being MetricQ). These serialization and deserialization overheads are prohibitive for high data rates. The inefficient encoding alone limits throughput, as we show in our evaluation in Chapter 5. Furthermore, none of the presented alternatives includes a flexible data compression scheme, where compression can be configured per sensor. Another issue is that not all systems natively support nanosecond timestamps. This is problematic for all use cases that require such high temporal resolutions. Many solutions exploit the increasing nature of timestamps and apply some kind of delta encoding.

2.4.2 Compression of Sensor Data

As discussed earlier, it is crucial for many application scenarios that the available storage space is used as efficiently as possible. Moreover, the throughput for both compressing and decompressing data should be as high as possible. There is a vast body of literature targeting compression of sensor values. We specifically investigate compression of floating-point and timestamp data. For floating-point data, we further distinguish between lossy and lossless compression methods.

Lossless Compression of Floating-Point Data

There are several lossless compression schemes for floating-points data. The different algorithm make different assumptions about the structure of the data to compress. Lindstrom et al. designed fpzip [74] for spatially correlated multidimensional data (in most cases, points lie on an evenly spaced, multidimensional grid). Such data often results from scientific simulations or any other computations that are based on regularly sampled continuous functions. The SPDP [23] compressor is developed by exhaustively searching for the best-performing candidate for list of datasets in a set of over nine million systematically generated compressors. Burtscher et al. propose FPC [17], which tries to predict the next value based on one of two predictors [37, 98]. It then computes the XOR of the predicted and the actual value and compresses leading zeros. Similarly, as Pelkonen et al. describe for their Gorilla time series database [90], data can also be compressed by simply calculating the XOR of consecutive values. The same compression approach is used for compressing floating-point values in InfluxDB. Based on the compression approaches in FPC and Gorilla, Bruno et al. propose TSXor [16], improving both compression rate and throughput. Chimp [71] is another improvement over the XOR approach used in Gorilla, optimizing the encoding based on empiric investigation on various real-world datasets. Cayoglu et al. [21] investigate a new data encoding by proposing an improved *shifted XOR* for calculating the difference between floating-point values. Instead of directly reducing data size, shuffle filters in HDF5 [121] and the approach taken by the blosc library [119] precondition data to be easier to compress by general-purpose compressors like zstd [24] or lz4 [77]. The work of Gómez-Brandón et al. specializes in industrial time series data, but is designed for integer data (so floating-point data first needs to be quantized to an appropriate integer format) [38].

Lossy Compression of Floating-Point Data

In addition to lossless compression, real-valued time series data can also often be compressed in a lossy fashion. This is particularly interesting for multidimensional numerical arrays representing continuous or smooth functions. Such data often results from scientific simulation experiments. Lindstrom et al. developed zfp [73], which is a lossy encoding for small blocks of multidimensional floating-point (single- or double-precision) data that exhibits spatial correlation. The encoding can also be used in a

lossless mode. Similarly, the lossless floating-point compressor fpzip [74] can be used in a lossy mode, but zfp performs better in the lossy case. The SZ lossy compression framework[4] is optimized for scientific simulation data [30, 31, 72, 109, 138]. Its latest version, SZx, improves performance by only relying on lightweight operations [136]. The framework also supports a variety of application-specific use cases.

Compressing Timestamps

There are several techniques for efficiently compressing timestamps. Many of them rely on the observation that differences (or deltas) between consecutive timestamps are smaller in value than the actual timestamp values [29]. Encoding smaller values makes it possible to use fewer bits, using one of many alternatives [131] like Variable Byte [29] or Simple-8b [7]. Delta-encodings often operate on blocks of values to avoid decoding too long sequences of values. Another optimization that is specifically useful for timestamps is delta-of-delta encoding, where only the difference to the last *delta* is stored. When the individual deltas are identical or at least very similar, this makes it possible to use few bits for storing timestamps. Lemire et al. [69] present an efficient method to pack the resulting bits of the delta-of-delta sequence. Since we expect timestamps to be very regular in industrial settings, our approach leverages a delta-of-delta method that has been shown to perform and compress well in many cases [69, 131].

2.4.3 Other Related Aspects

For some aspects that are not core to our work, there are also a variety of previous publications.

Approximate Query Processing

Approximate query processing (AQP) is a technique to provide approximate results to complex queries in a much faster way than obtaining exact results. AQP is useful when exact results are not required or when the exact computation is too expensive or time-consuming. For time series data, queries spanning large time ranges quickly get expensive because of the underlying number of raw samples.

AQP techniques can be broadly categorized into two categories, online aggregation and offline synopsis generation [3, 70]. Online aggregation techniques select samples online, at query time, and use these samples to answer queries. Offline synopsis generation generates some kind of reduced representation of the data based on prior knowledge (e.g., the expected types of the queries) and uses this synopsis to speed up queries. For example, instead of scanning an entire table, an AQP system might use a random sample of the data to provide a quick estimate of the result. This can reduce

[4] `https://szcompressor.org/`

the processing time for the query considerably, as well as the amount of data that needs to be processed and transferred.

Previous work for time series data successfully used offline synopsis generation, i.e., pre-computing windowed sketches [26] that can be used to answer queries. In MetricQ [52], a multi-resolution hierarchy of precomputed aggregations is used to speed up aggregate or visualization queries. This is very similar to our approach, but uses a different data storage model. According to the classification of Cormode et al. [26], this is a sketch-based approach. Chaudhuri et al. argue that it is important to offer an approximate query mode to users, in particular for interactive queries [22]. Another approach relies on identifying similarities between time series. This helps to avoid redundant storage of materialized, pre-computed views on data [91]. This focuses more on scaling aggregate queries along the dimensions of the dataset instead of the length of the time series.

Distributed and event stream storage systems

A lot of research effort has been spent on distributed streaming storage and processing systems. Some use cases and problems discussed in such works [97, 137] are similar or related to our use case. However, distributed stream processing systems—despite the name—are a different category of systems. These systems consider event streams and the challenges that come with it [14, 32, 47, 48, 68, 86]. This is fundamentally different from our usage scenario in both data type and data rate or regularity. An event is usually much more complex (it may arrive in the form of text or structured text) than a single sensor reading (one 4-byte floating-point value) and arrives with a much more irregular frequency.

3 HAQSE: System Overview

In this chapter, we provide a high-level overview of our sensor data processing and storage system. We call our system **HAQSE**[1], an acronym for **H**ierarchical **A**ggregation and **Q**uery **S**tor**E**. HAQSE fulfills the requirements and fills the gaps identified in the previous two chapters. As an introduction, Section 3.1 provides an intuitive explanation of the problems HAQSE solves and where it fits into the whole sensor data storage, processing and analysis pipeline. Next, in Section 3.2, based on the definitions from Section 2.1 we explain the interfaces for sensor stream ingestion and data queries and how HAQSE organizes the stored sensor stream hierarchy. In Section 3.3, we present the individual components that make up the system and explain how they interact with each other. More detailed explanations of the inner workings of these components follow in Chapters 4 to 6. In Section 3.4, we give an overview of our implementation and the software stack used in HAQSE.

3.1 Highlights of HAQSE

HAQSE is designed to address the problem of preprocessing and permanently storing large quantities of structured sensor data. As sketched in Figure 3.1, it is meant to be deployed as a storage component consuming multiple, independent, multivariate sensor streams, precomputing aggregations on data ingestion, permanently storing these streams, and providing the stored data to visualization and analysis clients. HAQSE specifically puts its focus on the three aspects stream ingestion input rate, stream query performance and storage efficiency for noisy sensor data. It is targeted at consuming continuous, long-running sensor streams that adhere to a fixed sensor schema (see Definition 2) and originate from a single source. It can ingest data faster than any existing time series management system we are aware of and incorporates an efficient way for compressing noisy floating-point values.

3.1.1 Main Concepts and Ideas in HAQSE

In the following paragraphs, we summarize the key design goals of HAQSE.

Efficient Stream Input HAQSE is built to consume multiple independent, multivariate, high cardinality, high stream rate sensor streams that adhere to a defined sensor

[1] HAQSE is pronounced like the Bavarian word “Haxe” (*/ˈhaksə/*).

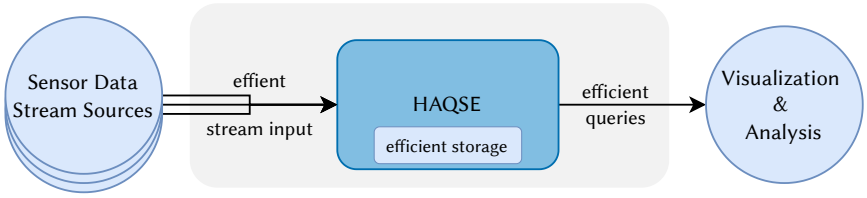


Figure 3.1 Overview: HAQSE is designed to consume multiple sensor data input streams, permanently and efficiently storing this data and providing it to analysis and visualization users.

schema. It comprises an efficient binary input protocol that is based on gRPC client streaming [42]. This is one of the key aspects for enabling high throughput on data ingestion. The input may consist of individual samples ψ, but it also allows the input stream to consist of sample batches $\psi_{i..j}$ (multiple timestamps and associated sensor values). As we show in Section 5.6, HAQSE enables ingestion of sensor source streams faster than any comparable system we are aware of. HAQSE efficiently computes windowed aggregations on the input sensor source stream, leading to several derived, aggregated sensor streams. Since these derived streams are all related, we can optimize the calculation of aggregation values based on the greatest common window size. In Section 7.4, we show that it is possible to deploy HAQSE in demanding real-world sensor processing scenarios, where it can effortlessly keep up with the generated stream rates.

Efficient Storage Storage efficiency is an important aspect for continuously generated sensor data. It is no question *that* (in most cases, old) parts of the data must be deleted at some point. The question is only *when*, and how large the time span covered with available data can be, given a certain storage size. Moreover, different data streams have different data properties, so there is no single compression approach that works best for all kinds of data sources. Consequently, it is crucial to include a modular compression approach into the core design of the system. For HAQSE, we chose an open-source storage format that comes with this property: Apache Parquet [116]. It supports different lightweight encoding schemes, but also supports various more sophisticated compression algorithms. There are encoding schemes for repetitive data values as well as for evenly spaced timestamps that are ubiquitous in time series data. We specifically design one combination of encoding and compression to target noisy floating-point sensor data. As we show in Section 4.4, this compression technique can reduce the compressed size of several publicly available data sets, all while increasing compression and decompression throughput.

Efficient Querying We provide an interface for efficiently retrieving range-based excerpts of the stored sensor streams. The query interface is based on Apache Arrow Flight [8]. This is again a gRPC service with existing implementations in several

programming languages, enabling easy adoption by clients. Queries return Apache Arrow record batches, which are easy and efficient to consume and process. Again, these record batches are binary encoded, avoiding any deserialization overheads on the client. To speed up querying large time windows, HAQSE supports retrieval of precomputed windowed aggregations. These precomputed aggregations can provide *min*, *max*, *mean* or other windowed summaries of the raw data[2]. One or multiple appropriate aggregation levels are selected for answering queries. This is useful, e.g., for visualizing months of sensor data sampled at dozens of kHz. In addition to that, machine learning models can be trained on precomputed mean values of sensor data[3]. HAQSE can be queried as data is streaming in, allowing for concurrent seamless aggregation and storage of data. We also propose a method for querying geographically distributed sensor data.

3.1.2 Non-Goals of HAQSE

To better understand what HAQSE is good at, we also discuss what HAQSE is not designed to be used for. These are deliberate design decisions that make it possible for HAQSE to fill the gaps where other systems cannot satisfy application requirements. We derived these system properties by analyzing the industrial application requirements from Section 2.3.

Multiple, concurrent sources for one stream Existing systems are able to receive data from multiple sources into one storage unit[4] concurrently (or are even optimized for such a workload). This is necessary especially for Internet of Things (IoT) use cases where hundreds or thousands of distributed sources write to the same database or table with one or multiple different identifying tags or labels. Many existing general-purpose systems [54, 93, 126] handle this case very well. HAQSE, on the other hand, is built to efficiently ingest from a *single* sensor source stream that produces a multivariate, high data rate stream of sensor samples. This avoids dealing with unifying the individual, concurrently arriving messages and many challenges that this implies, like ensuring serialized access to the final data structure.

Reordering items in the input stream This is related to the previous non-goal, but can still be considered a separate aspect. When data is received from multiple sources (but also in other cases), unsynchronized or drifting clocks as well as delays in stream transport make it necessary to reorder the arriving samples according to

[2] Aggregation functions need to be commutative and associative in order for HAQSE to be able to apply them on the sensor stream. This is explained in more detail in Section 5.3.

[3] Such models can be used to, e.g., detect slowly developing changes in the sensor stream and thus do not require the full stream resolution.

[4] One storage unit is, e.g., a table in a classical database management system. Depending on the specific system, the concept of such a storage unit is named differently.

their timestamp. Since HAQSE only has a single data source, we expect timestamps to arrive in strictly increasing order without backward jumps. Actually ensuring strictly increasing timestamps is a challenging task on the upstream sensor processing system, but other work has shown that there are viable approaches to achieve this [49, 60, 82].

Irregular or unknown sampling frequencies Precomputing windowed aggregations is not very meaningful for irregular or unknown sampling frequencies. We built HAQSE for input of sensor streams that are strictly increasing and evenly spaced at a known Δt. Streams in industrial settings satisfy these criteria since the sampling and processing are often done by special real-time-capable hardware that ensures these properties. These assumptions make data aggregation and retrieval more efficient. It is also possible to put irregularly sampled data into HAQSE, but precomputing aggregations for these cases is not as meaningful as for fully regular sampling (depending on the chosen aggregation window lengths).

Storage of non-numeric values General-purpose database systems are able to also store non-numeric data like variable length text or even more complex things like "events". HAQSE is not designed to do this. Instead, it focuses solely on storing information that can be represented by individual integer or floating-point numbers. Structured, more complex data types like frequency spectra are supported by splitting these types up into its individual components. Many components of HAQSE (e.g. the storage layer) would already be capable of storing other types of data, and it would be possible to support such use cases as well. However, this clearly is not the main focus of HAQSE.

Ensuring durability after every sample Many existing systems ensure durability after every sample—or, at least, every batch of samples contained in one "transaction"—sent to the respective system. This property requires frequent synchronizations to disk, essentially limiting the achievable data rate by the frequency these synchronizations can be performed on a given system. Since HAQSE targets continuous data stream applications, this explicit synchronization to disk is detrimental and excludes some application scenarios, so we chose to employ a relaxed durability strategy.

Flexible schemata Some systems support flexible schemas [54]. This is not the intended use case for HAQSE. The applications we envision all have a clearly defined schema. This makes it possible to preallocate memory for ingesting data based on the schema and thus optimize for receiving data that always follows this schema. We plan to integrate enhancements like the addition of new schema fields in a future version of HAQSE.

While these aspects seem to limit the utility of HAQSE, it helps to sharpen the focus of its field of use. Furthermore, focusing the system on certain aspects—as indicated above—helps create an efficient implementation.

3.2 Data Model and System Interfaces

Having established an understanding of the main parts and features of HAQSE, we now define its interfaces and data model. As shown in Figure 3.2, HAQSE offers two main interfaces: one for concurrently receiving multiple sensor data streams on the input side and another one for making this data accessible for querying. This section defines these two interfaces and the underlying data models.

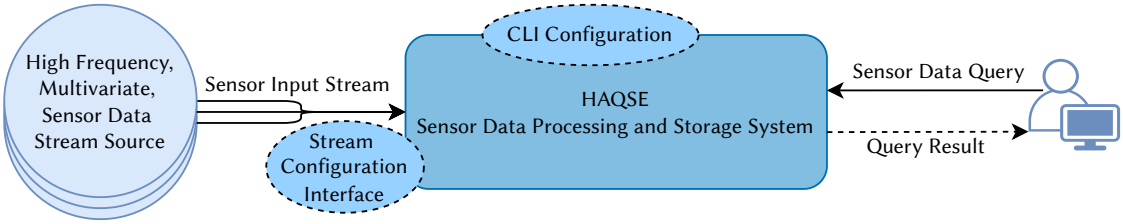


Figure 3.2 Sensor Data Processing and Storage System Interfaces

3.2.1 Sensor Data Stream Input

HAQSE is designed to consume multiple *sensor source streams*, as defined in Definition 4. Such a sensor source stream must first specify its sensor schema $\mathcal{S}$, i.e., types and names of the sensors, as well as the stream resolution Δt. Then, the actual stream of sensor samples ψ is received. The samples can either be sent individually or in sample batches. As we show in Section 5.5, using sample batches is necessary for most efficient data input.

3.2.2 Query Interface

One of the main goals of our system is to enable efficient retrieval of large timespans of sensor data for a variety of use cases like explorative data analysis or as input data for other analytic scenarios. For that reason, HAQSE supports range-based querying from a sensor stream hierarchy as defined in Definitions 10 and 11. In order to query the system, a JavaScript Object Notation [59] (JSON) encoded query request must be sent to HAQSE. As a response, the client receives a stream of sensor-ordered sample batches that contain the requested data in a resolution that best matches the request. Similar to the ingestion interface, the query interface yields binary encoded data. This way, serialization and deserialization overheads can be avoided.

3.2.3 Hierarchical Data Model

To have data ready for data queries, HAQSE uses different logical concepts for storing sensor streams. This is visualized in Figure 3.3. For each sensor source stream, HAQSE allocates one *stream bucket*. This means that at the top level, HAQSE's data model consists of multiple independent stream buckets (see Figure 3.3), each with its own schema. Each bucket, in turn, contains a stored sensor stream hierarchy Ψ^H—as defined formally in Section 2.1. First, every bucket contains a *stored sensor stream* that holds samples ψ of the original sensor source stream and in the original stream resolution Δt. In addition to that, there optionally can be multiple, derived streams that store aggregated versions of the original stream with a gradually increasing Δt. These aggregated streams also have a derived schema, where each field of the original stream has multiple, derived fields that represent the various aggregation functions precalculated on ingestion. This derived schema is indicated on the right part of Figure 3.3 (box titled *Aggregated Stream 1*). These aggregations are calculated on tumbling windows on the sensor data of the source stream.

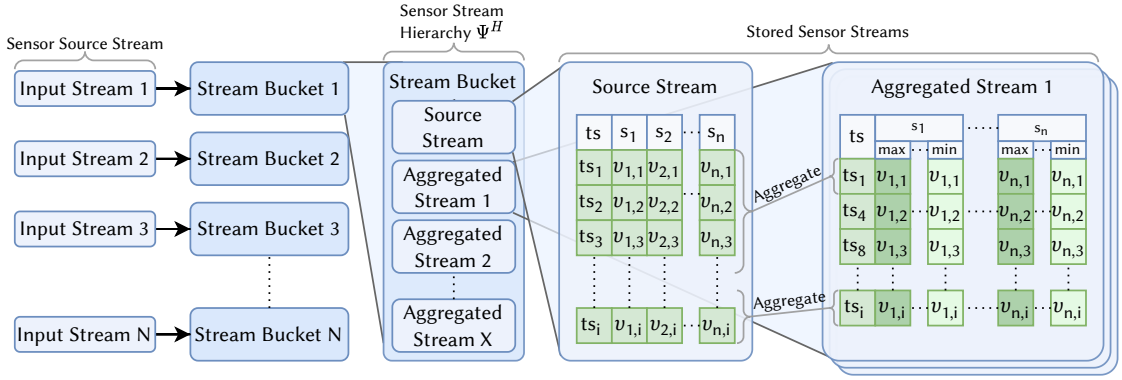


Figure 3.3 Overview of hierarchical data model in HAQSE.

The streams with different resolutions in one bucket make querying large time spans, e.g., for visualization, much faster. In HAQSE, all stored sensor streams of one sensor stream hierarchy are stored independently. This makes it possible to have different retention policies for each of the streams of the stream hierarchy with different resolutions.

3.3 System Architecture Overview

In this section, we explain the high-level software architecture of our system. We introduce the individual components and explain how they interact with each other. The requirements from Section 2.3 motivate design decisions that are distributed across

the various components of our software architecture. We will refer to these requirements when describing the details in the respective subsections. But first, we explain the main components on the basis of Figure 3.4. Having defined the data stream input and query interfaces (left and right in Figure 3.4) in Section 3.2, we now focus on the internals of our system:

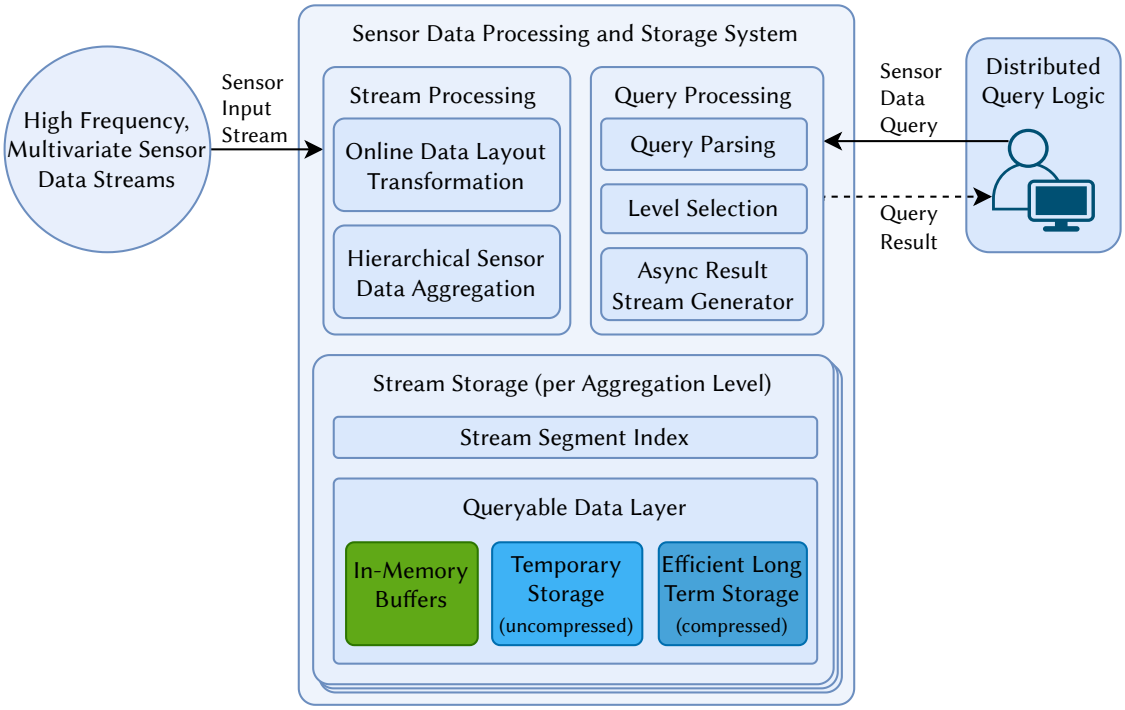


Figure 3.4 Sensor Data Processing and Storage System Components.

- Independent of any sensor stream interfaces, HAQSE is designed to efficiently store and manage hierarchical stored sensor streams Ψ^H. This is the responsibility of the *stream storage* component (bottom), which we further explain in Section 3.3.1.
- Before any data can be stored, however, we need to receive it on the input interface and transform and process it. This logic is contained in the *stream processing* component (top-left), which we explain in Section 3.3.2. This component produces a layout-transformed original data stream and several derived, aggregated sensor streams.
- Finally, client applications *retrieve* sensor data from our system by sending queries to the *query processing* component. Logic for retrieving sensor data from multiple, distributed instances of our system is located on the client. We give an overview of all query processing-related aspects in Section 3.3.3.

Both the stream and query processing components rely on and interact with features integrated in the stream storage component. For this reason, we start with an overview of this component first.

3.3.1 Stream Storage

The stream storage component represents a central part of our system, as all other components communicate with this component in some way, either directly or indirectly. It consists of the Stream Segment Index and the different data layer implementations that provide access to the actual stored sensor streams Ψ. The Stream Segment Index is an internal, tree-based, in-memory data structure that enables fast lookups for range-based queries. When dealing with large databases, it is a key component for addressing Requirement 10 (Query Speed). It manages stream segments, which represent contiguous, disjunct and ordered segments of stored sensor streams. Each segment is physically stored in one of the available data layer implementations, depending on the lifecycle of the segment (see Figure 3.5).

Figure 3.5 Lifecycle of segments of the stored sensor stream.

After sensor samples are received via the input interface, they are first put into an in-memory buffer. This optionally uses a double-buffering approach and helps to fulfill Requirement 5 (High-Bandwidth Stream Consumption). This buffer is required for two processing tasks: the layout transformation processing logic and the hierarchical batch aggregation functionality. Both of these are explained in more detail in Section 3.3.2. The size of the in-memory buffers are limited by the usable RAM. This is often a scarce resource that is shared with other processes that run on the same hardware. To reduce the required amount of RAM, our design includes a temporary storage area. This stores data in a format similar to the in-memory buffers, in uncompressed form. This temporary storage is needed for addressing Requirement 11 (Online/Offline Data Queries). The underlying files satisfy two important characteristics: First, they are readable in other contexts before being completed and closed by the writing process. This is important for handling queries requesting the data from recent time ranges that is stored in these files. Second, once terminated, these files contain a file footer that provides an index into the stored sensor stream contents of the file. In the next storage area, we add data compression to enable a compact representation on permanent storage, addressing Requirement 1 (Storage Efficiency for Floating-Point Data). We view compression as an integral part of our system, taking into account the special properties of sensor data.

In all of our design decisions, we had Requirement 10 (Query Speed) in mind, so this is reflected in several parts. First, the Stream Segment Index ensures fast access to the respective stream segments. Second, by exploiting the internal structure of the various storage formats, we limit the amount of data that needs to be read for range-based access. Third, we decide to use a binary data interface, avoiding costly serialization and deserialization. Lastly, we design a compression scheme that is fast to decompress, while still being able to achieve high compression ratios.

The details of all storage-related parts of our system, as well as our approach for compressing floating-point data, are presented in depth in Chapter 4. Aspects related to writing data to the respective storage areas, as well as updating the Stream Segment Index, are discussed in Chapter 5. Everything related to data queries, in particular reading the Stream Segment Index and reading from the various storage areas, are discussed in Chapter 6.

3.3.2 Stream Processing

As discussed in Section 3.2.1, sensor data is received in a time-ordered layout on the input interface. In contrast to this, as we explain in more detail in Chapter 4, data is stored in a sensor-ordered or columnar layout for further processing and long-term storage. Consequently, we need to transform the input data stream into a columnar layout. For this, we use the in-memory (optionally double-buffered) storage area mentioned in the previous subsection. Once the in-memory buffer is filled, its contents are written to the temporary storage area as one contiguous stream segment. Multiple such in-memory buffers are put into one such temporary file. Then, multiple temporary files are combined into a compacted file. We eventually end up with several such compacted files as the final destination of stored sensor streams. These compacted files employ data compression, ensuring an efficient representation on disk. Compaction is executed asynchronously in a background thread. Once a compacted file is completely written, the respective temporary files are deleted and the Stream Segment Index is updated accordingly.

Additionally, as discussed in Section 2.1, it is useful to calculate windowed aggregations for multiple different window sizes to speed up retrieval of large time spans. This results in a multi-resolution hierarchical data model, as sketched in Figures 2.2 and 3.3. In our system, we perform data aggregation on the in-memory, sensor-ordered batches of sensor data, achieving high computational efficiency. The resulting derived data streams are managed as independent sensor streams. Each such sensor stream (original and derived) is processed identically by the stream storage logic.

3.3.3 Query System

The query processing component implements retrieving data from HAQSE. As mentioned before, the query system accesses the Stream Segment Index, ensuring that

range-based queries can locate the requested parts of the data rapidly. The index accesses data from all the different storage areas, making it possible to also retrieve most recent sensor data.

For supporting distributed queries, the query system also provides information about upstream or downstream systems that can be queried from the query client (Requirement 13, Transparently Querying Multiple Sites). To exploit this information, clients run logic for dealing with distributed queries. In case a query cannot be fully satisfied[5], the distributed query logic forwards queries to other nodes potentially containing the data. We discuss the details of our query system in Chapter 6.

3.3.4 Combining Components in a Holistic System

The various methods used in our system are partly orthogonal and can be employed independently. As one example, the proposed efficient compression for floating-point sensor data is useful without the rest of the system. Yet, combining them in a holistic, integrated system creates synergies, ensuring that all discussed aspects work together in a seamless and efficient way. In Chapter 7, we show how HAQSE—as a complete system—can be integrated into and used as storage component in an existing industrial sensor processing system. Using the example of a production-grade combustion monitoring system, we also demonstrate that HAQSE is ready to be deployed in a production environment.

3.4 Implementation

This section describes the main technologies we use for implementing and evaluating our ideas in HAQSE.

- As programming language, we choose C++20 [58]. This provides a good trade-off between language features, control of runtime behavior (especially memory management) and there exists an extensive set of high-quality external libraries. We build HAQSE with g++ version 11.3.0 [111][6].

- As long-term storage file format, we rely on the Apache Parquet format. We also contributed to the specification during the course of our work and added new functionality to the Java and C++ reference implementations. At the time when we started implementing HAQSE, the best (feature-, stability- and performance-wise) implementation for writing and reading Apache Parquet files was available

[5] This is the case, e.g., if the requested time range for sensor data is not fully available anymore on a node or a query resolution is not available on a certain node.

[6] We do not expect significant performance variations when compiling HAQSE with a different compiler, e.g., Clang [92] or the Intel C++ Compiler [27].

in the Apache Arrow C++ implementation of Apache Parquet[7]. Timestamps are stored using the timestamp type of the respective format (Apache Arrow[8] and Apache Parquet[9]). In both cases, this means that timestamps are stored as the number of nanoseconds since UTC, encoded as 64 bit integer.

- Both our data input interface (see Section 3.4.1) and query interface (see Section 3.4.2) are based on gRPC. gRPC works on top of HTTP/2. It supports several ways of authentication and can encrypt traffic between client and server, fulfilling important security requirements. More specifically, HAQSE's query interface is based on Apache Arrow Flight, which basically is a wrapper around a gRPC service to send Apache Arrow columnar format data.

- We use the Apache Arrow columnar format internally for representing and processing sensor samples. The Apache Arrow columnar format is a perfect fit for representing columnar data that is loaded from Apache Parquet files. We also use components of the Apache Arrow ecosystem, e.g., the Apache Arrow Flight framework, for our query interface. This ensures that data is efficiently queryable since there is almost no serialization overhead. A wide range of other libraries and frameworks also adopt parts of the Apache Arrow framework. This also makes integrating further functionality based on the Apache Arrow ecosystem easy.

- We use different ways of concurrently executing various processing tasks in parallel. All external triggering happens via gRPC: receiving data is done directly through a gRPC service (see below), querying data from HAQSE is done through Apache Arrow Flight, which is just a wrapper around a gRPC service. In both cases, we rely on gRPC's synchronous request processing. This internally uses a thread pool to handle requests, creating new threads if required. Internal asynchronous processing tasks like compaction or asynchronous reading data for queries are also performed in thread pools. We use one thread pool for compaction and one for asynchronous reading data for queries.

We package HAQSE in a Docker image that is based on Ubuntu 22.04.

3.4.1 Input Interface Definition

From a client's perspective, one of the most interesting aspects is how to get sensor samples into HAQSE. As mentioned before, the ingestion interface is realized via a gRPC protocol. The gRPC service definition is shown in the following Protocol Buffers listing:

[7] The C++ implementation of Apache Parquet is located in the Apache Arrow project repository at the time of writing.

[8] https://github.com/apache/arrow/blob/apache-arrow-11.0.0/format/Schema.fbs#L242

[9] https://github.com/apache/parquet-format/blob/apache-parquet-format-2.9.0/LogicalTypes.md#timestamp

```
service ServerInput {
  rpc StreamSensorData(stream SensorStreamElement) returns (StreamStatus) {}
}
```

It shows that the client is required to send a stream of `SensorStreamElement` messages. As response, after the client has stopped streaming messages to HAQSE, it receives a `StreamStatus` message, indicating whether the call was successful. Messages of type `SensorStreamElement` are defined as follows:

```
message SensorStreamElement {
  oneof StreamDefOrContent {
    // impl needs to enforce that first message is always the StreamDefinition
    StreamDefinition stream_definition = 1;
    // all messages after the first one need to be SensorData messages
    SingleSampleData single_sample = 2;
    SampleBatchData batch_samples = 3;
  }
}
```

This shows that a `SensorStreamElement` must contain one of three message types: a `StreamDefinition`, a `SingleSampleData` or a `SampleBatchData`. When starting a stream, the client first needs to send a `StreamDefinition` message, containing a definition of the sensor stream that follows. The details of this `StreamDefinition` message are presented in the next listing, showing the excerpt from the interface protocol definition:

```
enum FieldType {
  Bool = 0;
  Int8 = 1;
  Int16 = 2;
  Int32 = 3;
  Int64 = 4;
  UInt8 = 5;
  UInt16 = 6;
  UInt32 = 7;
  UInt64 = 8;
  Float32 = 9;
  Float64 = 10;
  TimeStampNanos = 11; // nanoseconds since UTC 1970-01-01 00:00, encoded as int64
}
message SchemaField {
  string name = 1;
  FieldType type = 2;
}
message StreamSchema {
  repeated SchemaField fields = 1;
}
message StreamDefinition {
  string stream_id = 1;
  StreamSchema schema = 2;
```

```
  uint64 time_resolution = 3;
  // ...
}
```

When starting the stream with the `StreamDefinition` message, clients can also specify stream configuration parameters. To keep this section concise, we exclude these parameters from the listing and do not elaborate on them here. Details of these stream configuration parameters can be found in Appendix A.2.

The actual sensor data that should be sent to HAQSE then needs to adhere to the following protocol definition:

```
message SingleSampleData {
  int64 timestamp = 1; // nanoseconds since UTC 1970-01-01 00:00, encoded as int64
  bytes stream_content = 2; // encoded measurement values, same order as in schema
}
message SampleBatchData {
  uint64 sample_count = 1;
  bytes timestamps = 2; // nanoseconds since UTC, encoded as a sequence of int64
  bytes values = 3;
}
```

The actual sensor samples ψ are sent via either a `SingleSampleData` or a `SampleBatchData` message. For a `SingleSampleData` message, the timestamp is encoded as a single `int64`. The actual sensor measurement values v are encoded using bytes that conform to a little-endian C-compatible binary representation. For this, all these binary representations are simply concatenated in the `stream_content` attribute in the order as the sensors σ appear in the schema.

For `SampleBatchData` messages, the number of samples are explicitly encoded as **`uint64`** `sample_count`. This message type provides flexibility regarding the encoding of timestamps and values. For that reason, we simply use **`bytes`** as type for both. Currently, the only supported encoding for timestamps is a series of `sample_count` **`int64`** values. The actual values v are encoded in a column-oriented, i.e., sensor-ordered, fashion, again conforming to a little-endian C-compatible binary representation. This means that the `values` attribute contains, in order, all values v that belong to the first sensor, then all v of the second sensor, etc. Explicitly encoding the number of samples in the `SampleBatchData` message enables future improvements to the encoding of the `values` or `timstamps` attributes of the message. As one example, the timestamps may be delta encoded, reducing the message size and saving bandwidth.

When all sensor data is sent, the gRPC connection can be closed and the success status is returned to the client.

3.4.2 Query Interface Definition

HAQSE requires clients to use a simple custom query definition for retrieving data from the system. These queries need to be specified using a JSON object that follows the structure of the example shown in the following listing:

```
{
  "signals": ["sensor0", "sensor4"],
  "stream-id": "my-stream-name",
  "start": 1509969600000000000,
  "end":   1687950671000000000,
  "max-points": 1000,
}
```

This example query requests data for the two sensors named *sensor0* and *sensor4* of the stream named *my-stream-name*. Only data matching the queried time range between `start` and end will be returned to the client. Additionally, the client requested a maximum number of 1000 points. This means that in this particular case, the client does not require more than 1000 samples for its processing task, e.g., visualization. On the server side in HAQSE, this `max-points` attribute is used for selecting an appropriate aggregation level from the hierarchy, if available. Otherwise, a coarser aggregation level is returned, or the query returns an empty result set. In both cases, the metadata part of the response indicates that the query could not be fully satisfied.

This JSON object needs to be put into an Apache Arrow Flight `Ticket` as a UTF-8-encoded byte sequence. The created ticket can be used to call the `DoGet` method of the Apache Arrow Flight service:

```
rpc DoGet(Ticket) returns (stream FlightData) {}
```

As a result, HAQSE will return a stream of `FlightData` messages. Each such message contains an Apache Arrow record batch with data matching the query definition. This record batch consists of one or multiple columns per requested sensor, depending on whether the result uses aggregated data (multiple columns, one for each queried aggregation) or not (one column containing the original stream values). Additionally, it contains one timestamp column, encoded as a nanosecond `arrow::Type::TIMESTAMP`[10]. Details of the Apache Arrow Flight protocol are defined in the respective protocol specification [8]. In addition to that, the response contains metadata information about the query result. Appendix A.3 shows a code snippet for a basic query client in Python.

[10] An Apache Arrow `TIMESTAMP` is internally represented by a 64 bit integer, see `https://github.com/apache/arrow/blob/apache-arrow-11.0.0/format/Schema.fbs#L242`

4 Efficient Storage of Industrial Sensor Data

Industrial installations continuously generate sensor data and analysis results at very high rates. As described in Section 2.2, the generated data is used in a wide range of different application scenarios. One large category of use cases is the analysis of large amounts of historical sensor data. Analysts and engineers require efficient access to measurement values of huge time spans of historical sensor data. In this chapter, we focus on two main aspects related to this: storage model and compression.

On the one hand, deciding on a storage model is a crucial point in our design of HAQSE. It has functional and performance implications on all other parts of the system. On the other hand, an efficient compression approach is fundamental for achieving good storage efficiency. While saving storage space and transmission bandwidth is important, that is not the only aspect to consider. Compressing data must also be fast enough for the desired data rates, and decompressing data should not slow down (or even speed up, if possible) analysis tasks.

In Section 4.1, we first discuss the storage model. For this, we discuss how HAQSE is designed to consume sensor data streams and how data is supposed to be deleted again (Section 4.1.1). Then, in Section 4.1.2, we show possible approaches of how to map the sensor data stream concept from Section 3.2 to actual physical data layouts. In Section 4.2, we present a compression scheme for floating-point encoded sensor data. After that, in Section 4.3, we explain how our approach maps to the Apache Parquet file format. Finally, in Section 4.4, we evaluate our solution, showing that our system not only improves compression efficiency for a typical dataset, but also increases compression and decompression throughput compared to prior art.

Publication Information
The approach described in this chapter, in particular compressing floating-point data, has previously been published in [66]. The author is the principal author, with conceptual contributions by all other authors. The third author (Martin Radev) significantly contributed to the design, analysis and especially implementation in Apache Parquet of the developed Byte Stream Split approach.

4.1 Storage Model

In this section, we discuss the storage model of HAQSE. This comprises the data write and read pattern our system is built for as well as the physical organization of data on disk. For this, we have to take into consideration Requirements 1, 10 and 5, i.e., how data is typically written, how it is accessed, and that we require a compact representation on disk. So first, we quickly discuss what types of operations we want to support for adding content to and removing it from the database. After that, we describe the design of the physical data organization in HAQSE.

4.1.1 Adding and Deleting Data

HAQSE is designed to consume continuous streams of sensor data with increasing timestamps, each coming from a single source. Thus, we design and optimize HAQSE for an append-only workload, similar to other time series management systems [54, 93, 126]. This means that data can only be appended to an existing database. All data that has been written to the database will never be updated or changed again. Once measured and persisted in the database, there is no reason why existing sensor measurements should ever be altered, especially in an industrial context. As data is flowing in, all these data stream values are written to independent, time-partitioned segments (represented by, e.g., one file per segment). By using this strategy, a single segment represents a certain time span of the sensor stream (see Figure 4.1).

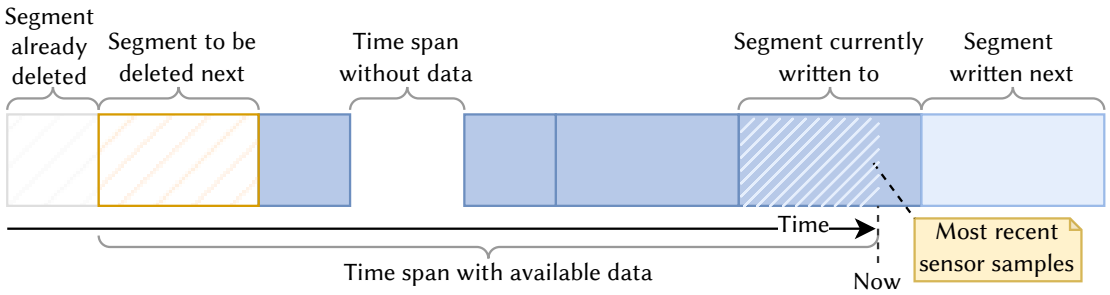


Figure 4.1 Sensor data lifecycle. Sensor data is continuously flowing in, written to time-partitioned files in an append-only manner. By using this strategy, each segment can be deleted in one block, removing all sensor data contained in the respective segments.

Since sensor data streams for monitoring industrial assets create vast amounts of data continuously, at some point, data will need to be deleted according to a certain retention policy. For HAQSE, we use a strategy similar to existing time series databases, where whole segments or partitions of the existing sensor data stream are deleted in one operation. For example, QuestDB uses *data partitions*[1], InflxuDB uses *shards*[2] and

[1] `https://questdb.io/docs/operations/data-retention/`
[2] `https://docs.influxdata.com/influxdb/v2.5/reference/internals/shards/`

TimescaleDB uses the concept of *hypertables* that consists of multiple child tables, so-called *chunks*[3]. Dropping data is achieved by simply dropping complete segments, removing the respective time span as contained by the segment from the database. In practice, for automated operation, this will often be the oldest time span, such that a contiguous part of the most recent data is available. This is, however, not a necessity; it would also be quite conceivable to design more complex retention strategies where time segments to keep longer are marked in some way.

While this model seems relatively restrictive, it is sufficient for many real-world industrial applications; one such example is presented in Section 7.4. Specializing for the introduced workload makes certain optimizations possible. It would be feasible to allow other data interactions, e.g., updating or deleting individual sensor measurements. However, these would be very costly operations. In the scope of this thesis, we do not address any other potential data interactions than the ones described above. HAQSE also does not currently allow other interactions.

4.1.2 Data Layout

As explained in Section 3.2, the input to our system is a multivariate, time-ordered (and potentially batched) stream of sensor data. For permanent storage, there are different ways to persist such a sensor data stream. In Figure 4.2, we sketch four possible layout alternatives. Listed with their advantages and disadvantages, they are:

(a) Writing the data stream in the same order as it arrives at the interface, sample by sample: *time-ordered* or *row-oriented*. This has the advantage that the stream can be processed and written to permanent storage *as is*, on a per-sample basis (or per batch). This means no in-memory buffering is necessary, so this approach has a small memory footprint. The processing overhead is also minimal.

The major disadvantage is that this layout is suboptimal for many analytics use cases. First, since access happens at the granularity of operating system pages (at least), more data is read than what is actually needed. Consequently, a lot of unused parts of the data stream must be read from persistent storage, especially when only a small subset of sensors is required. Related to this: when the sensor data stream is stored in a compressed fashion, the *complete* stream must be decompressed to read data, even if data for only a single sensor is requested. This is how the state-of-the-art system we integrate HAQSE into currently stores data (cf. Section 7.2.2).

(b) Splitting up the sensor stream into completely separate streams: one stream of timestamp information and one stream per sensor in the source stream. This has the main property that the individual streams are fully decoupled. Thus, for

[3] `https://docs.timescale.com/timescaledb/latest/how-to-guides/hypertables/`

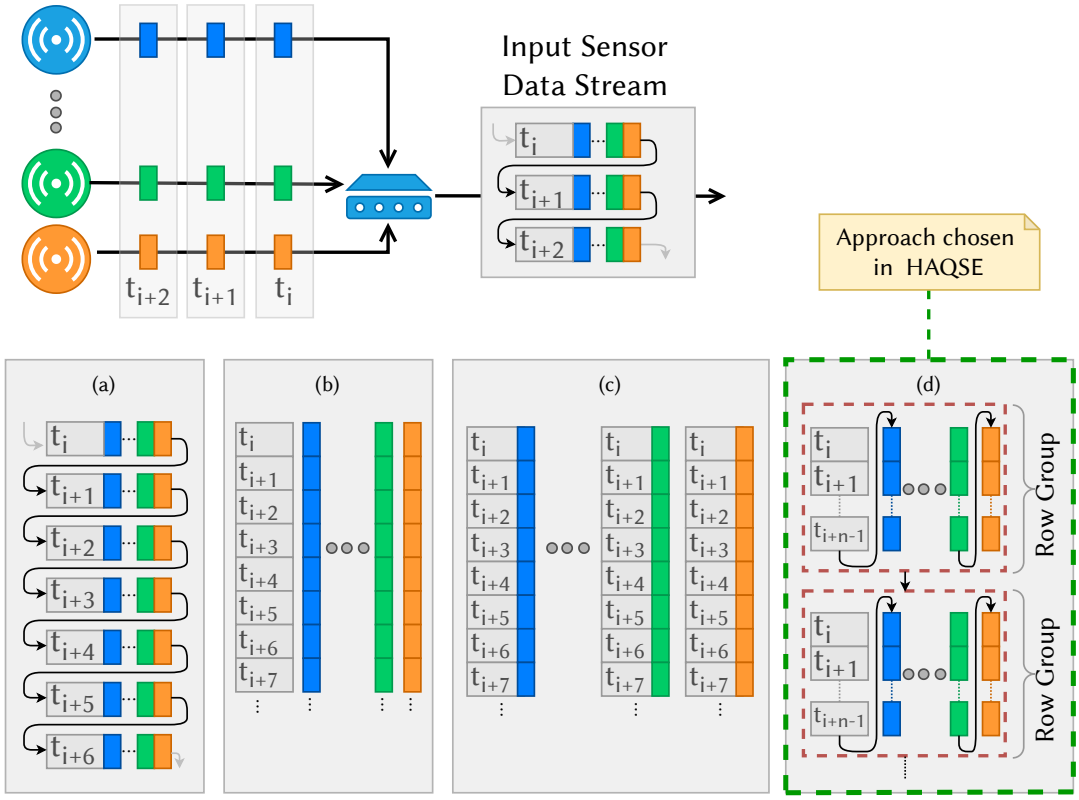


Figure 4.2 Illustration of the different possibilities for physical organization of sensor data on a storage medium. The black arrows in boxes (a) and (d) indicate how data is organized linearly in memory.

retrieving data, only the streams of interest must be read and possibly decompressed. However, this also means that there must be an additional mechanism to associate the correct timestamp information with the actual measurement values. Furthermore, when writing data for large schemas, there is a large number of open (file) streams to manage. This can quickly become a burden for the operating system, especially on resource-constrained systems. QuestDB [95] uses this layout, but does not compress data in the individual files.

(c) Splitting up the sensor stream into separate timestamp-measurement pairs and persisting each such stream independently, duplicating the timestamp information. This is similar to the previous alternative but duplicates the timestamp information for each sensor. This has the advantage that individual streams are fully self-contained and can be deleted (also partly) completely independently. Duplicating the timestamp column, however, increases the required storage space (even though this can be compensated partly by using an efficient encoding for timestamps). Additionally, this approach also requires a large number of open

streams during writing. MetricQ [52] and InfluxDB [54] employ this storage layout.

(d) Writing the data stream as sequence of *sensor-ordered* batches, also called *columnar* row groups. A *row group*[4] is a set of rows that forms one contiguous storage unit, as illustrated by Figure 4.2 (d). For in-memory structures, we also use the term *record batch* as defined by the Apache Arrow project. This approach requires a layout transformation step when ingesting time-ordered data streams, buffering data before one row group can be fully populated. It has several advantages regarding storage efficiency since (possibly similar) measurements from a single sensor are compressed and stored consecutively. The advantage in comparison with the previous two approaches is that there is only one file stream that needs to be managed on the operating system level. This is beneficial for both writing and reading, especially for high schema cardinalities. Additionally, timestamp information is only stored once, and the association between timestamps and measurement values is established via the file structure internally.

Columnar systems have been shown to perform better for analytic use cases in general [1]. For long-term persistent storage in our system, we decide to use alternative (d), a layout consisting of sensor-ordered, columnar record batches or row groups. This layout allows us to fulfill a number of requirements. First, it enables columnar compression, compressing timestamp and measurement data for each sensor individually. Compression happens on a column-by-column basis inside one row group. This makes it possible to achieve very good compression rates since measurement values originating from the same sensor are compressed as one unit. Second, it allows selectively decompressing only the sensor columns that are to be retrieved for a certain use. Third, we avoid duplicating timestamp information, saving storage space. This layout also maintains the structure of the input data stream, making time series joins of sensors in one particular schema, e.g., for sensor correlation, a trivial operation. Finally, when only a certain subset of row groups contains relevant information (e.g., because the query limits the time range accordingly), only the data in the respective row groups need to be decompressed. Details of how this can be achieved are discussed in Section 4.3.1.

In addition to the actual data storage layout, another crucial aspect is storage of meta information. This includes essential information like the stream schema, i.e., which sensors are contained in the stream, how they are named or which data type is used to represent values of each sensor. In addition to that, there might be index structures containing information on how to quickly access certain parts of the data or statistical information, useful for pruning candidates in range-based queries. For time series data, this might be information about the timespans contained in a certain

[4] We borrow the term *row group* from the Apache Parquet terminology. An Apache Parquet file is organized in a sequence of row groups. More details are discussed in Section 4.3.1.

segment, pointers to the respective parts in the storage files or statistical information about which range of values is contained in each segment. Furthermore, meta information includes properties of the storage format itself, i.e., what type of compression is used, the number of contained values, block sizes, etc. This information alone is not enough in many cases. Thus, it is augmented with application-dependent metadata like measurement units, or more detailed descriptions of the sensors. Assuming some type of file-based storage for the actual sensor streams data, there are several—not mutually exclusive—possibilities to store this kind of metadata:

(a) *Separately.* Metadata is stored separately from the actual data files, e.g., in additional files or an external database. The main advantage of this alternative is that data only needs to be stored once for a particular stream. If these separately stored files contain information essential for reading data, this approach requires some kind of management to keep actual data files and the meta information synchronized. Additionally, all data *must* adhere to the shared information; this makes unforeseen changes to metadata more challenging engineering-wise.

(b) *Header.* Meta information is stored at the top of the file, before all actual measurement values. Since the schema is known before the actual stream contents need to be stored, storing meta information at the top in a header is easy to implement. It just requires this information to be prepended to every file. Headers cannot contain structural information when the file is being streamed to since not all content—and thus, not all structure—is known at the beginning of a file.

(c) *Interleaved (repeated headers).* Similar to the above, but having repeated headers or meta information interleaved with actual data values increases flexibility. It allows putting different parts in a file stream with unpredictable order. This allows streaming data to files and also allows reading from these files before they are fully written (by iterating over the contents).

(d) *Footer.* Meta information is written at the end of the file. This is mainly useful for writing structural information when all content and structure is known; all this data can be collected while writing the file. This structural information can contain statistics or indices, potentially speeding up data retrieval. A main disadvantage of having essential information in a file footer is that it needs to be fully written before the file can be read again by independent processes.

As we show in Section 6.3, structural and statistical information in files is crucial for fast queries on large datasets. We thus employ a storage format that provides such information in its footer. Since we also want to be able to read data from unfinished files, we also use a temporary format that uses header information and interleaved information. In its current version, HAQSE only stores very basic stream information separately: stream names and resolutions are stored in the file system as folder names. All other information, in particular schema information, is stored redundantly in each

file containing data. This makes it possible to take care of aspects like schema evolution by creating a new file once a new schema is required.

We postpone the discussion of how to transform the incoming sensor stream data to a columnar layout until Section 5.2. So, for the remainder of this section, we assume data layout to be batches of sensor-ordered row groups.

4.2 Two-Step Floating-Point Compression

As explained previously, in many use cases, it is desirable to have a database spanning large time spans to retrieve data from. However, in industrial contexts where data is generated continuously and autonomously, the covered time range will almost always be limited by the available amount of physical storage space[5]. In addition to that, some parts of the data are transferred to other systems either on a regular basis or as reaction to special events. In any case, it is of paramount importance to use the available storage space and network bandwidth efficiently. One way to achieve this is to compress data on permanent storage or before transmission by using lossless or lossy compression schemes. This directly impacts the required storage space and network bandwidth for data transmission. However, it generally also increases the computational cost incurred by the compression when writing data. Likewise, the same holds true for reading when data needs to be decompressed first. Moreover, compression is a highly data-dependent step in the processing pipeline. For these reasons, it is essential to consider data compression from the beginning and integrate it tightly into the sensor data processing system.

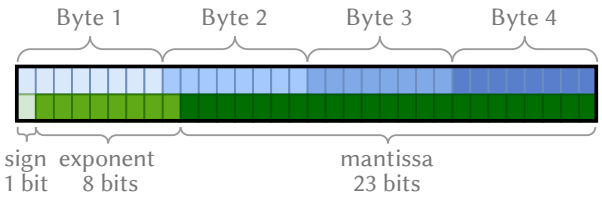


Figure 4.3 Layout of an IEEE 754 single-precision floating-point number in memory.

As analyzed in Section 2.2.3, most sensor data is generated by sensors measuring continuous physical phenomena. We design our approach to work with the ubiquitous *IEEE Standard for Binary Floating-Point Arithmetic* [50] representation for real numbers since this is *the* standard representation used in almost all production analytics code.

[5]While the amount of data storage space in cloud systems is virtually unlimited, all long-term storage comes at a certain cost. Since operation must also be tractable economically, the storage space in the cloud is limited by financial considerations rather than by technical limitations. However, eventually, storage space is always limited by one or the other factor.

Figure 4.3 illustrates how the format uses the 32 bits of a single-precision floating-point number. In the remainder of this section, we therefore assume this binary representation of floating-point numbers. However, there is no direct dependency on this representation and our approach should also work for other, recent alternatives in representing and processing real numbers, e.g., posits [44].

We base the compression component of our system on a proven [119, 121] compression approach, consisting of 1) a fast reversible reorganization step and 2) the usage of a production-grade general-purpose compressor. This approach reduces the implementation complexity, but it is very well suited for noisy sensor data that often covers only a relatively low dynamic range when compared to the full range offered by the binary floating-point representation. The base scheme is similar to the approach taken by the Blosc library [119] or the shuffle filter in HDF5 [121].

The rationale behind our approach is to prepare the stream of floating-point numbers so that the general-purpose compressor is much more efficient on some parts of the reorganized data stream—both in terms of compression effectiveness and speed. For this, we take a fixed-length window of data for a single series of values and reorganize the floating-point binary representations of this series. Our reorganization scheme is called Byte Stream Split: it splits a contiguous block of a stream of values into multiple, more "similar" streams of bytes, as visualized in Figure 4.4. These intermediate byte streams are then concatenated and compressed using a general-purpose compressor, e.g., zstd.

In the specific example in Figure 4.4, the first bytes of the 4-byte IEEE 754 representation of three very similar floating-point values are identical (0x40). The second bytes are very similar (0x15 and 0x13). In contrast to that, for the other two bytes (representing the least significant bits in the mantissa), there is no easy-to-recognize pattern. Reorganizing the stream as indicated in the example, the first half of the concatenated stream contains mostly very similar values, which is efficient and fast to compress. The second half remains hard to compress since it contains the mantissa's noisy (high-entropy) part.

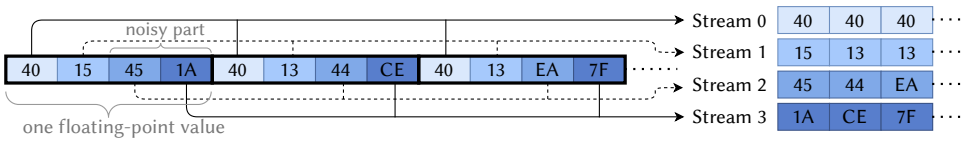


Figure 4.4 Single-precision floating-point example for Byte Stream Split. The example shows how the beginning of a simple stream of floating-point values is transformed into four separate byte streams. The values are: 2.3323427 = 0x4015451A, 2.3010745 = 0x401344CE and 2.3111875 = 0x4013EA7F.

For reconstruction, we just decompress the compressed stream and combine the bytes from the resulting byte streams to reconstruct the original floating-point stream.

This reconstruction just requires information about the block size as well as the data type, or, more precisely, the size of one value in bytes, of the stream.

An alternative and similar splitting approach we investigated is *component split*. For this, values are not split on byte boundaries, but rather on the component boundaries of the floating-point representation. For single-precision values, this would result in a stream with only the sign bits, one stream with only the exponents, and three streams with the mantissa information (one stream with seven bits, two with eight bits). While splitting on component boundaries intuitively appears to be promising, there are a couple of disadvantages of this approach. First, the sequence containing the sign bits and the one containing the 7-bit mantissa part both require padding for block sizes that are not a multiple of eight. Second, this approach is harder to map efficiently to hardware since it requires non-aligned bit accesses. Third, preliminary experiments also showed that the compression efficiency is not better than the Byte Stream Split approach, as visualized in Figure 4.5. Because of these reasons, we did not further investigate the component split approach in more detail.

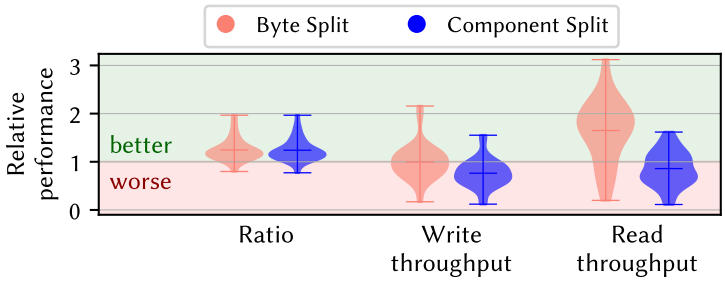


Figure 4.5 Preliminary comparison of the two different floating-point splitting approaches for a range of floating-point datasets (see Section 4.4.5). Each dataset is represented by one dot, the mean of all dots of one group is represented by a horizontal bar. For every dataset, we measure the relative improvement when compressing the transformed stream compared to when compressing the original stream. While compression ratio is improved for both approaches equally well, Byte Stream Split performs better on average regarding both write and read throughput.

In summary, Byte Stream Split is a conceptually extremely simple, yet effective, approach when combined with a standard compression algorithm (as we show in Section 4.4). The basic implementation just requires a simple nested loop structure that can be implemented in few lines of code in any programming language. This simplicity was one of the key arguments that convinced the Apache Parquet community to include the approach as the first—and thus far, only—encoding applicable for floating-point data. Nonetheless, the approach can also be mapped to highly efficient machine code, as we explain in the next section.

4.3 Implementation in Apache Parquet

Apache Parquet [129] is a widely used columnar storage format in the data analytics community. Apache Parquet is a column-oriented storage format integrated into popular computing and data analytics frameworks such as Apache Spark, Apache Arrow and Pandas, and is suitable for efficient representation of tabular data.

We implement the two-step compression approach in a fully working system based on Apache Parquet. Since our approach extends the specification of the format, we first sketch aspects of the format that are required to understand our implementation. Then, we explain how the implementation of the proposed reorganization step can be mapped efficiently to vectorized instruction sets of modern CPU-architectures.

4.3.1 Apache Parquet: Format Details

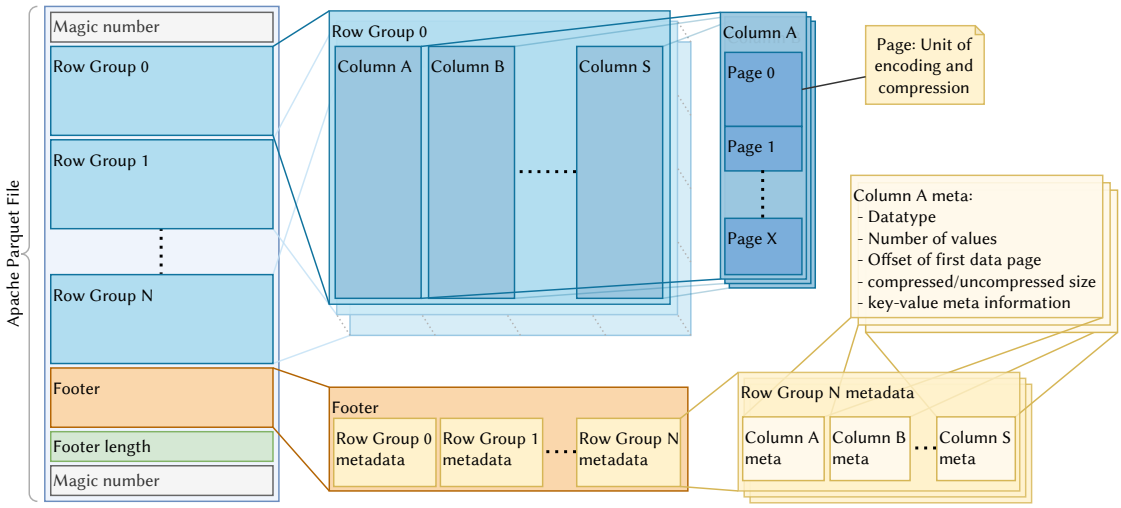


Figure 4.6 Individual components and construction of the Apache Parquet file format, adapted from the Apache Parquet file format documentation [116].

From a logical, high-level perspective, an Apache Parquet file contains a list of rows, each conforming to a schema of fields. Fields are possibly nested, making it possible to represent more complex and hierarchical objects. This allows the format to support storage of complex hierarchical object structures. Fields can be *optional*, in which case the value for a particular row can remain undefined. Furthermore, field values can contain multiple values of the defined type if the field is *repeated*. If a field is neither optional nor repeated, it is required: there must be exactly one value for each row in the schema. If all fields are required, an Apache Parquet file represents a simple table structure with no empty table cells. So far, we only use required fields in HAQSE. Having the ability to leave individual cells in the table empty (having optional fields) is an interesting option for future extensions to HAQSE, though.

What is more interesting to our implementation of efficiently storing floating-point sensor data is the physical layout of Apache Parquet (see Figure 4.6). The logical list of rows is split up into *row groups*, implementing the desired storage format from Section 4.1. A row group, in turn, consists of a set of *column chunks*. Each column chunk holds all data for a particular column inside a certain row group. Again, a column chunk is further divided into several *data pages*. A data page contains the actual encoded data values, metadata information, and serves as the smallest unit of compression. In Apache Parquet, there are two complementary steps for transforming a CPU-usable, in-memory representation of data values to the persistent representation on disk (and vice versa): data encoding and data page compression. In the simplest case, data encoding is just the *plain* mapping and uses the in-memory binary representation. For certain types of data, encodings can also work as lightweight compression methods like run-length-encoding or dictionary compression [115]. Data page compression just takes the data stream resulting from the encoding step and applies one of several supported standard compression schemes such as zstd [24], lz4 [77] and others [118]. Data pages can be decompressed individually, allowing efficient projection to columns and selective queries.

All structural information of an Apache Parquet file is stored in its footer. This footer contains the schema information and offsets of the row groups in the file. Beyond that, the footer may contain statistical information like minimum and maximum for each column in every row group. This information can speed up selecting matching row groups for range-based queries.

The specification of Apache Parquet is steadily extended and evolving. While there are many more interesting aspects in the format, the features sketched here already make it an appropriate choice for HAQSE.

4.3.2 Two-Step Compression

For the implementation of our two-step compression scheme, we use the encoding and compression functionalities of Apache Parquet. Our implementation is based on the C++ implementation of Apache Parquet, which is part of the Apache Arrow project [112]. With that, we rely on the general-purpose compressors integrated into Apache Arrow, greatly simplifying our implementation.

We implement Byte Stream Split as a new *encoding* in Apache Parquet. In contrast to other encodings in Apache Parquet, this encoding does not reduce the data size on its own, but prepares the input for compression. The implementation consists of an *encoder* for writing and a *decoder* for reading. Again, we use the existing infrastructure in Apache Parquet to store the size of blocks—in this case, the *page size*—necessary for decoding data.

To ensure that the additional Byte Stream Split step is implemented efficiently, we evaluate three alternative implementations: a simple implementation using two nested loops and two manually vectorized implementations using Streaming SIMD Exten-

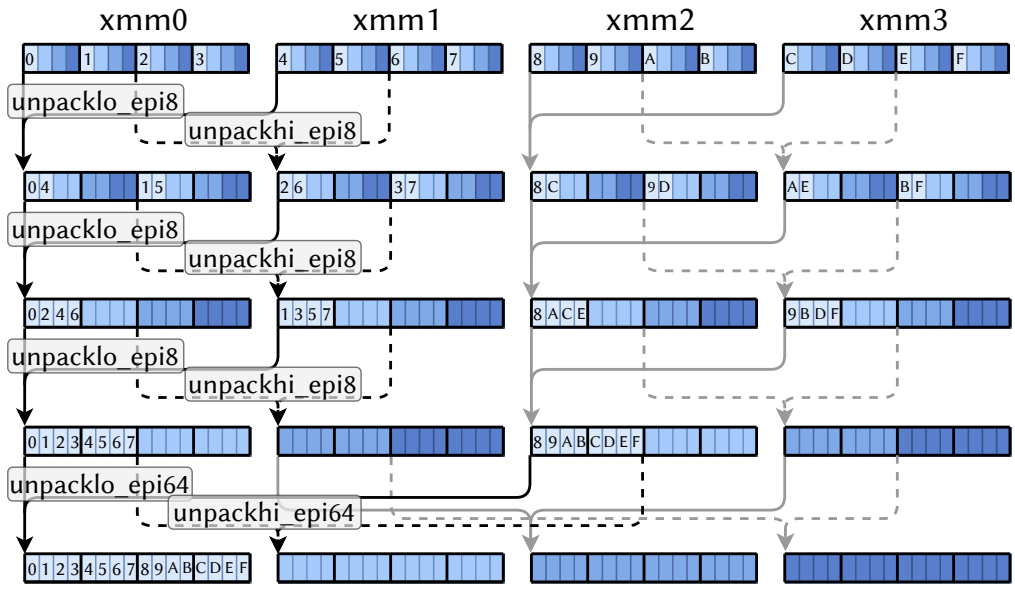


Figure 4.7 Single-precision floating-point Byte Stream Split encoder transformation sequence using SSE unpack intrinsics. This processes 16 single-precision floating-point values simultaneously. The grayed-out arrows represent the same operations in the respective stages, and the annotations are left out for clarity.

sions [57] (SSE) and Advanced Vector Extensions [57] (AVX) instruction sets. The simple implementation loops over the input, splits each floating-point value into its constituent bytes and scatters them to the corresponding streams (cf. Figure 4.4). While this simple implementation serves as a baseline that can be used on any system, we determined that the compiler does not efficiently auto-vectorize all code paths. One example is the encode path for double-precision data, which is significantly slower than our hand-tuned version. The SSE and AVX implementations use a combination of `shuffle`, unpack and `permute` intrinsics with different lane and stride sizes.

As an example, we describe the single-precision version utilizing SSE-intrinsics, using a sequence of dependent stages. This transformation sequence requires four stages for the encode-transformations (see Figure 4.7) and two stages for the decode-transformations (see Figure 4.8). It processes 16 consecutive single-precision values simultaneously, split up into four 128-bit SSE registers. After loading data into the registers, the encoder applies four steps using unpack intrinsics to distribute the bytes. The encoding transformation finishes after four stages. At that point, each SSE register contains the bytes of 16 floating-point values for each corresponding output stream. Then, the encoder stores the register contents in the intermediate buffers. The decoder works analogously but only requires two stages (see Figure 4.8).

We evaluate the different implementations in Section 4.4.3.

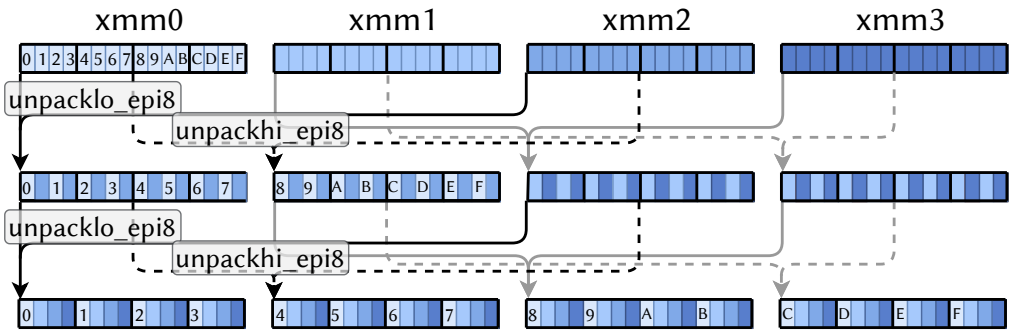


Figure 4.8 Single-precision floating-point Byte Stream Split decoder transformation sequence using unpack intrinsics. This processes 16 single-precision floating-point values simultaneously. The grayed-out arrows represent the same operations in the respective stages, and the annotations are left out for clarity.

4.4 Storage Efficiency and Throughput Evaluation

In this section, we provide an empirical evaluation of our implementation. We focus on the compression step individually in this part of the thesis and postpone a more extensive evaluation of the streaming aspect until Section 5.5. We propose a number of evaluation questions in Section 4.4.1 that help to characterize the various aspects of our approach's performance. Then, we describe our experimental setup and go over the evaluation aspects one by one. We finish this section with a short discussion of our results and findings.

4.4.1 Evaluation Questions

Our experiments in this chapter are organized around the following *compression question sets* (CQSx):

- CQS1: How efficiently can Byte Stream Split be mapped onto the instruction set of different CPU architectures? How does the performance depend on the block size used for Byte Stream Split? How much can the manually vectorized versions speed up Byte Stream Split in comparison to the compiler-provided automatic vectorization?

- CQS2: How do dataset properties influence the effectiveness of Byte Stream Split in the context of Apache Parquet?

- CQS3: How much can Byte Stream Split improve the compression ratio of general-purpose compressors for a list of publicly available datasets? How does it compare to other, specialized lossless floating-point compressors?

- CQS4: How much does it speed up compression and decompression for an example sensor dataset from a real-world industrial application? Which compression algorithms and settings work best?
- CQS5: How does our compression approach compare to other systems in terms of storage efficiency for the example sensor dataset?

4.4.2 Experimental Setup

Our approach is meant to apply to high-end server systems and low-power edge computers. Thus, we evaluate performance on two types of systems: One is equipped with a powerful server CPU. The other is a passively cooled low-power edge system, typically found in industrial installations. The details of those two systems are listed in Table 4.1.

Table 4.1 Configurations of the systems used for evaluations

	Server System	Edge System
CPU	Intel® Xeon® Gold 6136 Processor 3.00 GHz, 12 cores (24 threads)	Intel® Atom® x5-E3940 Processor 1.60 GHz, 4 cores (4 threads)
RAM	4×16 GiB DDR4 @ 2666 MT/s	2×4 GiB DDR3 @ 1866 MT/s
TDP	150 W	9.5 W
ISA	SSE & AVX	SSE
Network	10 Gbit Ethernet	1 Gbit Ethernet

Both systems run Ubuntu 20.04 and use the *performance* CPU frequency scaling governor. Our experiments are implemented in C++, compiled with g++ compiler version 9.3.0 and use the options '`-O3 -march=native`'. As our implementation of Byte Stream Split is upstream in the Apache Arrow and Parquet-MR[6] libraries, we use the Conda packages `arrow-cpp` and `pyarrow` version 0.17.1 and link against this version of the library. Unless specified otherwise, we use default parameter values in the libraries for our experiments.

We use the *0-order entropy* $H(X)$ (or short just *entropy*) to estimate the compressibility of datasets X [100]. It is calculated as

$$H(X) = -\sum_{x \in X} p(x) \log p(x),$$

where $p(x)$ is the relative frequency of each unique value x of the set of values X. The 0-order entropy estimates how random the data is and how well it can be compressed using entropy coding. It is only an estimate because this is not the actual entropy for sources with memory, where consecutive samples are not independent of each other.

[6] Parquet-MR contains the Java implementation of Parquet.

4.4.3 CQS1: Performance of Byte Stream Split

At first sight, adding a further processing step in the encoding/compression path might seem like imposing quite a bit of additional overhead. However, our experiments show that the overhead is low, assuming the memory must be copied to another buffer anyway. To illustrate this, we design a separate microbenchmark, testing different implementation alternatives for the encoding and decoding paths of Byte Stream Split. We measure throughput of encode and decode stages, which helps to understand the impact of our vectorization scheme. The first version implemented is just a simple nested loop structure without explicit vectorization. In this version, we rely on the compiler to automatically generate efficient code. The other two implementation variants explicitly use SSE or AVX intrinsics, respectively. We test our implementation on blocks of memory of different sizes on both of our evaluation systems to better understand the runtime behavior. We average the throughput over 256 runs of encoding (or decoding) a buffer of the specified block size. Input and output buffers are both accessed before the measurement to ensure warm caches. The results of these experiments are shown in Figure 4.9.

Figure 4.9 reveals several interesting properties of our Byte Stream Split encoding. First, smaller block sizes can be processed with higher throughput. This is due to the CPU cache hierarchy, where smaller block sizes fit entirely in the CPU's cache. This also indicates, at least for larger block sizes, that the encoding is bound by memory bandwidth instead of computational intensity. When the block sizes are small enough, though, we can observe that the decode step is faster than the encode step. This can be explained by noting that fewer instructions need to be executed for the decode step (cf. Figures 4.7 and 4.8). Furthermore, encoding (and decoding) single-precision data is faster than encoding (decoding) double-precision data for small buffer sizes. In addition to that, we can observe that the fastest vectorized implementations are almost as fast as memcpy for larger block sizes. It should also be noted that the compiler cannot automatically vectorize the double-precision version of the encode path[7]. This justifies adding a manual SSE implementation for that case to improve performance. In contrast, as expected for the memory-throughput bound encoding kernels, using AVX-instructions does not lead to a huge increase in performance on the Xeon platform.

To summarize these results, our approach can be efficiently implemented on architectures with support for vector instructions. Considering that compression and decompression rates of standard compression algorithms are considerably lower than the rates we measure here, we expect that Byte Stream Split will not harm the compression or decompression throughput. We confirm this expectation in later experiments.

[7] General remark: we experienced lower performance with gcc 7.5 for some code paths, so compiler technology is improving in that aspect.

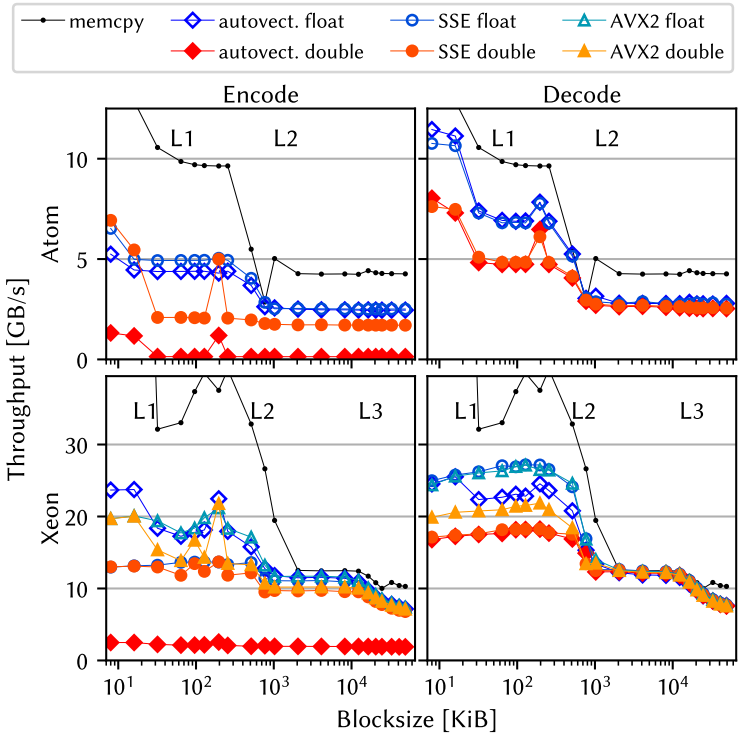


Figure 4.9 These graphs show encode (left plots) and decode (right plots) performance of execution on the two systems, Atom (top) and Xeon (bottom). The different Byte Stream Split implementation alternatives are represented using differently colored measurement points. Each graph shows the throughput of the different implementation alternatives for different block sizes. The caching hierarchies of the systems are highlighted using light green background colors and Lx annotations to better explain the performance. Note the different scales for the throughput axis on the two different systems. The AVX measurements are only performed on the Xeon system since this instruction set is not available on the Atom CPU.

4.4.4 CQS2: Influence of Dataset Properties on Byte Stream Split Effectiveness

This section analyzes the impact of various dataset properties on the effectiveness of the presented Byte Stream Split approach. We perform this analysis in the context of Apache Parquet, which we have chosen as the long-term storage format for HAQSE. We take a short excerpt of raw sensor data (see Figure 2.1) for this analysis, normalizing it to zero mean and a standard deviation of one. The dataset is approximately normally distributed as shown on the right of Figure 2.1. The resulting dataset used for these experiments contains 1 000 000 single-precision floating-point samples, i.e., it consumes 4 MB when stored in an uncompressed way.

For each analysis, we save the data to an Apache Parquet file and change certain dataset properties as well as Apache Parquet format settings. Our experiments write data using five different configurations:

- **uncompressed**: no encoding, no compression. This just serves as a baseline.
- **dict**: Apache Parquet dictionary encoding, no additional compression. This is advantageous for large row group sizes, so we report this as an alternative lightweight compression method.
- **dict+zstd**: Apache Parquet dictionary encoding, zstd compression. Same as above, but data pages are additionally compressed using zstd.
- **zstd**: no encoding, zstd compression. This serves as a general-purpose compression baseline.
- **bss+zstd**: Byte Stream Split encoding, zstd compression. Our proposed approach.

We report the achieved compression ratio with respect to the uncompressed 4 MB dataset size, dividing the size of the resulting Apache Parquet file by 4 MB. Sensible values for constant parameter values are determined by preliminary experiments.

Impact of Relative Values Scale

For this analysis, we vary the standard deviation of the dataset, i.e., we change the absolute value range. We do this by scaling the complete dataset with a constant factor, keeping the mean value at zero. We use an Apache Parquet row group size of 250 000. The results of this analysis are shown in Figure 4.10.

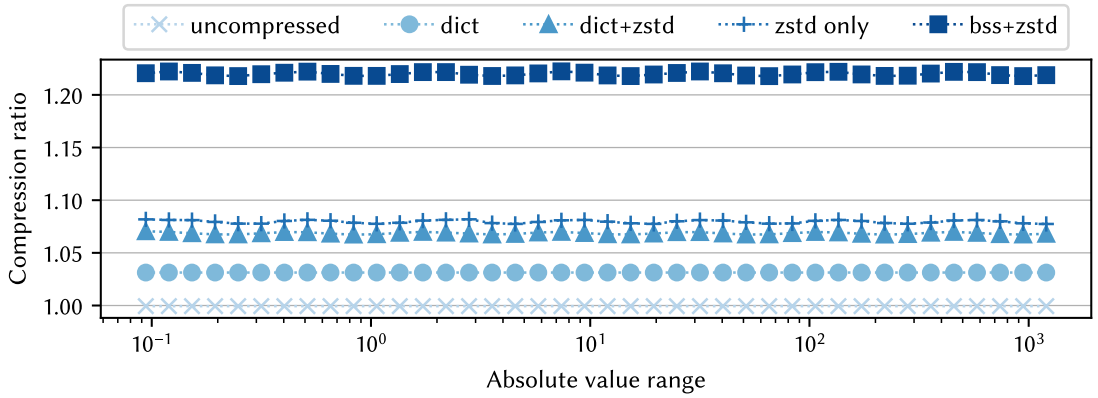


Figure 4.10 Influence of relative dataset scale on the compression ratio of the various tested encoding/compression alternatives. The x-axis shows the absolute value range, i.e., $max(ds) - min(ds)$ for the values in the tested dataset ds.

This plot shows that the different scale factors have almost no influence on the compression ratio of any tested compression variant. This similarity is expected since the scale factor primarily changes the value of the exponent part of the numbers. The

distribution in the composing bytes (which determines our approach's effectiveness) does not change. The entropy for all resulting datasets (with different scale factors) is constant at ≈16.9. Our proposed approach *bss+zstd* is the best method in Apache Parquet.

Impact of Average Value Magnitude

Contrasting the previous experiment, we now change the signal's magnitude. We do this by adding a constant to all values of the dataset. The standard deviation is kept at one. As before, we use an Apache Parquet row group size of 250 000. The results of this analysis are plotted in Figure 4.11.

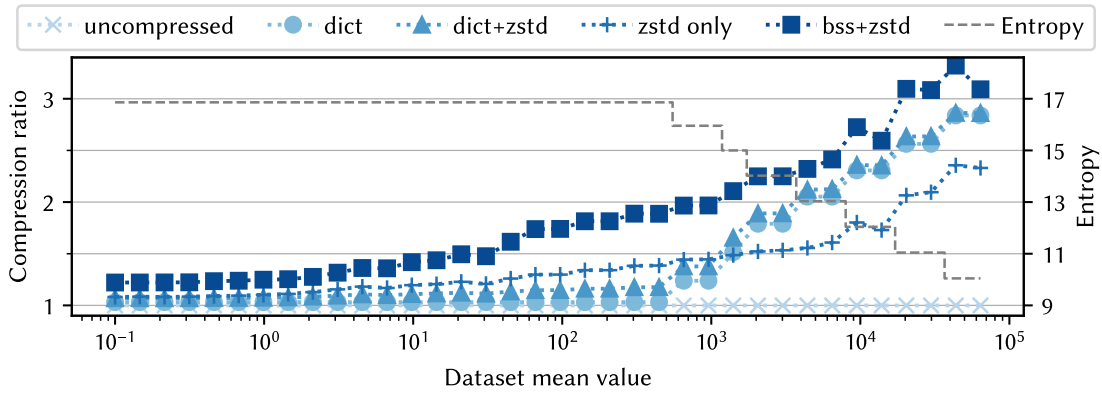


Figure 4.11 Influence of average value magnitude on the compression ratio of the various tested encoding/compression alternatives.

This plot shows that an increasing absolute value leads to gradually improved compression effectiveness of our approach. The value's increased magnitude can explain this improved effectiveness. There are not enough bits in the mantissa to represent all the information of the original value anymore, leading to higher quantization errors and reducing the entropy in the generated dataset. A single-precision floating-point number can only represent roughly seven decimal digits. The effectiveness of dictionary encoding and entropy coders (e.g., zstd) improves with reduced entropy. Our *bss+zstd* approach provides gradual improvements already for datasets where the effectiveness of dictionary encoding is unchanged. Like before, *bss+zstd* is always the best method.

Impact of Compression Unit Size

In this last part of the effectiveness evaluation, we evaluate the impact of the compression unit size. For this, we vary the row group size that is used for writing the resulting Apache Parquet file. We take the normalized dataset for this experiment: the

dataset has a mean of zero and a standard deviation of 1. We vary the row group size in exponential steps from 4000 to 1 000 000.

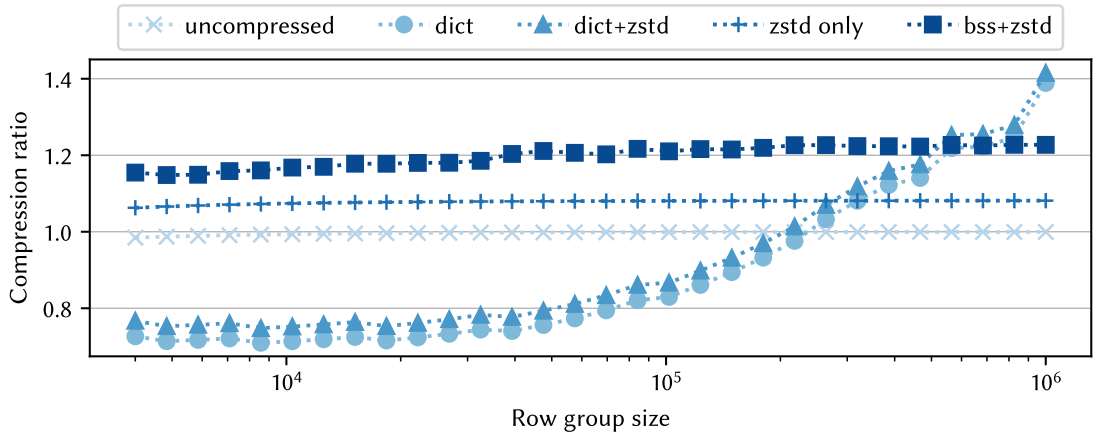


Figure 4.12 Influence of row group size on the compression ratio of the various tested encoding/compression alternatives.

The results in Figure 4.12 show several interesting aspects. First, the row group size has a significant impact on the achieved compression ratio for dictionary-encoded data. For row group sizes smaller than roughly 250 000, the additional storage space required by the dictionary leads to a total storage space consumption higher than the uncompressed baseline. However, this additional dictionary overhead pays off for row group sizes larger than ≈550 000. In these cases, the dictionary-encoded alternatives provide the best compression ratio. Our proposed approach bss+zstd inherits the properties of the underlying compressor (zstd in this case): The row group size has a minor influence on the resulting compression ratio (but, again, larger row group sizes are beneficial).

Summary

This section summarizes the insights gained regarding the effectiveness of Byte Stream Split:

- Byte Stream Split performs well regardless of the dataset size to compress (in Apache Parquet, this corresponds to the row group size). This is in contrast to dictionary encoding, which works especially well for larger row group sizes (where values repeat more often).
- Linearly scaling a dataset does not impact the effectiveness of our proposed approach.
- Byte Stream Split works well for datasets with a small value range compared to the mean value. This is common for sensor measurements with values fluctuating around a constant signal component.

- Our experiments in this section show that applying Byte Stream Split before zstd is always better than only using zstd. Apart from that, our approach inherits the properties of the underlying compressor.

In the next section, we confirm and extend these results by evaluating our approach for different floating-point data sets.

4.4.5 CQS3: Compression Ratio Performance

In this section, we compare our approach against other methods for lossless floating-point data compression. We use a collection of publicly available datasets for evaluating our Byte Stream Split approach in terms of compression ratio.

Datasets

The datasets provided by Burtscher et al. [17] were originally used to benchmark the FPC [17] and SPDP [23] lossless floating-point compressors. They contain single-precision [18] and double-precision [19] samples from three categories: observational data, results of numeric simulations and messages with numeric data. The datasets are listed in Table 4.2. This table contains the dataset *size* in MB and the *0-order entropy* (see Section 4.4.2).

Experiment

We compress each dataset, in both variants, single and double precision with each of the following compression alternatives: zfp [73], fpc [17], fpzip [74], SPDP [23], ZStandard [24] (using default settings, tagged zstd), and a combination of Byte Stream Split and ZSstandard (tagged with bss + zstd). We calculate the compression ratio as the size of the uncompressed dataset divided by the size of the compressed result.

Results

The resulting compression ratios are plotted in Figure 4.13 (omitting results for the alternatives zfp and fpc[8]).

The results show several interesting properties of our approach. First, applying Byte Stream Split before the actual compression is better than just applying ZStandard for almost all tested datasets. The only exceptions are the *num plasma* and the *obs error* double-precision datasets, where *zstd* alone achieves a slightly better compression ratio than *bss+zstd*. In a majority of cases (19 out of 26 tested datasets), the *bss+zstd* approach yields the best compression ratio. It is noteworthy that *bss+zstd* does not

[8] The results for zfp are not competitive for any of the datasets. For fpc, there are only results for double precision.

Table 4.2 Uncompressed size and entropy of floating-point datasets [17] used for experimentally evaluating Byte Stream Split. Single-precision datasets are half the size of the respective double-precision dataset reported here.

Dataset	Size (MB)	Entropy (bits)	
	double-precision	*double*	*single*
Observational data (obs): *measurements from scientific instruments*			
error	59.28	17.804	17.785
info	18.05	18.068	18.068
spitzer	189.00	17.359	17.359
temp	38.08	22.251	22.185
Numeric simulations (num): *results of numeric simulations*			
brain	135.27	23.971	23.580
comet	102.38	22.039	21.880
control	152.12	24.140	24.082
plasma	33.46	13.651	13.650
Parallel messages (msg): *numeric messages sent by a node in a parallel system*			
bt	254.05	23.667	22.754
lu	186.13	24.470	24.293
sp	276.67	25.030	23.282
sppm	266.07	11.238	8.400
sweep3d	119.91	23.411	19.691

clearly fail for any of the datasets. In case it is not the best alternative, it is almost always very close in achieved compression ratio to the best alternative. Exceptions are *obs error* and *obs info* datasets, where SPDP compresses considerably better. It should be noted, though, that SPDP was *designed* for this list of datasets. The double-precision version of these two datasets compresses best with fpc (not shown in the figure; compression ratios are 2.28 and 2.04 for *obs error* and *obs info*, respectively). For all other datasets, fpc is not among the best-performing alternatives.

4.4.6 CQS4: Compression Performance on Sensor Data

In this section, we investigate the overall performance of our two-step compression approach for a real-world, industrial sensor dataset. As before, we measure compression ratio. Additionally, for this set of experiments, we also measure compression and decompression throughput. We write to and read from persistent storage since this is closer to the real-world usage scenario than just keeping data in memory. We use a relatively large time window of 1 GB of data (250 million single-precision values) of

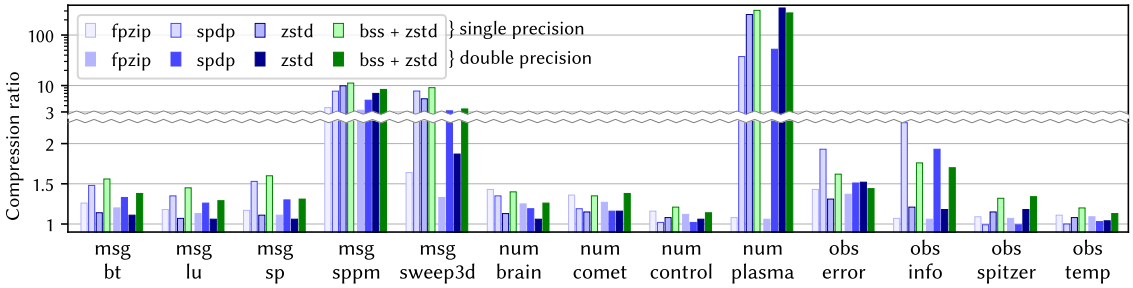


Figure 4.13 Comparison of Byte Stream Split/zstd (bss + zstd) with other lossless compression alternatives. Note that the lower part of the plot has a **linear scale** while the upper part is scaled **logarithmically** to better show the differences for highly compressible datasets. The lighter-colored bars (left half for each dataset) indicate the results for single-precision datasets. Solid colored bars (right half) show results for double-precision datasets.

the raw sensor data stream (see Figure 2.1) as benchmark dataset, corresponding to roughly 163 minutes of uninterrupted data sampled at 25.6 kHz.

We test different encoding/compression configurations using the following approach: Using a test driver implemented in Python, we load 1 GB of data into an Apache Arrow `Table` object, and call `write_table()` from the `pyarrow.parquet` package with the respective encoding and compression settings. This results in data being written to an Apache Parquet file on a local Solid-State Drive (SSD). This Apache Parquet file contains all data in a single row group. After that, we fully read the file that has been just written to measure read performance. To avoid operating system buffer caching effects, we clear the Linux page cache after writing and before reading the files[9]. This also ensures that all data *must* be read from persistent storage. The compression ratio is computed as the number of bytes in the Apache Arrow table divided by the size of the resulting Apache Parquet file. For throughput measurements, we take the uncompressed data size and divide it by the time needed for compressing and writing/reading and decompressing to/from persistent storage. We average these throughput results across ten runs.

The results of this compression performance evaluation are visualized in Figure 4.14. They show that our approach improves the best state-of-the-art variants in all three metrics: compression ratio, write and read throughput.

With respect to compression ratio, the results show that all tested general-purpose compression algorithms benefit from first applying Byte Stream Split. This underlines that our approach does not depend on any particular general-purpose compressor but can be applied to a wide range of compressors. In addition to that, applying Byte Stream Split works well regardless of compression level (for those compression algorithms that have a compression level setting: zstd [24], gzip [34], and Brotli [4]). In-

[9] When working with files in Linux, the page cache [128] caches file contents from slow persistent storage devices in RAM. This hides latencies and considerably speeds up working with files located on, e.g., Hard Disk Drives (HDDs).

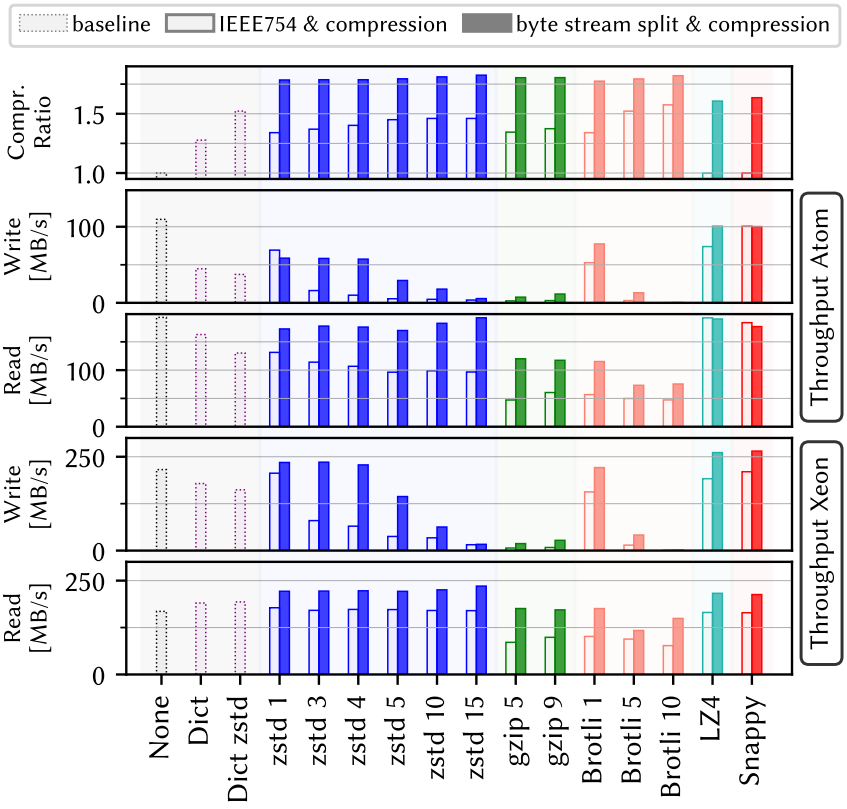


Figure 4.14 Comparison of compression ratio, write and read throughput for Atom and Xeon platforms across a set of encoding/compression combinations. The first measurement (None) is there for reference, providing a baseline of the I/O-system's speed. We are using *dictionary encoding*—a fast and lightweight alternative encoding in Apache Parquet—as an additional baseline.

stead, applying Byte Stream Split improves the compression ratio more than any improvement in changing compression level could yield. Additionally, our approach enables significant improvements for fast and easy compression algorithms like LZ4 [77] and Snappy [39]. These two compression algorithms almost do not compress the data streams at all without preprocessing. We also want to note that *all* tested combinations with Byte Stream Split provide higher compression ratios than the best alternative in Apache Parquet without Byte Stream Split (Brotli with a compression level of 10). The left three bars show the alternatives available in Apache Parquet before the addition of Byte Stream Split. It is evident that the presented method represents a significant improvement for efficiently storing floating-point data in Apache Parquet, making it an interesting format choice for many applications requiring such storage.

Looking at write and read throughput, applying Byte Stream Split *improves* throughput numbers in almost all cases, even though an additional processing step is performed. For the system with the Atom CPU, there are some combinations where the

additional step slightly decreases the write throughput (zstd 1, Snappy). This slight throughput reduction is due to the less powerful CPU being fully utilized in these cases. On the contrary, for the powerful Xeon CPU, writing with Byte Stream Split enabled is always faster. A similar behavior can be observed for reading and decompressing data: only Byte Stream Split combined with LZ4 and Snappy perform slightly worse when writing on the Atom system. Everything else is faster with Byte Stream Split.

Next, we look at a combination of throughput and compression ratio results. For the few combinations where read or write throughput are minimally reduced, we argue that the improved compression ratio will be worth the additional effort. For all other cases, activating Byte Stream Split for the tested dataset is always better in all three metrics.

We conclude that Byte Stream Split is beneficial if the throughput gap between storage and CPU is large. Moreover, our method is well-suited for simple, high-throughput compression algorithms, extending the design space especially on systems with less powerful CPUs.

4.4.7 CQS5: Storage Efficiency Analysis for Real-World Dataset

For this last set of experiments, we analyze the storage efficiency of different state-of-the-art systems and our approach as implemented in Apache Parquet. In contrast to the previous experiments, for this part of the evaluation, we analyze a complete dataset covering multiple sensors and including 64 bit timestamp information. We take a representative, real-world dataset that consists of roughly one hour of sensor data. This data comes from 14 sensors monitoring the combustion process of a gas turbine (for details of the installation we took the data from, see Section 7.4) at a sampling rate of 25.6 kHz. The original dataset is available in a proprietary, binary format (Argus Data Format (ADF)), described in more detail in Section 7.2.3. In this dataset, all sensor values are represented as 32 bit single-precision floating-point values. As a baseline, the raw size for this dataset is approximately $3600\,\mathrm{s} \times 25\,600\,\mathrm{Hz} \times (14 \times 4\,\mathrm{B} + 8\,\mathrm{B}) = 5.9\,\mathrm{GB}$.

We convert or load the dataset into various storage formats or systems. First, we report the size of two variants of the original ADF format: once uncompressed and once compressed with zstd [24]. Then, we provide results for a variety of Apache Parquet encoding and compression alternatives. As a baseline, we use an uncompressed Apache Parquet file, using the `PLAIN` encoding and no compression for any of the columns. We also report the result when compressing all sensor columns with zstd and encoding the timestamp column with the `DELTA_BINARY_PACKED` method available in Apache Parquet. Finally, we apply Byte Stream Split to all sensor columns before compressing them with zstd, and again use delta encoding for the timestamp column. We also import data into three different state-of-the-art time series management systems: InfluxDB (using the ILP), QuestDB (manually defining the schema to force single precision for floating-point columns) and TimescaleDB (via SQL). We additionally apply manual chunk compression for TimescaleDB [125] and report that result. We measure

the required storage space on disk after the import is finished with the method listed in Table 4.3.

Table 4.3 Different storage systems and format and how storage size is measured.

System/Format	Measured via
ADF uncompr.	File size
ADF zstd	File size
Parquet uncompr.	File size
Parquet zstd	File size
Parquet bss+zstd	File size
QuestDB	`du db/<id>/` (see [41, 94])
InfluxDB	`du engine/data/` (see [41, 55])
TimescaleDB	`SELECT before_compression_table_bytes + before_compression_toast_bytes FROM chunk_compression_stats('<id>');`
TimescaleDB compr.	Chunks are manually compressed [125], then: `SELECT after_compression_table_bytes + after_compression_toast_bytes FROM chunk_compression_stats('<id>');`

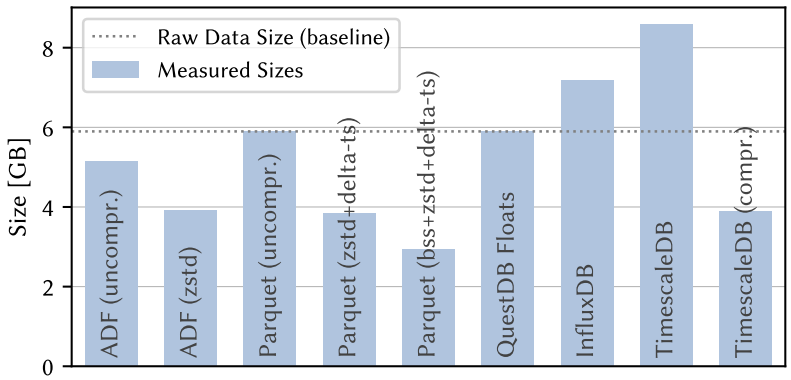


Figure 4.15 Comparison of storage efficiency between different storage system alternatives for a real-world dataset.

The results of this analysis are visualized in Figure 4.15. They reveal several interesting properties. First, the uncompressed ADF consumes less space than the raw data baseline. This can be explained by a very high storage efficiency for the timestamp information in the ADF format (for details, see Section 7.2.3). Compressing ADF with zstd brings the storage consumption further down to 3.9 GB. When stored uncompressed in Apache Parquet, the required storage size is approximately equal to the raw data size,

as expected. The first compressed Apache Parquet alternative (compressing the sensor columns with zstd and applying delta encoding for the timestamp column) shrinks the size to 3.8 GB, slightly less than the compressed ADF file. Applying the presented Byte Stream Split encoding to the floating-point columns before compressing them with zstd further reduces the required size to 2.9 GB. This represents the most efficient way concerning storage space consumption among all tested alternatives. Out of the three time-series management systems, in their respective default configuration, QuestDB performs best. Since it does not compress the data, the required storage space is approximately equal to the raw data baseline. It should be noted that QuestDB requires the table to be defined using a `CREATE TABLE` statement. Otherwise, when implicitly created using the ILP, `float` fields would be stored as `doubles`, doubling the required storage space. In contrast to that, InfluxDB does not offer that option. Yet, the required storage size is not doubled because the underlying TSM files use compression for both timestamps and sensor values. TimescaleDB performs worst in our experiment and requires 8.6 GB. This value excludes storage space required for database index data. When enabling compression for the hypertable and manually triggering compression (in contrast to automatic compression happens after a configurable time has passed for the to-be-compressed chunk), the required storage space can be reduced to 3.9 GB, which is the best value for all time series management systems.

4.4.8 Evaluation Summary

To summarize our evaluations in this chapter, we conclude that the presented two-stage approach represents a storage-efficient and performant way to store floating-point data. The Byte Stream Split processing step can be implemented efficiently on contemporary hardware architectures. It provides a simple, yet effective preprocessing step for a wide range of general-purpose compressors. Apache Parquet proves to be a good choice as long-term storage format, especially when comparing it to the storage requirements of other time series management systems. We also showed that the approach works well for a typical sensor processing dataset. We thus choose the combination of Byte Stream Split and zstd for floating-point sensor data together with the `DELTA_BINARY_PACKED` encoding for timestamps as the default configuration for storing data in HAQSE.

5 Sensor Data Stream Transformation & Processing

In this chapter, we discuss all aspects related to sensor data stream processing. This is a crucial component of our system, as it performs all processing steps that are necessary to enable efficient data retrieval when querying sensor data from the system. These steps are layout transformation, windowed batch aggregation, and compression. In Section 5.1, we first give an overview of the complete data input pipeline and the involved components. In Section 5.2, we discuss the successive steps the data of a single sensor stream passes through: starting from sensor stream input until finally writing data to persistent, compressed and long-term storage. Then, we propose a compute-efficient hierarchical, windowed batch aggregation scheme, which we explain in Section 5.3. In Section 5.4, we describe the environment in which we perform our evaluation and explain the tools that are necessary for our experiments. Then, in Section 5.5, we extensively evaluate individual aspects of our implementation. Finally, in Section 5.6, we compare our system to other state-of-the-art time series data management systems in terms of ingestion performance.

Publication Information
Parts of the approach described in this chapter have previously been published in [66]. In this work, the author designed and developed the principal approach for streaming sensor data to Apache Parquet files. While this work builds the basis for the streaming architecture described in this chapter, it still misses several important aspects covered in this dissertation.

5.1 Overview of Sensor Data Stream Processing

Figure 5.1 shows a sketch of the data flow for a single sensor source stream in HAQSE. This stream of multivariate sensor measurements originates from a single source and is received by HAQSE via the interface defined in Section 3.2. It is first transformed to the sensor-ordered layout described in Section 4.1.2 and buffered *in memory*. After that, it passes through a *temporary storage* area, storing the transformed sensor data in temporary files. Multiple such files are finally combined, compressed and permanently stored in *compacted storage*. This scheme (first green row in Figure 5.1) enables efficient sensor stream ingestion and concurrent query processing.

In addition to that, as mentioned above, data is aggregated into streams of multiple levels of coarser time resolutions. This happens in the batch aggregation component, sketched on the left of Figure 5.1. Whenever an aggregation window for one or multiple aggregation levels is terminated, this component emits the result of this last aggregation window. These aggregation results are subsequently processed in the same way as the original source stream regarding storage and query handling (bottom, lighter green rows in Figure 5.1).

The details of query processing are discussed in Chapter 6. However, data queries rely on a data structure that is populated and updated during stream ingestion: the Stream Segment Index (sketched in the right part of each row of Figure 5.1), one index instance per stream level. Hence, already in this section, we explain how the Stream Segment Index is updated as data is received and flowing through the stages. This enables queries to retrieve a consistent state of all available segments.

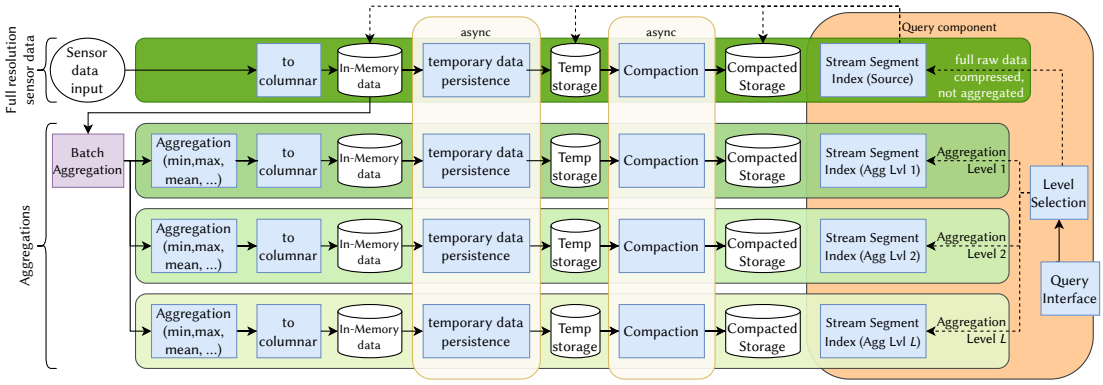


Figure 5.1 Overview of stream ingestion in HAQSE.

Each of the streams (original source stream and aggregated versions) is handled by the same processing logic. Many parts of this pipeline can be parameterized to meet the performance requirements of the application, adapted to the stream characteristics and adjusted to the underlying hardware resources and capabilities. The details of this processing pipeline are described in the following sections.

5.2 Three-Stage Stream Ingestion Pipeline

As explained in Section 4.1.2, we use a sensor-ordered layout for persistent storage. Contrary, as discussed in Section 3.2.1, the natural order of a sensor data stream in industrial sensor monitoring systems is time-ordered. As outlined in that section, sensor samples ψ arrive in one of two forms: as atomic samples for one particular instant ψ_i or as sample batches $\psi_{i..i+bs-1}$ with a batch size bs. In the first case, each sample contains exactly one value v_i for each sensor and the timestamp ts_i. In the latter case, the sample batch covers bs consecutive samples, containing bs timestamps ($ts_{i..i+bs-1}$) and

multiple, associated measurements $v_{i..i+bs-1}$ for each sensor. The challenge of HAQSE's ingestion pipeline is to process these incoming samples such that—eventually—they are stored in the sensor-ordered and compressed storage format described in Section 4.1.2. In addition to that, we already need to consider the basics of the query functionality of HAQSE (which we will describe in more detail in Chapter 6). For this query functionality, it should be possible to retrieve all data in the system at all times, independent of where in the ingestion pipeline it is located at query time. The next subsections present a three-stage approach we designed to consume high-rate sensor data streams, while being able to query all that sensor data. After that, we show what technologies we use to realize this approach in HAQSE.

5.2.1 Layout Transformation to Sensor-Ordered Buffer

First, when data is received on the interface, it is decoded from the message representation to an internal, native representation. Depending on the chosen message format, this decoding might be trivial (in case the message representation is identical to the native representation) or already require some data processing (in case a different message format is chosen). Once decoded, the native representation of the sensor values are buffered in a preallocated, *temporary buffer*. This buffer is allocated when receiving the first message from the stream. It is organized such that values from one sensor are placed contiguously in memory, as visualized in Figure 5.2. It allows a certain number of samples to be buffered in memory, depending on the configurable *temporary row group size*[1], that determines the number of rows in this buffer. This buffering carries out the layout transformation from the time-ordered input layout to the sensor-ordered layout in the buffer. In other words, the individual measurement values of the different sensors in the schema for one ψ_i are scattered into the sensor-ordered layout of the buffer.

Together with the schema of the sensor stream, the temporary row group size determines the physical size in bytes of the buffer:

$$\text{buffersize} = \text{temporary row group size} \times \text{schema\_bytesize}$$

where

$$\text{schema\_bytesize} = \texttt{sizeof(type}(ts)\texttt{)} + \sum_{\sigma \in \mathcal{S}} \texttt{sizeof(type}(\sigma)\texttt{)}.$$

This temporary buffer can optionally be backed by a memory-mapped file, avoiding filling the operating system's swap space unnecessarily in case physical memory is limited. This is useful, especially for large temporary row group sizes. Backing the memory by a file can reduce the amount of memory that needs to be physically present

[1] We borrowed the term *row group size* from the Apache Parquet terminology, where data is organized in row groups, each of which has a certain *row group size*.

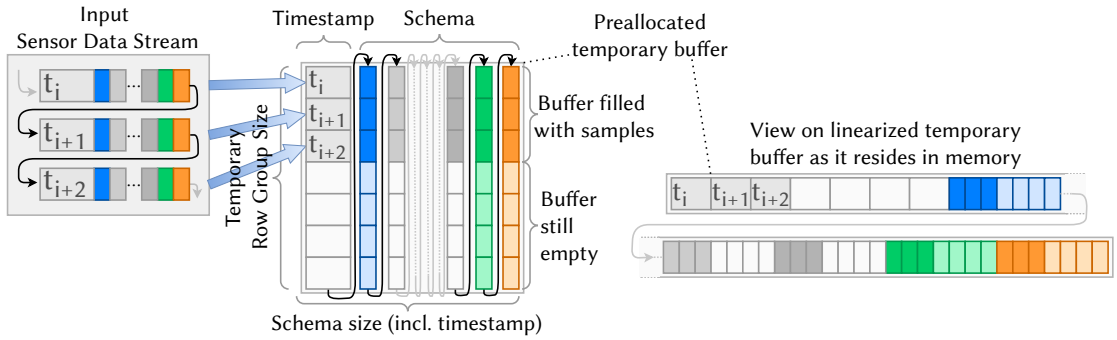


Figure 5.2 Sketch of how input stream data is mapped to the temporary buffer. The black arrows indicate the physical organization of the stream and the buffer. The right part of the figure illustrates how the sensor-ordered layout actually resides in the linear memory space.

in RAM, but requires the operating system to write back memory to persistent storage regularly. While this limits the achievable throughput, it can be helpful for slower sensor data streams.

Once a new temporary buffer is started and filled with at least one row, a new stream segment is added to the Stream Segment Index. As explained in more detail in Section 6.1, stream segments store the range covered by the respective sensor data. For row groups stored in the temporary buffer discussed in this section, this range is open-ended since we expect the stream to be arriving at a fast pace, and we want to avoid fast-changing updates to the stream index. The added segment is identified by the timestamp ts_i of the first sample ψ_i contained in the buffer. The end timestamp of this segment is left empty.

5.2.2 Intermediate Buffering In Temporary Columnar Files

Once the temporary buffer is filled completely (or the stream is terminated), it is written to *temporary files* in persistent storage. These temporary files serve as an extended buffer between the fast, temporary buffer (which is limited in size) and the compressed, long-term compacted storage. There are several reasons why it is advantageous to write uncompressed data to temporary files instead of directly to compressed files. First, most recent data segments are highly likely to be queried. This could be for near-real-time analyses, for dashboard applications showing current values or other—in many cases autonomously—recurring queries that retrieve the most recent history. Thus, it makes sense to provide access to these segments with as little overhead as possible. Second, having an intermediate storage buffer, the final compacted row group size can be independent of the temporary row group size. This increases the flexibility of the processing pipeline. Moreover, it potentially increases storage efficiency in the compressed representation, as the share of constant overheads in the compression format diminishes. Finally, reading partially written compacted files requires managing

this information somewhere. While this would be possible (all necessary information is available), it unnecessarily complicates the implementation.

Temporary files are designed to contain multiple temporary row groups. The binary representation of these row groups does not necessarily need to be identical to the one in the temporary buffer. However, for performance reasons, the general data layout should reflect that of the temporary buffer. In addition to that, it must be possible to read contents from these files while they are still not completed. Otherwise, without additional metadata or structural information, it would not be possible to retrieve data for segments that are stored in these temporary files. This is achieved by having repeated headers in the file, indicating the size of the following parts with the actual sensor data. This layout requires unfinished temporary files to be read from front to back when queried. For such queries, iterating over too many row groups in temporary files could be unfavorable for performance. Hence, after a certain number of temporary row groups has been persisted to a temporary file, structural information is appended in a footer and this file is closed. Then, a new file is created and the next fully buffered temporary row groups are written to the new file.

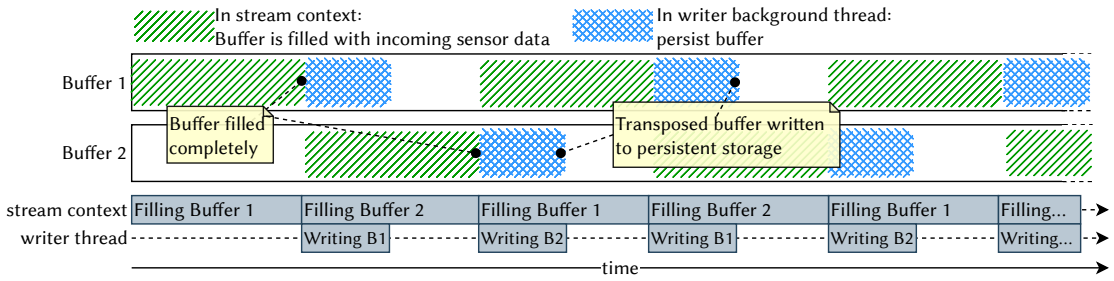


Figure 5.3 Illustration of double buffering scheme. While one of the buffers is filled with new sensor data received from the stream, the other one (which has already been filled previously), is written out to a temporary file.

It is possible to have multiple instances of preallocated temporary buffers. This makes it possible to write fully buffered data in the temporary buffer to the current temporary file, while another temporary buffer instance can already ingest (and transform the layout of) more data, as illustrated in Figure 5.3. This double-buffering scheme helps to increase the processing throughput in certain situations. We analyze this in detail in Section 5.5.

Once a temporary buffer has been fully written to a temporary file, the following actions are performed in an atomic update step on the Stream Segment Index:

- The entry for the fully written temporary buffer is removed from the Stream Segment Index. There may already be another entry for a temporary buffer in the index for consuming stream data; this is left untouched.

- If the temporary file existed before and thus, the entry for it is present in the Stream Segment Index, it is removed, as it contains outdated information of the temporary file. In particular, the covered segment range is not correct anymore.
- An updated entry for the temporary file is added to the Stream Segment Index, reflecting the correct time span covered by the temporary file.

This is done in one atomic step, protected by a data structure guaranteeing mutual exclusion. After the index is updated, the filled temporary buffer that was just written to a temporary file is available again for ingesting sensor stream data.

When a temporary file is finished (which implies renaming the file currently being written to), the following actions are performed:

1. Create a hard link[2] to the file using its new name. The new name is the first timestamp contained in the file.
2. Atomically update the Stream Segment Index:
   - Remove entry for unfinished file.
   - Add entry for renamed file.
3. Unlink the file system entry with the old filename.

The whole process of writing data to temporary files creates a list of several temporary, uncompressed files. Since we target compressed files for long-term storage, in the next step, these files are compacted. This compaction step is described next.

5.2.3 Compaction: Combining and Compressing Sensor Data from Intermediate Files

Once a certain, configurable number of temporary files is filled, these files are combined in a *compacted file*. These compacted files act as the long-term storage destination of all stored sensor streams. The contents of such a compacted file use the final data layout described in Section 4.1.2, using a *compacted row group size* to organize the resulting row groups. While not required, in most practical cases, this compacted row group size will be larger than the *temporary row group size* of the temporary buffer and files. Furthermore, it is reasonable—but, again, not mandatory—to define the compacted row group size to be an integer multiple of the temporary row group size.

In addition to that, these compacted files are compressed, such that the long-term storage requirements are minimized. As presented in Section 4.1.2, the actual compression is performed independently for each column, i.e., the timestamp column and each sensor column are compressed separately. This enables changing compression parameters individually for each sensor.

[2] A hard link to a file is an independent directory entry (inside the same file system) to the file (more precisely, its contents). Creating a second hard link to a file makes it possible to reference a single file by multiple names [11].

```
1  class Compactor:
2    """Manages compaction cfg and state"""
3    def compact_files(self, list_of_files, target, row_group_size):
4      """Compact files in `list_of_files`, write to output file `target`.
5         Each row group of the output file is filled with a maximum of `row_group_size` rows. """
6      self.list_of_files = list_of_files
7      self.schema = extract_schema(list_of_files)
8      self.row_group_size = row_group_size
9      idx_state = IndexState()
10     while idx_state.file_index < len(self.list_of_files): # iterate as long as more data is available and
11       idx_state = self._fill_next_row_group(idx_state, target) # fill target rowgroup by rowgroup
12
13   def _fill_next_row_group(self, idx_state, target) -> IndexState:
14     """Fill one row group of target"""
15     row_group_writer = target.append_rowgroup()
16     for c in self.schema.columns:                          # iterate over columns
17       wrtr = row_group_writer.next_col()                   # create a column writer
18       new_state = self._fill_col_chunk(wrtr, c, idx_state) # new_state should be equal for all columns
19     return new_state
20
21   def _fill_col_chunk(self, col_writer, col, idx_state) -> IndexState:
22     """Fill one column of the current row group, write to `col_writer`"""
23     new_state = idx_state
24     remaining_size = self.row_group_size
25     while remaining_size > 0 and new_state.file_index < len(self.list_of_files):
26       # there is space in row group and more data available
27       cur_file = self.list_of_files[new_state.file_index]
28       rb = read_rb(cur_file, new_state.record_batch_offset)
29       col_data = rb.column(col)
30       offset = new_state.rb_row_offset
31       vals_to_take = min(len(col_data) - offset, remaining_size)
32       col_slice = col_data[offset:offset+vals_to_take]
33       col_writer.write(col_slice)
34       new_state.advance(vals_to_take, len(col_data), cur_file.rb_count())
35     return new_state
```

Listing 5.1 Compaction algorithm.

The actual algorithm for compacting files is outlined in Listing 5.1, with a compaction index state helper structure shown in Listing 5.2. When compacting a list of temporary files, after initializing the necessary state (lines 6–9), we fill the target file row group by row group (line 11). This is done as long as there are remaining rows in one of the files to be compacted (line 10). For each of the row groups to be filled, in turn, we iterate over all columns in the file schema (line 16) and fill them with data from the temporary files (line 18). Filling the columns (line 21) reads data as long as the column chunk is not fully filled and there is still data to read (line 25). The actual reading happens in units of record batches of the temporary files. These temporary files are opened as memory-mapped files. This has the advantage that only memory for columns that are actually read are mapped to physical memory. For other columns of the record batch, this is delayed until the respective column is read. We write as many values as there is space in the current row group, or all data in the current record batch that has still not been written, whatever value is smaller (line 31). In case there is not enough data to

fill the last row group of the resulting file, the actual number of rows in that last row group is smaller than the configured row group size. After reading selected contents from the temporary file (lines 29 and 32), and writing them to the column chunk of the current row group (line 33), the index state is advanced (line 34). This progresses the state object defined in Listing 5.2 to the next set of rows (line 9), record batch (line 11) or temporary file (line 14). The writing step in line 33 performs all data encoding and compression steps.

```
1  @dataclass
2  class IndexState:
3    """This class tracks how much has already been read"""
4    file_index: int = 0
5    record_batch_offset: int = 0
6    rb_row_offset: int = 0
7
8    def advance(self, vals_taken, num_rows_in_rb, num_rb_in_file):
9      self.rb_row_offset += vals_taken                # progress row offset in currect record batch
10     if self.rb_row_offset >= num_rows_in_rb:        # if record batch is exhausted
11       self.record_batch_offset += 1                 #   progress to next record batch
12       self.rb_row_offset = 0                        #   and start over with first row
13       if self.record_batch_offset >= num_rb_in_file: # if current file is exhausted
14         self.file_index += 1                        #   progress to next file
15         self.record_batch_offset = 0                #   and start with first record batch
```

Listing 5.2 Compaction algorithm state helper structure.

After the `compact_files()` function has returned, the target Apache Parquet file is closed. We use `fdatasync()` to ensure that all data actually lands on persistent storage. Once that call returns, we can assume that the compacted file is fully written to storage and the following actions are performed:

1. The Stream Segment Index is updated atomically:
   - The entries representing the temporary files are removed.
   - A new entry for the compacted file is added.
2. The temporary files are removed from the file system.

This compaction process is a resource-intense task. It reads all data from temporary storage, which may need actual reading from persistent storage, depending on how large the page cache of the operating system is. This aspect is experimentally evaluated in Section 5.5.6. Furthermore, executing encoding and compression steps require a considerable amount of CPU processing resources. Finally, the resulting file is written to persistent storage, again putting load on the I/O-system. For these reasons, and since we do not want to interrupt stream ingestion, compaction is performed in a separate background thread.

5.2.4 Implementation in HAQSE using gRPC, Apache Arrow & Apache Parquet

In this section, we describe how we implement the three-stage streaming approach just presented. For receiving sensor streams we define a gRPC client-streaming protocol (see Appendix A.2). The message stream of this protocol requires a stream definition as the first message. After that, the actual sensor samples or sample batches can be received by HAQSE. The actual samples or sample batches are represented by an opaque byte buffer, providing flexibility with respect to the actual encodings in the gRPC message. Per default, this buffer just contains the C representation in little-endian format of all data sent. This ensures that the overhead for receiving data is small. For sample batches, data is organized in a sensor-ordered layout, similar to the targeted data layout.

The temporary buffer for transforming the layout is just a single contiguous byte buffer. For writing timestamps and the individual sensor values, we calculate offsets into the buffer. These offsets, however, must be aligned to multiples of the alignment requirement of the respective data type for reading from the buffer. For that reason, we add padding such that each column ends on an eight-byte boundary (ensuring that both `alignof(double)` and `alignof(int64_t)` are satisfied). This guarantees that every (next) column starts with the same alignment. This is sufficient for writing all supported data types. Columns can optionally be aligned on the size of full cache lines. We did not notice any performance effects of aligning column buffers on cache lines, though.

It also would be possible to allocate separate buffers for each column. This, however, incurs an increased cost when the number of columns becomes large, because the number of virtual memory mappings that need to be managed by the operating system increases accordingly.

The row counter of the temporary buffer is implemented using an atomic integer, `std::atomic_size_t`. Together with a release-acquire memory order [28, 58], this ensures that querying data retrieves all rows written up to the point of the query. In contrast to protecting the row counter increment with a lock, this works without a context switch to the operating system and thus, has a low overhead.

Once the temporary buffer is filled, it is written to temporary files in the *Apache Arrow IPC format* [9]. This format consists of a stream of column-oriented *Apache Arrow Record Batches*, corresponding to the buffered row groups we use for transforming the data layout. It can be read without any additional information just by reading the stream from front to back. Additionally, before the file is closed, a footer is written after all record batches. It contains an index that points to the individual record batches of the stream. This speeds up searching for a particular timestamp in a finished file: since we know that the timestamps are ordered, we can exploit binary search instead of linearly iterating over record batches.

Finally, we compact multiple Apache Arrow IPC format files to one Apache Parquet file according to the algorithm explained in Section 5.2.3. An important implementation detail is to indicate to the operating system when the already compacted parts of a file are not needed anymore. This is done via the `madvise()` system call [76], using the `MADV_DONTNEED` advice for fully read memory pages. This keeps the resident set size low and considerably speeds up compacting large temporary files.

Locking the Stream Segment Index is achieved by employing a `std::shared_mutex`. Read accesses acquire a shared lock on the `shared_mutex`, updating the index requires an exclusive lock.

5.3 Hierarchical Windowed Data Aggregation

As discussed in Chapters 1 and 2, in many analytics scenarios, it is necessary to query large timespans[3]. However, it is not always necessary or even prohibitive to use the full temporal resolution of sensor data streams. Instead, in many cases, it is sufficient to retrieve windowed aggregations of sensor data streams. This can dramatically reduce the amount of data that needs to be transferred to a client requesting data. In situations like interactive data exploration, it might be both satisfactory and necessary to rely on precomputed aggregation results: Satisfactory because small inaccuracies will not change the user's perception of the visualized timeline. Necessary because latencies of more than 100 ms may be detrimental to the user's flow of thought [33]. In HAQSE, the required *hierarchical aggregates* on tumbling windows of the input stream are thus computed already during data ingestion and are eponymous for HAQSE's name.

In our approach, we exploit our assumption that the input sensor stream is sampled at a relatively stable sampling rate of $\frac{1}{\Delta t}$. In our aggregation component, we compute windowed aggregations for several derived levels of user-configurable aggregation resolutions Δt^{ω}. These aggregation resolutions are not required to have a certain relation to each other. In practice, however, using a fixed factor between aggregation level resolutions has several advantages. First, it is easier to understand for users. Second, this also makes the computation of aggregations more efficient. Finally, it makes most sense regarding the ratio between query performance and storage efficiency.

For real-world setups, the number of aggregation levels can quickly grow to more than ten. Thus, it is important to ensure an efficient and scalable approach for computing these aggregations. In HAQSE, we exploit the sensor-ordered, columnar data layout that we described in the previous section for this.

[3] Timespans are only large with respect to the sampling rate of a certain data stream. A timespan of one year is relatively small when data is sampled daily, only containing 365 values or a bit more than 1.4 kB (assuming single-precision, without timestamp information). In contrast, one year of data of a sensor stream sampled at 100 kHz contains more than three trillion values and requires more than 12 TB of uncompressed storage space (single-precision, without timestamps).

5.3.1 Formalization of Batch Aggregation

We first establish the necessary concepts and terminology to explain our approach.

We define an *aggregation window specification* $\omega = (\Delta t^{\omega}, o^{\omega})$ that uniquely partitions time into tumbling windows. These windows are aligned to an arbitrary, but fixed origin of time o^{ω}, e.g., Unix-epoch or a defined duration after that. Δt^{ω} is the length of the window specification, indicating both the size of the aggregation windows created by ω and the timestamp difference between two consecutive aggregation results. Based on such a window specification ω and a certain timestamp ts, $\mathcal{W}_{ts}^{\omega}$ is an actual aggregation window. We define the so-called *window id* that uniquely identifies a certain window $\mathcal{W}$:

$$id\left(\mathcal{W}_{ts}^{\omega}\right) = \left\lfloor \frac{ts}{\Delta t^{\omega}} \right\rfloor .$$

Using this definition, two aggregation windows $\mathcal{W}$ derived from the same window specification ω are identical iff $id\left(\mathcal{W}_{ts_a}^{\omega}\right) = id\left(\mathcal{W}_{ts_b}^{\omega}\right)$.

We also define the greatest common sub-window of a set of window specifications $\Omega = \{\omega_1, \omega_2, ...\omega_n\}$ with matching ω_o:

$$\hat{\omega} = gcd_{\omega}(\Omega) = gcd_{\omega}(gcd_{\omega}(gcd_{\omega}(\omega_1, \omega_2), ...)\omega_n)$$

where $gcd_{\omega}(x, y)$ represents the function to calculate the greatest common divisor of the window sizes Δt^{ω} of the respective window specifications x and y.

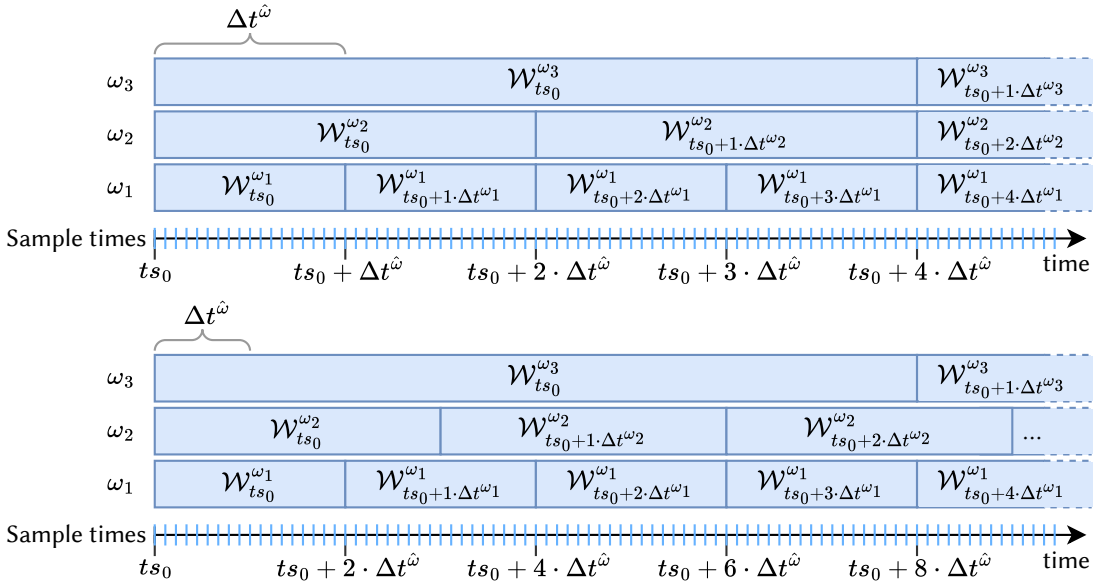


Figure 5.4 Example of greatest common sub-window $\hat{\omega}$ for two sets of hierarchical window specifications.

If the window specifications are defined such that *larger* window specifications are multiples of the smallest aggregation window specification, $\hat{\omega}$ is equal to the aggregation window specification with the smallest Δt^{ω}. This is visualized in the upper example of Figure 5.4. In other cases, such as in the lower part of Figure 5.4, this is not the case.

When aggregating, a set of different aggregation functions $\mathcal{F} = (f_1, f_2, \ldots f_n)$ is evaluated independently for each sensor σ in the respective schema. All $f_k \in \mathcal{F}$ must be associative and commutative so that it is possible to compute and combine partial results. The actual computations of aggregation functions $f_k \in \mathcal{F}$ happen for a particular aggregation window $\mathcal{W}^{\omega}$. We call the samples $\psi_{i..j}$ that fall into the aggregation window $\mathcal{W}^{\omega}$ the *aggregation batch*. All samples $\psi_{i..j}$ in an aggregation batch have the same window id i.e., $id(\mathcal{W}^{\omega}_{ts_i}) = id(\mathcal{W}^{\omega}_{ts_j}) \ \forall i..j$. For each σ in the schema, the respective functions in $\mathcal{F}$ are applied iteratively on all measurement values v^{σ} of the samples $\psi_{i..j}$ in the aggregation batch:

$$f_k(\ldots f_k(f_k(v^{\sigma}_i, v^{\sigma}_{i+1}), v^{\sigma}_{i+2}) \ldots, v^{\sigma}_j)$$

Since each f_k in $\mathcal{F}$ is associative and commutative, we can pre-aggregate these results based on $\hat{\omega}$ into so-called *preaggregations*. One (if the window specification ω is identical to $\hat{\omega}$) or multiple (for all other window specifications) such preaggregations can subsequently and iteratively be merged into the actual aggregation results. This makes it possible to reduce the number of computations in comparison to a naive implementation, where each aggregation level is calculated individually on the input samples.

5.3.2 Aggregation Process

This section describes the actual procedure of calculating aggregations. For calculating the various aggregation levels, we track the so-called *aggregation state* for each of the aggregation levels individually. This aggregation state consists of the current window $\mathcal{W}$ identified by its id $id(\mathcal{W})$ and the accumulated intermediate result. For each sensor sample ψ that is received from the stream, we perform the sequence of actions that is visualized in the flow chart in Figure 5.5. This sequence is triggered only once per received sample batch $\psi_{i..j}$, as soon as all samples of the batch have been put into the temporary buffer.

For each received sample ψ with timestamp ts, we calculate the id of the corresponding window $\mathcal{W}^{\hat{\omega}}_{ts}$ of the greatest common sub-window specification $\hat{\omega}$. If the internal aggregation state is not initialized (e.g., in case this is the first sample ever received), we initialize this state with the calculated window id $id(\mathcal{W}^{\hat{\omega}}_{ts})$. Additionally, we also initialize the state of each of the actual aggregation levels L using their respective window-ids $id(\mathcal{W}^{\omega_L}_{ts})$. Next, we check whether the processed sample ψ triggers an aggregation. This is the case if the window $\mathcal{W}^{\hat{\omega}}_{ts}$ is different from the one stored in the

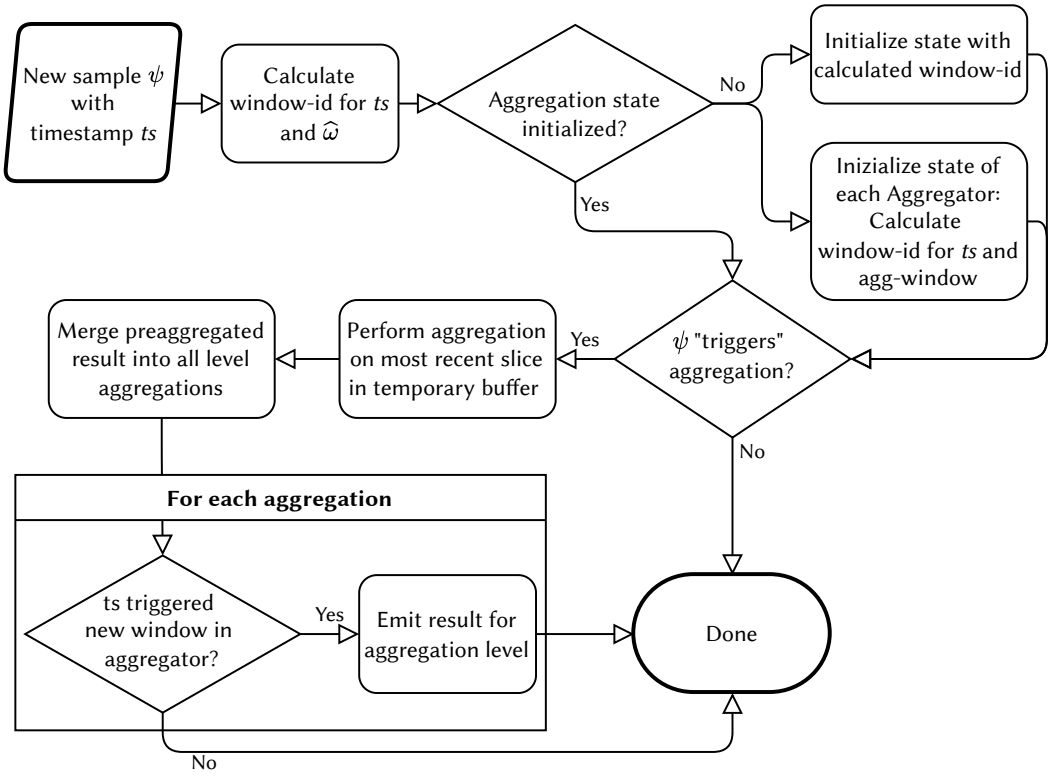


Figure 5.5 Flow chart showing batched aggregation process.

aggregation state, i.e., if the window-id calculated for ts is larger than the one currently stored in the aggregation state. This never happens if the state was previously uninitialized. Additionally, since we perform computations on the in-memory buffers, if the last row of these in-memory buffers is written, this also triggers an aggregation. These two different conditions for triggering an aggregation are visualized in Figure 5.6.

Whenever the sample triggers an aggregation, we take all samples that belong to the window $\mathcal{W}_{ts}^{\hat{\omega}}$ from the respective temporary buffer and perform the actual calculation on the batch of samples for each function in $\mathcal{F}$. The result is a preaggregated calculation result. This preaggregated calculation result is merged into the result state of all the actual aggregation levels.

Once that is done, all actual aggregation levels are checked: if $id(\mathcal{W}_{ts}^{\omega_L})$ is larger than the state stored for the respective level, the window is terminated. This means that the final aggregation result of the previous window (which was just fully calculated) is emitted. This value is then processed in the same way as samples from the original input stream, as discussed in Sections 5.1 and 5.2.

When a sample ψ does not trigger an aggregation, nothing is done. Depending on $\hat{\omega}$ and the size of the temporary row group buffer, this case happens relatively often and is one of the reasons why this aggregation logic has a low overhead. We expect this approach to scale well because of two further reasons. First, aggregations are computed

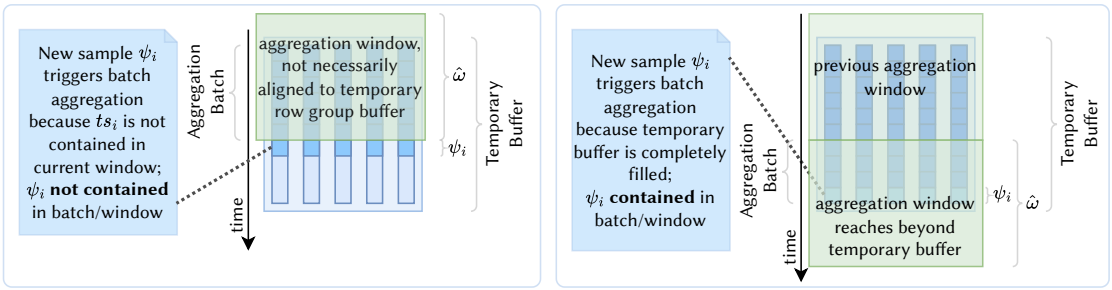


Figure 5.6 The two possible ways an aggregation is triggered. Left: the timestamp ts_i of the sample ψ_i is not part of the *active* window, so all buffered samples falling into the active window that are contained in the temporary buffer are used for computing aggregations. This does not include ψ_i. Right: the sample ψ_i fills the temporary buffer, all samples falling into the active window—including ψ_i—are aggregated.

on contiguous in-memory arrays of relatively recently added sensor values. This implies that there is a high chance that this data is available in caches. Furthermore, the linear access pattern can be executed very efficiently on modern CPU architectures. Additionally, this access pattern makes it easy for the compiler generate machine code that exploits vectorized instructions for the actual computation of aggregations. Second, the actual computation of mergeable aggregations only happens once. The results are then reused multiple times for the various aggregation levels. This works especially well for standard cases where increasingly coarse aggregation levels are multiples of each other.

5.4 Experimental Setup

In this section, we describe our experimental setup to extensively evaluate the various parameters of HAQSE. We also use this setup to compare HAQSE to other time series management systems with respect to sensor stream ingestion rates.

We begin with an overview of the general architecture and the components involved (Figure 5.7). As it can be seen in this figure, data generation is performed by a *Data Generator* component, which generates a stream of emulated sensor data. We explain this component in more detail in Section 5.4.2. Since the different databases in our comparison in Section 5.6 have different input protocol interfaces, we need to prepare the generated sensor data stream to be compatible with the respective input protocol. This is done by a *stream adapter* component that generates an appropriate sensor data stream, further explained in Section 5.4.3. We perform our experiments on a variety of different hardware configurations, as explained in Section 5.4.1.

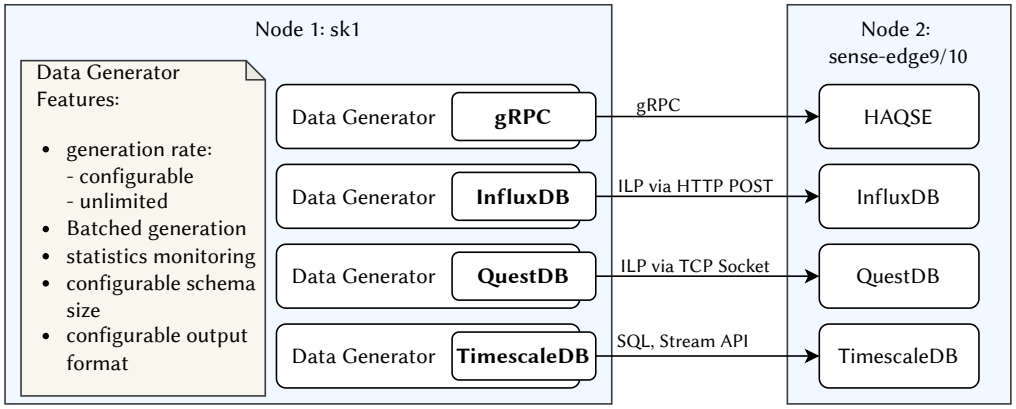


Figure 5.7 Setup of input stream benchmarks. Data generator and stream adapters are always running on *caps-sk1*. The actual databases under test run on another system.

Table 5.1 Overview of the different machines we use for our stream ingestion experiments.

Name	CPU (all Intel®)	Cores (Threads)	RAM [GiB]	Persistent Storage
caps-sk1	Xeon® Silver 4116	12 (24)	12 × 8	1 TB HDD
sense-edge9	Core™ i5-8365UE	4 (8)	1 × 8	1 TB HDD, 256 GB SSD
sense-edge10	Atom® E3845	4 (4)	1 × 8	1 TB HDD, 64 GB SSD

5.4.1 Hardware and Software Environment

The experiments performed in this chapter run on a number of different nodes, connected via the faculty network[4] as sketched in Figure 5.8. The systems available for these experiments are listed in Table 5.1. All systems listed in this table are connected to the faculty network via a Gbit Ethernet network interface card. The node *caps-sk1* runs Ubuntu 20.04. The other two nodes, *sense-edgeX*, run Ubuntu 22.04. The HDDs employed for storing sensor data use the ext4 file system [122]. The operating system is installed on a separate SSD for the two *sense-edgeX* systems. *caps-sk1* is used to run the data generator presented in Section 5.4.2 together with the stream adapters presented in Section 5.4.3. During the experiments, the machines are mostly used exclusively for our benchmarks, but this cannot be guaranteed; the same holds for the faculty network. The actual time series management systems are all deployed using container images. Both *sense-edgeX* systems use Docker 20.10 to run these containers.

We needed to take special care in case of all experiments that write data to (and read from) disk in this chapter. Details of this, together with baseline throughput numbers, are presented in Section 5.4.4.

[4]We use the network of the main computer science building at TUM for our experiments.

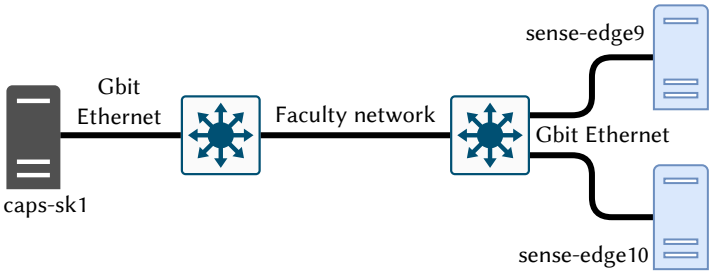


Figure 5.8 Sketch of the network topology for the experiments in this chapter.

5.4.2 Data Generator

For the evaluation of our system, we require a data source that is easy to control and can deliver a high data rate. Since one of the key features of our system is the ingestion of sensor data streams with a high stream rate or high schema cardinality (or both), the source for our experiments must be able to deliver such a high stream rate. Thus, to simulate varying input stream workloads for our system, we design and implement a data generator that implements the gRPC client streaming interface introduced in Section 3.2. This generator has the following functionality that can be controlled via command line interface (CLI) configuration parameters:

- **Stream schema** It is possible to configure the stream schema, and most importantly, vary the number of sensors for which a data stream should be generated. In all our experiments, we generate data for schemas that consist of a varying number of single-precision floating-point sensors (32 bit $\hat{=}$ 4 B per value). We restrict our tests to single-precision floating-point values since this is the main use case we are targeting. The stream schema of cardinality N consists of columns named `v0`, `v1`, ..., `v(N-1)` and will be named `teststreamN`, unless otherwise noted.

- **Sampling rate** The stream's average sampling rate can be emulated. This is done by repeatedly pausing stream generation for the required time after a slice spanning one millisecond of data has been generated. Although this rudimentary sample rate control mechanism can not perfectly mimic the behavior of real-time sensor analysis systems, it is sufficient for our benchmarking purposes. There is also the possibility to have an *unlimited* stream rate, which means that the generator simply generates data as fast as it can, without any artificial pauses. We use this unlimited stream rate extensively in our experiments in order to assess the throughput limits of the state of the art and our system.

- **Batch size** As discussed earlier, the client streaming interface supports batched input. As we will see in our evaluation results, this is an important feature for

efficient stream processing. For that reason, our data stream generator also supports an arbitrary batch size. This makes it possible to perform an in-depth design space exploration for varying batch sizes.

- **Payload coding** The actual payload data generated from the data generator uses IEEE 754 for floating-point data. This is the same as the internal representation of the generator and the adapter component (see Section 5.4.3). While it might be interesting—especially for low-bandwidth scenarios—to create and examine different on-the-wire representations for sending data, we leave this as future work. The various adapter implementations (described in the next section) targeting existing systems transform the binary encoding to the required text protocol for the experiments.

- **Setting parameters for HAQSE** HAQSE can be configured using a variety of parameters changing functionality and performance. The generator supports settings these parameters on a per-stream-basis. This makes it possible to execute a variety of parameter studies, as Section 5.5 will show.

- **Data generation statistics logging** The generator supports logging stream statistics in a regular, configurable interval. This makes it possible to retrieve performance characteristics during and at the end of stream generation. We use this as one method to analyze behavior over time of stream consumption as well as to retrieve benchmark summary results. Logging is implemented by printing the last set of statistics after a configurable time interval has passed. If there are no new statistics (because, e.g., no new batch has been sent since the last output), the current output iteration is skipped. We use an interval size of one second for all experiments in this chapter.

Especially the last point of this list is important for our experiments: The generator statistics log output serves as the base for all our evaluations. On the one hand, this allows reporting experiment summaries. On the other hand, it also allows analyzing behavior over time in the experiments.

Our data generator produces sequences of normally distributed random numbers. A certain amount of data is generated once and reused for subsequent invocations to ensure that random number generation is not the bottleneck in our system. For most of our tests, the actual values of the generated data do not matter. They only matter when writing compressed representations of the data to persistent storage and measuring the achieved throughput. For these experiments (Sections 5.5.6 and 5.5.7), we report the resulting compression rates.

For large payload sizes, we decouple gRPC message generation from the sending logic and execute both in a separate thread of execution. We synchronize this producer-consumer scenario with the help of a shared message queue. The generator is implemented in C++20 and uses gRPC 1.45.

5.4.3 Stream Adapters

For comparing HAQSE to state-of-the-art time series management systems (which we present in Section 5.6), we developed several sensor stream adapters. All these adapters consume the gRPC-protocol-based data stream as produced by the data generator (see Section 5.4.2). When a new stream is started, the adapters first connect to the respective time series management system and perform all system-specific initialization steps like authentication or table creation. The adapters then process the gRPC data stream and generate the protocol that is required by the respective client interface. They receive and buffer the incoming data stream (which may be batched in arbitrarily sized chunks) and produce batches of the respective client protocol. All tested time series management systems use a text-based protocol. The size of the output batches is configurable and independent of the adapter input batch size. Choosing a good protocol batch size depends on the respective database; it is essential to tune this parameter for achieving good input performance (see Section 5.6). We implement the adapter in C++ to ensure low runtime and memory overheads[5].

Specifically, we implement the adapter for InfluxDB, QuestDB, TimescaleDB and HAQSE. For HAQSE, the adapter would actually not be necessary, but we add it for fairness reasons and because it decouples data generation from actually sending sensor data to HAQSE. Our implementation for the other adapters uses two threads, synchronized via a shared queue. In one of the threads, the adapter generates the respective, text-based, batched input protocol message strings and puts them into the queue. In the other thread, one prepared protocol string is taken from the queue and sent using the respective transport protocol. This design ensures that the adapter component is not a throughput bottleneck in our setup since it decouples the compute-intense[6] protocol string generation from the actual sending.

For **HAQSE**, we simply forward the received gRPC stream by copying and reassembling the gRPC message.

For **InfluxDB**, we manually implement generation of input batches. Each input batch consists of several input rows, encoded using the ILP (cf. Listing 5.3). These ILP batches are then sent to InfluxDB via Hypertext Transfer Protocol (HTTP) POST requests. Our implementation uses curl [103] to send these requests.

For **QuestDB** we use the `c-questdb-client` package (version 2.1.1) for C++ clients. This generates the ILP[7] internally via API calls and handles sending the generated ILP batch via a Transmission Control Protocol (TCP) stream socket. In contrast to manually

[5] A first version based on Java client APIs for the respective time series management systems turned out to be problematic, especially for high-throughput configurations. One reason was a memory allocation problem in a dependency of the gRPC Java library, using direct byte buffers [87]. These byte buffers were not released fast enough, eventually leading to an out-of-memory error.

[6] The main processing step that makes string generation compute-intense is the conversion of floating-point data to text. Details are described when answering PQ3 in Section 5.4.5.

[7] On their website, the QuestDB authors state that data input via the ILP is the fastest available alternative.

generating the ILP in the case of InfluxDB, the QuestDB client library internally uses double-precision values for their API. As a result, the generated ILP is larger than the alternative with single-precision values we generate manually.

For writing data to **TimescaleDB**, we use the `libpqxx` library to connect to our instance of TimescaleDB and interact with it. We implement input via the `stream_to` [63] library interface, which showed the best performance results in our tests, especially when compared against regular SQL `INSERT` statements.

```
teststream4 v0=3.14159,v1=2.1828182,v2=1.414,v3=3.456789 59479627988000000\n
```

Listing 5.3 One example line for the ILP for a schema of four sensors (v1-v4) named `teststream4`. The last integer in the ILP indicates the timestamp. The ILP is used for InfluxDB and QuestDB.

Regarding the implementation, it is interesting to note that creating textual representations for IEEE 754 floating-point numbers is actually a compute-intensive workload. For generating the ILP for InfluxDB and for generating SQL statements for TimescaleDB, we use the `libfmt` [120] library. Internally, `libfmt` uses the dragonbox algorithm [62] for fast string generation of floating-point numbers. The `c-questdb-client` library is implemented in Rust [104] and uses the Ryū algorithm [2]. Both have been shown to be significantly faster than using standard serialization methods integrated into programming languages (like, e.g., `std::to_string(float)`) [2, 62]. For all three cases, input string generation, however, remains a computationally intensive task, as we will show in Section 5.4.5.

Data generation and any stream preparation always happens on a single node, the actual time series management systems are (in most cases) running on a different node. This setup tries to emulate the typical setting in industrial sensor data processing, where sensor processing systems send sensor data streams to a processing and storage node, located in the same network or in the cloud. It would also be possible to let the adapter component run on the receiving node (the same node as the database). However, we chose to keep this setup as sketched in Figure 5.7 since this is closer to real deployments. Furthermore, running the components like that reduces the amount of processing the receiving node has to perform.

5.4.4 Validating the Experimental Setup: HDD Performance

Several of our experiments write data to persistent storage. More specifically, HAQSE writes data to the locally installed HDDs. We chose HDDs in our experiments since they represent a typical storage medium in real industrial deployments. Our experiments in Section 5.5 show that, in many cases, throughput is limited by the write throughput of the installed HDD. As others have investigated before [36, 108, 135],

write throughput on HDDs is not constant across all physically available blocks. Blocks that are located on the outer part of the disk can be written faster, as the linear velocity of the outer lanes is faster than that of the inner lanes of the platters[8]. Since we do not want our experiments to be influenced by variations in HDD write throughput, we only use the first 100 GB (10 %) of the HDD. We achieve this by creating a partition that only uses the first 10 % of the logical block addresses. This partition is formatted with ext4. The blocks of this partition are located on the outer parts of the platters and deliver a relatively stable write throughput, as shown below.

Our experiment for determining the write throughput baseline is very simple. We just write to a file that is located on the 100 GB ext4 partition of the installed HDD and measure the time this takes. We do this with the help of dd [75], using the following command:

```
dd bs=1M count=$cnt if=/dev/zero of=/hdd/ext4-part/testfile conv=fdatasync
```

The output of this command shows the number of bytes written as well as the measured duration. Since we're mainly interested in throughput when writing large files, we vary `$cnt` from 50 to 10 000. This results in files with sizes from 50 MB to 10 GB. We execute the command 100 times for each of the tested sizes.

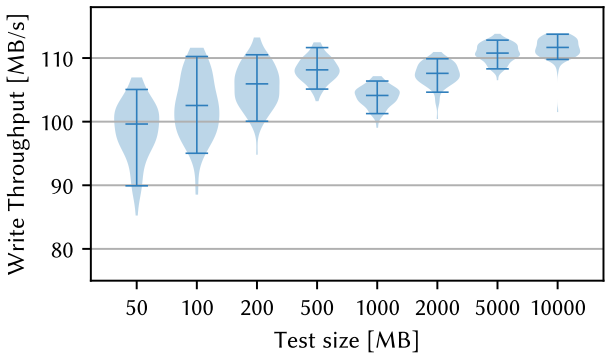


Figure 5.9 HDD write throughput test results. Each size is tested 100 times. The horizontal bars show 5th, 50th (median) and 95th percentiles.

The results of this experiment are plotted in Figure 5.9 and show several aspects. First, for this simple test, the achieved throughput slightly depends on the size of the written file. Similarly, the variance in results is larger for smaller test file sizes. Both of this can be explained by the constant overheads[9] associated with each test that just

[8] The angular velocity of a HDD is constant. As a result, the same length passes the write-head in a shorter amount of time on the outer lanes of the platters.

[9] These overheads are HDD seek times, file system overheads like file creation, and other constant processing overheads.

have a larger impact on the smaller files (the experiment for the smallest tests took less than a second). The rates are within a relatively stable range, both overall and even more within a particular tested size. The variance was considerably larger when utilizing the whole disk.

We want to highlight that the chosen experimental setup generates performance results that will not reflect behavior in real systems. Employing a (mostly empty) partition that uses only the first 10 % of the HDD represents the best case in terms of write throughput. We still perform our benchmarks in this environment. This is the only tractable way to create comparable results and exclude effects that stem from inconsistent HDD write throughput values. Our main goal is to assess the performance of HAQSE; this is only reasonably possible with reproducible experiments.

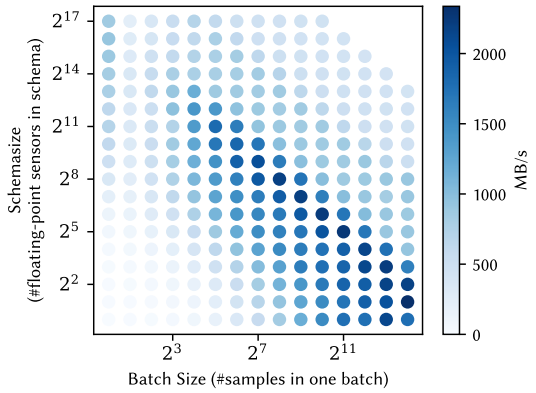
5.4.5 Validating the Experimental Setup: Generator & Adapters

Before we do any actual evaluations, we want to ensure that our experimental setup is suitable for measuring the ingestion throughput of the various existing time series management systems and HAQSE. For this, we perform a set of preliminary tests on the evaluation environment itself to ensure that it is not the bottleneck in our experiments. We want to answer these *preliminary questions* first:

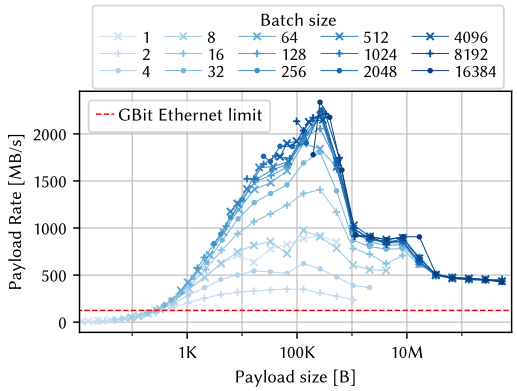
- PQ1: At what rate can our data generator generate data?
- PQ2: At what rate can the gRPC data stream be sent via Gbit Ethernet?
- PQ3: At what rate can the adapters for the state-of-the-art systems generate the respective input format?

By answering these questions, we establish a baseline that allows us to assess the results of the later experiments with respect to this baseline. This makes it possible to interpret the results of the HAQSE parameter studies accordingly. Furthermore, it enables a fair comparison between HAQSE and the state-of-the-art systems.

We perform all our measurements via the logging output of the generator. This approach is appropriate since this best reflects how real stream sources would perceive a time series management system as a stream sink. The generated stream receives backpressure from the system under test via gRPC, so the measured stream rate on the generator will eventually be identical to the ingest rate at the stream sink. This measurement methodology has the side effect that at the beginning of the experiments, the measured throughput values might be higher than what the time series management systems are able to sustain. For that reason, we average our measurements over a time period that is long enough for the measured rate to stabilize (specified for each experiment separately).



(a) Achievable data stream rates with data generator on a single node color-coded depending on schema and batch size. Missing data in the top right of the plot is due to a too large payload size of over 1 GiB.



(b) Achievable data stream rates with data generator on a single node. For reference, we also show the theoretical limit for Gbit Ethernet.

Figure 5.10 Analysis of generator speed on a single node.

PQ1: At what rate can our data generator generate data?

In order to answer the first question, we perform a set of experiments measuring the performance of our data generator. For this, we implement a throughput-test receiver server for the gRPC service interface that just drops all received stream messages sent by the generator. This gRPC test receiver can receive a maximum message size of 1 GiB and is implemented in C++. For this experiment, we let both the gRPC test receiver and data generator run on *sense-edge9* to avoid any network throughput limits. We then vary the schema size in powers of two from 1 to 2^{17} and the batch size from 1 to 16 384. For each combination of schema and batch size, we let the generator run for 30 seconds at unlimited generation rate and record the average throughput numbers after each run. The results for these experiments are visualized in Figure 5.10.

As shown in Figures 5.10a and 5.10b, the achieved throughput generally increases with schema and batch size up to a payload size of slightly more than 100 kB. Small payload sizes result in limited throughput values because of the larger impact of constant gRPC processing overheads. We tested this by sending the same message repeatedly, avoiding any non-gRPC-related overheads. This resulted in almost identical throughput numbers.

This already reveals one interesting property of HAQSE: small gRPC payloads will limit the achievable throughput just because data cannot be transmitted at higher rates with gRPC for such cases. Furthermore, it shows that small payload sizes should be avoided when high throughput is required, and that we need to choose large enough batch sizes to avoid a bottleneck in the ingestion protocol.

Medium-sized packages (around 100 kB) achieve very high numbers in this experiment. This can be explained by the reduced impact of constant processing overheads on the one hand. On the other hand, data sizes are still small enough to fit in the caches of the respective processing cores (256 KiB L2 cache per core). As payload sizes increase further, these caching advantages diminish. For medium-sized payloads, using larger batch sizes is beneficial for throughput. This is due to constant processing overheads in data generation, which have a negative influence especially for small batch sizes.

The rate reduction for larger payload sizes (>10 MB) is caused by two aspects. First, just generating large payloads causes the data rate to drop since CPU caches are not large enough to hold all that data anymore (the shared L3 cache has a size of 6 MiB). In addition to this, just sending larger gRPC messages alone (without the generation overhead) also causes the rate to drop. Parallelizing message generation from the actual sending is thus beneficial for large payload sizes: processing-intense sections are executed in parallel on different cores of the CPU.

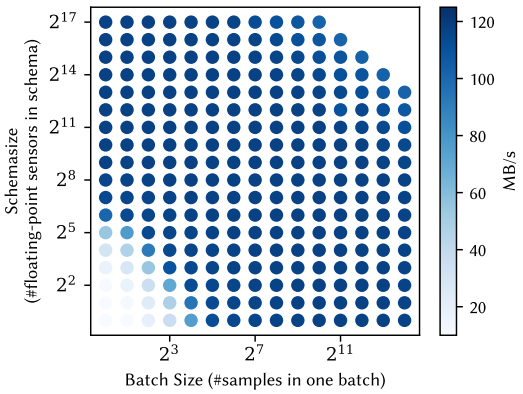
However, as it can be clearly seen in Figure 5.10b, the rate is greater than the theoretical limit of Gbit Ethernet for almost all cases. The only exception to this are cases where the payload size is less than approximately 300 B. We conclude that this is sufficient for most evaluated scenarios.

PQ2: At what rate can the gRPC data stream be sent via Gbit Ethernet?

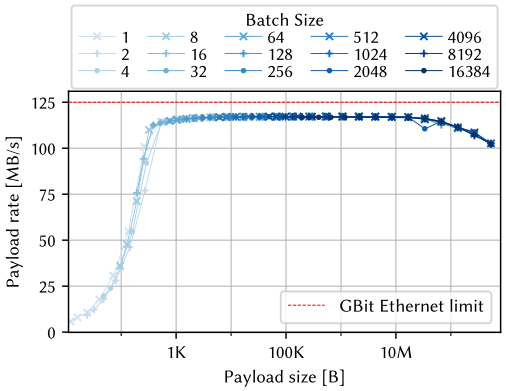
Next, we analyze how the data generation rate differs when source and target are located on different nodes connected over Gbit Ethernet. This is an interesting baseline when interpreting the performance results for HAQSE later on. So, in this paragraph, we repeat the experiment from above, but let the data generator run on *caps-sk1*. All other experimental parameters stay the same.

The results visualized in Figure 5.11 show the achieved rates when sending the generated samples over the previously specified gRPC protocol. The throughput saturates at roughly 117.6 MB/s. This is very close to the practical limit of the network: measuring the TCP bandwidth between the two nodes with `iperf` 2.0.13 [10] yields a maximum of 117.65 MB/s. For small payload sizes—resulting from either small schema or batch sizes—again, the rate is considerably lower due to constant overheads in sending and processing smaller packets. For large payload sizes, interestingly, the rate is also limited by gRPC—repeated sending of the same large message yields similar performance results.

As a consequence of this and the previous experiment, we conclude that the batch size should be chosen depending on the schema size when evaluating throughput of other components of the system. Ideally, the batch sizes should be chosen such that the payload size is around 100 kB.



(a) Achievable data stream rates with data generator sending data via Gbit Ethernet, color-coded depending on schema and batch size.



(b) Achievable data stream rates with data generator sending data via Gbit Ethernet. For reference, we also show the theoretical limit for Gbit Ethernet.

Figure 5.11 Analysis of generator speed when sending generated samples over Gbit Ethernet.

PQ3: At what rate can the adapters for the state-of-the-art systems generate the respective input format?

The input format and protocol depend on the respective database, as annotated in Figure 5.7. InfluxDB and QuestDB use the ILP for ingesting data, TimescaleDB inherits its ingestion functionality from PostgreSQL and uses SQL or a similar, text-based stream format. For InfluxDB, one or multiple ILP lines are sent via an HTTP POST request. For QuestDB, these ILP lines are sent through a TCP stream. For TimescaleDB, the pqxx library uses the PostgreSQL native wire format [124] to send SQL statements or the input stream format [63], the latter one providing a much better ingestion performance.

In order to assess the performance of our setup, we measure individual parts of the experimental data pipeline. For InfluxDB, we measure generation rate of ILP and how fast it can be sent via HTTP POST requests. For this, we create a dummy HTTP receiver in Python that just drops the received protocol messages. We batch 500 lines of the ILP into one POST request for this to avoid the protocol overhead to be too dominant for these tests because of small packet sizes[10]. For QuestDB, we do the same: we let the adapter generate the ILP string and send it through a TCP stream. On the other end of the stream, we use a dummy TCP receiver (again, implemented in Python), that just ignores all incoming data. Again, we asynchronously flush this buffer only after every 500 lines. For TimescaleDB, we test how fast SQL statements or stream protocol text can be generated. TimescaleDB requires the PostgreSQL native wire format for input. Since this is the only option to talk to PostgreSQL (and thus TimescaleDB), we consider this as part of the database and do not measure it separately in this section. For these

[10]We provide a more detailed evaluation of the dependence on the batch size in Section 5.6

experiments, we let the generator run for 60 s for each parameter combination. We show our results in Figures 5.12 and 5.13.

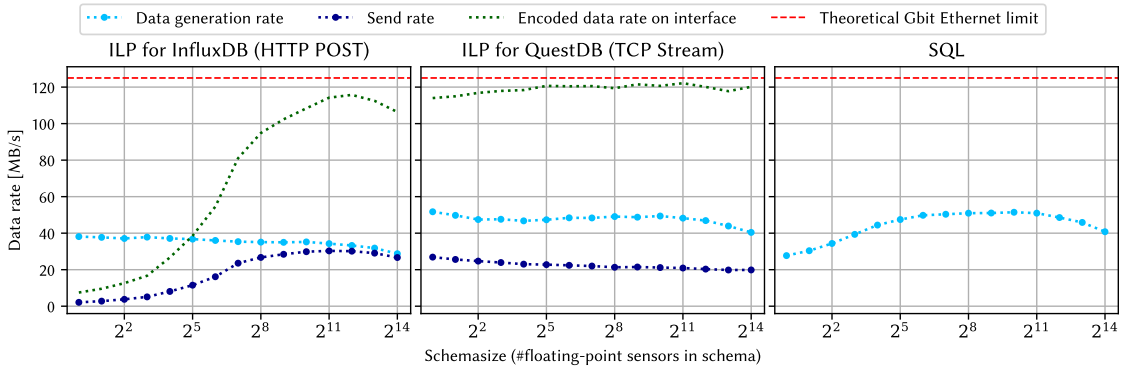


Figure 5.12 Data generation and transmission rates as a function of schema size for ILP (separately for InfluxDB and QuestDB) and SQL[11]. The green dotted lines represent the calculated rate on the interface, based on the raw rate (dark blue) and the encoding efficiencies from Figure 5.13 for the respective schema size. It is clearly visible that small schemas (and therefore small messages) are a bottleneck for the ILP protocol for InfluxDB. ILP for QuestDB (sent via TCP) is much more performant (for small schemas) and fully saturates the Gbit Ethernet connection for all tested schema sizes. For large sizes, the send rate for QuestDB is slower than the ILP version over HTTP used for InfluxDB. The reason is that the QuestDB version always sends double-precision numbers and thus has a less efficient textual representation.

These results reveal several interesting aspects of our adapter implementation and the employed protocol.

Even for only generating the protocol string, the adapters achieve raw data rates around 50 MB/s. Additionally, the generated string has an encoding overhead of a factor of 2.5–8, depending on protocol, schema size, and the actual data values (see Figure 5.13). Consequently, when data is being sent over Gbit Ethernet, sending the encoded values is limited by the throughput limit of Gbit Ethernet. This is indicated by the respective dotted lines in Figure 5.12. Especially for the *QuestDB send* case, the encoded rate almost fully saturates the network interface.

When including the actual send overhead, it also becomes evident that the protocol overhead of HTTP POST requests in case of InfluxDB is much higher than the approach chosen by QuestDB with a raw TCP socket, especially for small HTTP packet sizes.

Another thing to note is that the `c-questdb-client` library only provides an interface for sending double-precision data, thus unnecessarily increasing the size of the generated ILP string.

[11] As explained in the text, we do not measure the send rate for SQL. Consequently, the plot for SQL does not show *send rate* or *encoded data rate on interface*.

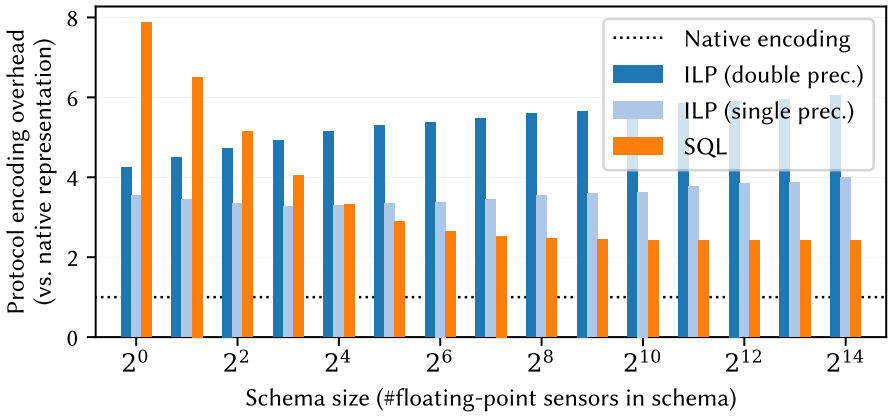


Figure 5.13 Encoding overhead as a function of the schema size for ILP (double precision, as generated for QuestDB), manually generated ILP (single precision, used for InfluxDB) and SQL. The overhead of the ILP is higher when generated with the QuestDB client library since it interprets and serializes values as double precision. In contrast to this, the manually generated ILP uses single precision, resulting in a more efficient encoding. Since batched ILP input is just a concatenation of individual ILP lines, there are no gains in efficiency with larger batch size. For SQL, the constant overhead of the `INSERT` command is higher (the plots show values for inserting a single row), but since no column names need to be specified, the efficiency improves with greater schema sizes. For SQL batched input, the constant overhead for small schema sizes would diminish further (not visualized in this plot).

Summary To summarize these preliminary experiments, we are able to show that data generation and string generation with rates of up to 50 MB/s (as raw payload encoding) are fast enough for saturating Gbit Ethernet network connections. As we will show in the next section, all these rates lie factors above the achievable rates of the existing time series management systems under test. This underlines that our experimental environment is suitable for comparing HAQSE fairly against the other time series management systems in Section 5.6.

5.5 Stream Consumption: HAQSE Parameter Evaluation

In this section, we extensively evaluate HAQSE with respect to its stream ingestion performance and how it depends on the various parameters. The goal of this section is to analyze the limits of each of HAQSE's components and how the achieved throughput depends on the parameters relevant to the respective component. This information can be used as a guideline during deployment to identify sets of parameters suitable for a certain application scenario. We also aim to identify parameters that do not have a

large impact on throughput: these parameters can be left at reasonable default values without negatively impacting performance.

In this section, all tests are performed with HAQSE running on *sense-edge9* (see Section 5.4). Since many processing steps in HAQSE deliver higher throughput values than Gbit Ethernet, we perform some experiments in two different setups. On the one hand, we let the generator run locally—on the same node as HAQSE—on *sense-edge9*. This makes it possible to identify the throughput behavior of the various components in HAQSE without the input stream being limited by Gbit Ethernet throughput. On the other hand, we let the generator run on *caps-sk1*. This shows how the respective component would behave in a more realistic scenario when data is received via Ethernet. In both cases, since we want to find out the throughput limits of HAQSE, the generator uses the unlimited generation rate option. The maximum possible generation rate depends on schema and batch size, as shown in Figure 5.10. We report the average ingestion throughput, calculated as the total number of bytes divided by the duration of the gRPC client streaming call. The total number of bytes is the number of sent samples multiplied by the schema size in bytes, including the 8 B timestamp (as specified in the input interface protocol, see Section 3.4.1). Since we only use single-precision floating-point values for these experiments, the size per sample is $(8 + 4 \times \text{schemasize})$ B. Streaming duration in the generator is measured as the difference between the wall-clock time before data generation starts and just after the gRPC streaming call returns. At that point, all data processing in HAQSE has finished.

For the rest of this section, we systematically evaluate the individual components of the ingestion pipeline, following the steps shown in Figure 5.1. Before that, we classify the various parameters into different categories.

5.5.1 Parameter Classification and Methodology

There are several parameters that may have an influence on sensor stream ingestion performance. First, there are *application-dependent* parameters that are defined by the application, like the number of sensors in the schema and the sampling rate. Second, there are *configuration* parameters in HAQSE that determine how a certain stream is processed, e.g., how many aggregation levels should be calculated for a certain input stream. Third, there are *internal configuration* parameters that determine how HAQSE processes data, which may depend on application-specific or other configuration parameters. These have no impact on the functionality of stream processing, but affect performance. One example of such a parameter is the temporary row group size or how many temporary files are compacted into one compressed file. These parameters can be chosen freely within the limits of the underlying hardware.

When evaluating the effect of a certain parameter on the resulting performance, we keep other parameters constant. Unless otherwise noted, we choose these constant parameters based on the results of other or preliminary experiments. More specifically, we use typical values that provided reasonable performance in these preliminary

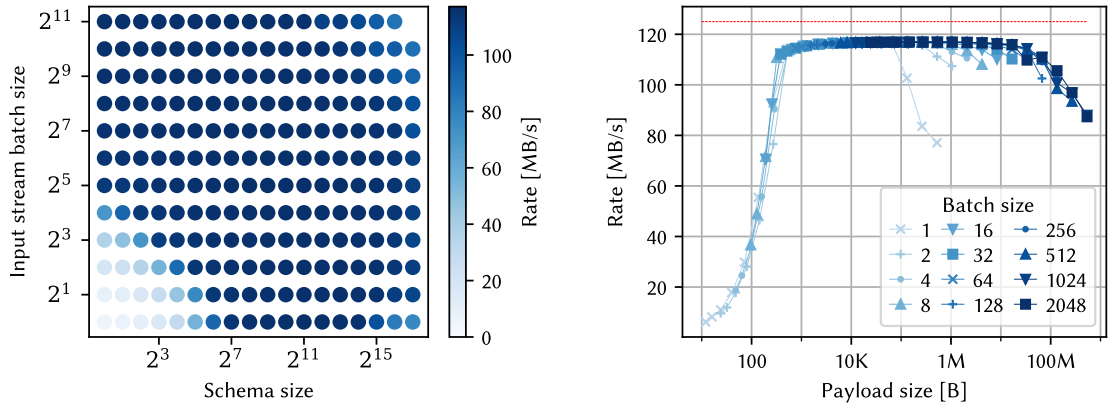


Figure 5.14 Throughput of only the layout transformation step when receiving data via the network interface.

experiments. Since the number of sensors is one of the most important parameters defined by the application, we perform all experiments in this chapter for a large range of varying schema sizes.

5.5.2 Layout Transformation Evaluation: Batch Size

In this experiment, we evaluate the first part of the pipeline: At what rate can HAQSE transform the layout of the incoming sensor stream? How does this depend on the schema size? How does the input batch size influence this?

For this set of experiments, we only perform the layout transformation step—no aggregations are calculated, no data from the temporary buffers is written to persistent storage, and consequently, no compaction or compression happens. Instead, the contents written to the temporary buffer are just ignored and overwritten by the next samples of the input stream. Consequently, only one such temporary buffer is required. We fix the temporary row group size to 1024: this means that every column is mapped to one or two full memory pages (4 KiB) on the utilized system[12].

We vary the batch size of the generator as well as the input schema size in increasing powers of two. We let each combination of parameters run for 10 s and report the average rate achieved at the end. It is not necessary to run each experiment for longer—there is no I/O buffering that might influence the throughput, and the observed rate does not change when running the experiments for a longer duration. The results of this experiment with the data generator running on *caps-sk1* (data being sent via Gbit Ethernet) are visualized in Figure 5.14.

[12] Columns containing 4 B values, i.e., all sensor columns (single-precision floating-point), consume one full memory page. Columns containing 8 B values, i.e., the timestamp column (64 bit integer), consume two memory pages.

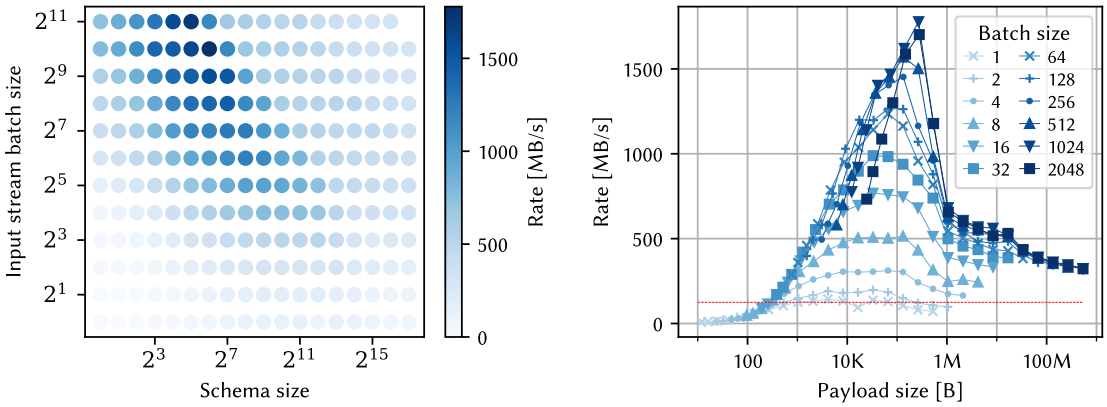


Figure 5.15 Throughput of only the layout transformation step when data is generated locally.

This figure shows that up to a payload size of roughly 0.5 kB, the throughput is heavily limited by the rate of the incoming data stream (cf. Figure 5.11). Then, up to a size of approximately 100 kB, the layout transformation step can sustain the full practical network bandwidth of 117.6 MB/s for all combinations of batch and schema sizes. For very large schema sizes (the right end of the various batch size lines in the right plot of Figure 5.14), the rates start to drop. This behavior can be explained by a disadvantageous but unavoidable access pattern to CPU caches, where recently used cache lines get repeatedly and systematically evicted from the various cache hierarchy levels. This is most problematic for small batch sizes, where only few values are updated in a particular cache line before it is replaced with the contents of another location in the temporary buffer. After that, for payload sizes larger than 10 MB, we observe a combination of two effects. First, the generator is not capable of delivering a high-enough data rate via Ethernet, as seen in Figure 5.11. Second, very large message payload sizes do not fit into operating system buffers. Since receiving data via gRPC and applying layout transformation in HAQSE are handled sequentially in this case, processing is slowed down further by the induced waiting times.

To investigate the potential performance of this step for faster networks, we repeat the same experiment, but let the generator run on the same node as HAQSE (both run on *sense-edge9*). This avoids the network bandwidth limitation apparent from Figure 5.14 and shows the potential performance of this processing step if there would be no network throughput limit. The results of this set of experiments are shown in Figure 5.15.

For small payloads—again, as seen previously in Figure 5.10—this shows that the throughput is limited, caused by the constant gRPC message processing overhead. Contrary to this, the throughput for larger payload sizes increases significantly. The behavior generally is very similar to the one shown in Figure 5.10, however, with a slightly reduced throughput peak. The reduction is relatively small, however, and the throughput achieved by HAQSE's layout transformation step is well above the Gbit

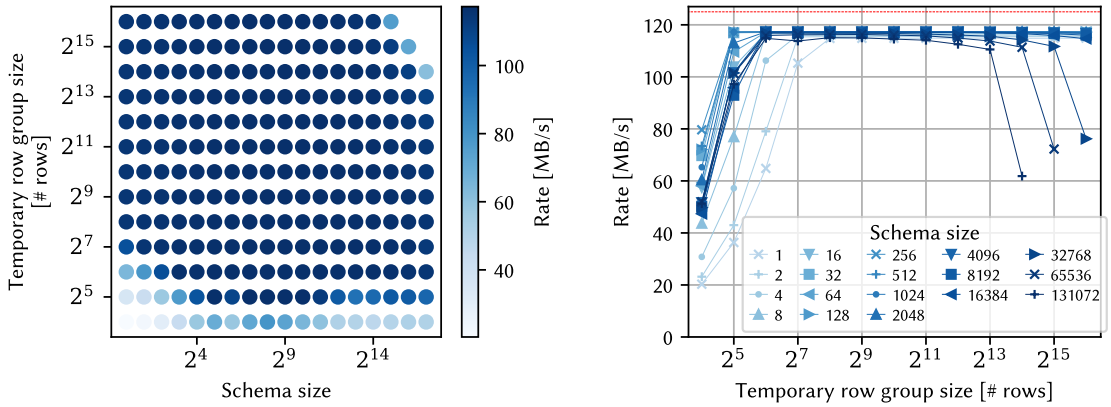


Figure 5.16 Behavior of HAQSE with varying temporary row group sizes, the sensor stream is received via Gbit Ethernet. The missing data points in the top right of the left plot indicate regions where HAQSE was not able to allocate a buffer of the requested size and thus prohibited streaming.

Ethernet limit for almost all cases where gRPC message generation and processing is not the limiting factor. The notable exceptions are observed for experiments with small batch sizes. Again, this is caused by the limit in gRPC transmission of small messages.

Summary To summarize the result in this section, we note that the overhead in small payload sizes limits the achievable throughput. Consequently, small schema sizes profit from sufficiently large input batch sizes. For the following experiments in this section, we thus choose a batch size such that the gRPC payload size is at least 10 kB. This ensures that the respective results are not negatively influenced by a limit introduced by a suboptimal batch size.

5.5.3 Layout Transformation Evaluation: Size of Temporary Buffers

We now evaluate the effect of differently sized temporary buffers. For these experiments, we fix the batch size at 64 since this provided a relatively stable rate across all schema sizes in the previous experiment. We vary the size of the temporary row group buffer in powers of two, and do the same for the schema size. All other parameters are kept identical to the previous experiment. The results for the experiments performed over network are shown in Figure 5.16.

These plots show several interesting aspects regarding the choice of the temporary row group size. First, for very small temporary buffers (product of small schemas and small temporary row group size) as well as for temporary buffers with less than 64 rows (lower left and bottom row of left plot in Figure 5.16), the throughput is significantly lower than for larger temporary row group sizes (≥ 256). This is due to constant

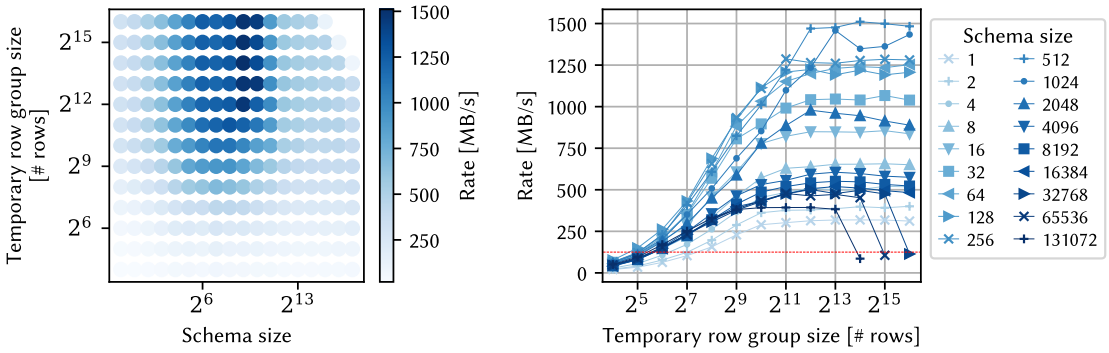


Figure 5.17 Behavior of HAQSE with varying temporary row group sizes, the sensor stream being generated locally. The missing data points in the top right of the left plot indicate regions where HAQSE was not able to allocate a buffer of the requested size and thus prohibited streaming.

processing overheads once a row group is fully buffered—HAQSE was not designed for such small row groups, as this would invalidate the advantages of the desired columnar layout.

Then, for very large temporary buffers, there is not enough physical memory to hold the complete buffer, so the operating system needs to start swapping out memory-pages to disk. This can be seen in the top right area in the left plot and the right-most data points for the three largest schema sizes in the right plot of Figure 5.16. Swapping out pages to disk reduces the rate to approximately 60 MB/s. Finally, as seen in the previous sections, the rate for larger schema cardinalities is slightly reduced (even if the buffer still fully fits into the available memory), especially for large temporary row group buffers.

Again, we are interested in actually seeing the throughput bottleneck in HAQSE without the Ethernet limitation. For this, as before, we repeat the experiment, but let the data generator run on the same node as HAQSE, *sense-edge9*. The results are plotted in Figure 5.17.

These plots show that increasing the temporary row group size generally improves performance across all tested schema sizes. Increasing it beyond a certain size, however, neither yields further benefits nor dramatically reduces the achieved throughput. The exact temporary row group size of this saturation point depends on the schema size. The maximum throughput is achieved with temporary row group sizes $\geq 2^9$ (largest tested schema size) or $\geq 2^{12}$ (medium-sized schemas). Increasing the temporary row group size beyond 8192 does not yield benefits for any of the tested schema sizes. Again, when physical memory is not sufficient to hold the complete temporary row group buffer, throughput degrades dramatically.

To still allow transforming input data to very large temporary buffers, it is possible to buffer data in memory-mapped files and let the operating system take care of writing data out to disk. Preliminary experiments in that direction indicate that this

works, but reduces throughput considerably. For streams that do not require such high data rates—especially streams originating from aggregation—this is an interesting alternative to consider. We postpone a more detailed analysis of this alternative way of buffering to future work; for now, we assume to have enough physical RAM available.

Summary To summarize the results of this section, we note that any choice of a temporary row group size of at least 256 is large enough for realistic scenarios where data is received via Ethernet. Values below that or above the physical limit of the underlying hardware should be avoided. Increasing the value beyond a schema-dependent maximum value ranging from 512 to 4096 does not yield any throughput benefits for this particular step in the processing pipeline. A schema-independent default value of 1024 seems like a reasonable choice, considering the experimental results.

5.5.4 Persisting Data to Temporary Files

In all experiments so far, HAQSE processed data purely in-memory, no data has been written to persistent storage. In this section, we analyze how writing data from the temporary buffer to temporary files behaves in HAQSE. For this, we perform several experiments for different combinations of parameters. When writing temporary row groups to persistent storage, all writes are buffered in the operating system page cache. We keep the operating system defaults for the virtual memory subsystem, which is responsible for writing back dirty pages to disk. Since we do not manually force data to disk via `fdatasync()` or a similar method (cf. Section 5.2), measuring the throughput requires longer running experiments[13]. So, in contrast to the previous experiments, we now let each combination of parameters run for two minutes. We use a batch size of 64 for the generator.

We assess the impact of the following parameters:

- First, we let HAQSE write out the temporary buffers *directly* in raw form, without any additional processing or transformations. Raw output is implemented using a single `write()` call. This appends the complete temporary buffer to a file. This test yields a baseline that we use to assess the overhead associated with writing data to Apache Arrow IPC format files. When HAQSE writes data in the Apache Arrow IPC format, we measure the accumulated overhead stemming from two sources: the processing overheads in the Apache Arrow library and the slightly increased amount of data (caused by the additional metadata information) required to be written.

[13] It takes some time until there are enough dirty pages in the page cache in order for the Linux kernel flusher thread to start writing back data to the actual storage device [128]. Only when no more free pages for writing data to are available, and the buffers of the storage device are filled, the backpressure coming from write rate limit of the storage device is measurable.

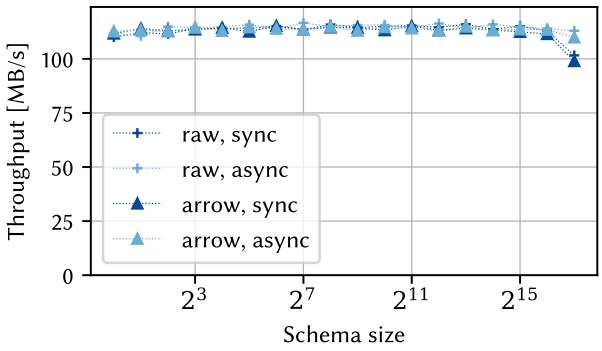


Figure 5.18 Comparison between raw and arrow serialization as well as synchronous and asynchronous writing: there is almost no differente between the various variants.

- Second, we examine the impact of using a second temporary row group buffer. This second buffer makes it possible to write out data to temporary files asynchronously in a background thread while ingestion happens in the other buffer.

- Finally, we vary the temporary row group size. This enables us to see the impact of different temporary buffer sizes on write throughput.

Figure 5.18 shows the difference between Apache Arrow IPC format and writing the temporary row group buffer in raw form. This is done for both the synchronous (one temporary buffer) and the asynchronous (two temporary buffers) case. The row group size is fixed to 1024. This shows that having two buffers and asynchronously writing out data has no measurable impact on the achieved throughput for most cases. The only exception is when writing a schema size of 2^{17}, where the performance of the synchronous case slightly decreases. Additionally, writing out data in Apache Arrow IPC format does not have a huge impact on the measured throughput. This shows that using the Apache Arrow IPC format is an appropriate format choice for persisting data to temporary files. For larger schema sizes, the slightly reduced performance is within measurement inaccuracies.

Figure 5.19 shows the impact of using an increased temporary row group size. As the plot shows, up to a certain schema size, there is no measurable difference between different row group sizes. For larger schemas, using a larger row group size of 16 384 leads to smaller average throughput numbers. Using a second temporary buffer and writing data to disk asynchronously, this effect is only visible for larger schema sizes.

This result indicates that having a larger total temporary row group (product of schema size and row group size) is detrimental to write throughput. This can be avoided by asynchronous write back or simply by choosing smaller row group sizes.

The experiments so far were performed with the data generator running on *caps-sk1*. We repeated the experiment with the data generator running locally on *sense-edge9*. The results look almost identical, which reveals that the throughput is really limited by

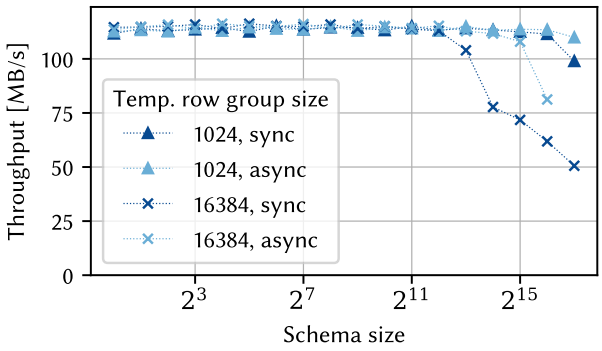


Figure 5.19 Influence of temporary row group size on throughput when persisting files to Apache Arrow IPC format files: large temporary row group sizes lead to reduced performance.

the HDD write throughput, not the Ethernet limit on the input interface (whose limit is only slightly above that of the HDD).

Summary As long as the chosen temporary buffer sizes remain within reasonable limits, our results indicate that there is not much benefit in using double buffering in the application layer. Instead, operating system and library buffering is enough for both input and output. For input, data is buffered in the network stack of the operating system as well as in the gRPC library. For output, data is buffered in the operating system's page cache. Copying data from and to these buffers is orders of magnitude faster than disk throughput and Gbit Ethernet. Consequently, as long as the temporary buffer is reasonably sized[14], the existing operating system and library buffers and threads are sufficient. A value of 1024 for the temporary buffers also works well for this part of the pipeline and thus confirms the results from Section 5.5.3.

5.5.5 Batch Aggregation Isolated

In this section, we evaluate the batch aggregation processing step and how the various configurable parameters influence the achieved throughput. We systematically evaluate the various parameters that have an influence on the performance in the following subsections. To ensure that we only measure the impact of the aggregation-related functionality, we do not write out any data to persistent storage and use a single temporary buffer for this set of experiments, as in Sections 5.5.2 and 5.5.3. Again, we report the average throughput measured in the generator. For each set of parameters, we let the experiment run for 60 seconds.

[14]The temporary buffer size must be large enough so that constant overheads can be amortized. It must also be small enough so that the required memory copy is fast enough to not block the application for too long.

In comparison to all other parts of the processing pipeline, the aggregation step has a relatively high computational intensity, performing many floating-point operations. For that reason and due to thermal throttling, during the experiment time, the CPU frequency was not fully stable. After the 60 seconds duration of the experiments, though, the reported average throughput stabilized and the reported values were reproducible reliably.

As discussed in Section 5.3.2, the stream input batch size has an influence on how often aggregation is triggered, so we first analyze the impact of this parameter. After that, we vary the various aggregation-specific parameters and analyze their behavior on throughput. We analyze the impact of the following parameters:

- Number of aggregation levels
- First aggregation factor
- Subsequent aggregation factors
- Temporary row group size

We let the generator run locally since, as apparent from the results, the throughput is mostly considerably larger than Gbit Ethernet limit.

Influence of Input Batch Size

To understand the impact of varying input batch sizes on the possible aggregation throughput in HAQSE, we first fix all aggregation-specific parameters and vary the input batch and schema size. We compute aggregations for a single aggregation level, setting the aggregation factor to 64. The temporary row group size is set to 1024. We vary the batch size as powers of two from 1 to 1024 and perform tests for schema sizes from 1 to 32 768 single-precision floating-point fields (again, using powers of two). The results of this experiment are visualized in Figure 5.20.

The plots show that aggregation throughput heavily depends on the input batch size. This can be explained by the fact that after every received batch, aggregation is potentially performed for the new data. In particular, this means that the constant overhead for triggering aggregation is accumulated as the batch sizes become smaller. Depending on the schema size, the throughput improves up to a certain batch size. For small schema sizes, throughput is improved for batch sizes as large as 1024. For larger batch sizes, performance begins to diminish again. This result shows that there are several possibilities to improve the performance of the batch aggregation component. First, it is apparent that the constant overheads play an important role. Thus, if possible, these overheads should be identified and reduced. Furthermore, it would also be possible to decouple the calculation of aggregation batches from the input batches, i.e., to calculate aggregation only after a minimum number of samples or after a certain time. Generally, as long as the batch size is small enough (less than 1024 in our experiments), throughput is highest for medium-sized schemas (32–256).

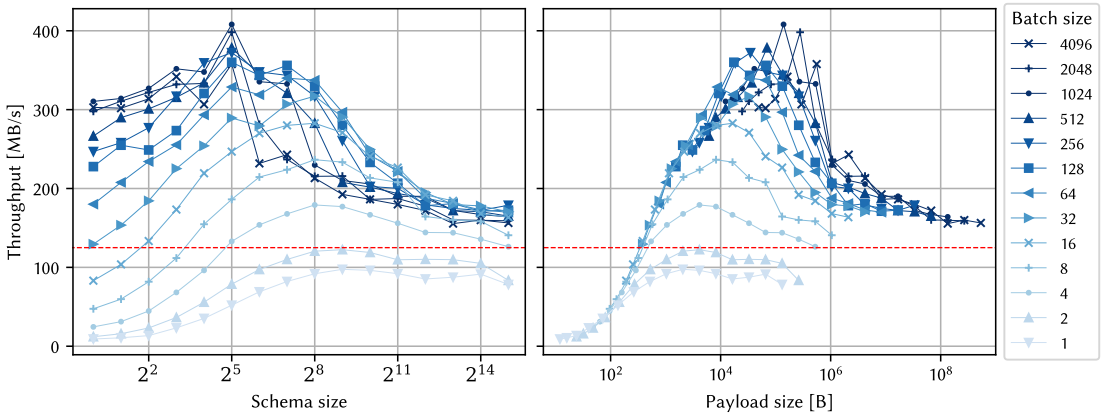


Figure 5.20 Influence of batch and schema size on achieved throughput of batch aggregation. Both plots show the results of the same experiments, once as a function of schema size (left), and once as a function of payload size (right).

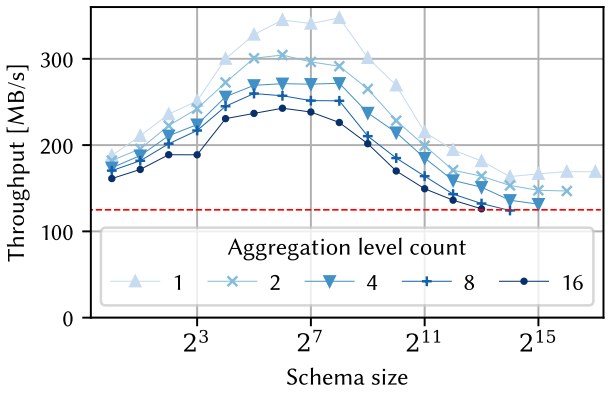


Figure 5.21 Impact of aggregation level count on the aggregation throughput.

For all subsequent experiments in this section, we choose a batch size of 64.

Impact of Number of Aggregation Levels

For this experiment, we vary the number of aggregation levels. We fix the first aggregation factor to 64, all subsequent aggregation factors are 2. The temporary row group size is 1024. The results are plotted in Figure 5.21.

This plot shows that increasing the number of aggregation levels slightly impacts overall throughput, while the general behavior across different schema sizes remains similar. Compared to the impact of a smaller or larger schema size, throughput is much less impacted by changing the number of aggregation levels. The additional impact of increasing the number of levels decreases for a larger number of aggregation levels. This is expected since the amount of work increases only logarithmically with additional aggregation levels.

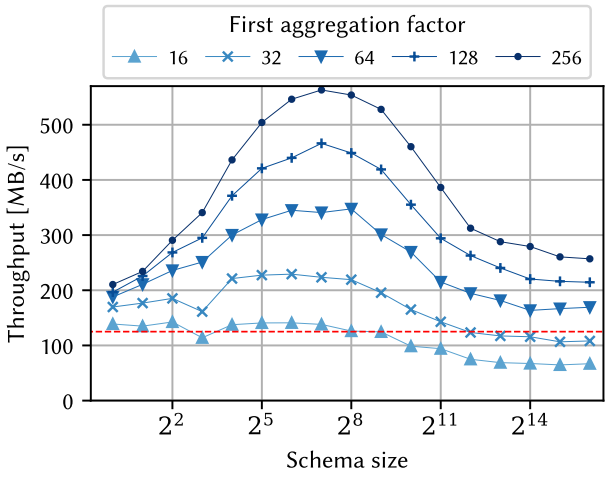


Figure 5.22 Impact of different values for the first aggregation factor.

A higher number of aggregation levels also requires more temporary buffers. For that reason, and since for large schema sizes the available physical memory was not sufficient to allocate the required buffers, there are no results for large schema sizes and many aggregation levels in Figure 5.21.

Impact of First Aggregation Factor

For this experiment, we vary the first aggregation factor, i.e., how many samples are contained in an aggregation window of the first aggregation level. In the previous experiment, this was 64; now, we vary that factor in powers of two from 16 to 256. We fix the number of aggregation levels to one. As before, the temporary row group size is 1024. The results are plotted in Figure 5.22.

The plot shows that the influence of the first aggregation factor is significant on the achieved throughput. This is expected since the actual computation of aggregations happens in batches of the respective factor. This means that for smaller factors, this calculation happens more often and produces more aggregation data. The behavior is consistent across different schema sizes, but for very small first aggregation factors (16 and 32), there is no real peak anymore for medium-sized (32–1024) schemas.

Impact of Subsequent Aggregation Factor

Next, we vary the other aggregation factors, which determine the size of the aggregation window for subsequent aggregation levels. For these experiments, the first aggregation factor is set to 64. We use a total of six aggregation levels and, as above, a temporary row group size of 1024. The results of these experiments are shown in Figure 5.23.

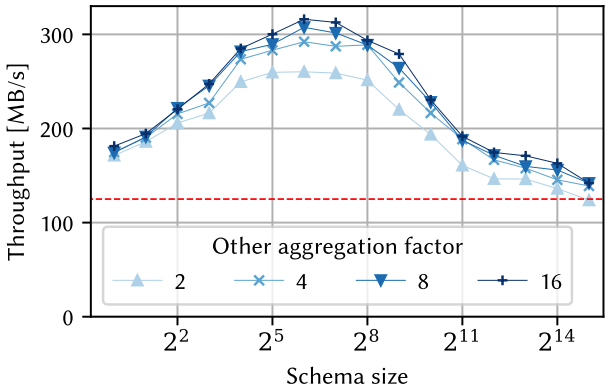


Figure 5.23 Impact of other aggregation factors on the achieved throughput.

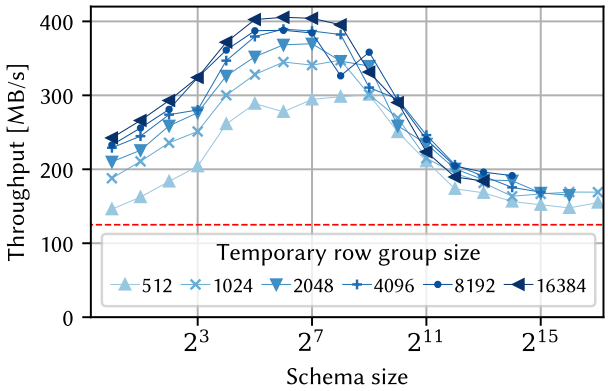


Figure 5.24 Aggregation temporary row group size.

This plot shows that the throughput is only slightly impacted by varying the other aggregation factors. This means that this factor can be chosen more or less freely without too much negatively impacting performance[15].

Temporary Row Group Size

In this experiment, we evaluate the impact of the temporary row group size on the aggregation performance. We use a temporary row group size of at least 512 since smaller row group sizes might have a negative impact on the achievable throughput, as discussed previously (cf. Figure 5.17). We calculate one aggregation level with a factor of 64 and increase the temporary row group size in powers of two, up to 16 384. The results are shown in Figure 5.24.

[15] If the same range of aggregation resolutions should be covered and a smaller factor between aggregation levels is chosen, a greater number of aggregation levels is required. This, as discussed previously, does have an impact on the achievable rate (cf. Figure 5.21).

This plot shows that larger temporary row group sizes generally yield higher throughput rates. For larger schema sizes (starting at roughly 512 in our experiments) the throughput for different row group sizes is converging. This result is different from the observations from Section 5.5.3, where throughput did not increase noticeably for row group sizes beyond a value of 4096. Nevertheless, a value of 1024 that was previously (Sections 5.5.3 and 5.5.4) found to work well also provides good performance numbers for aggregation.

Aggregation Summary

While the individual subsections contain all the details, we want to summarize the results of our aggregation experiments here.

- The chosen **batch size** has a high impact on the achieved throughput: especially small batch sizes of less than ten are prohibitive in terms of performance. This is partly due to gRPC input, but partly also due to the fact that aggregation processing is coupled to the input batch size. Batch sizes larger than 256 do not yield any benefit but instead, lead to slightly reduced throughput values.

- The **number of aggregation levels** to compute only has a moderate impact on throughput. It should be noted, however, that increasing the number of aggregation levels has a high impact on the memory footprint of the application.

- In contrast to this, the first **aggregation factor** has a large impact on the achieved throughput. This property needs to be taken into account when configuring HAQSE for a certain use case. In contrast to this, other factors have almost no impact on the performance.

- In contrast to previous experiments, a higher **temporary row group size** is beneficial for aggregation performance. However, also considering memory footprint and the previous results, 1024 manifests as a reasonable choice for the temporary row group size.

- Generally, medium-sized **schemas** (16–512) work best. This leads to a bell-shaped performance curves in many of our result plots (Figures 5.21 to 5.24)

When the batch size is chosen reasonably, *almost all* measured throughput values lie above the theoretical Gbit Ethernet throughput limit. This indicates that generally, aggregation is not expected to have a huge negative impact on the overall performance of sensor stream ingestion in HAQSE.

5.5.6 Compaction Isolated

In this section, we evaluate the throughput of the compaction process. As we want to understand the influence of the various parameters on the compaction process in an isolated fashion, we design a separate set of experiments for this part of the evaluation. These experiments mimic how the compaction process would be executed in HAQSE. They avoid other processing tasks that would run concurrently in HAQSE like, e.g., layout transformation or concurrently writing to (the next set of) temporary files, so these experiment represent the best case of processing compaction.

All processing logic in these experiments is performed in a single process and consists of the following two steps: first, temporary Apache Arrow IPC format files are created; then, the contents of these files are compacted and written to a single Apache Parquet file. For each benchmark run, we generate sensor samples with the same logic that is used in our data generator (cf. Section 5.4.2). We create temporary files in the same way and order HAQSE would create them and write the generated data to these files. This ensures the operating system caching behavior is similar to the real case. This means that when compaction starts—depending on the employed parameter values—some parts of the to-be-compacted files are still available in the page cache in physical memory. After that, we measure the execution time of the compaction function, reading the data just written to temporary files and writing it to compacted Apache Parquet files. At the end of the compaction, data for the compacted file reaches the disk by a call to `fdatasync()` (like for normal operation inside HAQSE). Thus, the measured duration includes all necessary disk write times.

For this set of experiments, we compress sensor data using the following combination of parameters (cf. Section 4.4.6): floating-point sensor values are compressed using Byte Stream Split and zstd, timestamps use the Apache Parquet `DELTA_BINARY_PACKED` encoding [115]. Unless otherwise noted, the total achieved compression ratio is around 1.90. We report throughput rates as raw number of bytes to write divided by the execution time of the compaction call. For that reason, and because data is compressed before writing it to disk, some numbers reported here are significantly higher than what we presented in Section 5.4.4.

We evaluate the behavior with respect to the following parameters:

- **Total data size**. A larger total size results in more data being written to temporary files, subsequently read from these files and finally written to compacted Apache Parquet files. With a growing total size, the share of temporary file content that can be kept in the operating system page cache is reduced. This means that the parts not available in the page cache need to be read from the actual persistent storage medium on compaction. In contrast, for smaller total sizes, the page cache might be able to fully contain the required data pages in physical memory so that fewer bytes actually need to be read from disk.

- **Temporary file count**. Assuming a fixed total size and temporary row group size (and thus, total number of temporary row groups), there are different ways how these temporary row groups can be distributed to temporary files. This experiment shows how well our system handles cases with few huge files or a lot of very small files.

- **Schema size**. Larger schema sizes result in larger strides reading column data from subsequent row groups of temporary files. This is decreasing memory locality and makes file system prefetching less effective when reading temporary files.

- **Temporary row group size**. As above, a smaller temporary row group size might make disk prefetching less effective, as data from more temporary row groups needs to be loaded for filling a single compacted column. This depends on the chosen schema size.

- **Compacted row group size**. Larger compacted row group sizes reduce the constant processing overheads of filling and compressing the compacted column chunks. On the other hand, data from more temporary row groups must be read, potentially leading to an increased pressure on the page cache.

We evaluate each of the listed parameters in the following subsections, also considering dependent or related parameters for each. We repeat each combination of parameters three times (except for the experiments for the *total size* tests, which are repeated five times). The measured throughput values are distributed similarly to the results in Section 5.4.4. To preserve clarity, we use the median of our measurements for visualization in the following plots. This is robust to outliers and thus is a reasonable representative value for the measurements.

Total Size

We start by analyzing the impact of the total data size to compact. For this experiment, we use a schema size of 128, a temporary row group size of 1024 and a compacted row group size of 65 536. We perform the experiment for three different temporary file count values: once with having all temporary data in a single file, once with four temporary files, and once with 16 temporary files. We start the experiments for a total number of temporary row groups of 64 and increase that number in powers of two up to 32 768. This corresponds to a total uncompressed data size ranging from 32.5 MiB to 16.25 GiB. The results are plotted in Figure 5.25.

This plot show that for a small enough total size to compact, the achieved throughput is substantially higher than for very large total sizes ($\geq$ 8 GiB). The reason is, as mentioned before, that for a total data size that does not fit in the available physical memory of 8 GiB, data is written to and read back from the storage disk. Apart

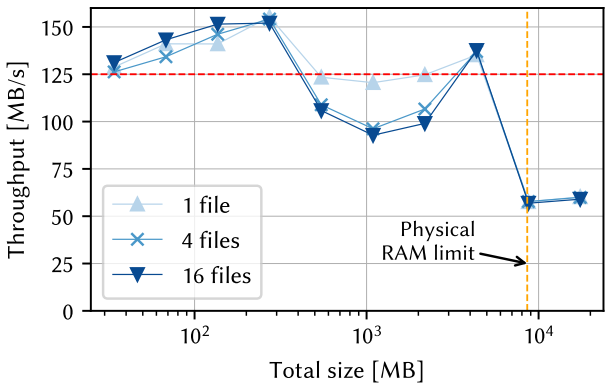


Figure 5.25 Compaction performance as a function of total size to compact.

from that, the throughput is highest for a medium total size of approximately 250 MB with 154 MB/s. The actual HDD write throughput that is necessary for this is around 80 MB/s as data is compressed with a rate of roughly 1.92. The number of temporary files compacted to one compressed Apache Parquet file does not have a significant impact on the general behavior. For sizes of around 1 GB, using only a single file is beneficial compared to using multiple files. We analyze the impact of the number of temporary files next.

File Count

We now measure the impact of the number of temporary files on compaction throughput. Again, we fix the schema size to 128, use a temporary row group size of 1024 and a compacted row group size of 65 536. As the total size has a significant influence on the performance, we perform the test for four different total sizes: 250 MB, 1 GB, 5 GB and 10 GB. We vary the number of files in the following way: we start with one file, such that all necessary row groups are contained in a single temporary file and increase the number of files in a roughly exponential fashion. We end with as many files as row groups, such that there is only one row group per temporary file. The results of these experiments are shown in Figure 5.26.

This plot shows a few interesting properties. First, for very large temporary files (number of files is very low, left part of plot area), the performance is very good. This can be attributed to the intelligent management of row groups in the temporary files that are already fully read. As discussed in Section 5.2.3, we use `madvise` to tell the operating system when certain page ranges are not needed anymore. This makes it possible for the operating system to keep only a small part of the temporary files in the page cache. A previous version that did not implement this performed poorly for very large files, as it put high pressure on the page cache. Then, except for a total size of 1 GB, performance is relatively stable. Only for a very large number of files and a large total size of 10 GB (where each temporary file contains a diminishing amount of

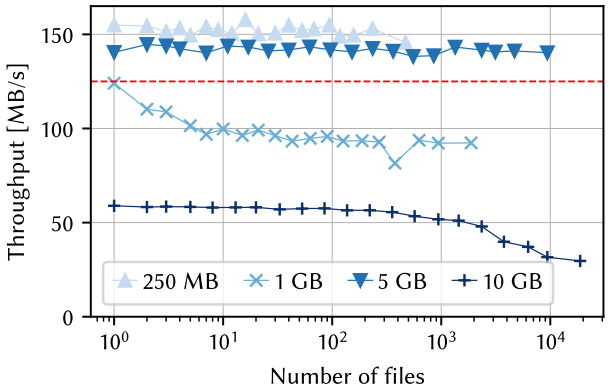


Figure 5.26 Compaction performance as a function of file count.

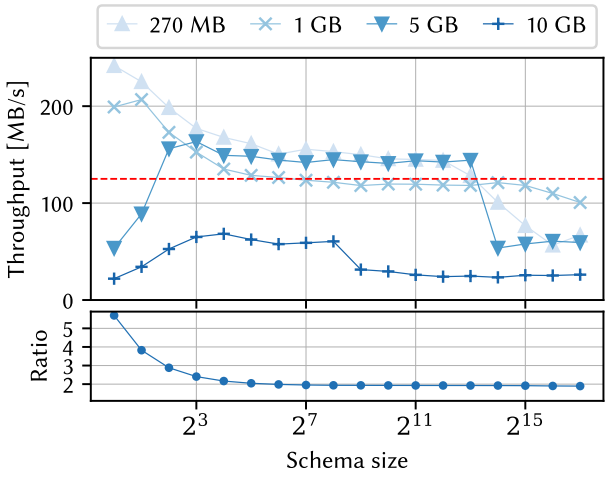


Figure 5.27 Compaction performance as a function of schema size.

data), the achieved throughput goes down noticeably. This can be explained by the accumulated constant overheads of opening many small files from disk.

Thus, it is advisable to keep the number of temporary files to compact small for deployments if ingestion throughput is important.

Schema size

Being one of the main parameters determined by the application, the schema size influences performance considerably, as shown by the previous experiments. We also assess the impact of this parameter on the compaction performance. We use a temporary row group size of 1024, a compacted row group size of 65 536 and use a single temporary file. We repeat the experiments for four different total sizes: 270 MB, 1 GB, 5 GB and 10 GB. The results are shown in Figure 5.27.

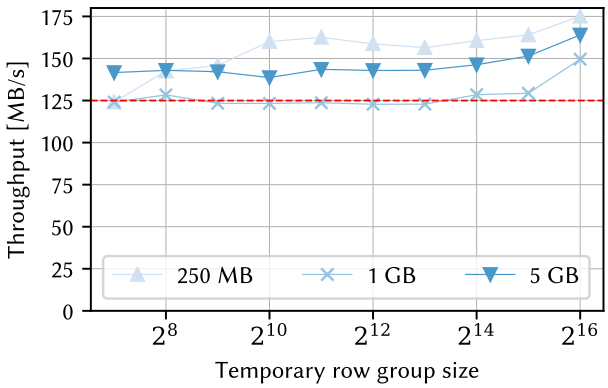


Figure 5.28 Compaction performance as a function of temporary row group size.

This plot shows—as expected when looking at the results of previous experiments—that larger total sizes can not be compacted as fast as smaller total sizes. This is due to data being read from persistent storage instead of RAM during compaction. Interestingly, for schemas larger than 256, the achieved throughput is again reduced quite a bit. This can be explained by prefetching mechanisms of the operating system not being as effective for the access pattern exhibited in these cases.

For total sizes that fit into the available physical memory, this cannot be observed for schemas of similar size. However, for schemas larger than 8192 fields, a similar effect can be observed. For schemas smaller than eight fields, where the total size of a compacted row group in our experiments is less than 3 MB, the achieved throughput is also slightly more limited in the 5 GB and 10 GB case. Contrary to this, for all experiments with a smaller total size, the achieved rate is considerably higher for small schemas. This can be explained by the efficient encoding of timestamp data. The difference between consecutive timestamps created by our data generator is always exactly Δt. For that reason, these timestamps can be compressed very efficiently using Apache Parquet's `DELTA_BINARY_PACKED` encoding. Since the timestamp column represents a large share of the total data size for small schema sizes, the total compression ratio is higher for such small schema sizes. This is shown in the bottom plot of Figure 5.27. As the amount of data that needs to be written to persistent storage is reduced drastically, and since we report throughput values for the uncompressed size, these values are higher for small schema sizes.

Temporary row group size

For measuring the impact of the temporary row group size, we fix the compacted row group size to 65 536. We vary the temporary row group size from 128 to 65 536. The schema size is fixed at 128. We repeat the experiments for three different total sizes: 250 MB, 1 GB and 5 GB. We always use a single temporary file. The results are shown in Figure 5.28.

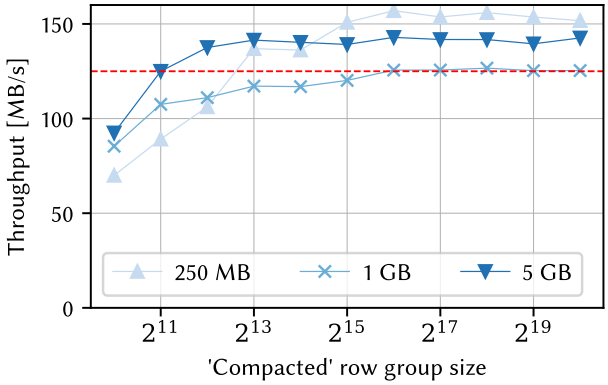


Figure 5.29 Compaction performance as a function of compacted row group size.

It can be seen that the temporary size does not have a huge influence on the overall throughput. Only very small temporary row groups slightly reduce the achieved performance. When the temporary row group size is closer to the compacted row group size of 65 536, the achieved throughput is best. For all tested parameter combinations, the throughput is higher than what Gbit Ethernet can achieve in our experimental setup. Consequently, regarding compaction, the choice of temporary row group size has only a minor influence on achievable throughput. The previously determined default temporary row group size value of 1024 also works well for compaction.

Compacted row group size

Finally, we evaluate the impact of the compacted row group size on throughput. For this, we fix the temporary row group size to 1024 and set the schema size to 128. We vary the compacted row group size from 1024 to 1 048 576. We repeat the experiment for the same total sizes as in the experiments from the previous section: 250 MB, 1 GB and 5 GB. The results are visualized in Figure 5.29.

The plot shows that except for very small compacted row group sizes of up to 8192, where constant processing overheads have a noticeable impact, throughput is largely independent of the chosen compacted row group size. As seen before, the performance varies for different total sizes to compact. Varying the compacted row group size has only a minor influence on the resulting throughput.

Summary of Isolated Compaction Experiments

In this section, we summarize the insights gained from all the compaction experiments.

- One of the most important aspects regarding compaction performance is to keep the *total data size* to compact *small enough*, i.e., below the limit of physical memory. The problem for larger total sizes is caused by the currently implemented

compaction trigger logic in HAQSE: compaction is only triggered when all data to compact is available. Alternatively, it would be possible to trigger the compaction process for one compacted row group as soon as enough data for one such compacted row group is available. This would increase the chances that data is still fully available in the operating system's page cache. We defer this optimization to future work.

- The number of files to compact should be kept small. This ensures that the constant overheads associated with opening files amortizes over the amount of data in one such file. Similarly, our results show that for compaction, temporary files cannot be *too* large, as long as they are smaller than the physical memory limit.
- The compacted row group size should be large enough. A size of 65 536 worked well for all our tested cases. Increasing it further did not provide major performance improvements, but also did not degrade performance.
- Similarly, the temporary row group size should not be chosen too small. All our experiments with a temporary row group size of at least 1024 worked well. Sizes above that all work well regarding compaction in our experiments.
- Even though it is determined by the application, we want to note that large schema sizes (more than 8192 columns) reduce throughput considerably (depending on other parameters).

Overall, the results presented in this section show that the compaction step can be executed fast enough for data received via the Gbit Ethernet interface for many configurations. For completeness, it should be noted that in real deployments, compaction performance will not be as stable and the resulting throughput values may be considerably lower. As mentioned in Section 5.4.4, this is due to HDDs not being able to deliver stable performance for all available blocks. We want to note again that compressing data is highly beneficial, increasing the throughput beyond that of the employed HDD.

5.5.7 Full Pipeline Test

Up to this point, we have tested all ingestion-related components of HAQSE individually. In this section, we perform experiments that test the ingestion pipeline as a whole. This shows the effective overall throughput HAQSE can deliver in our experimental setup. We provide experimental results for a variety of scenarios and evaluate how the individual parameters influence the resulting performance. For all tests in this section, the data generator runs on *caps-sk1*. This limits the maximum achievable performance to approximately 117.65 MB/s (cf. Section 5.4.5).

From the previous experiments, we derive a default set of parameters. The temporary row group size is 1024, and we use only a single temporary buffer. For compaction,

we use a compacted row group size of 16 384 and target a compacted size of 200 MB. Aggregation is disabled, and by default, we use an input batch size of 32. For each experiment, we then vary one of these parameters to assess its influence on performance. Each combination of parameters is run for ten minutes.

Input Batch Size

We start by investigating the impact of the input batch size on ingestion throughput. We use a small set of different batch sizes to show the behavior of HAQSE. The results of our experiments are plotted in Figure 5.30.

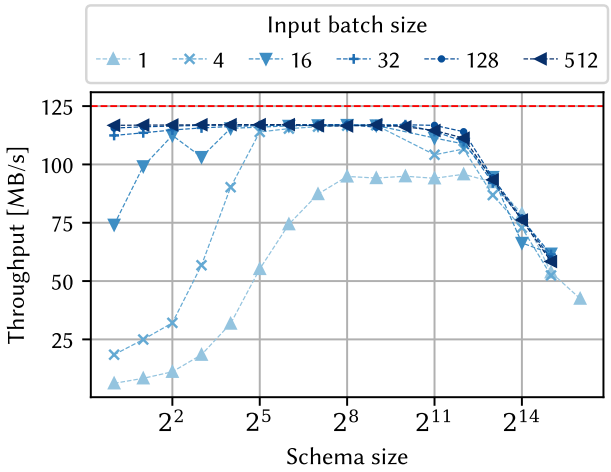


Figure 5.30 Impact of input batch size on overall ingestion throughput.

This figure shows that for smaller input batch sizes, the rate is limited by the gRPC interface for smaller schema sizes (as previously discussed in Section 5.5.2). Using an input batch size of 32 or more fully saturates the Gbit Ethernet input interface for all schema sizes up to approximately 4000. This confirms our previous results. For schema sizes of 8000 or more, the rate is limited by another factor: as seen in Section 5.5.6, compaction for larger schema sizes limits throughput. For a batch size of 1, the throughput never reaches the Gbit Ethernet limit.

Temporary Row Group Size & Temporary File Writing Strategy

We evaluate the next step in the processing pipeline: data is put into temporary buffers and persisted to temporary files. There are two main parameters: the temporary row group size as well as the number of temporary buffers. We test two different temporary row group sizes: 1024 and 16 384.

As shown in Figure 5.31, there is no measurable performance difference when changing the temporary row group size. This confirms the results of Section 5.5.3. Schema sizes larger than 4000 are again limited by compaction performance.

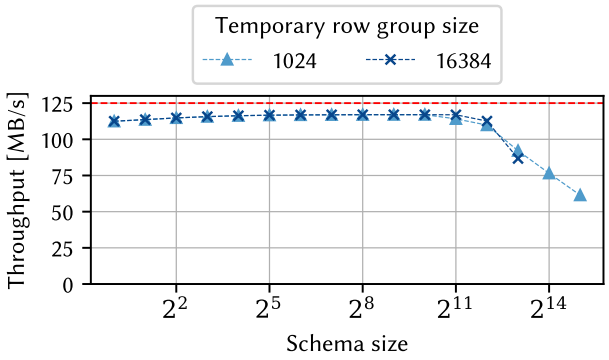


Figure 5.31 Impact of temporary row group size on overall ingestion throughput. The last two points for the case with a temporary row group size of 16 384 are missing because the resulting compacted file sizes (consisting of only a single row group) are larger than 1 GB and thus are badly comparable.

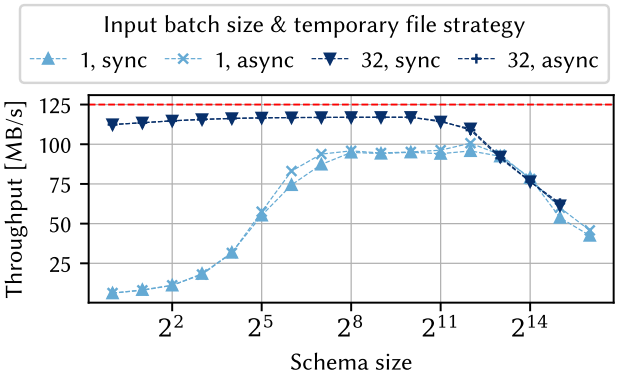


Figure 5.32 Impact of strategy for writing temporary row groups to temporary files on overall ingestion throughput.

As shown in Figure 5.32, having a second temporary buffer is only beneficial for small batch sizes. The impact is so small that, in most cases, this second buffer is not worth the additional memory required.

Compacted File Size

In this set of experiments, we evaluate the impact of the compacted file size. For this, in addition to the baseline size of 200 MB, a larger file size of 2 GB is used.

The results in Figure 5.33 show that throughput goes down noticeably for larger compacted file sizes. This is expected when considering the results from Section 5.5.6.

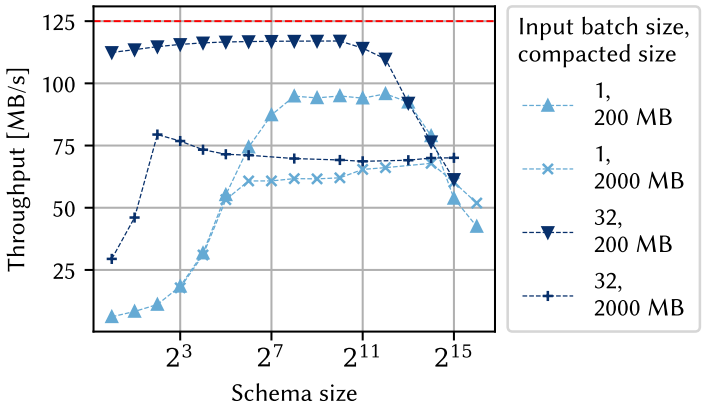


Figure 5.33 Impact of compacted row group size on overall ingestion throughput.

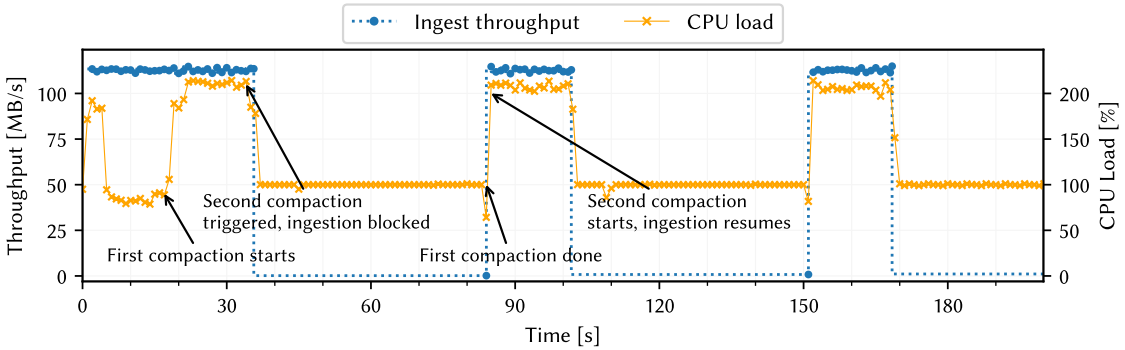


Figure 5.34 Timeline for configuration with a large compaction setting (compacted target size of 2 GB, input batch size 32, schema size 1).

Figure 5.34 shows the behavior over time for one particular combination of parameters[16]: the input batch size is 32, the compacted target size is 2 GB, and the schema size is 1. This plot shows the first 200 seconds of that experiment and indicates very clearly that performance is actually limited by compaction. Compacting data is slower than data ingestion in this case. As shown in Figure 5.34, the first compaction starts after a little less than 20 seconds. The CPU load rises to above 200 % at that point, showing that HAQSE can effectively exploit multiple cores on the employed CPU (cf. Table 5.1). After approximately another 18 seconds, the second compaction is triggered, causing the ingestion to be blocked. Only after the first compaction is done after roughly 85 seconds, ingestion resumes for approximately 18 seconds. To summarize this behavior, the compaction queue is repeatedly filled faster than the compaction background thread can process it. This, in turn, causes data ingestion to be blocked as long as the previous compaction process is not finished.

[16]This specific set of parameters is chosen because it shows the behavior in a very pronounced way. For other parameter combinations, the resulting compaction intervals are considerably shorter or not noticeable at all.

Aggregation

As the last experiment in this section, we activate aggregation in HAQSE for the received streams. We set the first aggregation factor to 100. All subsequent factors are set to ten, and we increase the number of aggregation levels up to six.

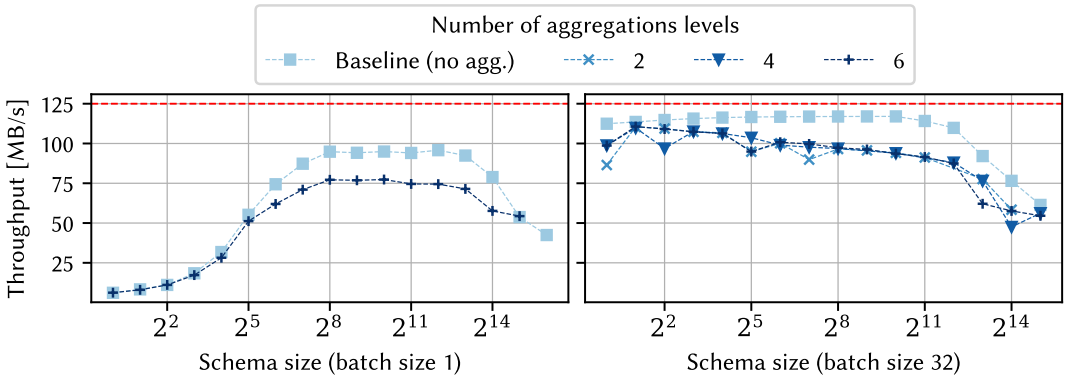


Figure 5.35 Impact of aggregation on overall ingestion throughput.

As the results in Figure 5.35 show, this slows down overall throughput. Calculating multiple aggregation levels, however, has no negative impact on performance.

Full-Pipeline Experiments Summary

In general, the results gathered in the full pipeline experiments from this section confirm the results from the previous, component-wise results. Especially the compaction workload turns out to be one of the primary throughput limiting factors in certain configurations. Independent of any other parameters, performance drops considerably for schema sizes of more than 4096. This drop is due to compaction blocking input, as observed in Section 5.5.6.

5.5.8 Summary of HAQSE Parameter Evaluation

In this section, we summarize the results of this section and draw conclusions regarding good parameter choices for HAQSE.

- Throughput for small payload sizes is limited by gRPC overhead. This is in particular relevant for small schema sizes: if possible, larger batch sizes of at least 32 should be chosen.
- A temporary row group size of 256 or more is fast enough for Gbit Ethernet use cases. Increasing the size beyond a certain point (4096 for medium-sized schemas) does not yield any benefits. When the temporary buffer does not fit into memory, performance suffers drastically. Our experiments suggest that a temporary

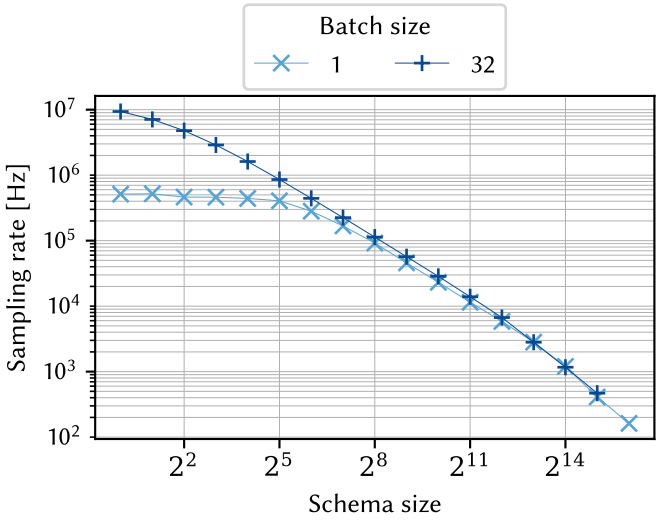


Figure 5.36 Supported sampling rate of HAQSE depending on schema size and input batch size.

row group size of 1024 represents a good default value. This was confirmed by experiments for layout transformation, when persisting data to temporary files and in our isolated aggregation experiments.

- Using two temporary buffers and writing data to temporary files in an asynchronous fashion is not necessary. Copying to and from operating system buffers (network stack, page cache) is fast enough and having an additional temporary buffer can be considered a waste of memory.
- Regarding compaction, our full pipeline tests show that this step in the ingestion pipeline is the limiting factor for schemas larger than 4000. The results of our isolated compaction experiments indicate that the chosen approach of compacting data only when a certain number of temporary files has been accumulated can be improved.
- When aggregation is enabled, ingestion throughput is slightly reduced in comparison to ingestion only. The actual number of aggregation levels, on the other hand, does not have a large impact on throughput.
- Our results generalize to different network types: slower networks are expected to further limit throughput. Faster networks, e.g., 10 Gbit Ethernet, remove the network bottleneck on ingestion. However, as our full pipeline experiments show, most workloads will be limited by the disk throughput during compaction.

Figure 5.36 shows the sampling frequency HAQSE supports for the following configuration: using a single temporary buffer with a temporary row group size of 1024,

a compacted target size of 200 MB, a compacted row group size of 65 536, without aggregation. Up to a schema size of 32, HAQSE supports sampling rates of more than 400 kHz when using the worst-case input batch size of one on the tested system. When using larger input batch sizes, the supported sampling rates are even higher for small schema sizes. Even for a schema size of 16 384, the sampling rate may still be around 1000 Hz. These values indicate that on the tested system, HAQSE is fast enough for all requirements of current applications (cf. Section 2.2.2).

5.6 Stream Consumption: Comparison with State-Of-The-Art

In this section, we compare our system to the three state-of-the-art time series management systems InfluxDB, QuestDB and TimescaleDB in terms of data ingestion throughput. Specifically, we want to answer the following research questions:

- RQ1: At what rate can state-of-the-art time series management systems consume sensor data streams? How does this depend on input batch size and schema size?
- RQ2: How does HAQSE compare to this?
- RQ3: What influence does the chosen hardware have on the rate?

For this, we employ the evaluation environment and components described in Section 5.4. Specifically, we use the setup sketched in Figure 5.7. For these experiments, our aim is to run the systems under test using their default configuration. We deploy each of them using the officially provided Docker images. We only slightly change some configuration parameters in order to avoid timeouts or out-of-memory failures. Versions and details are listed in Table 5.2. Unless noted otherwise, the databases run on the *sense-edge9* system and use the integrated HDD as storage location. This setup best matches the targeted industrial deployment scenario.

For this comparison, as before, we configure our data generator to generate single-precision floating-point numbers (4 B per value). We let it generate data in batches of 128 samples. As we showed in Section 5.4.5, this batch size ensures that the achieved data generation rate is large enough to not negatively influence our experiment results.

Table 5.2 Versions and configuration parameters for databases under test.

Database	Version	Configuration
InfluxDB	`influxdb:2.3`	`INFLUXD_STORAGE_WRITE_TIMEOUT=60s`
QuestDB	`questdb/questdb:6.5.3`	`QDB_LINE_TCP_MSG_BUFFER_SIZE=1048576`
TimescaleDB	`timescale/timescaledb:2.7.2-pg14`	`timescaledb.enable_2pc=false`
HAQSE	`haqse:v22.280`	`aggregation=false`

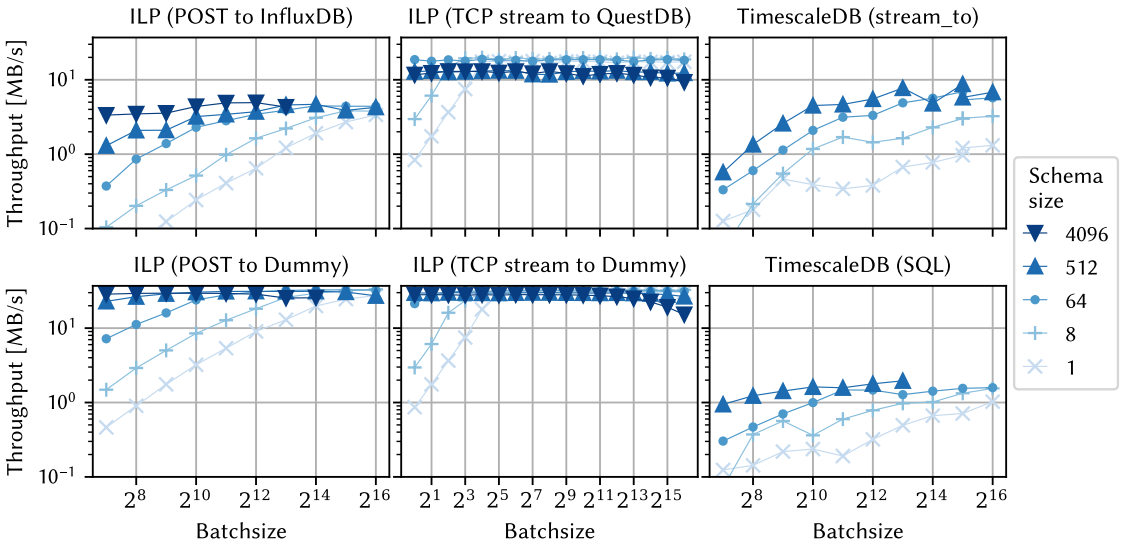


Figure 5.37 Sustained throughput depending on schema and batch size for InfluxDB, QuestDB and TimescaleDB (top row). The bottom row shows the baseline rate without actual database for the two ILP cases (InfluxDB and QuestDB). For TimescaleDB, it shows the achievable rate using SQL insert statements. The two TCP stream experiments (center plots) test a wider range of batch sizes. This is to show for which combination of parameters this ingestion protocol is less performant (generally, it is very stable performance-wise).

5.6.1 RQ1: State-Of-The-Art System Ingest Rates

In this subsection, we want to answer RQ1: at what rate can state-of-the-art time series management systems consume sensor data streams? For answering this question, we test multiple parameter configurations for the various state-of-the-art time series management systems. In particular, we vary schema size of the stream and the batch size used to send data to the respective system[17]. For this set of experiments, we let each test run for 30 seconds. Note that we still generate with a batch size of 128 at the generator to ensure no bottleneck between generator and adapter.

In addition to testing the rate achieved with the actual time series management system, we also provide a baseline for each of the three systems: For InfluxDB, this baseline is a HTTP POST server that just drops all received HTTP packets. For QuestDB, we test sending to a TCP receiver dropping all received data. For TimescaleDB, we provide numbers showing the throughput achieved by using SQL.

The results for these experiments are shown in Figure 5.37. As these plots show, InfluxDB, QuestDB and TimescaleDB ingestion throughput differs to a large extent. There are several interesting insights these plots reveal:

- The behavior for InfluxDB and QuestDB differs significantly. This is notable, in particular, since both QuestDB and InfluxDB use the same textual representation

[17] This is the output batch size of the adapter, not to be confused with the generator batch size

of the input stream (ILP). The reason for this difference can be best explained by inspecting the result plots for the respective dummy receivers. For InfluxDB, the overhead imposed by the HTTP POST requests leads to a significantly reduced throughput for small schema sizes and small batch sizes. Since the dummy receiver does no real work with the received stream data, it is apparent that the input method alone (HTTP POST) drastically limits the possible throughput to a large extent. In contrast to this, for QuestDB, the ILP is sent via a TCP socket stream. This enables throughput behavior that is almost not affected at all, neither by the batch size nor by schema size. Only very small payload sizes (product of schema size and batch size) are limited in throughput, similar to small payload sizes in HAQSE's gRPC input protocol.

- Again looking at the results of the dummy receivers, the maximum rate for InfluxDB slightly outperforms that of QuestDB. The reason for the different limits are the different encoding overheads for QuestDB and InfluxDB, as previously shown in Figure 5.13. In both cases, though, the maximum throughput for the encoded ILP without the respective protocol overhead (HTTP/TCP) (almost) fully saturates the Gbit Ethernet connection bandwidth (113.3 MB/s for InfluxDB, 117.2 MB/s for QuestDB). Comparing that to the values that can be achieved with the actual data base as receiver of the stream, we see that for both systems, the ILP does not seem to be the only bottleneck: for both systems, the achieved rates are well below the dummy receiver rates.

- For TimescaleDB with SQL, the maximum achievable rates are considerably lower than for the other two alternatives. Using the standard SQL method for streaming data to the tables saturates at around 2 MB/s for large schema or batch sizes. As promised by the `libpqxx` documentation, the `stream_to` method for ingesting data delivers a significantly higher throughput, especially for larger schema and batch sizes.

- For TimescaleDB, large batch sizes led to high memory utilization. For very high batch sizes and schema sizes, errors occurred such that some test runs failed.

- QuestDB performs best for small schema sizes, in contrast to the other two systems. For InfluxDB, the rate is mostly dependent on the HTTP packet payload size. If this payload size is large enough, the maximum rate on InfluxDB does not seem to depend on the chosen schema size. For TimescaleDB, larger schema cardinalities are beneficial for throughput. This can be explained by the row-oriented nature of the underlying PostgreSQL database.

To summarize these experiments, QuestDB delivers a highly stable ingestion rate of around 15 MB/s, almost independent of the chosen parameters. In particular, it also performs well for small batch sizes. Compared to the other two systems, QuestDB delivers the highest ingestion rate: InfluxDB reaches approximately 5 MB/s, TimescaleDB

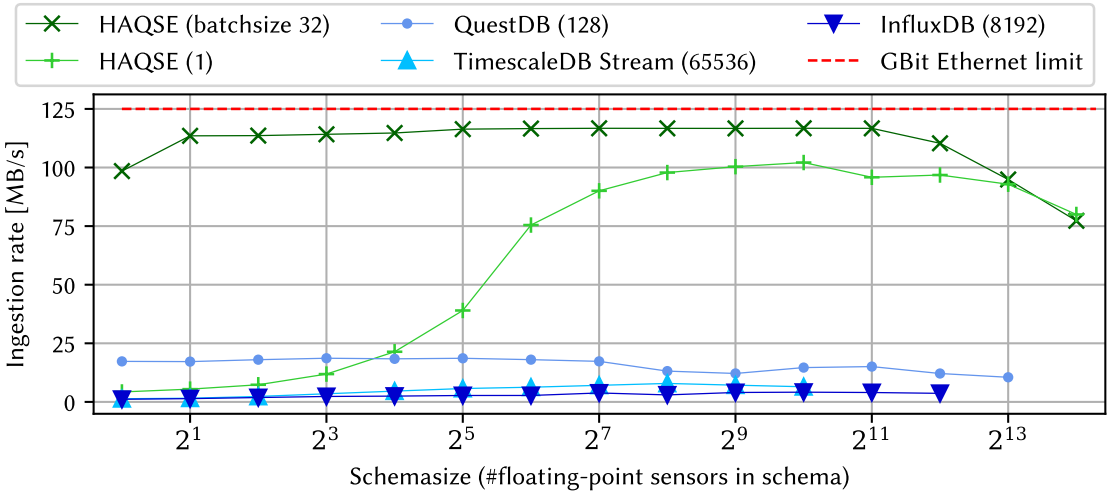


Figure 5.38 Throughput comparison of HAQSE with InfluxDB, QuestDB and TimescaleDB.

yields around 8 MB/s for certain configurations. In addition to that, InfluxDB and TimescaleDB heavily depend on the schema and the chosen batch size.

5.6.2 RQ2: Comparison of HAQSE With State-Of-The-Art Systems

We now fix the batch size parameter from the previous experiments to values where the respective databases show good throughput values. For InfluxDB, we fix this value to 8192, for QuestDB we use 128 (QuestDB shows good performance for almost all batch sizes, though). For TimescaleDB, we use the `stream_to` method and set the batch size to 65 536. Using these parameters, we compare the ingestion throughput to two ingestion configurations of HAQSE (using an input batch size of 1 and 32), varying only the schema size. To have a fair comparison, we do not activate aggregation in HAQSE for this set of experiments. Furthermore, we set the temporary row group size in HAQSE to 1024 and use a compacted row group size of 16 384. We use an uncompressed target size of 500 MB for compacted files and write all data to a single temporary file before that.

The results of this experiment are shown in Figure 5.38. It can be clearly seen that even for a batch size of 1, HAQSE outperforms existing databases in almost all configurations and reaches throughput values as high as 100 MB/s. For a batch size of 128, HAQSE achieves ingestion rates that fully saturate the Gbit Ethernet interface for almost all tested schema sizes. The next best contender is QuestDB with achieves a relatively stable ingestion rate of 10.5 MB/s to 18.6 MB/s, even though the batch size is relatively low.

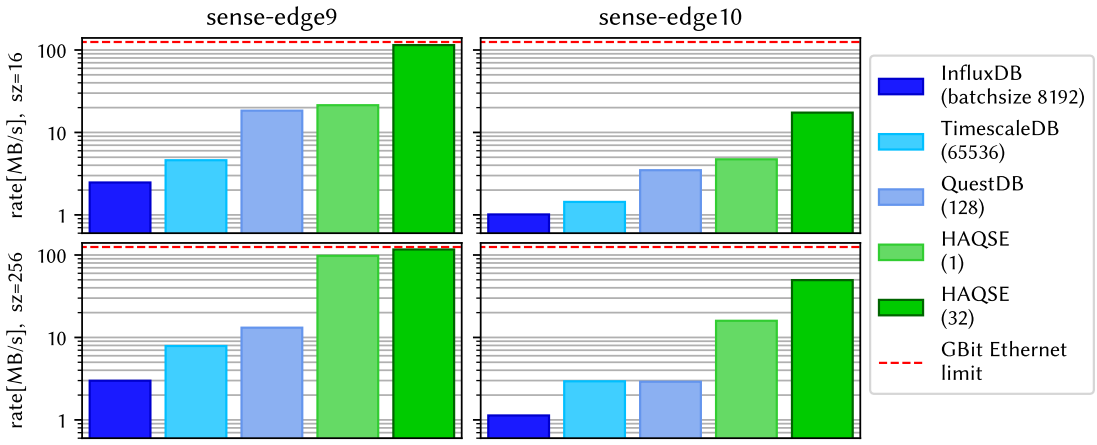


Figure 5.39 Throughput comparison of HAQSE with InfluxDB, QuestDB and TimescaleDB on different hardware configurations. Note the logarithmic scale on the y-axis.

After that, TimescaleDB follows with a maximum input rate of 7.9 MB/s. It should be noted, though, that for smaller schema sizes, this rate is considerably lower. Furthermore, there are several factors to be aware of. First, we use a batch size of 65 536 for sending data to TimescaleDB. This is magnitudes larger than for any of the other tested systems. Second, we use the `stream_to` interface. This works since our adapter is implemented in C++ and uses the pqxx library. However, it will be much more challenging to implement this for clients written in different languages like Java or Python. Third, PostgreSQL—and thus also TimescaleDB—only supports schemas up 1600 columns. Larger schemas would need to be split up into multiple database tables.

Lastly, InfluxDB achieves rates of up to 4.2 MB/s. Another thing to note is that InfluxDB failed to ingest a data stream with 16 384 sensors. For larger batch sizes, this problem appears even for smaller schemas, indicating that too large messages cannot be handled on the system under test. Thus, for large schema sizes, large batch sizes should be avoided. Large batch sizes also lead to an increased memory consumption.

We did not investigate the limiting component (and whether there is a single component that limits throughput) for any of the state-of-the-art systems. It should be noted, though, that none of the systems can come close to the performance HAQSE offers, simply because the encoding overhead of text-based protocols prohibits rates that come close to raw Gbit Ethernet.

5.6.3 RQ3: Influence of Hardware on Ingestion Rate

We now expand the comparison from above. In particular, we are interested in understanding the impact of the available processor on the ingestion rates. We do this by running the databases on systems with a less powerful CPU, *sense-edge10*. The generator still runs on *caps-sk1*, and we repeat the experiments from above.

The results of this are visualized in Figure 5.39. It is apparent that HAQSE outperforms existing systems also on less powerful hardware. The less powerful CPU on `sense-edge10` degrades performance of all tested systems, indicating that the processing speed is important for all time series management systems under test.

5.6.4 Comparison Summary

We summarize our comparison results in this section.

- The text-based input methods of existing systems have a high influence on the best possible ingestion rate, limiting achievable throughput considerably. Furthermore, using HTTP POST for InfluxDB is a bad choice for small batch sizes in terms of throughput. Other systems cannot achieve rates anywhere near the maximum throughput of HAQSE due to the input protocol alone.
- However, the input protocol is not the (only) limiting factor for the tested configurations.
- HAQSE achieves higher rates because it has less strict durability guarantees. InfluxDB and TimescaleDB *guarantee* that data is synchronized to disk after the insertion call finished successfully (response to the HTTP POST request for InfluxDB, transaction ok status code for TimescaleDB). QuestDB uses a slightly more relaxed method, relying ”on OS-level data durability” [95]. This needs to be taken into account when interpreting the results.

HAQSE fills a gap in the design space for applications that do not require durability guarantees as strictly as offered by the examined system. Our results also show that HAQSE works well across a range of different hardware configurations.

6 Querying Sensor Data

In this chapter, we explain how our system enables fast and efficient data retrieval. In Section 6.1, we present an overview of the query functionality, discuss the core data structure that enables fast data queries and explain the basic query handling logic. After that, in Section 6.2, we explain how our system supports distributed queries. In Sections 6.3 and 6.4, we evaluate our ideas.

Publication Information
Parts of the approach and experiments described in this chapter have previously been published in [66] and [65].

6.1 Efficient Query Processing

As discussed in Section 2.3, there is a set of requirements regarding data retrieval that we want to satisfy with HAQSE (Requirements 10 to 15). The work described in the previous two chapters builds the foundation for fulfilling some of these requirements. Having quick responses to queries (Requirement 10) requires a data format suitable to support this and an ingestion layer that processes the incoming sensor streams accordingly. To query most recent ranges and historical data efficiently (Requirement 11) requires an appropriate processing logic when consuming sensor streams. Finally, providing approximate answers to range-based queries (Requirement 12) is only efficiently possible if suitable approximations are calculated before the query is executed.

However, while a lot of preparatory work has already been presented, the actual query logic and the supporting data structures are described in this section. These are essential to create an efficient query system.

6.1.1 Stream Segment Index

As described in Section 2.3, our main focus regarding data retrieval are range-based queries. The primary data structure that enables fast range-based access to the stored sensor streams is the Stream Segment Index. A rough sketch of how the Stream Segment Index is organized is visualized in Figure 6.1.

For this Stream Segment Index, we exploit that stored sensor streams are continuous streams that have a natural ordering with respect to the sensor sample timestamp ts_i. For each level of the aggregation hierarchy (see Section 3.2.3), a separate such Stream

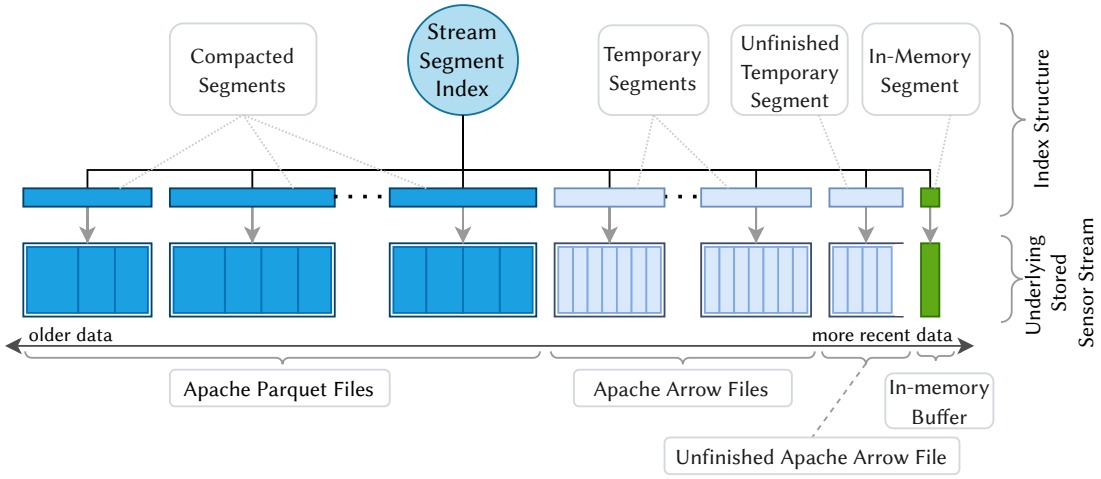


Figure 6.1 Overview of the Stream Segment Index in HAQSE.

Segment Index is created. Specifically, for each level of the aggregation hierarchy, the stored sensor stream is split along the time dimension into disjoint *stream segments*. Each such stream segment represents an ordered continuous segment of a stored sensor stream (of either the original sensor stream Ψ^O or a precalculated aggregation Ψ^A from the sensor stream hierarchy Ψ^H). A segment-id identifies each stream segment. We use the start timestamp of the segment as id since this uniquely identifies the segment. Furthermore, each segment contains all information for accessing the sensor samples of the segment. For file-based segments, this is the filename; for in-memory streams this is a pointer to the buffer containing the samples. Additionally, segments contain the covered time range as meta-information.

Based on these segments, we use a red-black-tree [12] to build an in-memory index that sorts segments according to their start timestamp. This helps to locate segments for range-based queries quickly. The index and the contained stream segments are independent of the actual underlying physical storage type. This allows simultaneous usage of different storage technologies. In fact, for supporting online use cases (Requirement 11), most recent parts of the data are stored in memory buffers, while historical data is stored in compressed form on permanent storage.

In comparison to the total number of samples stored in the system, the Stream Segment Index only contains a tiny fraction of entries. During ingestion, there is usually one entry for the in-memory buffer. Here, the ratio of number of samples per index entry is typically lowest: there is one entry per *row group size* samples. After that, there is one index entry per Apache Arrow or Apache Parquet file, each one containing potentially several million samples. An important feature of these two storage formats regarding fast lookup is that they include information about the file structure. This makes further drill-down to the queried range in these files very efficient.

This index may be accessed concurrently by the query (for reading from it) and ingestion (for writing) components. For that reason, there needs to be a mechanism that ensures that concurrent reads and writes do not lead to an inconsistent state of the data structure. In HAQSE, we do that by locking a mutex data structure when accessing the index. For reads, the mutex is locked in shared mode, allowing multiple concurrent reads. For writing, the lock needs to be acquired exclusively.

6.1.2 Query Processing Logic

There are two possible ways to retrieve data from HAQSE: using a *range-based query* or a *hierarchical range-based query* (cf. Definitions 8 and 11). The main difference in terms of processing the query is the selection of the queried aggregation level. The rest of the query processing is handled identically.

The level selection is done as specified in Definition 11: the appropriate aggregation level of the multi-resolution aggregation hierarchy is determined from the query by using the level with the largest resolution smaller than the requested query resolution $\Delta t^{\max}$.

Once the appropriate stored sensor stream Ψ of the sensor stream hierarchy Ψ^H is determined, HAQSE executes the following steps to prepare the query response. First, from the corresponding Stream Segment Index, a list of candidate segment ids matching the query range (cf. Listing 6.1) is retrieved. This list of candidate segments contains all segments that have a non-empty intersection with the query range.

```
1  def query_segs(q_range):
2    segments = []
3    seg_id = first_segment(seg_index, q_range)
4    while not beyond_query_end(seg_id, q_range):
5      p = create_from_id(seg_id)
6      segments.append(p)
7      seg_id = next_seg(seg_id)
8    unsatisfied_rg = calc_unsatisfied_rg(segments)
9
10   return segments, unsatisfied_rg
```

Listing 6.1 Find matching segments for query range

For this, we determine the first candidate segment via the red-black tree index (Line 3) and then iterate over the segments in the tree as long as the segments are in range (Line 4). For each of the segments in the range, we create a segment provider from the segment id (Line 5) and put it into the resulting list of candidate segments (Line 6). The calculated candidate segments contain the timestamp range covered by available data, so we can already calculate what parts of the query cannot be satisfied at that point

(Line 8). Thus, this information can instantly be sent to the client as metadata before any data needs to be read from segment data storage. Similarly, we also handle when the requested resolution cannot be satisfied by the available data. In this case, we still provide data with the best resolution available but inform the client that better resolutions might be available on other nodes. The client can then already start processing the lower-resolution data.

```
1 def query_data(q_range, signals, segments):
2   result = []
3   for s in segments:
4     seg_data = s.query_data(q_range, signals) # specification of projection/filtering
5     result.push_back(seg_data)                # collecting exec plans for segments
6   return QueryPlan(result)                    # return a stream handle
```

Listing 6.2 Server Query handling

In the second step, we perform actual data querying, as described in Listing 6.2. For each of the candidate segments (Lines 3–5), we create an execution plan. This execution plan consists of several parts:

- Reading samples: the underlying files or buffers are prepared for reading the actual sensor samples in the segment. For segments that are backed by files, this triggers opening the files in a background thread.
- Projection: The underlying stored sensor stream is projected to the requested sensors $\sigma_{i..j}$ of the query specification. This projection is pushed down to the storage layer such that only the respective columns of the respective storage files are actually read once the results are streamed to the client.
- Filtering: Filters are set up such that only samples that are part of the queried range are part of the result. These filters are also pushed down to the storage layer in order to avoid reading any unnecessary data.

Once this execution plan is constructed, we return a result stream handle to the client. This means that the client can then start reading result data, and only then data is actually read from storage.

6.2 Distributed Querying

Our approach for distributed querying mainly targets scenarios as described in Figure 6.2. In these cases, certain parts of hierarchical sensor streams are sent to storage and compute cloud hierarchies for further processing or analysis. Consequently, various views on the sensor measurements exist on different geo-distributed nodes in

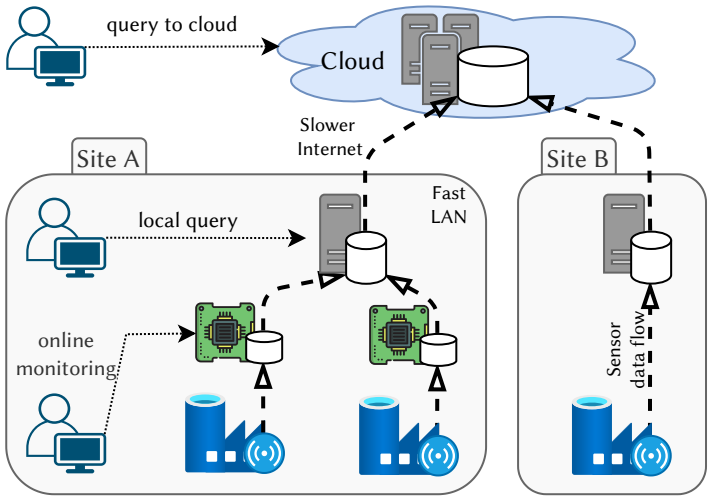


Figure 6.2 Distributed Sensor processing and analysis scenario overview. Sensor data is continuously flowing from the sensors to the cloud along the storage and compute continuum. Our system provides the same view of the data, no matter which part of the infrastructure is queried (first) and where the requested data is actually stored.

an edge-to-cloud continuum. For HAQSE, we design a simple mechanism that allows clients to retrieve data that is distributed across such an edge-to-cloud continuum.

When a client sends a query to a certain node, the query is processed locally, only on that particular node. Local processing is performed as described in Section 6.1. The query response consists of two parts: a quick metadata response and the actual local sensor data query result.

An overview of distributed query execution logic is visualized in Figure 6.3. The core idea is that all logic for distributed data retrieval is located and executed on the client-side. This way, there is no need for communication among the servers during query processing, greatly reducing overall complexity: no shared state needs to be managed, and the communication follows a simple client-server model. The client drives querying the various servers. It can already start processing data once it has received results from the first queried node. This is especially useful for interactive scenarios like visualization. Once results from other nodes are available, they can be used to progressively update the visualization.

```
1 def query(q, url):
2   result_list = []
3   queried_nodes = set() # set of already queried nodes
4   query_node(q, url, result_list, queried_nodes)
5   return result_list
```

Listing 6.3 Entry points for client query logic

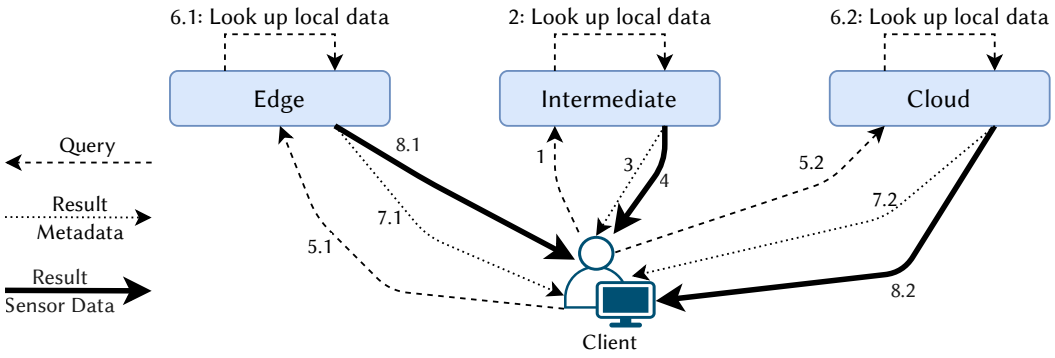


Figure 6.3 Sequential steps of distributed querying. The client first requests data from one of the nodes, in this case, the *Intermediate* node in the middle (step 1). Then, the query is processed locally (step 2). As first part of the query response, HAQSE sends the appropriate metadata (steps 3). This metadata contains information about the other nodes and allows the client to forward the query to these nodes, in this case, *Edge* and *Cloud* (steps 5.1 and 5.2). Since this is possible by only reading the Stream Segment Index, the client receives this first part of the response quickly and can forward the modified queries immediately. This happens simultaneously to receiving actual sensor data from the first queried node (step 4). The other nodes process the query (6.x) and send metadata (7.x) and result set data (8.x).

```
def query_node(q, url, result_list, queried_nodes):
  if url in queried_nodes:
    return # query each node only once
  queried_nodes.update([url])
  result = perform_query(q, url) # executes query in background thread
  result_list.append(result) # future to retrive q-result
  # schedule forwarding once metadata is received
  forward_query_on_receive(q, result, result_list, queried_nodes)
```

Listing 6.4 Client query logic for querying a single node

The client starts with querying any of the nodes in the infrastructure hierarchy (Listings 6.3 and 6.4). Typically, the node that is closest in terms of latency is queried first. When this query cannot be fully satisfied by the queried node, the metadata response contains information about which other nodes exist and what range to query. With an appropriate implementation of the queried server, this metadata response can be sent very quickly, before any actual data is sent. The client then modifies the query accordingly and sends it to the respective nodes (Listing 6.5). For this, it uses the original query and adapts the time range so that only the unsatisfied parts are requested. This process continues until all parts of the query range are satisfied or there are no other nodes to query.

```
def forward_query_on_receive(q, result, result_list, queried_nodes):
  nodes = extract_nodes(result)
  for ts_s, ts_e in extract_ranges(result):
    modified_query = modify_query(q)
    for n in nodes:
      query_node(modified_query, n.url, result_list, queried_nodes)
```

Listing 6.5 Query forwarding logic

6.3 Evaluation of Single Node Querying

In this section, we evaluate performance for data retrieval from a single HAQSE server. For this, we investigate the impact of the various configurable parameters in HAQSE. In particular, we focus on the storage-related parameters since these have the highest influence on query performance. Network latencies and throughput limitations are not considered here; these aspects are evaluated in Section 6.4. We first explain the evaluation environment used for our experiments. Then, we describe our experiments and discuss the results.

6.3.1 Evaluation Environment

We perform all our experiments in this section on the *sense-edge9* node also used in the evaluation of the previous chapter (see Section 5.4.1). For each experiment, we generate one or multiple datasets with a specific combination of parameters. We do this with the help of the data generator from Section 5.4.2. A Python-based query client performs multiple queries on the generated datasets. As shown in Figure 6.4, this query client executes its logic on the same node *sense-edge9*.

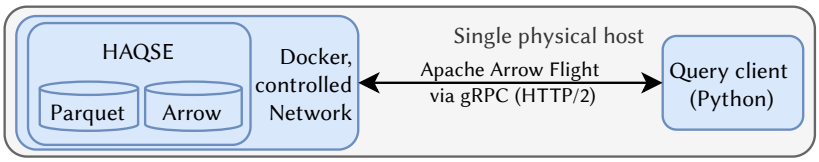


Figure 6.4 Rough sketch of the single node evaluation environment.

Python Query Client

The query client is implemented in Python (version 3.9), using the `pyarrow` and `pyarrow flight` packages (version 11.0.0). It runs on the same physical machine as the Docker container running HAQSE to avoid any uncontrolled network bandwidth and latency effects (see Figure 6.4).

Measurement Methodology and Metrics

For every query, we measure the total query time $t_{\mathcal{A}}$. $t_{\mathcal{A}}$ is the duration from start of the query until all result data is received from the queried HAQSE server. We use the function `perf_counter()` from the Python `time` package to measure runtimes of queries: We take one measurement just before sending the query, and one measurement right after the last part of the result set has been received. $t_{\mathcal{A}}$ is the difference between the two `perf_counter()` measurements. Queries are executed sequentially from a single query client. This means that only one query is active at the same time. We report the results in form of value distributions, indicating maximum, minimum and median values.

Datasets and Query Definitions

The datasets used in this section are generated by the generator from Section 5.4.2. All generated datasets only store the original resolution. No aggregations are calculated since this is not necessary for the aspects we want to investigate in this section. The datasets created in this section consist of sensor streams with different schema sizes. All sensors in the generated datasets use single-precision floating-point (32 bit, 4 B) as data type. This is expected to be the most frequently used data type in real-world deployments. Unless otherwise noted, we use the following default parameters for the datasets[1] used in our experiments and query definitions:

Apache Arrow settings We use a record batch size of 1024 and do not use compression for generating the Apache Arrow IPC format files.

Apache Parquet settings We use a default row group size of 65 536. The Apache Parquet data page size is set to the default value of 1 MiB. By default, we compress all data in the compacted Apache Parquet files. Timestamps are encoded using the `DELTA_BINARY_PACKED` [115] encoding. All sensor measurements use the Byte Stream Split encoding together with zstd (see Chapter 4 for details). We use the library's default compression level settings for zstd.

Query settings By default, the query range is selected randomly and uniformly from the total available range. This ensures that all data is equally likely to be queried, reducing the chance of biased measurements for specific query ranges. Each query is repeated twice. Depending on the total dataset size, for the first query, it is rather unlikely that the queried time range is in the operating system's page cache. For the subsequent identical query, in contrast, it is highly likely that the data required to process the query is already available in the page cache. Executing each query twice makes it possible to evaluate the difference between the two scenarios. Other query parameters (and parameter ranges) are determined by executing preliminary experiments and deducing reasonable values.

[1] Experiments in the previous chapters (see Sections 4.4 and 5.5) showed that these values work well for ingestion and compression. Therefore, we take the same values for our query experiment datasets.

Query results contain one 4 B floating-point value per requested sensor. Additionally, independent of the request, result sets contain a 8 B nanosecond timestamp per sample. The total uncompressed data size as well as the result set size can be calculated according to the following formula:

$$\text{total\_byte\_size} = \text{sample\_count} \times (8\text{B} + \text{schema\_size} \times 4\text{B})$$

All generated data is stored on and served from the installed 1 TB HDD.

6.3.2 In-Memory Segments

When sensor samples are received in HAQSE, they are first put into a temporary buffer, managed by the in-memory segment (see Figure 6.1). Since all data is available in memory, retrieving data from this segment does not require reading any data from persistent storage. Nevertheless, data needs to be filtered and selected according to the two dimensions specified in the query: time range and selected sensors. Selecting the sensors happens by assembling a new temporary record batch that discards all sensor columns not required in the result set. The time range is filtered by scanning the timestamp column and generating a filter mask. This filter mask is applied to all selected sensor columns. Querying data from these in-memory segments thus represents a good baseline with respect to all other types of segments; retrieving data from all other segments always requires opening at least one file from the storage directory and thus, will always be slower.

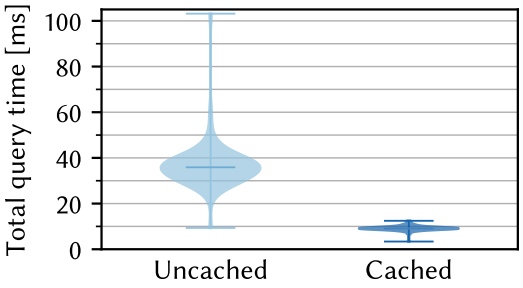
To measure query speed, we use a buffer with an in-memory row group size of 8192 and generate a sensor stream with a schema size of 8. We generate 8191 samples and then pause (but do not terminate) data generation. This makes sure that samples remain in the buffer and avoids the contents of the buffer being written to a temporary Apache Arrow file.

We then query with the following parameters. The query filters for a time range representing 1000 samples. Furthermore, the query selects three sensors from the stream schema. We perform 200 such queries and record $t_{\mathcal{A}}$. Since data is already in memory, there is no difference between first and subsequent query for this case.

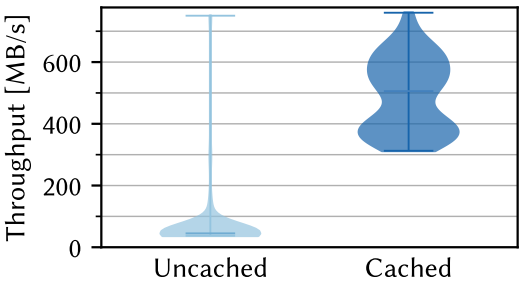
Queries take around 920 µs on average, with a median of 895 µs and a 95^{th} percentile of 1180 µs. This shows that almost all queries can be handled in less than 1 ms on the given setup when answered from the in-memory segment.

6.3.3 Page Cache Effects

Query performance heavily depends on the performance of the underlying storage hardware. In many interactive scenarios, however, similar parts of the data are requested repeatedly. For example, subsequent queries might only slightly alter the requested query time range. In such cases, a large part of the queried data is already



(a) Difference in $t_{\mathcal{A}}$ between retrieving 1000 rows (20 kB) for the first time (no data in page cache) and for subsequent queries (data is present in page cache).



(b) Throughput difference when reading 10^6 rows (20 MB) for the first time (likely no data in page cache) and for subsequent queries (data is present in page cache).

Figure 6.5 Influence of page cache on query performance.

present in the operating system's page cache. Consequently, fewer or even no further parts need to be loaded from actual storage, speeding up such subsequent queries.

With the experiments in this subsection, we evaluate the performance difference between uncached and cached data queries. For this, we generate a large dataset of 100 GB. The generated stream uses a schema size of eight and stores data in uncompressed Apache Parquet files. The database consists of 400 files of 250 MB each. In total, this dataset contains 2.5 billion samples or 20 billion single-precision floating-point sensor measurements. Performing the experiment on such a large data set ensures that not all parts of the data can be in page cache at the same time (there are only 8 GiB of RAM on the *sense-edge9* system we use for testing). We perform 100 queries and execute every query twice. The first query is thus very likely to request data that is not present in the page cache, while the second one is very likely to hit data in the page cache[2].

The results of this experiment are shown in Figure 6.5. Figure 6.5a shows the difference in distribution of $t_{\mathcal{A}}$ for a query of 1000 samples. For uncached data, 50 % of queries take more than 35 ms, with some taking longer than 100 ms. On the other hand, data that is present in cache usually takes less than 10 ms and in some cases less than 4 ms. Our implementation of HAQSE performs as many I/O operations as possible in background threads. This includes opening Apache Arrow and Apache Parquet files and locating the start offset inside the file. This part of query processing cannot be avoided either in the cached case.

Similarly, for throughput of larger result sets, reading cached data is considerably faster (see Figure 6.5b). Retrieving data from storage is clearly limited by the disk's throughput: the median value of 45 MB/s is slightly below the throughput that can be

[2]Without actually monitoring page faults, it is not guaranteed that data for a particular query comes from the page cache. However, the resulting measured distributions are sufficiently different, so we do not consider it necessary to actually monitor page faults.

delivered by the underlying HDD. Once data is available in the page cache, data can be retrieved with a throughput varying from 300 MB/s to 750 MB/s.

In both cases, it is evident that there is a large difference in performance between the two cases. For interactive data exploration tasks, where we expect similar parts of the data to be retrieved in subsequent queries, this effect is highly beneficial for the experienced performance. In most of the following experiments, we exploit data being available in page cache, but we specify what numbers we report in each case.

6.3.4 Unfinished Apache Arrow Segment

Sensor data in HAQSE passes through a number of different storage steps in its lifecycle (cf. Figures 3.5 and 6.1). The first of these steps where data is persisted to disk is storage in unfinished Apache Arrow files. In contrast to the later two steps (finished Apache Arrow and Apache Parquet files), there is no structural information that can be used to quickly locate the relevant parts that contain the time range of interest[3]. This lack of structural information means that finding the queried time range requires sequential iteration over the file contents until the respective part is located. With an increasing size of temporary Apache Arrow files, also the amount of data that must be iterated grows.

In the experiments in this section, we analyze this behavior. For this, we create three different databases, each containing a single Apache Arrow file: one with 500 MB, one with 2.5 GB, and one with 10 GB. Each database contains a stream with 16 sensors. We perform 100 different queries, each retrieving 1000 samples for three sensors.

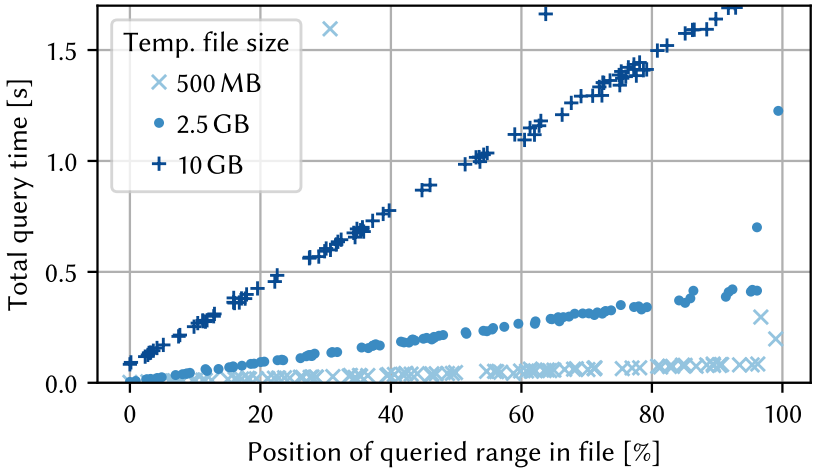


Figure 6.6 Behavior of linear search in unfinished temporary files.

[3]It would be possible to manage such structural information externally and use this information to speed up finding segments in unfinished Apache Arrow files. We do not deem this useful for now, considering the added implementation complexity.

The results are shown in Figure 6.6. They show that the total query time linearly depends on the start position of the queried range. In all three cases, the linear iteration behavior corresponds to roughly 75–80 million samples (rows) per second being iterated. This rate depends on the schema size: increasing the schema size leads to a decreasing iteration rate.

Repeatedly accessing the range does not improve query speed. This is due to all required timestamp data being likely in page cache anyway in this scenario. Furthermore, loading the actual measurement values once the correct time range is located does not contribute noticeably to the experienced query time. The few outliers that do not follow the linear behavior (not all visible in plot, as some of them are cut off) happen more often towards the end of the file. This supports the hypothesis that most timestamp data is already in page cache in this experiment, but data at the end of the file is not in all cases.

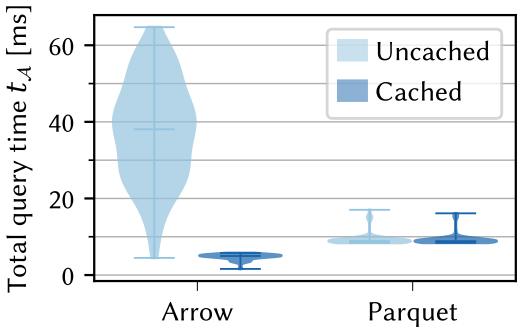
The results in this section show that linear search in the underlying data files is not a scalable approach for quickly locating the respective time range. As a consequence of these results, it is advisable to not let the size of temporary files grow too large, i.e., configure a limit around 250 MB if query performance for such recent data is important. Another viable approach would be to store structural information externally, e.g., in the Stream Segment Index. We keep that optimization as future work. The following sections show that Apache Arrow and Apache Parquet files perform significantly better by exploiting structural information in the footer.

6.3.5 Apache Arrow vs. Apache Parquet

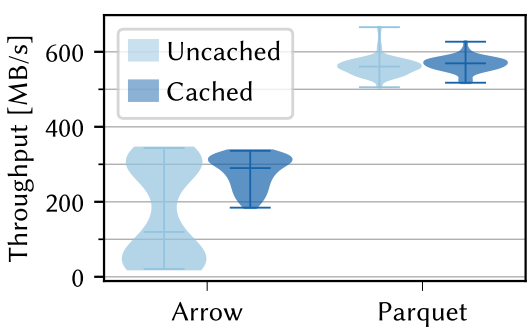
Apache Arrow and Apache Parquet files both have structural information in the footer to avoid the problems with large file sizes as indicated in the previous subsection. While Apache Arrow files only provide pointers to the contained record batches, Apache Parquet files also provide column statistics. These make it possible to determine the first row group that matches a range-based query by only using this statistics information. In the experiments in this section, we evaluate the performance difference between the two approaches.

For this we create two HAQSE data sets, each with 16 sensors and comprising roughly 35 million samples. This corresponds to 2.5 GB of uncompressed timestamp and measurement value data. One of the datasets consists of a single Apache Arrow file, the other one consists of a single Apache Parquet file. We perform 100 queries on each data set for two different result set sizes (10^3 and 10^6 samples). Each query retrieves data for three sensors and is executed twice.

The results of this experiment are shown in Figure 6.7. For the small result set size, Figure 6.7a shows the distribution of $t_{\mathcal{A}}$ for the different cases. It is apparent that querying the Apache Arrow file has the largest spread in the uncached case. Interestingly, the cached Apache Arrow case is the best performing in this experiment. This can be explained by the fact that when data is in cache, it can be used *as-is*, with-



(a) Difference in $t_{\mathcal{A}}$ between retrieving 10^3 samples (20 kB) for Apache Arrow and Apache Parquet files.



(b) Throughput difference when reading 10^6 samples (20 MB) for Apache Arrow and Apache Parquet files.

Figure 6.7 Comparison between retrieving data from Apache Arrow and Apache Parquet.

out any further processing like decompression. In case of the Apache Parquet dataset, queries have a very consistent behavior with respect to $t_{\mathcal{A}}$. There is no large difference between the cached and uncached case. This can be explained by the fact that locating the correct row group only requires information from the footer. This information very likely resides in the page cache for all queries (except for the very first). For the larger result set, the Apache Parquet case clearly outperforms the Apache Arrow case, providing almost twice the throughput. Again, the achieved throughput for Apache Parquet is very consistent around a rate of 550 MB/s.

The consistent behavior of retrieving data from Apache Parquet can be explained by the availability of statistics information in the parquet footer. Only few parts of the file need to be read in order to find the correct location in the file. These parts of the footer are very likely already in the page cache, so no data needs to be loaded from disk. Additionally, reading the measurement values then only requires reading few compressed pages from actual storage and decompressing them. Since data is stored compressed, a larger range can be held in the page cache.

6.3.6 Influence of Parquet Parameters

Apache Parquet files can be parameterized in several ways. In this section, we look at each of these parameters and quantify its influence on query performance. We examine the influence of the Apache Parquet footer size, row group size, data page size and compression.

Parquet Row Group Size and Footer Overhead

There is a trade-off between the number of row groups in a file and the size of each row group. The two conflicting goals are the following:

- On one hand, the number of row groups should be kept small, as every row group results in a small storage overhead in the footer of the Apache Parquet file. This footer is read every time the file is opened.
- On the other hand, the size of row groups should be small enough such that the statistical information in the footer is useful for quickly locating a certain time range. In the extreme case of only one very large row group, the information in the footer is almost useless for locating a certain timestamp.

As shown in Section 6.3.5, the statistical information in the footer of Apache Parquet files helps to quickly locate time ranges in the file. While this is beneficial in many cases, a large footer increases the constant overhead when opening an Apache Parquet file. In order to show the influence of this, we create several data sets, each containing 250 million rows, but each with a different row group size. This implies that for files with very large row group sizes, there are only few row groups in the file. The stream uses a schema size of eight, so each file contains 10 GB of uncompressed data. We request 1000 and 1 000 000 samples containing three sensors. We perform 25 different queries and repeat each query twice. We compute median, and 5^{th} and 95^{th} percentile of each second query and plot the results in Figure 6.8.

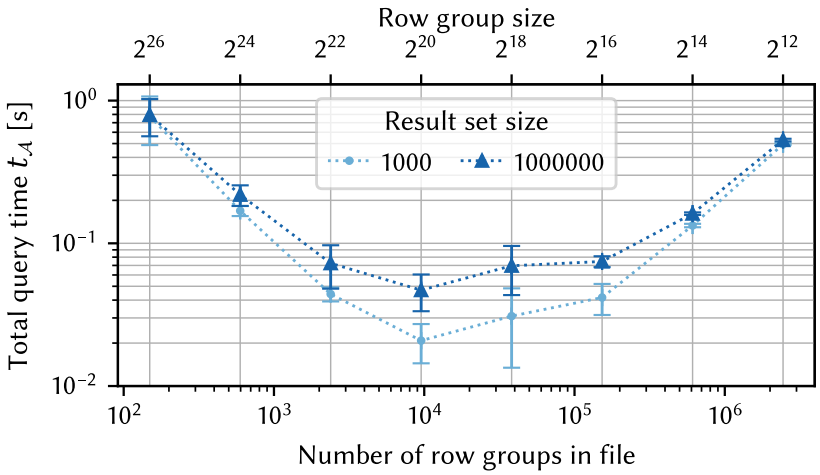


Figure 6.8 Total query time $t_{\mathcal{A}}$ depending on row group size in a 10 GB Apache Parquet file. The bottom x-axis shows the *number* of row groups in the respective file, the top x-axis shows the corresponding row group size.

This figure shows that both extremes—very large and very small row group sizes—lead to suboptimal query performance. Having only few, but very large row groups (towards left side of the plot) makes locating the queried time range inefficient. The timestamp column of a very large row group needs to be scanned sequentially, including the decompression of a large amount of unneeded data. Currently, all column data (of the queried sensors) of every row group that has matching statistics is read from the Apache Parquet file. While this can be optimized further, it would require substantial

engineering effort in the underlying Apache Parquet library implementation. Instead, it is advisable to keep row group sizes small. On the other end, having a high number of row groups in the file (towards right side of the plot) slows down initial reading of metadata. This is underlined by the fact that there is almost no variation in query time, regardless of the result set size. The footer is up to 45 MB in this case for a row group size of 2^{12} (corresponding to approximately 2.5 million row groups). In this specific example, row group sizes between 2^{16} and 2^{22} perform best.

This evaluation shows that both a very large number of row groups per file and very large row group sizes should be avoided for best query performance. This can be achieved by using row group sizes in the range found above and limiting the total number of rows per file.

Influence of Parquet Data Page Size

The smallest readable unit in Apache Parquet files are so-called data pages [114]. In this part of our evaluation, we investigate the trade-off between many small and few large data pages. Data pages have no influence on locating the queried time range. While the Apache Parquet format generally would support it, statistics in data page headers are not implemented currently. Data pages represent the unit of compression, i.e., when reading data, one complete data page is decompressed as an atomic processing step. This means that the page size mainly has an influence on sequential read throughput.

For this experiment, we thus try to provoke large sequential reads in Apache Parquet files: we use large values for the row group size to have many long scans. This best shows the influence of differently-sized data pages. We generate multiple databases with eight sensors and 25 million samples, i.e., 1 GB of uncompressed measurement values. This data is present in a single Apache Parquet file with a row group size of 2^{23}. We vary the data page size of these files from 4 KiB to 16 MiB. We query one million rows and three sensors. Each query is performed twice, and we only include the duration of the second query in our results. The resulting distribution of the query time is shown in Figure 6.9.

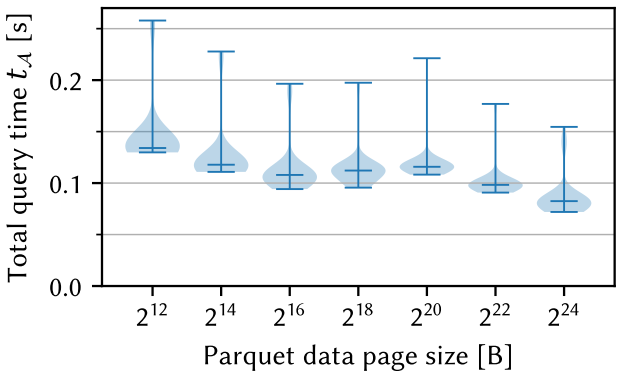
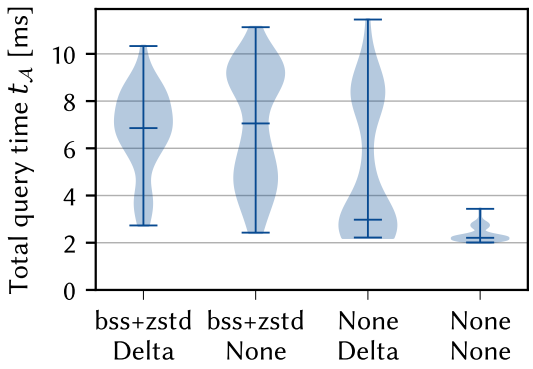
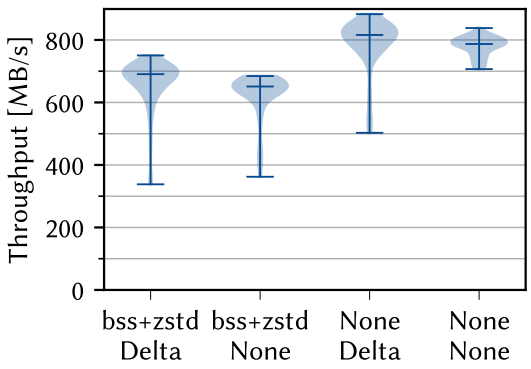


Figure 6.9 Data Page size experiment results.



(a) Difference in $t_{\mathcal{A}}$ when retrieving 10^3 rows (20 kB) for different timestamp/sensor measurement compression combinations.



(b) Throughput difference when reading 10^6 rows (20 MB) for different timestamp/sensor measurement compression combinations.

Figure 6.10 Influence of compression options on query performance.

These results show that smaller page sizes are generally less performant than larger page sizes. The difference, however, is negligible when compared to the influence of other factors. In addition to this, large data pages only have an influence on sufficiently large row groups. Even without any compression, the default value of 1 MiB implies that all row group sizes $\leq 131\,072$ only contain a single data page for 64 bit timestamp data[4]. For compressed data, this row group size limit is increased by the factor of the compression ratio. This means that there is no reason to change the default value of 2^{20} (1 MiB).

Influence of Apache Parquet Compression

As last parameter that is specific to the Apache Parquet file format, we examine the effect of compression on the query performance. For this, we create four databases with varying compression settings. Each database contains 16 sensors and 14 million samples, i.e., 1 GB of uncompressed data in four equally sized files. For timestamp compression, we evaluate the uncompressed and `DELTA_BINARY_PACKED` encoding alternatives, for floating-point compression we use uncompressed and the combination of Byte Stream Split and zstd (cf. Section 4.4.7). We define 100 queries retrieving three sensors and 10^3 or 10^6 samples. We report the results for the second query (even if there is no huge difference for this experiment between cached and uncached results). The resulting distributions are shown in Figure 6.10.

The results show that querying uncompressed data performs best, both with respect to the total query time $t_{\mathcal{A}}$ for small result sets and for throughput of large result sets. Using bss+zstd compression for floating-point data slightly increases the query time

[4] One 64 bit timestamp consumes $8 = 2^3$ B. 1 MiB $= 2^{20}$ B $\rightarrow$ There is space for $\frac{2^{20}}{2^3} = 2^{17} = 131\,072$ rows in one data page with a default size of 1 MiB.

and reduces throughput. This can be explained by the fact that when data is served from the page cache, the additional decompression step incurs overhead that is not necessary when using uncompressed data. Decoding the delta-encoded timestamps does not have a large negative influence on the resulting performance on average. However, the slowest queries are considerably slower ($t_{\mathcal{A}}$ is 4–5 times higher) and provide less throughput when compared to the uncompressed case.

6.3.7 Scaling Stored Stream Schema

Streams with a high schema cardinality increase the amount of data in the footer of Apache Parquet files. Since all this footer information is read when opening files for data retrieval, it takes longer to open files for streams with a high schema cardinality (cf. Section 6.3.6). In the experiments in this section, we quantify the impact of large schemas on the query performance.

For this, we create several sensor stream databases containing 1 to 65 536 sensors (using increasing powers of four). Each database is designed to contain 1 GB of uncompressed measurement values, split up into four files of 250 MB each. We create 50 queries, each one retrieving 1000 samples of one sensor from the respective stream. Each query is performed twice, and we report results for the second query. The results are shown in Figure 6.11.

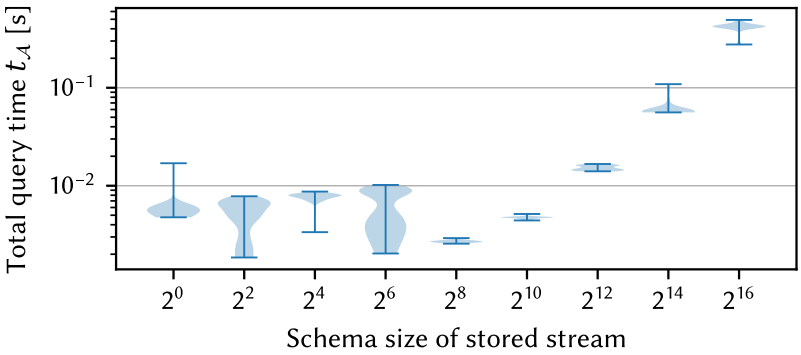


Figure 6.11 Influence of schema size of stored stream on total query time $t_{\mathcal{A}}$.

The results show that the query time increases considerably with increasing schema size. This is despite the fact that the number of total rows in databases with a larger schema is considerably lower (less than 4000 samples for the largest schema). This can be explained by the increasing size of the Apache Parquet footer that needs to be read on each query when opening the file, confirming the results from Section 6.3.6.

6.3.8 Scaling Result Set Size

There are two main dimensions within which result sets can grow: on the one hand, the number of requested samples can be increased, on the other hand, the number of sen-

sors can be increased. We evaluate the performance for scaling these two dimensions in this section.

An overview of the behavior is shown in Figure 6.12. This plot is generated from experiments on the dataset described in the query schema paragraph below.

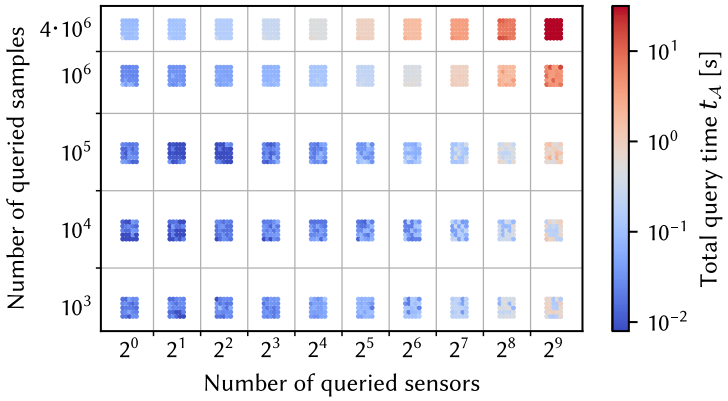


Figure 6.12 Overview of impact of result set size on total query time $t_{\mathcal{A}}$. The result set size depends on two dimensions: number of samples and number of sensors in the result set. The plot shows the color-coded distribution of $t_{\mathcal{A}}$ for each paramter combination (each small dot in the colored rectangles represents one experiment run).

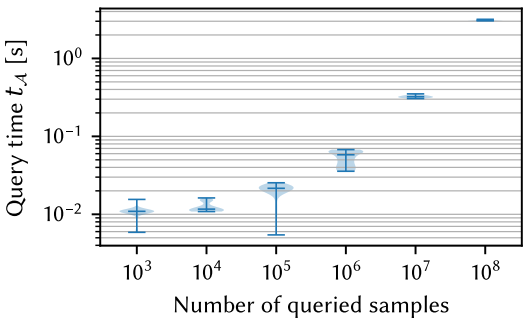
This plot shows that scaling the number of queried rows performs much better. We investigate two specific configurations in more detail below.

Scaling Number of Requested Samples

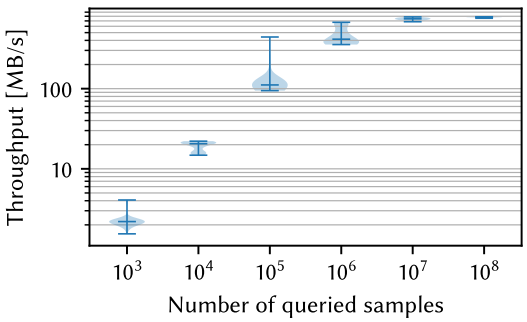
The number of samples required for analytic tasks varies considerably depending on the exact application. In this part, we evaluate how well HAQSE scales when increasing the number of samples to be retrieved. To test this, we create a database of approximately 140 million samples with a schema of 16 sensors. This corresponds to an uncompressed size of 10 GB, and each Apache Parquet file in the dataset contains 250 MB of uncompressed data. We query four sensors from the database. We increase the number of samples to retrieve from the database in powers of 10. For each combination of parameters, we create 50 queries and execute each one twice. We take results from each second query and plot the results in Figure 6.13.

Figure 6.13a shows the total query time $t_{\mathcal{A}}$ as a function of the number of retrieved samples. For few queried samples ($\leq 10^4$), the query time is roughly constant: the number of samples queried has no measurable influence on the query time $t_{\mathcal{A}}$ for such small results. After that, the time approaches a linear behavior with an increasing number of samples. This can also be seen in Figure 6.13b: for many samples retrieved (10^7 – 10^8), the throughput approaches a rate of 800 MB/s.

This result shows that HAQSE is able to fulfill Requirement 10 (Query Speed): small queries can be executed in few milliseconds, larger queries provide high throughput.

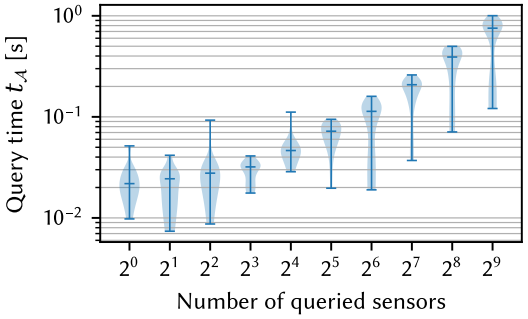


(a) Influence of number of samples in result set on total query time $t_{\mathcal{A}}$.

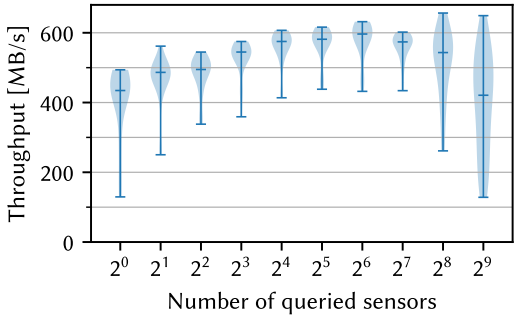


(b) Influence of number of samples in result set on query throughput.

Figure 6.13 Influence of number of samples in result set on query performance. Uncompressed result set size ranges from 24 kB to 2.4 GB.



(a) Influence of number of sensors in result set on $t_{\mathcal{A}}$ for queries retrieving 1000 rows. Uncompressed result set size ranges from 12 kB to 2 MB.



(b) Influence of number of sensors in result set on query throughput for queries retrieving 10^6 rows. Uncompressed result set size ranges from 12 MB to 2 GB.

Figure 6.14 Influence of number of sensors in result set on query performance.

Scaling Query Schema

For this set of experiments, where we scale the queried sensor schema, we create a dataset with an uncompressed total size of 10 GB. We create roughly five million samples for 512 sensors. Each Apache Parquet file in the dataset contains 250 MB of uncompressed data. We increase the number of queried sensors in powers of two from 1 to 2^9. We retrieve 10^3 and 10^6 samples. For each combination of parameters, we create 25 queries, uniformly and randomly selecting a query range. Each query is executed only once, in order to show uncached behavior (in contrast to the many cached results before). The results are shown in Figure 6.14.

The results for few queried samples (1000) are shown in Figure 6.14a. They show that up to a certain number of queried sensors (roughly 10), the query time $t_{\mathcal{A}}$ increases only slightly. After that, the query time $t_{\mathcal{A}}$ increases linearly with the number of queried sensors. Overall, there are large variations in the results due to the fact that we perform

every query only once. While there is a certain chance that parts of the data are in the page cache, most of the queried data is not and needs to be loaded from the HDD that is used as storage medium.

In Figure 6.14b, the throughput for a larger amount of queried samples (10^6) is shown. These results show that median throughput slightly increases up to a certain point (2^6 sensors). For schema sizes larger than 2^7, the median value decreases. In addition to that, the variability in throughput increases drastically. This can be explained by the fact that for very broad result schemas that contain many rows, the pressure on the page cache is very high. The uncompressed result set of each query contains more than 2 GB of data. All this data must be loaded from (compressed) storage, in some cases (but not in all) invalidating large parts of the contents previously available in the page cache.

6.3.9 Concurrent Ingestion and Data Retrieval

In this section, we present a very basic analysis of how querying data influences data ingestion and vice versa. For this, we let our data generator from Section 5.4.2 generate a sensor stream of eight sensors. The generator runs locally on the same system and produces batches of 128 samples. In the first phase of the experiment, the data generator generates sensor samples for 120 s. After that, in the second phase, we still continue generating data, but in addition to that, start querying data. We do this for 60 s, and then stop data generation. In the third phase, we continue performing data queries for roughly 60 more seconds. HAQSE is configured to write data to Apache Arrow files containing roughly 20 MB. Five such files are compacted to Apache Parquet files, resulting in an uncompressed target size of 100 MB. We query one sensor and 1000 samples, uniformly and randomly selecting a query range. Each query is executed once. The results for this experiment are shown in Figure 6.15.

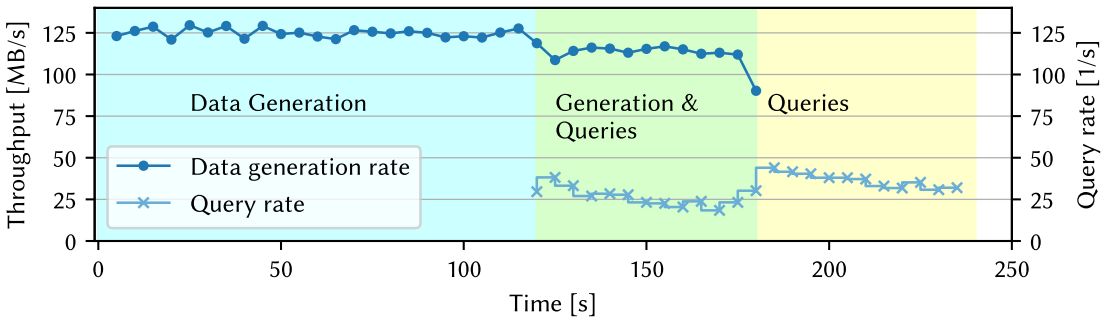


Figure 6.15 Timeline of concurrent sensor stream consumption and querying.

They show that HAQSE consumes the generated stream at roughly 125 MB/s. This results in approximately 15 GB of uncompressed data or 375 million samples. When queries start after 120 s, data consumption in HAQSE is slightly reduced to 115 MB/s.

During that time frame, data can be queried at roughly 20 to 30 queries per second. As soon as data generation is stopped, the query rate increases to roughly 40 queries per second.

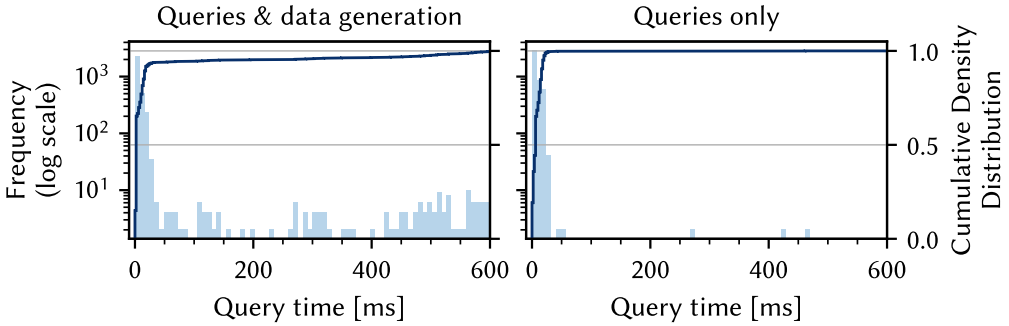


Figure 6.16 Distribution of total query times $t_{\mathcal{A}}$ for two different scenarios. Both plots show the absolute frequency (light blue bars, left y-axis) and the cumulative distribution (darker blue line, right y-axis) of measured query times $t_{\mathcal{A}}$. The x-axis shows query time bins with a width of eight milliseconds. Scenario on the left: queries are executed while HAQSE is consuming sensor streams. Scenario on the right: only queries are executed, no other processing is happening simultaneously.

Figure 6.16 shows the query time distribution for second and third phase. The majority of queries can be executed in less than 10 ms in both cases. However, the two distributions show that a few queries take considerably longer (> 400 ms) when executed while data is consumed in HAQSE. This also becomes evident when looking at percentiles. While the 90^{th} percentiles of 18.3 ms and 17.9 ms are still close together, the 95^{th} percentiles (140 ms and 19.9 ms) are already far apart.

The overall parameter space for concurrent sensor stream ingestion and data queries is vast. The experiment here covers the most critical aspects and ranges that affect the real-world scenarios targeted in this thesis. We leave it to future work to explore this space more exhaustively (targeting different scenarios and trade-offs).

6.3.10 Single Node Query Evaluation Summary

This section summarizes the results from all experiments in Section 6.3.

- There is a considerable positive effect of repeatedly querying similar time ranges due to the operating system's page cache. This is especially beneficial for interactive data exploration scenarios.

- In general, queries producing small result sets fully yield these results in less than 50 ms. If data is already available in the operating system's page cache, data for most queries can be delivered in approximately 10 ms.

- Query throughput for large (cached) results is well above the Gbit Ethernet bandwidth limit, indicating that, for such scenarios, throughput will be limited mostly by that bandwidth limit.

- Making use of the chosen compression approach does not slow down queries for most practical uses.

- Reading large Apache Parquet file footers is slow. It is interesting to explore whether it is possible to optimize this and how much can be gained for large schema sizes. Before such optimizations are implemented, it is advisable to avoid too large Apache Parquet footers. This is achieved both by limiting the number of row groups per Apache Parquet file and by keeping the number of sensors per file reasonably low ($< 10^4$).

- It should be avoided to let unfinished Apache Arrow files grow too big since sequentially scanning these files directly impacts total query time. Similarly, row group sizes of Apache Parquet files should not be chosen to be too large. Values around the tested default of 65 536 represent a good compromise.

- Changing the data page size of Apache Parquet does not have a huge impact on query performance. The default value of 1 MiB represents a reasonable default.

- Concurrent querying reduces ingestion performance, but does so only slightly.

Many aspects of the query performance evaluated in this section can be attributed to the performance characteristics of the storage format we choose for HAQSE. All the processing that is performed in stream consumption to eventually write sensor streams to Apache Parquet files now pays off when retrieving data from HAQSE.

6.4 Evaluation of Distributed Querying

In this section, we evaluate the distributed querying capabilities of HAQSE. In contrast to the evaluations in the previous section, the experiments here focus on the interaction between the query client and multiple instances of HAQSE.

6.4.1 Evaluation Environment

We let multiple HAQSE instances and the query client run on a single system. Therefore, we use a relatively powerful system for the experiments in this section.

Base system

We perform all our tests on a system with two Intel Xeon Gold 6136 CPUs (running at 3.00 GHz) with 320 GiB of RAM. The system is running Ubuntu 20.04. Our prototype is compiled with g++ version 10.3.0 with compiler flags `-O3 -flto -march=native` and linked against libarrow 7.0.0. The server is deployed in a Docker container (also running Ubuntu 20.04) to have a controlled network environment, as described next.

Network Environment

We design the experiments in this section to emulate geo-distributed deployment scenarios. Thus, we require a network environment with controllable latency and bandwidth. In our experiments, we artificially emulate these network properties. We do this by running our prototype inside a Docker container, using the `tc`-tool to set delays and shape traffic accordingly.

Data and Storage

We generate a random dataset to build a sensor stream database. Following an actual use case, this database contains values representing 16 artificial floating-point sensors. Details about the generated hierarchical stored sensor stream are shown in Table 6.1. The data spans roughly 34 days of sensor data. The sampling rate is 20 kHz (resolution of 50 µs), a typical rate for monitoring the combustion dynamics in gas-turbines[5]. This data corresponds to more than 950 Billion raw 32 bit floating-point measurements or 4.3 TB of raw data (including 64 bit timestamp information). Additionally, our database consists of 13 precomputed aggregation levels with five aggregations (min, max, mean, stddev, count) for each level and sensor. The first aggregation level has a resolution of 3.2 ms, i.e., the window for this first aggregation level contains one set of aggregations for every 64 raw samples. Further aggregation levels aggregate four samples of the previous level, the coarsest resolution thus is at roughly 14 hours and 55 minutes. In total, the database consists of 4.8 TB of data. This data is stored in uncompressed Apache Parquet files with a row group size of 312 500 on a NAS attached via a 10 Gbit Ethernet connection and mounted via SMB version 3.0.

Measurement Methodology and Metrics

For every query, we measure the total query time $t_{\mathcal{A}}$ (as introduced previously) as well as the metadata response time $t_{\mathcal{M}}$. As before, these times are measured using the function `perf_counter()` from the Python `time` package. $t_{\mathcal{M}}$ indicates the duration from query start until the metadata response has been received (see Figure 6.3). This

[5] This sampling rate is derived from our motivating example.

Table 6.1 Evaluation dataset and aggregation levels.

Level Number	Resolution	Sample Count	Size	Factor to previous	Number of files
0	50.0 µs	$6.0{\cdot}10^{10}$	4.30 TB	–	7639
1	3.2 ms	$9.4{\cdot}10^{8}$	366 GB	64	120
2	12.8 ms	$2.3{\cdot}10^{8}$	91.4 GB	4	30
3	51.2 ms	$5.9{\cdot}10^{7}$	22.8 GB	4	8
4	204.8 ms	$1.5{\cdot}10^{7}$	5.71 GB	4	2
5	819.2 ms	$3.7{\cdot}10^{6}$	1.43 GB	4	1
6	≈ 3.3 s	$9.2{\cdot}10^{5}$	358 MB	4	1
7	≈ 13.1 s	$2.3{\cdot}10^{5}$	89.3 MB	4	1
8	≈ 52.4 s	$5.7{\cdot}10^{4}$	22.3 MB	4	1
9	≈ 209.7 s	$1.4{\cdot}10^{4}$	5.60 MB	4	1
10	≈ 14.0 min	$3.6{\cdot}10^{3}$	1.42 MB	4	1
11	≈ 55.9 min	$8.9{\cdot}10^{2}$	369 kB	4	1
12	≈ 3.7 h	$2.2{\cdot}10^{2}$	107 kB	4	1
13	≈ 14.9 h	55	41 kB	4	1

first metadata response does not contain sensor data, but only query result meta information like the projected stream schema. We design our queries in such a way that the result contains data for three sensors and 1000 data points from the queried data stream. We report the results in the form of value distributions, indicating maximum, minimum, and median values.

6.4.2 Multinode Baseline

In this subsection, we analyze the behavior of our system under different network environments. Specifically, we change network delay and bandwidth. To better understand the influence of both of these parameters, in this section, we apply artificial delay and bandwidth limits independently. For all experiments in this section, we always perform the exact same query to ensure reproducible response times. We request 1000 data points for three sensors for a fixed range of roughly ten days. Results of queries hitting aggregated data contain *min*, *max* and *mean*, leading to a result set size of $1000 \times (8\,\text{B} + 3 \times (3 \times 4\,\text{B})) = 44\,\text{kB}$.

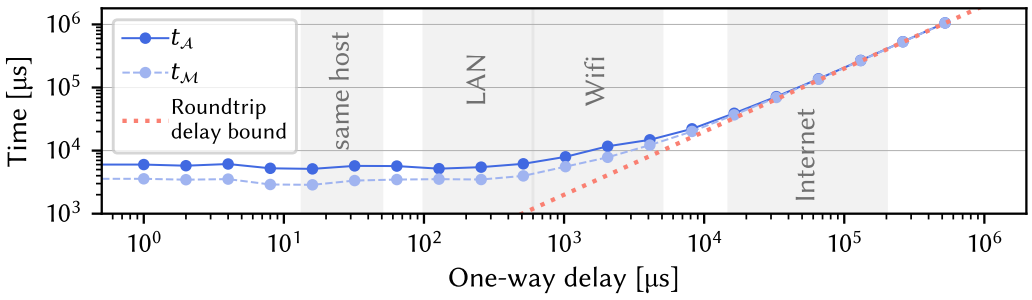


Figure 6.17 $t_{\mathcal{A}}$ and $t_{\mathcal{M}}$ for different network delays.

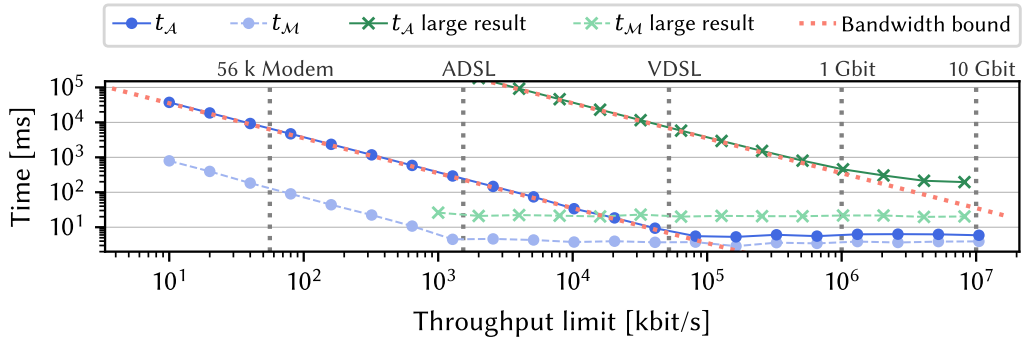


Figure 6.18 $t_{\mathcal{A}}$ and $t_{\mathcal{M}}$ for different network bandwidth limits for two result sizes 44 kB (1000 data points) and 44 MB (*large result*, 1 000 000 data points).

Varying network delay

In Figure 6.17, we show the behavior of our system under varying network delays. The figure shows that short network delays (those typically found in wired local networks like Ethernet) do not have a noticeable impact on $t_{\mathcal{A}}$ or $t_{\mathcal{M}}$. When data is transferred via a connection similar to a typical Internet connection [5], virtually all latency is introduced by the network.

Varying network bandwidth

Figure 6.18 shows the results when changing network bandwidth. Similar to the case above, transferring the query results fully saturates the connection (up to a certain bandwidth that depends on the query result set size). This means that $t_{\mathcal{A}}$ depends almost entirely on the bandwidth. Since the metadata size is magnitudes smaller, $t_{\mathcal{M}}$ is also a lot faster for lower bandwidth connections. The additional graphs for *large result* in Figure 6.18 yield a result set size of 1 000 000 samples with otherwise identical settings.

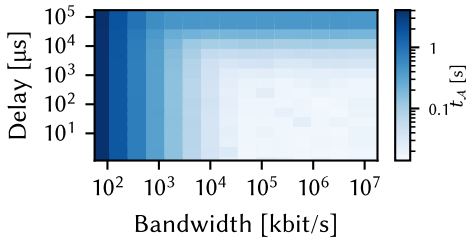


Figure 6.19 $t_{\mathcal{A}}$ (single host) for different bandwidth and delay combinations.

Varying Network Bandwidth and Delay

Figure 6.19 shows the combination of various network bandwidth and delay settings. In the bottom right of this plot, there is a lighter area where the network is *good enough* and does not have a negative impact on $t_{\mathcal{A}}$. Low network bandwidths (towards the left border of the plot) impact result data transfer times, increasing $t_{\mathcal{A}}$. High network delays (towards the top of the plot) induce large roundtrip times and thus, dominate query times, regardless of the available bandwidth.

6.4.3 Two-Node Analysis

This section analyzes how our system behaves in a two-node setup. This case applies to the classic edge-cloud scenario, where data is generated and initially stored in a remote edge environment and where aggregated overview data is available on a cloud instance. For this set of experiments, the cloud node contains levels 4–13. The edge node contains all levels of the dataset from Table 6.1. For these experiments, the query client is assumed to be located "near the cloud" and thus, queries the cloud node first.

We request 1000 data points at a resolution of 51.2 ms (level 3), which the cloud can only answer with an unsatisfying resolution of 204.8 ms (250 data points of level 4), and thus informs the client to request more detailed data from the edge server. Typically, the cloud instance will be reachable via a high bandwidth connection, while the connection to the edge system is often much more limited when accessed via the Internet.

As our previous analyses show, both network delay and available bandwidth have an impact on $t_{\mathcal{A}}$, so we systematically investigate the impact of each of these aspects.

Delay

We first show the impact of network delay without limiting the bandwidth. On the left of Figure 6.20, the plot shows $t_{\mathcal{A}}$ for receiving data from the cloud. On the right, it shows $t_{\mathcal{A}}$ for receiving data from both cloud and edge. As the client needs to wait for the response of the cloud before requesting data from the edge, the two roundtrip delays add up, which can be seen on the right plot. In contrast, the response from the

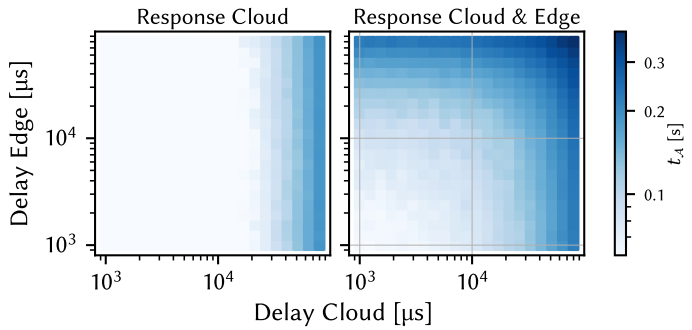


Figure 6.20 $t_{\mathcal{A}}$ from cloud (left) and cloud and edge (right).

cloud is entirely independent of the delay to the edge node since the two queries are fully decoupled.

Bandwidth

As shown in previous work, delays to both cloud and edge systems can vary considerably [5] when accessed over the Internet. In addition to that, systems might also be queried locally with a very low delay.

This is why we perform our bandwidth experiments for multiple delay scenarios. We define three representative delay scenarios: *Local Network* (RTT of 0.4 ms), *Low-Latency WAN* (RTT of 10 ms) and *High-Latency WAN* (RTT of 60 ms). We perform our experiments for all nine combinations of cloud/edge latency scenarios. For each scenario, we vary the bandwidth on both nodes from 128 kbit/s to 1 Gbit/s.

The results for this experiment are shown in Figure 6.21. The qualitative behavior is similar in all nine scenarios, but minimal values for $t_{\mathcal{A}}$ (top right area of each plot) depend on the respective delay scenario. In contrast, in low-bandwidth settings, the network delay has no significant impact anymore, as the time to transfer results dominates. The slight asymmetry in each of the nine plots stems from the fact that four times more data needs to be transferred from the edge. The missing noise in the high-bandwidth areas (top right) in all the high-latency scenarios shows that the latency of our system does not have a notable impact on overall $t_{\mathcal{A}}$ in these cases.

6.4.4 Real-World Multi-Node Case

We evaluate our system in a setting with three server nodes, as sketched in Figure 6.22. This scenario is adopted from a real-world use case for monitoring gas-fired power plants. On-site, a standard local network with 1 Gbit/s bandwidth is deployed. Sensor data is stored in an edge node (Node 1) as well as an intermediate on-site server equipped with additional storage space (Node 2). Overview data is available in a cloud instance (Node 3).

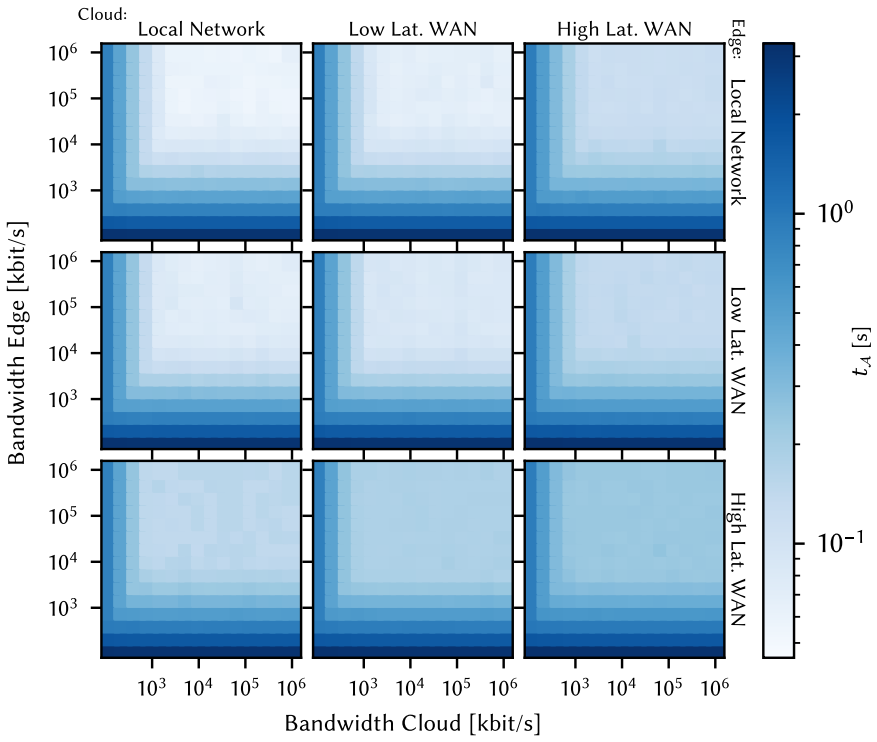


Figure 6.21 $t_{\mathcal{A}}$ for different latency scenarios of cloud (columns) and edge (rows) nodes. For each plot in the 3×3 grid, we vary bandwidth for edge and cloud nodes from 128 kbit/s to 1 Gbit/s.

We derive a dataset from the one described in Table 6.1 by selecting specific time spans and aggregations levels for each of the nodes. The resulting datasets are listed in Table 6.2, and the ranges covered are sketched in the right part of Figure 6.22. Node 1, being closest to the data source, contains all aggregation levels but only covers a limited time span of the most recent data. The second node contains all levels except for the raw data (L_0) and covers a slightly longer time span. The last node in the hierarchy misses the most recent data (which might not have been transmitted yet) and contains only coarse aggregations.

We define three representative query session scenarios (labeled *QSx*). The query sessions emulate an interactive user zooming into or out of the data, requesting three sensors in a resolution that results in at least 1000 data points. The three query session scenarios are shown in blue in the top row of Figure 6.23. The first scenario (*QS1*) queries data only available on the third node and zooms into the data in several steps. This emulates root-cause analysis for historical data, where a local user investigates an event for which only a rough timeframe was specified. In the second scenario (*QS2*), an on-site user starts from an overview of the complete time range and zooms in deep to a certain event. In the last scenario (*QS3*), the user zooms out from real-time monitoring

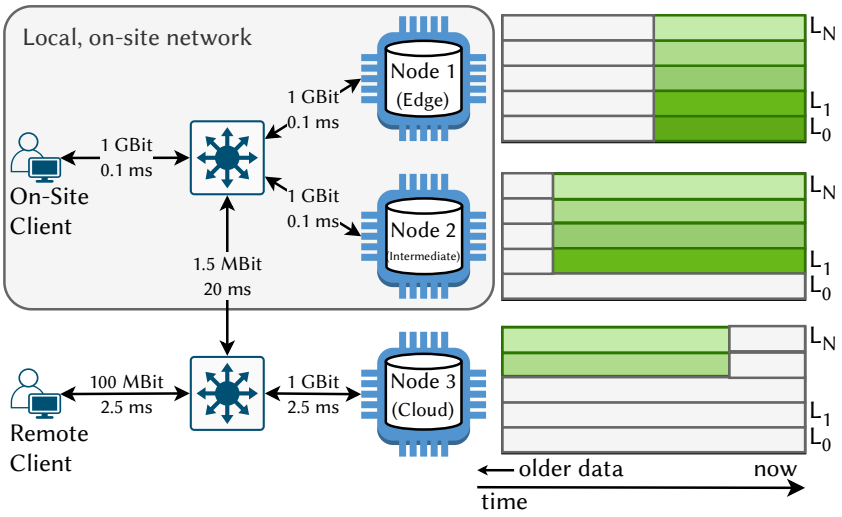


Figure 6.22 Setup of the experiment for Section 6.4.4. On the left, we show the network topology settings using typical values between clients and the three server nodes. On the right, we roughly sketch (in green) which part of the data (time range, aggregation level) is available on which of the three servers.

to investigate the history of a certain sensor. The hatching in the first row of the plot also sketches which parts of the data are available on which node.

In addition to the query session scenarios, we define three query node scenarios (*QNx*): in each query node scenario, the first query is directed at one of the three server nodes in our system, using the client that is closest to the respective server (Table 6.3). The first two nodes (Node 1 and Node 2) are queried from a client in the same local network. Node 3 is queried from a remote user connected via the Internet. The sketch in Figure 6.22 also shows network topology, bandwidth, and delay between the client and servers. The effective bandwidth and aggregated round-trip-times (RTT) are listed in Table 6.3.

We perform each combination of *QSx* and *QNx*—i.e., a total of nine experiments—and visualize the resulting query times $t_{\mathcal{A}}$ in the lower three rows of Figure 6.23.

We do not control the exact number of points but try to find an aggregation level that best matches the query. As a consequence, the query result size varies to some degree. This explains the zigzag pattern and spike that can be seen, e.g., on the bottom right plot, where data size and transfer rates impact $t_{\mathcal{A}}$.

For *QS2*, when Node 1 is queried first, the query is forwarded to Node 2. This additional local roundtrip to Node 1 is hardly noticeable: the plot below (*QS2*, Node 2), where the roundtrip is not necessary, shows very similar values for $t_{\mathcal{A}}$.

In cases when a certain query can be fully satisfied from a particular node, no other query is performed, independent of which node serves the data. This can be observed

Table 6.2 Data distribution for Section 6.4.4.

Node ID	Total Size	Resolutions	Timespan
Node 1 (Edge)	2.08 TB	50 µs – 14.9 h (all)	2022–02–09 – 2022–02–23
Node 2 (Intermediate)	410 GB	3.2 ms – 14.9 h	2022–01–26 – 2022–02–23
Node 3 (Cloud)	88 MB	13.1 s – 14.9 h	2022–01–20 – 2022–02–14

Table 6.3 Client Server Connectivity Map for Section 6.4.4.

1st Node queried	Client	Agg RTT [ms] Local	Agg RTT [ms] Remote	Bandwidth [Mbit/s] Local	Bandwidth [Mbit/s] Remote
Node 1	On-Site	≈ 0.4	≈ 45.2	1000	1.5
Node 2	On-Site	≈ 0.4	≈ 45.2	1000	1.5
Node 3	Remote	≈ 45.2	≈ 10	1.5	100

in *QS2*, where the mid-level resolutions (Query numbers 3–12) can be fully served either by Node 2 or Node 3.

In the design of our system, fast responses have been a primary goal, even if the first result is only an approximation. The bottom-row plot for *QS2* (remote client) shows that our system supports exactly this use case. When the resolution on the cloud is not sufficient anymore to fully satisfy the query, the data can nevertheless be used as a first approximation of the requested data and can be completed once the higher resolution data (received later) is available. The same behavior, even more noticeable for the user, is experienced on the remote client for the first few queries for *QS3*, where the first result from Node 2 is available a lot earlier than the result from Node 1.

Overall, there are notable differences in $t_{\mathcal{A}}$ across the various query scenarios and nodes queried. However, most of these differences are due to varying bandwidth and delay settings, while the overhead of our system is minimal.

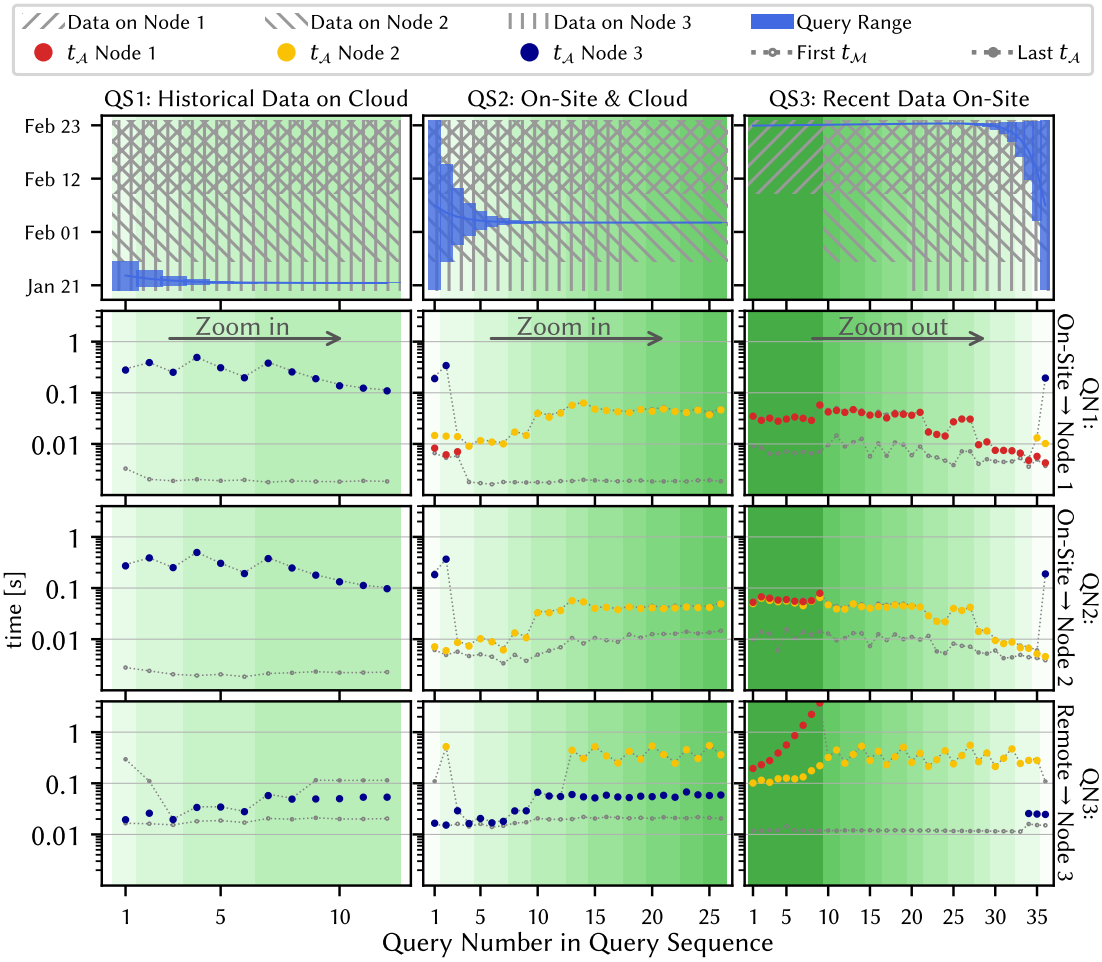


Figure 6.23 Multi-node use case query details and results for three different query sessions *QSx* (three columns). The three plots in the first row visualize two aspects: the query ranges used for the successive queries (marked in blue) and data time windows available on the three server nodes (hatched in gray). Data time windows are shown depending on the requested query resolution/level. The three bottom rows show $t_{\mathcal{A}}$ in the query sequence for each experiment. The green background indicates the queried level.

7 Integration into Industrial Infrastructure Monitoring Systems

This thesis is based on research performed in an application-oriented, industrial context. HAQSE addresses challenges derived from analyzing various real-world applications in this industrial context. The resulting approaches and scientific contributions can possibly benefit several parts of the sensor data processing pipeline in a wide range of industrial applications. As proof of concept, this chapter describes how the developed system can be integrated into an exemplary industrial sensor processing pipeline. We take the specific example from the introduction chapter (Section 1.1): a combustion monitoring system used in heavy-duty gas turbines. Such a system is developed by, e.g., IfTA GmbH. We demonstrate one possibility of how HAQSE could be integrated into that particular solution. More specifically, we describe how HAQSE is integrated into the existing system to receive, preprocess and store the generated sensor data. We also discuss how the existing analysis infrastructure can use HAQSE's hierarchical approximate query functionality and show how this improves sensor data analysis scenarios. In fact, many aspects of HAQSE are motivated by the experiences of engineers working with systems such as the one developed by IfTA GmbH.

This chapter is organized as follows. First, in Section 7.1, we discuss how—in general—HAQSE can be integrated into sensor processing systems. Then, in Section 7.2, we describe the most relevant components of IfTA's state-of-the-art combustion monitoring and protection system for gas-fired power plants. This overview helps to understand how HAQSE can be integrated into this solution. Next, in Section 7.3, we specifically explain how HAQSE can be integrated with the various components of IfTA's monitoring infrastructure. Finally, in Section 7.4, we demonstrate how HAQSE is integrated into the sensor data processing pipeline of a production-grade gas-fired power plant.

7.1 General Integration Guideline

Integrating HAQSE into an existing sensor processing pipeline requires a few steps. This section outlines what aspects need to be considered and how integrating HAQSE works in general. First, it is important that the rest of the system meets the requirements HAQSE assumes. Then, in the next step, the sensor stream input interface needs to be implemented to be able to stream sensor samples to HAQSE. Another essential part is retrieving data from HAQSE. Thus, the interface for querying original and ag-

gregated data streams needs to be implemented. Finally, like for any other storage system, it is also necessary to specify application requirements and configure HAQSE accordingly. We describe each of these steps in the following subsections.

7.1.1 Meeting HAQSE's Requirements

There are a few requirements that need to be fulfilled in order for HAQSE to be deployed in practice. These primarily reflect the aspects we discussed in Section 2.1.

Defined stream schema: The schema of a data stream needs to be clearly defined and may not change. If a schema change is required, a new stream must be created with a separate stream id. Schema changes include a change in the list of sensors (addition or removal of sensors; change in sensor name or type) and a change in the sensor source stream resolution.

Single source stream: HAQSE assumes each stream to only have one source. In HAQSE, different sources would be managed as different streams. In fact, separate sources often actually represent separate streams as they measure different phenomena. If, on the other hand, multiple such streams are indeed related, it can be desirable to integrate these streams into a single sensor source stream. This integration must happen in a separate system that prepares a single sensor source stream to send to HAQSE. This integration can happen in two dimensions: If the measured entity is the same, and the different streams contain samples with different timestamps, these different samples can be integrated by demultiplexing them into a single stream. Care needs to be taken regarding the order of timestamps (also see below). The other alternative is that multiple streams measure different phenomena but do so using identical sampling times. In this case, the different streams can be merged into a single stream, de-duplicating the timestamp information of the two streams. However, depending on the use case, keeping multiple streams and sending them as separate sensor streams to HAQSE might be more appropriate.

Synchronous, strictly increasing timestamps: Similar to other systems [95, 126], a stored sensor stream in HAQSE is required to have sensor values sampled synchronously. This means that measurements from different sensors must have identical or synchronous timestamps. In addition to that, for each sensor stream, timestamps must be strictly increasing. For multi-source streams, this can be hard to guarantee for a variety of problems like clocks being out of sync or different transmission delays. On the contrary, while it is also challenging, it is much easier to ensure for single-source streams.

Once these aspects are considered and ensured, in the next step, stream ingestion can be implemented.

7.1.2 Stream Ingestion

To insert sensor samples into HAQSE, the input interface presented in Section 3.4.1 needs to be implemented. Both a live sensor source stream and an existing sensor sample database can be used as the data source. The input interface is implemented utilizing a gRPC service. The actual sensor streaming is implemented with the help of a client streaming remote procedure call (RPC), i.e., a sensor data source executes an RPC method provided by HAQSE and uses it to stream sensor samples to HAQSE.

Specifically, this client streaming protocol consists of two parts, a stream definition and the actual sensor stream with measurement values:

1. First, the stream definition must be created based on the available data stream. It consists of the details defined in Section 3.2: the stream id, the stream schema and the stream resolution. In addition to that, it is possible to set further stream configuration parameters with this message. These are mainly the definition of the windowed aggregations to compute as well as the exact storage-related parameters like compression, row group and file sizes.
2. After the stream definition message has been sent, the actual sensor measurement values can be sent. This can happen either in sensor-ordered batches or by sending each sample individually. As shown in Section 5.5, if the generated stream rate is high, sending data in larger batches to HAQSE is highly beneficial.

To implement a data source, the gRPC interface specified in Appendix A.2 can be used to generate interface code in the programming language used for the respective client.

7.1.3 Query Processing

For retrieving data from HAQSE, an Apache Arrow Flight client has to be implemented in any of the supported programming languages. This client needs to execute the `DoGet` RPC as demanded by the Apache Arrow Flight specification. The query is specified as a JSON object. This contains the stream id and sensor names to retrieve data for. In addition to that, the time range of interest, as well as the number of data points to retrieve, are part of the query. More details regarding the implementation of the query interface are presented in Section 3.4.2. HAQSE responds with a stream of record batches. These record batches contain the requested data and can be processed however the respective application requires it. If data from multiple sources must be retrieved, the distributed querying logic from Section 6.2 must be implemented on the client.

7.1.4 Application Specific Configuration

Once the basic interfaces are connected, it is necessary to identify the application parameters and other internal HAQSE settings. These parameters are discussed in the

respective previous chapters. It is also essential to configure HAQSE according to the availability of hardware resources or even select appropriate hardware for deployment.

7.2 IfTA Infrastructure for Monitoring and Protecting Gas-Fired Power Plants

In this chapter, we demonstrate how HAQSE can be integrated into an existing sensor data processing system. Thus, this section describes the IfTA ArgusOMDS system developed by IfTA GmbH. We first provide a brief company overview (Section 7.2.1) and present the essential components of the IfTA ArgusOMDS system in Section 7.2.2. In Section 7.2.3, we explain the necessary details of the current file and stream format of the IfTA ArgusOMDS system to understand the integration details better. We describe the actual integration in Section 7.3

7.2.1 Application Context: IfTA GmbH

IfTA GmbH is a small company located in Puchheim, near Munich, Germany. It was founded in 1996 by Dr. Jakob Hermann based on his university research on active suppression of combustion dynamics in stationary gas turbines [45, 46, 99]. IfTA GmbH develops high-end vibration measurement technology. The developed system's use cases comprise combustion dynamics monitoring, rotor dynamic and torsional vibration analysis, among many others. Until today, its main field of application is monitoring and protecting the combustion process in stationary, heavy-duty gas turbines. IfTA GmbH develops both hardware and software to cover the complete sensor processing pipeline, starting after the sensor and ranging to sophisticated visualization and analysis solutions. Beyond that, IfTA GmbH continuously researches various related areas and offers consulting services for any kind of vibration problem. One fundamental product of IfTA GmbH is the IfTA ArgusOMDS system, which we describe next.

7.2.2 Existing Monitoring, Protection and Analysis Infrastructure: IfTA ArgusOMDS

The IfTA ArgusOMDS monitoring and protection system is a configurable and modular system of co-designed hardware and software. This system makes it possible to protect different classes of applications that require vibration monitoring. It can be integrated with other monitoring and control systems installed at a particular facility like a power plant or test bed. The system is specifically optimized for monitoring heavy-duty turbines in gas-fired power plants. It consists of several components, as sketched in Figure 7.1.

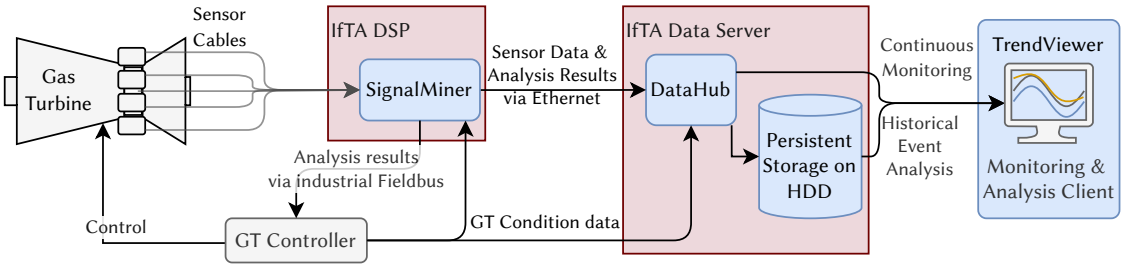


Figure 7.1 Overview of the IfTA ArgusOMDS monitoring and protection system. The red boxes indicate hardware components of the IfTA ArgusOMDS; the blue boxes indicate the most important software components.

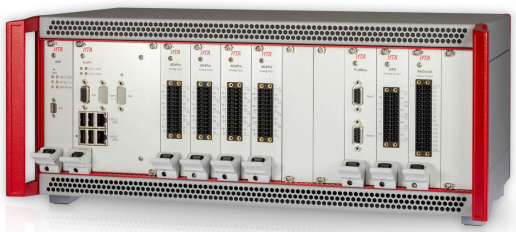


Figure 7.2 Example hardware configuration of an IfTA ArgusOMDS system [51].

At the core of the protection logic, the *IfTA SignalMiner* software is responsible for real-time analysis and protection. IfTA SignalMiner runs on special, real-time capable digital signal processing hardware. An exemplary hardware configuration is shown in Figure 7.2. It continuously analyzes short, tumbling or overlapping windows of sensor data. This sensor data is sampled from physically connected sensors[1] via integrated analog-digital conversion hardware channels. The analysis results calculated by IfTA SignalMiner comprise frequency spectra, frequency band analysis and many other application-dependent analyses. The raw sensor measurements and analysis results are used for two main purposes. First, via a real-time fieldbus network[2], they are sent to the gas turbine control system. This is done to inform the control system about problems in the combustion process or physical structure of the turbine in a timely manner, guaranteeing protection of the monitored machine. Second, raw sensor measurements and analysis results are sent to an instance of *IfTA DataHub*, running on a separate, industrial server computer. In IfTA DataHub, additional sensor data streams are collected and augment the raw sensor data and analysis results produced by IfTA SignalMiner. The sensor streams that flow into IfTA DataHub are stored persistently

[1]These are, e.g., acceleration sensors or high-temperature piezoelectric pressure sensors.
[2]Or an alternative connection to the gas turbine control system.

(and optionally aggregated over time) for later analysis tasks. This is done in a ring-buffer fashion, such that the oldest values get deleted on a regular basis. It uses time-ordered ADF files, as described in Section 7.2.3. At the same time, the sensor streams are provided as Argus Online Protocol (AOP) streams to further components in the system. One exemplary use of this stream is *IfTA TrendViewer*. This visualization and analysis software uses the stream to continuously visualize the most recent analysis results. In addition, IfTA TrendViewer can also be used to analyze and visualize historical sensor data streams, e.g., for investigating the root cause of a damage event. IfTA TrendViewer is typically executed on client devices like workstations or notebooks and connected through the internal plant network to IfTA's data server.

Generally, there is one IfTA ArgusOMDS system per monitored gas turbine. Thus, when multiple turbines are to be monitored and protected, all described components are replicated. However, parts like the IfTA data server hardware may be shared, as we will show in Section 7.4 for a specific deployment case.

7.2.3 IfTA Argus Data Format & Argus Online Protocol

In the IfTA ArgusOMDS system, sensor data streams are stored in a proprietary format: the Argus Data Format (ADF). In addition to that, IfTA DataHub offers the ability to send data streams containing live sensor data. This is done using the so-called Argus Online Protocol (AOP). While the AOP implements a concept similar to the Sensor Source Stream (Definition 4), the ADF represents a Stored Sensor Stream (Definition 7). The actual data stream sent in the AOP protocol is thus structurally very similar to the ADF: the ADF can simply be thought of as a serialized AOP stream or vice versa, streaming an ADF results in something very similar to the AOP[3]. Therefore, in the following, we only explain the necessary details for the ADF structure, as visualized in Figure 7.3. This explanation is also valid for the AOP.

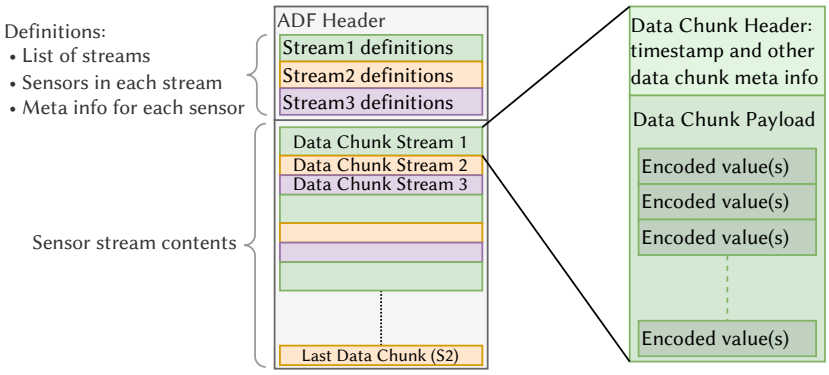


Figure 7.3 Rough sketch of ADF files and AOP stream.

[3]There are some minor technical differences in the implementation of the two formats. These are irrelevant to the discussions here.

Logically, an ADF file consists of several independent, multiplexed or interleaved sensor streams. On the highest level, the format consists of a header followed by the actual stream contents. The header contains information about the different sensor streams contained in the file as well as metadata about each of the sensors for every stream. The actual stream content consists of a sequence of *data chunks*. Each data chunk belongs to one of the sensor streams in the file. Data chunks of one such stream, especially those originating from the IfTA ArgusOMDS system, have strictly increasing timestamps. A data chunk is made up of a *data chunk header* and the actual stream payload. The header contains the stream identification (associating the data chunk with a stream), a timestamp that is valid for all the contents in the respective data chunk as well as additional information irrelevant to this discussion here. The stream payload then contains a binary representation of the actual sensor values. All data chunks of a certain stream have the same payload size.

In ADF, each sensor is represented by one of several possible signal types. This information determines how the stream payload is interpreted; it is encoded in the file header. Most interesting for the discussion here are the types *Raw Signal*, *Spectrum Signal* and *Value Signal* (also see Figure 7.5):

- *Raw Signals* contain a short window of a raw sensor signal, typically containing as many samples as one IfTA SignalMiner analysis window. Like for all other signal types, the respective header in the stream format only contains a single timestamp. This timestamp indicates the timestamp of the first raw sample in the data chunk. The timestamps of the subsequent measurement values in the chunk are derived based on the sampling rate. This sampling rate is specified in the meta information of the sensor stream, contained in the ADF header.

- *Spectrum Signals* represent frequency spectrum data. Such signals also contain multiple values, one amplitude value per frequency line of a spectrum. Again, important spectrum information like frequency resolution or the number of frequency lines is stored in the meta information available in the header of the sensor stream. The timestamp in the data chunk header is identical for all values in the spectrum.

- *Value Signals* are the simplest type of sensor. They only contain a single value per sensor and data chunk. The timestamp in the data chunk header is valid for all sensor values in the payload of the data chunk.

Due to the ordering of data chunks and the structure of the ADF, the format is a time-ordered or row-oriented format for *Spectrum* and *Value Signals*. For *Raw Signals*, data is actually partly sensor-ordered since all values of one sensor are contiguous for *Raw Signals* in the payload.

The ADF header stores additional information for all sensors. This information includes the measurement value unit, typical value ranges and textual sensor descrip-

tions. Since HAQSE does not (yet) have the ability to store such data, we do not need to discuss these aspects of the ADF in more detail.

The last data chunk of the ADF ends the file or stream. There is no footer or similar terminating part.

7.3 Integrating HAQSE into IfTA Monitoring Infrastructure

This section describes how HAQSE has been integrated into the state-of-the-art monitoring system IfTA ArgusOMDS. Before we start explaining the details of this integration, we discuss what tasks HAQSE can replace or augment in the existing infrastructure.

HAQSE's main objective is long-term storage and management of large-scale sensor data. In the current system, this task is (at least to some degree) performed by IfTA DataHub. For our integration into the rest of the existing system, we deploy HAQSE as a valuable *additional* component in the whole system. This decision implies that some data is stored redundantly in the two separate systems. Still, it makes the transition of the complete infrastructure—which includes several other tools, especially in the post-processing pipeline—much easier.

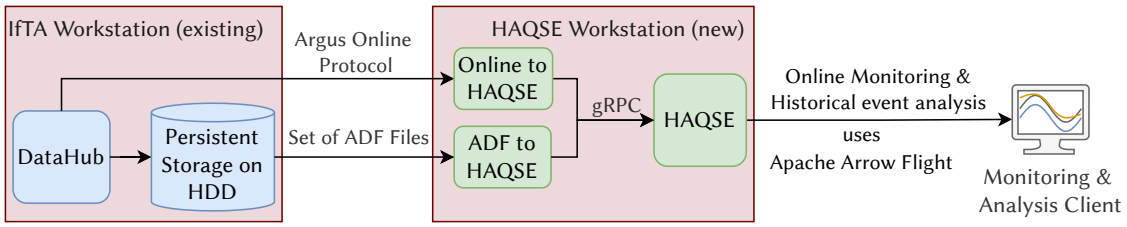


Figure 7.4 Rough sketch of how HAQSE is integrated into the existing software infrastructure of IfTA. The components sketched in green are added for HAQSE.

We integrate the two systems as shown in Figure 7.4. This figure shows HAQSE running on a separate physical machine. This deployment is one possible alternative rather than a strict requirement. Letting both systems run on a single machine is also a conceivable deployment alternative. There are two possible ways to get sensor data into HAQSE. First, it is possible to connect to the AOP stream offered by IfTA DataHub. Second, data can also be ingested from a set of permanently stored ADF files. Since both approaches have valid application scenarios, we implement both of them. Finally, the visualization and analysis client IfTA TrendViewer can be extended with functionality to query hierarchical sensor data from HAQSE.

The rest of this section is structured as follows. First, in Section 7.3.1, we explain how sensor data streams or collections of ADF files can be prepared for ingestion to

HAQSE. Next, in Section 7.3.2, we show that HAQSE can be used as a data source for driving interactive, exploratory data analysis of timespans that would have been unthinkable before. Lastly, in Section 7.3.3, we lay out how HAQSE builds the foundation for training a machine-learning-based anomaly detection system that uses combustion spectrum data. This new anomaly detection component dramatically benefits from the improved query functionality of HAQSE and would only be possible with a much higher effort in the current system.

7.3.1 Data Input: Connection to Argus Data Format & Argus Online Protocol

In a first step, it is necessary to get data into HAQSE. As outlined above and sketched in Figure 7.4, there are two possible ways to achieve this: reading from a collection of ADF files or connecting to and reading from an AOP stream. In both cases, the sensor signals described in the header of the ADF/AOP stream and selected for writing to HAQSE need to be mapped to HAQSE's sensor stream input model (cf. Section 3.2.1). As the two alternatives are structurally very similar (see Section 7.2.3), we make no distinction between the two approaches in the following explanation.

We describe mapping the three data types presented in Section 7.2.3 to their respective representation in HAQSE. This mapping logic is visualized in Figure 7.5.

- For *Raw Signals*, the data represents multiple consecutive measurements of a single sensor, resulting in multiple rows of one sensor stream column in the HAQSE data model. However, in HAQSE, all timestamps are stored explicitly, whereas in ADF, only one timestamp is stored. Consequently, we generate the derived timestamp values for all samples in a data chunk, as visualized in the top part of Figure 7.5. Moreover, since values are already partly sensor ordered, we can use the batched input in HAQSE's input stream protocol. Generating batches of the size equal to the number of values in one *Raw Signals* data chunk makes input into HAQSE more efficient and requires very little layout transformation processing.

- *Spectrum Signals* also contain multiple values, but each of these values belongs to the same timestamp. In HAQSE, sensors are always fixed-size scalar values. This fixed-size requirement makes it necessary to split up spectrum signals into separate sensor streams. Each of these sensor streams in HAQSE represents one of the contained frequency lines of the mapped spectrum signal. For mapping *Spectrum Signals* to HAQSE, we thus create a schema with one sensor per frequency line, as illustrated in the center of Figure 7.5. Data chunks of this type result in an input batch size of one.

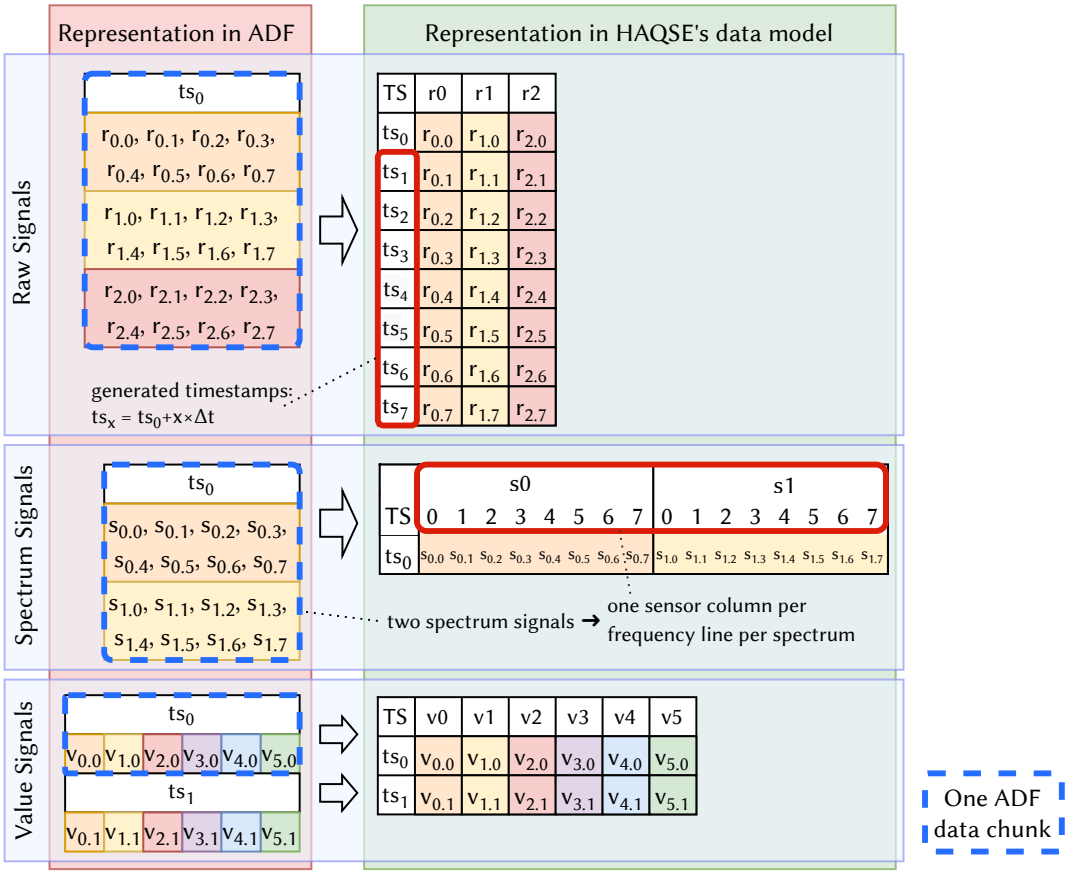


Figure 7.5 Different signal types in ADF and how they are mapped to HAQSE. The left part shows the ADF data chunk representation; the right part shows the representation in HAQSE's data model.

- *Value Signals* are the most straightforward case. They are mapped one-to-one from ADF to a HAQSE. Similar to *Spectrum Signals*, value signals generate individual input samples without batching.

As discussed, a sensor stream in HAQSE requires all sensors in a schema to be sampled at identical timestamps, and it requires all sensors to have a value for each timestamp. Thus, only sensors that fulfill this property can be put into a common schema. As a consequence, specifically for the ADF format, *Raw Signals* need to be in a separate sensor stream chunk (containing only *Raw Signals*) in order to be able to map this chunk (as a single sensor stream) to HAQSE.

Another requirement discussed previously (Section 2.1) is that HAQSE expects strictly increasing timestamps for the input data stream. We should note that all sensor streams that originate from IfTA SignalMiner fulfill this property. This is achieved by a control

loop on the real-time hardware. This loop ensures that the sample clock[4] is ticking—on average, over a long time period—exactly at the desired frequency. Long-term time drifts are avoided by controlling this loop with a global master clock. This global master clock is synchronized to some global time source via, e.g., a protocol like Network Time Protocol [82] (NTP).

As visualized in Figure 7.3, an ADF file stream contains many multiplexed sensor streams. For streaming data to HAQSE, we demultiplex this stream and generate multiple HAQSE compatible streams. For each of these resulting streams, a gRPC client streaming call is created.

For actually connecting to an AOP stream or reading a collection of ADF files, we create two separate CLI programs: an AOP adapter `online2haqse` and an ADF file reader adapter, `adf2haqse`. Both of these programs first read the ADF header (either from the AOP stream or the ADF file). Using the information in the header and the CLI configuration of the respective program, a stream description message is created and sent as the first message in the gRPC client streaming protocol. Then, the programs iterate over the contained data chunks, filtering for chunks (and signals in the selected chunks) configured to be sent to HAQSE. For each of these data chunks, the data chunk header is read first, extracting the timestamp information. Then, all selected signal values are read, and a gRPC message is created and sent via the client stream.

The resulting input streams have very different properties. They differ in schema size (very wide schema for *Spectrum Signals*, rather narrow schema for *Raw Signals*), sampling rate, and use different batch sizes for input. Thus, our adapter programs (acting as HAQSE gRPC clients) can be adjusted to allow configuring how HAQSE should process and store the respective streams. This includes storage parameters like row group size, compression algorithms as well as aggregation hierarchy settings.

Both adapter programs connect to a running HAQSE instance. For `adf2haqse`, the adapter requires a list of files to iterate. For `online2haqse`, the adapter connects to a running IfTA DataHub instance and uses one of the available AOP streams. Once all supplied files have been processed or the AOP input stream terminates, the gRPC client stream ends. At that point, all data is available in HAQSE.

7.3.2 Fast Interactive Exploratory Data Analysis

In this section, we describe how IfTA TrendViewer (see Section 7.2.2) can be extended to support HAQSE as data source. We first explain the typical interactive visualization workflow in IfTA TrendViewer based on ADF or AOP data sources. Then, we show the scalability issues of the current approach. Finally, we sketch how a HAQSE data source could be designed to overcome these scalability issues.

[4]The sample clock is responsible for driving the actual analog-digital signal conversion performed on the real-time hardware. In the IfTA ArgusOMDS system, this clock guarantees that all attached sensors are sampled at exactly the same instant (only ignoring errors introduced by different clock signal line lengths).

Visualization Workflow in IfTA TrendViewer

Figure 7.6 shows the user interface of IfTA TrendViewer. Before visualizing any data,

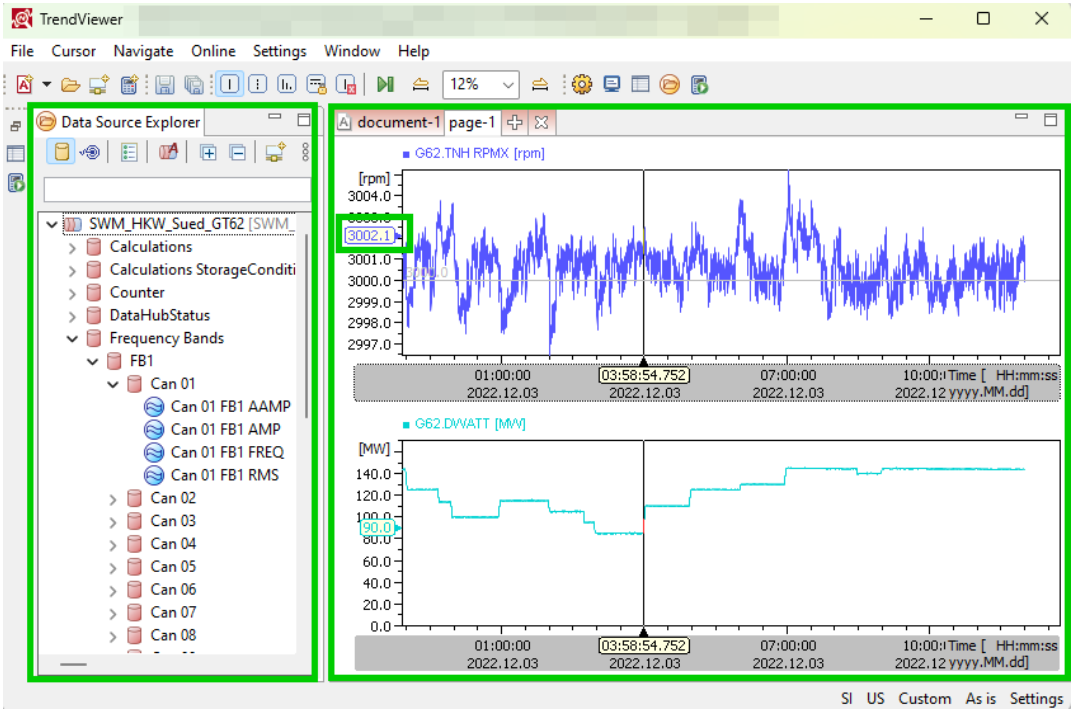


Figure 7.6 Graphical user interface of IfTA TrendViewer. The left green box highlights the data source explorer: this shows a hierarchy of all available streams and sensors from the ADF file. The right green box shows the plotting area: in this example, two signals are visualized in two plots. The small green box shows that IfTA TrendViewer displays the exact value of the current cursor position. This screenshot of IfTA TrendViewer is shown with permission from IfTA GmbH.

the user needs to create a data source (left part of the window). Currently, data sources in IfTA TrendViewer can be classified into two different categories. As one option, data is loaded from one or multiple files. This can be, e.g., ADF or CSV files. Alternatively, data can also directly come from a sensor source stream. IfTA TrendViewer can for example connect to an AOP stream offered by IfTA DataHub. This is implemented by writing the received data into a temporary file and reading data from that temporary file for visualization. In the following, we focus on the file-based method, as the stream-based method is internally handled in a similar way.

To create a data source, the user opens one or multiple files (or selects a sensor stream published via the AOP). As explained above, these files contain several sensor streams and cover a certain time range. When opening an ADF file, an index is constructed, creating a mapping from the contained timestamps to the positions of the data chunks inside the file. This requires reading through the entire file once. It is also possible to

use compressed files. In that case, these compressed files must first be decompressed completely in a local temporary folder; then, the uncompressed file is used as data source. Additional files need to be loaded when other time ranges are required for visualization.

Once a data source is created, the contained sensor streams and sensors can be explored in the data source explorer (see Figure 7.6, left part). One or multiple such sensors are then selected for visualization in one or multiple plots (see Figure 7.6, right part). For the discussion here, we restrict ourselves to the so-called *trend plot*, which shows the sensor values over time. Such a trend plot shows one line per selected sensor.

One of the key features of IfTA TrendViewer is the ability to interact with the plot and explore the available data. These interactions include zooming in and out of the time axis or shifting the time axis. All these actions modify the visualized time window. Furthermore, the same can be done on the value axis. If the data source is a live sensor stream received via the AOP, this updating can be done automatically so that IfTA TrendViewer updates the plot periodically with the most recent values[5]. When interacting with the plot with the time axis cursor, IfTA TrendViewer also shows the exact value of the plotted signal at the respective instant (this is also shown in Figure 7.6).

Scalability Challenges With Current Data Sources

The current file-based approach in IfTA TrendViewer has two main scalability issues. First, opening files (which may involve decompression) and creating the required mapping from timestamps to data chunks scales linearly with the file size. Second, reading measurement values from files for visualization also requires iterating over the complete part of the file that represents the time span to be visualized.

In this section, we primarily look at the overhead when reading measurement values from files. While opening large files may also interrupt the user's workflow, this only needs to be done once if intelligent caching strategies are implemented. For plotting sensor data in a trend plot, IfTA TrendViewer iterates over all samples for the respective sensor(s) in the selected time range. This approach works reasonably well for short time ranges as long as the number of data points is manageable (see below). However, when the time range spans billions of samples[6], iterating over the data quickly becomes a bottleneck. In addition to that, as discussed previously, the ADF is optimized for writing, not for reading individual signals.

For visualizing data such as in Figure 7.6, IfTA TrendViewer iterates over the whole part of the selected time range. The exact algorithm for drawing these lines is irrelevant for the performance aspects we analyze here: iterating over the measurement values

[5] IfTA TrendViewer can, for example, show the most recent 30 seconds of sensor data and update this plot every 200 ms.

[6] In the actual deployment example we present in Section 7.4, more than two billion raw data samples are generated per day.

in the file alone scales linearly with the amount of data and, thus, with the required time span.

We performed a simple experiment to illustrate the problem. On a typical workstation that stores data on an HDD, we iterated over an increasing amount of sensor samples and measured the execution time of such an iteration. The workstation is equipped with an Intel® Core™ i7-9700K CPU, running at 3.60 GHz. It has 32 GiB of 2666 MHz memory and is running Windows 10. The data is located on an internal 3.5" 500 GB Western Digital HDD. The resulting execution times are shown in Figure 7.7.

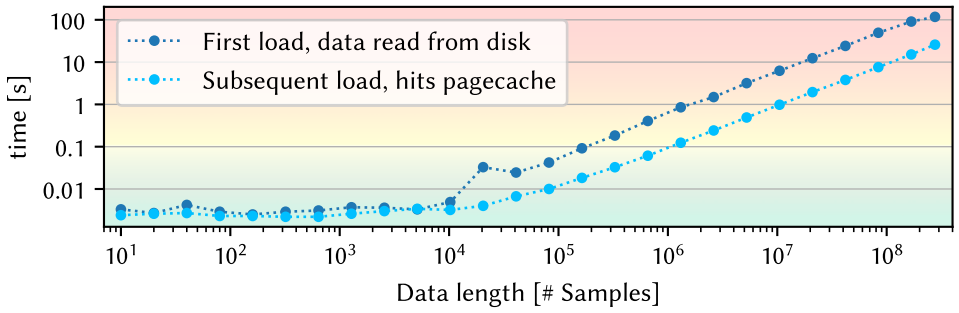


Figure 7.7 Time for reading data from storage for plotting as a function of data points to read. The background color indicates how the delay is perceived by users, using the 100 ms limit found in human-computer interaction literature [20, 33, 81].

This plot shows several aspects. First, there is a significant difference between reading data for the first time compared to subsequent reads. This is because the data is buffered in the operating system's page cache. For interactive data exploration, data is very likely to be present in page cache when updating the plot contents, so we now focus on the lower curve in the plot. Up to roughly 20 000 samples, there is no real difference in data loading times. Up to approximately 10^6 samples, reading data gets slower but is still fast enough for interactive settings [33]. For longer time windows to be visualized, the time for reading the data increases further, significantly degrading the user experience. It should be noted that cases with numbers such as 10^8 are not purely made up: when visualizing one hour of measurement values sampled at 25.6 kHz for a single sensor, sizes are already in that order of magnitude.

While the exact execution times and thresholds will vary depending on the specific environment, it is clear that the general linear behavior will stay the same.

Creating a HAQSE Data Source for TrendViewer

To be able to use data from HAQSE in IfTA TrendViewer, we create a new data source. This data source requires specifying the address of a running HAQSE server. Instead of opening an ADF file, users create a HAQSE data source. Once created, the available data streams and the underlying signals are queried from the server. This has the advantage that the time to create such a HAQSE data source is independent of the

amount of data available on the server. Users do not need to wait for the file to be loaded (or even for data to be decompressed). Creating such a data source takes a constant amount of time, typically in the order of a few seconds.

For the actual visualization, an appropriate query is created based on the current plot width and selected time range. The query can then retrieve min and max pre-aggregated values. If the query result actually uses pre-aggregated data, the min and max values can be used directly as input to the visualization layer. The resulting plot represents a very good approximation of the visualization result generated with the original resolution stream data. When visualizing large timespans, this is orders of magnitudes faster since the amount of data that needs to be loaded from HAQSE depends solely on the size of the plot[7]. As our results in Section 6.3 show, the latency caused by the roundtrip to the server and back is mainly determined by the network latency. We argue that this delay is hardly noticeable to users in local network environments.

A few aspects are currently missing in HAQSE to make its integration with IfTA TrendViewer smoother. We discuss these in Section 8.1.

7.3.3 Driving Machine-Learning-Based Anomaly Detection

HAQSE can also be used in more data-heavy scenarios. In this section, we describe an anomaly detection algorithm using HAQSE as a data source for training and inference.

As a first measure, the monitoring data and analysis results acquired and generated by the IfTA ArgusOMDS system are used for real-time protection. This real-time protection reacts within hundreds of milliseconds and detects quickly developing problematic behavior that may lead to machine damage. However, the spectrum data generated by IfTA ArgusOMDS can also be used to detect slowly developing changes in the observed behavior leading to damage events. An excerpt of a timeline of such a slowly developing damage event is shown in Figure 7.8. For such slowly developing problems, long time windows are required for algorithm development. IfTA GmbH develops a machine-learning-based anomaly detection algorithm.

This algorithm must be trained on several weeks of spectral data, thus requiring a significant amount of data (for a specific turbine monitoring configuration, see Section 7.4). The data pipeline that is used to train this algorithm uses HAQSE to retrieve aggregated versions of spectral data: Since the algorithm's goal is to detect slowly developing changes in the data, it is trained on averaged spectrum values. HAQSE can be configured to generate such aggregations on data ingestion, considerably reducing the amount of data that needs to be loaded for training the machine learning algorithm. Instead of raw values, only aggregated values are used for training. This can result in a data reduction factor of up to three orders of magnitude regarding data that

[7] Provided an appropriate sensor stream hierarchy is available on the chosen HAQSE server.

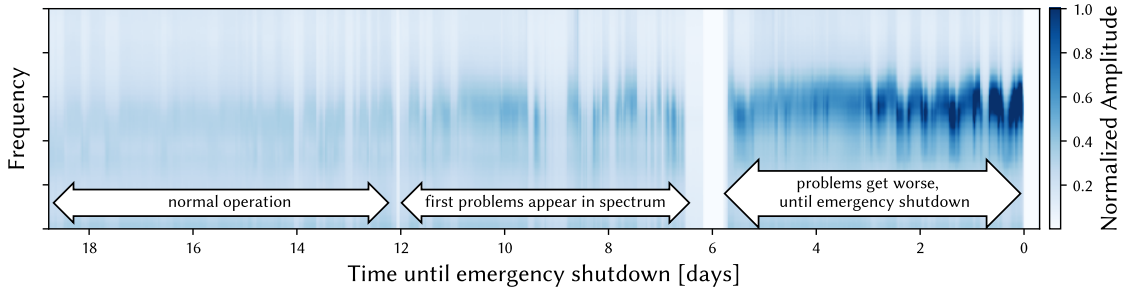


Figure 7.8 Excerpt of a frequency spectrum showing a slowly developing damage event. A planned shutdown roughly six days before the emergency shutdown could have been used to inspect the machine if the problem had been detected in the spectrum at that point. Even two days before the emergency shutdown, there would have been enough time to schedule a planned stop and investigate the problem that was (then) clearly visible in the spectrum.

needs to be read for training the machine learning model. Consequently, this data size reduction can shorten model training time considerably.

Since inference aims to detect slowly developing changes in machine behavior, a new prediction is generated in regular intervals. All spectrum data available for a particular turbine is used to calculate an anomaly score. For this application, HAQSE is queried regularly and provides the most current (non-aggregated) values for frequency spectra and operating data.

In summary, HAQSE provides a solid platform that can be used both for efficient training and inference in machine learning scenarios.

7.4 Integrating HAQSE into the Sensor Processing Pipeline of a Gas-Fired Power Plant

Using the integration tools presented in Section 7.3.1, we deploy HAQSE in the gas-fired power plant of a municipal services company. Specifically, we integrate HAQSE on a separate workstation in the IfTA ArgusOMDS monitoring infrastructure located at Heizkraftwerk Süd (HKW Süd), operated by Stadtwerke München GmbH (SWM) [102]. With this, we show that HAQSE meets the requirements of and can be integrated into a real-world industrial monitoring application.

The monitored plant site at HKW Süd consists of two combined cycle gas turbines, named *GT61* and *GT62*. They both use the same turbine type, a GE 9E [35] (see Figure 7.9). Each turbine has a nominal maximum power output of roughly 140 MW. The plant operates in a combined heat and power scheme, additionally acting as a source for district heating, supplying several neighborhoods in Munich with heat. Each gas turbine comprises 14 combustion cans, and a pressure sensor monitors each of these cans.

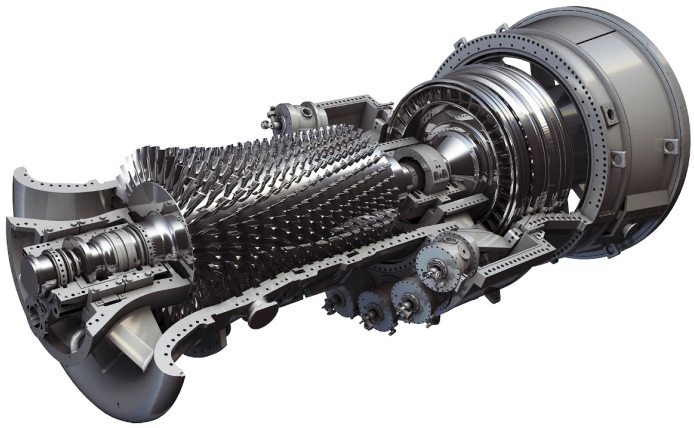


Figure 7.9 GE 9E gas turbine [35].

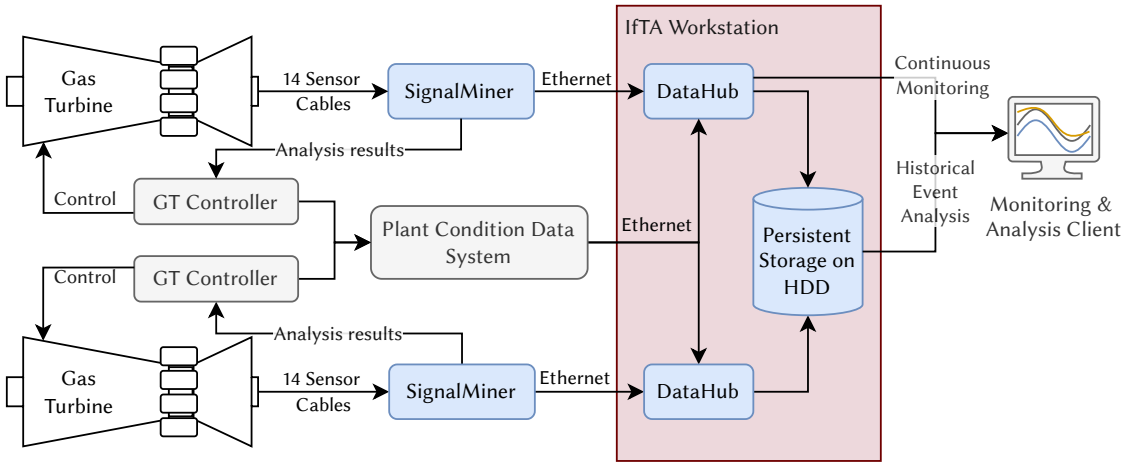


Figure 7.10 IfTA monitoring infrastructure at SWM.

The signals from these sensors are collected and analyzed in real-time by the IfTA ArgusOMDS monitoring system, one separate instance per turbine. The deployment for this particular site is sketched in Figure 7.10. The real-time monitoring hardware components are duplicated such that each turbine has one independent protection instance. Each such instance has a separate connection to the turbine's controller to communicate analysis results. In contrast to this, the two IfTA DataHub instances are running on a single workstation.

Both IfTA ArgusOMDS systems at HKW Süd are configured in the following way (summarized in Table 7.1). They sample the 14 sensors at a sampling frequency of 25.6 kHz and continuously analyze tumbling windows of 8192 single-precision (32 bit) floating-point samples (corresponding to an analysis window length of 320 ms). For each window, a frequency magnitude spectrum is calculated, resulting in a spectrum of

Table 7.1 Analysis specification of IfTA monitoring and protection system at HKW Süd of SWM.

Setting	Value
Sampling rate	25 600 Hz
Analysis window length	320 ms
Analysis rate	$(320\,\mathrm{ms})^{-1} = 3.125\,\mathrm{Hz}$
FFT Length and window sample count	8192
Size of output spectrum (per sensor)	3200 Lines
Number of sensors per turbine	14 (one sensor per can)
Approximate data rate per GT	2 MB/s

3200 single-precision floating-point frequency lines[8]. Based on the raw sensor stream and the calculated spectrum, IfTA SignalMiner computes more complex analyses in the time and frequency domain. All analysis results are sent to IfTA DataHub. In addition to that, some of these analysis results are communicated to the gas turbine controller. Simple analysis results and condition data contribute only little to the overall data rate. The major contributors are the raw data values and the frequency spectrum data. Hence, the total data rate HAQSE needs to process in this setup can be estimated as follows[9]:

$$\begin{aligned} r_{\mathrm{raw}} &= 25\,600\,\frac{\mathrm{values}}{\mathrm{s}} \times \frac{4\,\mathrm{B}}{\mathrm{value}} = 102.4\,\mathrm{kB/s} \\ r_{\mathrm{spectrum}} &= 3200\,\frac{\mathrm{values}}{0.320\,\mathrm{s}} \times \frac{4\,\mathrm{B}}{\mathrm{value}} = 40.0\,\mathrm{kB/s} \end{aligned} \tag{7.1}$$

Summing this up for two turbines, we get:

$$\begin{aligned} r_{1\mathrm{GT}} &= 14 \times (r_{\mathrm{raw}} + r_{\mathrm{spectrum}}) \\ &= 14 \times (102.4\,\mathrm{kB/s} + 40.0\,\mathrm{kB/s}) \\ &= 1993.6\,\mathrm{kB/s} \\ &\approx 2\,\mathrm{MB/s} \\ r_{\mathrm{total}} &= 2 \times r_{1\mathrm{GT}} \\ &\approx 4\,\mathrm{MB/s} \end{aligned} \tag{7.2}$$

As shown in Section 5.5, HAQSE should be able to easily consume this rate, even with many aggregation levels and compression active.

[8] The upper ≈20 % of the full frequency magnitude spectrum of 4192 lines are cut off due to the properties of the low-pass filter that is applied on the raw sensor samples to avoid aliasing effects in the frequency domain.

[9] When including additional analysis results, the resulting data rate is slightly (2–3 %) higher.

For the particular case of the HKW Süd, we use the `online2haqse` adapter and read the following six streams per turbine to put them into HAQSE (cf. Table 7.1):

- Raw sensor signal stream, containing 14 sensors (encoded as *Raw Signals*). A data chunk thus contains 14×8192 single-precision floating-point values per sensor, or roughly 460 kB.
- Spectrum stream, containing 14 frequency spectra (*Spectrum Signals*), each consisting of 3200 single-precision floating-point frequency lines. Such a data chunk has a size of roughly 180 kB.
- Four additional streams containing mostly scalar analysis result signals and condition data (*Value Signals*), only marginally contributing to the overall data rate.

The two IfTA SignalMiner instances create streams with identical names. Therefore, the `online2haqse` adapter prefixes these streams with the respective turbine name (*GT61* or *GT62*).

Utilizing the results from Section 4.4, we use the Byte Stream Split encoding and combine it with zstd for all floating-point sensors. The timestamp column uses a bit-packed delta-encoding (`DELTA_BINARY_PACKED`). All other columns are stored in an uncompressed way. One day of turbine data consumes roughly 176.5 GB of storage space when stored in uncompressed ADF files. When compressing these files with zstd, this size can be reduced to 127.9 GB. Storing the same data in HAQSE (without aggregation) consumes 112.1 GB, i.e., requiring only 63.5 % of the initial, uncompressed size. This further reduction in contrast to the zstd compression alone can be explained by the Byte Stream Split encoding for floating-point data. Adding aggregations (*min*, *max*, *mean*, *stddev*, and *count*) increases the consumed storage space to 129.6 GB for one day of gas turbine data. This storage size is a bit more than what the compressed ADF database consumes but enables quick overviews of the whole day (or even longer time spans) of sensor data.

8 Future Work and Conclusions

This chapter concludes the work presented in this dissertation. In Section 8.1, we outline shortcomings and discuss potential future research and development directions. Finally, in Section 8.2, we summarize the work and results presented in this thesis.

8.1 Future Work

Even though HAQSE improves over the state of the art in many aspects, there are still a lot of opportunities for improvement. Like any complex software project, this comprises software engineering improvements, but also includes other conceptual or technical aspects. We list these aspects in the following paragraphs.

Lossy Compression So far, our two-stage compression method in HAQSE has only been used for lossless compression. However, we also expect our approach to be effective for lossy compression with almost no modifications. This can be achieved by dropping the contents of the least significant bits of the mantissa. In order to validate this, the performance for this lossy scheme would need to be compared against other existing lossy compression algorithms like SZ3 and zfp [72, 73]. In addition to that, the influence of the amount of information loss on the respective application needs to be investigated.

Persisting Stream Segment Index In its current form, the Stream Segment Index is entirely stored in volatile memory (RAM). This has the following two implications: First, starting up HAQSE needs to open all files to recreate the index (even though not all file contents need to be read). When the number of files HAQSE manages is large, this process considerably slows down the creation of the Stream Segment Index and, thus, the startup of HAQSE. Second, for very large databases, the Stream Segment Index may unnecessarily require a large amount of RAM. Persisting the Stream Segment Index can help to avoid these two problems.

Memory Improvements for Stream Consumption Consuming sensor streams in HAQSE requires multiple kB per sensor in the stream schema. While this is unproblematic for small sensor schemas or systems with a lot of RAM available, it can be prohibitive especially for wide stream schemas or smaller systems. It would thus also

be worthwhile to investigate alternative ingestion methods that require less memory. Similarly, as shown in Section 5.5.6, the current approach for compacting data is not optimal for large compacted file sizes. Combining data from temporary files as soon as enough sensor samples are available might make compaction more memory-efficient and thus, faster.

Query Processing Range-based queries can be processed very quickly in HAQSE. There is, however, potential to explore query performance in more depth, especially regarding concurrent querying/ingestion scenarios. In addition, evaluating HAQSE with respect to multiple concurrent queries remains to be investigated.

The query interface, however, is limited to range-based data retrieval using the custom query protocol. To make adoption in third-party tools easier, offering a standard query interface would be desirable (e.g., SQL). Initiatives like Apache Arrow Flight SQL [117] offer a standardized way to achieve this with the employed technology stack. Another natural addition, once an interface as described above is available, is a more sophisticated query processing component. There is a general trend in the database processing field: query execution engines are being implemented in open source libraries such as Apache Arrow DataFusion [113] or Velox [79, 89] and thus, become a commodity. Integrating such a query engine in HAQSE can substantially improve its query capabilities. The underlying storage and in-memory processing technology provide a good starting point for integrating such systems.

Application Examples HAQSE is based on clearly defined concepts regarding sensor stream input and range-based data retrieval. While we show its utility for a real-world application in Chapter 7, other potential application scenarios could benefit from using HAQSE. One such example is data center monitoring, where HAQSE could be used as a solution for efficient long-term storage of monitoring metrics (e.g., as a component of projects like DCDB [84]). Furthermore, it would be interesting to see how HAQSE can be integrated as sensor storage system in large-scale physics experiments.

General HAQSE currently does not handle sensor metadata like units, signal description, or relations between stored sensor streams. For users of these sensor streams, this information is crucial when interpreting data. Furthermore, it enables improved user interfaces by supporting interaction based on metadata information. In addition to that, applications using sensor data stored in HAQSE like IfTA TrendViewer need to be adapted to make the most of the new possibilities HAQSE offers.

8.2 Conclusions

HAQSE makes it possible to consume, process and store high-volume sensor streams, and it makes the stored sensor data accessible via a query interface. As we show in

the respective evaluation sections, it handles each aspect very efficiently. It can consume sensor data at the rate of Gbit Ethernet, which is at least eight times faster than any other time series management system we tested (Section 5.6). Similarly, it stores data more efficiently than any other time series management system (Section 4.4.7). Our method makes it possible to compress data to 50 % of its uncompressed size when applied to a typical industrial dataset. The second best (existing) options achieve only around 66 % in that case. We showed that the employed compression works for a variety of different data sets (Section 4.4.5) and is faster than any other method (Section 4.4.6). By relying on pre-computed windowed aggregations, HAQSE can respond to range-based data queries at latencies that enable interactive data exploration, independent of the size of the requested time range. Our approach for integrating data from multiple, distributed instances enables distributed queries. By deploying HAQSE in real-world production environments (Chapter 7), we show that our approach satisfies the requirements of real applications. All these results make it possible for contemporary and future sensor processing applications to increase sensor sampling frequencies and data rates further, supporting hundreds of kHz for schemas that contain up to 100 single-precision floating-point sensors. For smaller schema sizes, HAQSE even supports sampling rates in the MHz range (Section 5.5.8). HAQSE is expected to speed up sensor data analysis considerably. Thus, it helps operators and engineers in decision-making and developing new sensor analysis methods.

Appendices

A Implementation Details of HAQSE

A.1 HAQSE File System Layout

We store metadata information (especially schema information) in HAQSE together with the data files containing the actual measurement values (Apache Parquet or temporary files, see Chapter 5 for details). Stream ids and resolutions are stored as folder names in the file system. The resulting file system layout looks like this:

```
haqse-root
├── <stream-id-1>
│   ├── o.....................................contains source stream
│   │   └── <source-resolution>
│   │       ├── <first-ts-in-file-1>.parquet
│   │       ├── <first-ts-in-file-2>.parquet
│   │       └── ...
│   └── a.....................................contains all aggregated streams
│       ├── <agg-resolution-1>
│       │   ├── <first-ts-in-agg-file-1>.parquet
│       │   ├── <first-ts-in-agg-file-2>.parquet
│       │   └── ...
│       ├── <agg-resolution-2>
│       ├── <agg-resolution-3>
│       └── ...
├── <stream-id-2>
├── <stream-id-3>
└── ...
```

A.2 gRPC Input Protocol Definition

The following protobuf code represents the full gRPC protocol definition for the sensor input interface of HAQSE.

```
1 syntax = "proto3";
2 package ifta.haqse.streamproto;
3
4 // only use lite messages (we don't need descriptors for now)
```

```
option optimize_for = LITE_RUNTIME;

service ServerInput {
  rpc StreamSensorData(stream SensorStreamElement) returns (StreamStatus) {}
}

enum BufStrategy {
  OneMemBuffer = 0;
  OneFileBuffer = 1;
  OneMemBufferNonTemporal = 2;
}

enum Compression {
  None = 0;
  zstd = 1;
  lz4 = 2;
  brotli = 3;
}

enum Encoding {
  Plain = 0;
  bss = 1;
  delta_packed = 2;
}

message ParquetComprSettings {
  Encoding e = 1;
  Compression c = 2;
  int32 c_lvl = 3;
  bool dict_enabled = 4;
}

message WriterCfg {
  uint32 bufCount = 1;
  uint32 rowGroupSize = 2;
  uint32 rowGroupsPerTempFile = 3;
  uint32 tempFilesPerCompacted = 4;
  uint32 rowGroupSizeCompacted = 5;
  BufStrategy bufStrategy = 6;
  ParquetComprSettings defaultCmpr = 7;
  map<string, ParquetComprSettings> typeComprMap = 8;
  string colWrtrStrategy = 9;
  string syncStrategy = 10;
}

message AggDef {
  uint64 resolution_nanos = 1;
  WriterCfg writerCfg = 2;
}

```

```
message AggStoreCfg {
  WriterCfg srcWriterCfg = 1;
  repeated AggDef aggs = 2;
}

enum FieldType {
  Bool = 0;
  Int8 = 1;
  Int16 = 2;
  Int32 = 3;
  Int64 = 4;
  UInt8 = 5;
  UInt16 = 6;
  UInt32 = 7;
  UInt64 = 8;
  Float32 = 9;
  Float64 = 10;
  TimeStampNanos = 11; // nanoseconds since UTC 1970-01-01 00:00
}

message SchemaField {
  string name = 1;
  FieldType type = 2;
}

message StreamSchema {
  repeated SchemaField fields = 1;
}

enum ContentType {
  PlainBinary = 0; // little endian plain C encoding
  MsgPack = 1;     // experimental, just for testing
}

message StreamDefinition {
  string stream_id = 1;
  StreamSchema schema = 2;
  uint64 resolution = 3;
  ContentType content_type = 4; // maybe this can be part of the actual sensor data
  AggStoreCfg agg_cfg = 5;
}

message SingleSampleData {
  int64 timestamp = 1;
  bytes stream_content = 2;
}

message SampleBatchData {
  uint64 sample_count = 1;
  bytes timestamps = 2;
```

```
  bytes values = 3;
}

message SensorStreamElement {
  oneof StreamDefOrContent {
    // implementation needs to enforce that first message is always the StreamDefinition
    StreamDefinition stream_definition = 1;
    // all messages after the first one need to be SensorData messages
    SingleSampleData single_sample = 2;
    SampleBatchData batch_samples = 3;
  }
}

enum StatusCode {
  Ok = 0;
  Failed = 1;
  Unknown = 2;
  NotImplemented = 3;
  StreamIncompatible = 4;
}

message StreamStatus {
  string message = 1;
  StatusCode code = 2;
}
```

A.3 Example HAQSE Query Client

The following short snippet shows how a basic HAQSE query client (without support for retrieving data from distributed instances) could be implemented in Python. This simple client just requires the `pyarrow.flight` package and its dependencies to be available. An example query is shown in Section 3.4.2.

```
import pyarrow.flight as flght
import json

def query(url: str, query: dict):
    clnt = flght.FlightClient(url) # create apache arrow flight client
    query_json = json.dumps(query).encode("utf-8") # encode query
    query_response = clnt.do_get(flght.Ticket(query_json)) # perform request

    # read record batches from resulting stream object and return as a list
    return [rb for rb in query_response]
```

A.4 Query Sequence

The following sequence diagram shows the sequence of actions when retrieving data from HAQSE.

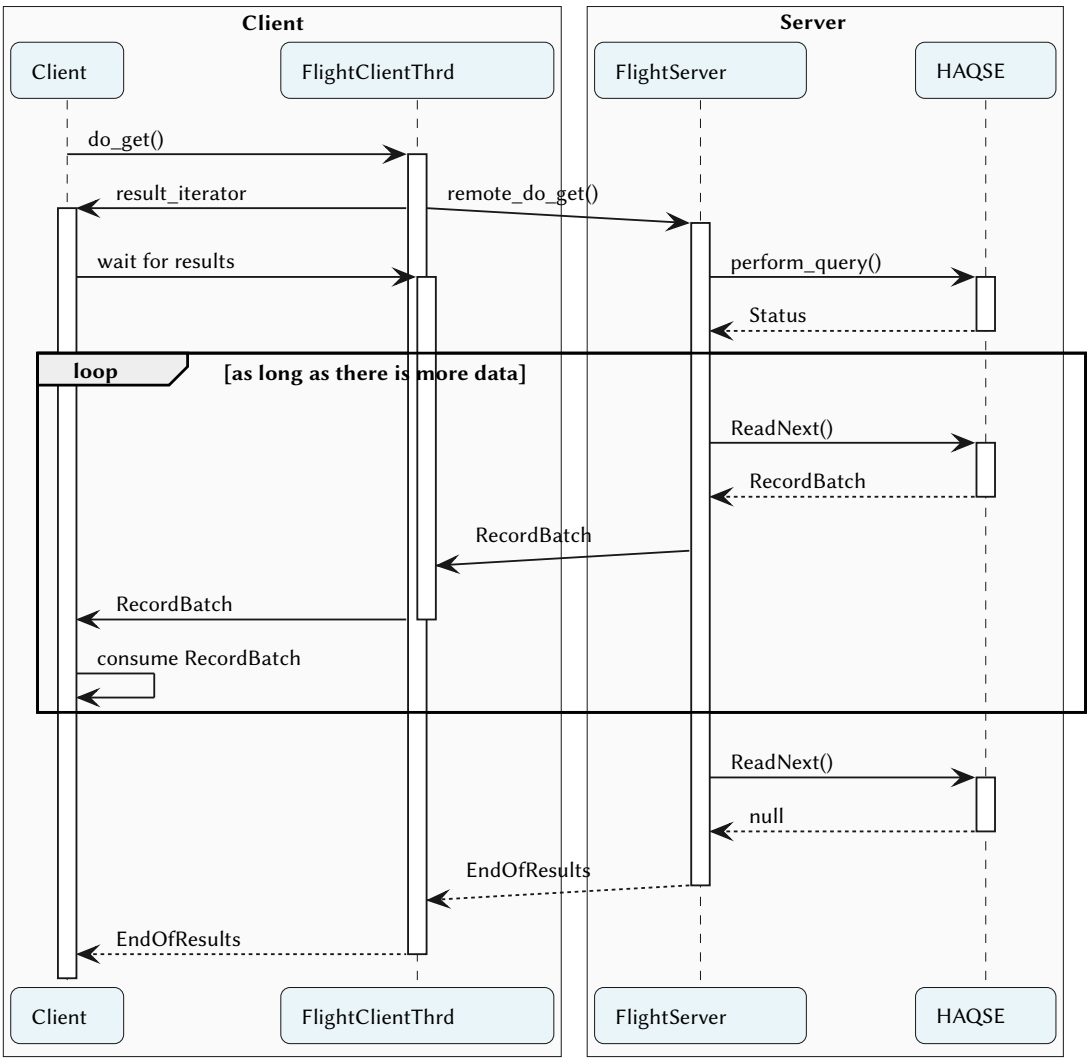


Figure A.1 Sequence diagram showing the interaction between client and server when querying data from a single HAQSE server.

Acronyms

ADF Argus Data Format. 70–72, 168–176, 181
AOP Argus Online Protocol. 168, 170, 171, 173–175
API Application Programming Interface. 9, 90, 91
AQP approximate query processing. 31
AVX Advanced Vector Extensions [57]. 58, 60–62, 198
CLI command line interface. 88, 173
CPU central processing unit. 7, 10, 19, 25, 26, 56, 57, 59–62, 69, 70, 80, 86, 95, 101, 107, 121, 128, 129, 153, 176, 198
CSV comma separated values. 28, 174
FFT Fast Fourier Transform. 3, 14, 20
HDD Hard Disk Drive. iii, 68, 87, 91–93, 106, 114, 118, 124, 139, 141, 150, 176
HKW Süd Heizkraftwerk Süd. 178–181
HPC High Performance Computing. 19, 29
HTTP Hypertext Transfer Protocol. 90, 96, 97, 125, 126, 129
ILP Influx Line Protocol. 28, 70, 72, 90, 91, 96–98, 125, 126
IoT Internet of Things. 35
ISA Instruction Set Architecture. 60
JSON JavaScript Object Notation [59]. 37, 45, 46, 165
NTP Network Time Protocol [82]. 173
RAM random-access memory. 7, 26, 40, 60, 68, 76, 87, 116, 140, 153, 183
RPC remote procedure call. 165, 197
SIMD Single Instruction, Multiple Data. 10
SQL Structured Query Language. 28, 91, 96, 98, 125, 126, 184
SSD Solid-State Drive. 68, 87
SSE Streaming SIMD Extensions [57]. 57, 58, 60, 61, 198
STFT short-time Fourier transform. 20
SWM Stadtwerke München GmbH. iii, 178–180
TCP Transmission Control Protocol. 90, 95–97, 125, 126
TDP thermal design power. 21, 60
TSM Time-Structured Merge Tree [56]. 27, 28

Glossary

Apache Arrow Apache Arrow is a development platform for in-memory analytics. 35, 43, 46, 51, 56, 57, 60, 68, 81, 104, 132, 138–143, 150, 152

Apache Arrow columnar format The Apache Arrow columnar format [9] includes a language-agnostic in-memory data structure specification, metadata serialization, and a protocol for serialization and generic data transport. 43, 197

Apache Arrow Flight Apache Arrow Flight [8] is an RPC framework for high-performance data services based on Apache Arrow data, and is built on top of gRPC and the Apache Arrow IPC format format. 34, 43, 46, 165, 184

Apache Arrow IPC format See Apache Arrow columnar format. 81, 82, 104–106, 112, 138, 197

Apache Parquet Apache Parquet [116, 129] is an open-source, column-oriented data file format designed for efficient data storage and retrieval. The contained columnar data is stored in one or multiple row groups. 10, 28, 34, 42, 43, 47, 51, 55–57, 59, 62–65, 68–73, 75, 80, 82, 112, 114, 116, 132, 138, 140–150, 152, 153, 189, 197, 198

Byte Stream Split An encoding for Apache Parquet, which helps to improve compression for floating-point sensor data. 10, 47, 54, 55, 57–63, 65–70, 72, 112, 138, 146, 181

Column Chunk A column chunk in the Apache Parquet format stores the data of one column of a row group and consists of one or multiple data pages. 197, 198

Data Page A data page represents one logical unit of encoding and compression in Apache Parquet. One or multiple data pages make up a column chunk. 197

gRPC gRPC is a modern open source high performance RPC framework that can run in any environment [43]. 34, 43, 45, 81, 88–90, 93–95, 99, 101, 102, 106, 111, 119, 122, 126, 165, 173, 189, 197

IfTA ArgusOMDS Monitoring and protection system developed by IfTA GmbH. Its main application is monitoring the combustion process in heavy-duty gas turbines for electricity generation. An overview is presented in Section 7.2.2. 166–170, 173, 177–179

IfTA DataHub Data collection and aggregation server of IfTA GmbH. 167, 168, 170, 173, 174, 179, 180

IfTA GmbH *IfTA Ingenieurbüro für Thermoakustik GmbH* is a small innovative company in Puchheim, Germany. It creates innovative systems for oscillation monitoring, specifically for protecting stationary, heavy-duty gas turbines against thermoacoustic instabilities. More background is provided in Section 7.2.1. Website: `https://www.ifta.com`. iii, 11, 163, 166, 174, 177, 197, 198

IfTA SignalMiner Monitoring and protection firmware developed by IfTA GmbH running on IfTA's realtime capable hardware. 167, 169, 172, 180, 181

IfTA TrendViewer IfTA's visualization and analysis software. This can be used for both online monitoring and analysis of historical data. 168, 170, 173–177, 184

Protocol Buffers Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. 43

Row Group A row group represents a continuous slice of multiple rows of the complete schema stored together as one unit in the Apache Parquet format. It consists of one or multiple column chunks. 51, 75, 197

Stream Segment Index A tree-based indexing data structure that manages segments of sensor data for fast retrieval. 10, 40, 41, 74, 76–78, 80, 82, 131–133, 136, 142, 183

Vector Instruction Vector instructions are processor instructions that exploit the vectorized processing capabilities of modern CPUs, e.g., instructions from the Intel SSE or AVX vector instruction sets. 10, 61

Bibliography

[1] Daniel Abadi et al. “The Design and Implementation of Modern Column-Oriented Database Systems”. In: *Foundations and Trends® in Databases* 5.3 (Jan. 2013), pp. 197–280. DOI: 10.1561/1900000024.

[2] Ulf Adams. “Ryū: Fast Float-to-String Conversion”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 270–282. ISBN: 9781450356985. DOI: 10.1145/3192366.3192369. URL: https://doi.org/10.1145/3192366.3192369.

[3] Pritom Saha Akash, Wei-Cheng Lai, and Po-Wen Lin. *Online Aggregation based Approximate Query Processing: A Literature Survey*. 2022. DOI: 10.48550/ARXIV.2204.07125.

[4] Jyrki Alakuijala and Zoltan Szabadka. *Brotli Compressed Data Format*. RFC 7932. July 2016. DOI: 10.17487/RFC7932. URL: https://rfc-editor.org/rfc/rfc7932.txt.

[5] Ahmed Ali-Eldin, Bin Wang, and Prashant Shenoy. “The Hidden Cost of the Edge: A Performance Comparison of Edge and Cloud Latencies”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476142. URL: https://doi.org/10.1145/3458817.3476142.

[6] Michael P Andersen and David E. Culler. “BTrDB: Optimizing Storage System Design for Timeseries Processing”. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 39–52. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/andersen.

[7] Vo Ngoc Anh and Alistair Moffat. “Index compression using 64-bit words”. In: *Software: Practice and Experience* (2010), n/a–n/a. DOI: 10.1002/spe.948.

[8] Apache Software Foundation. *Arrow Flight RPC*. Accessed April, 2023. 2023. URL: https://arrow.apache.org/docs/format/Flight.html.

[9] Apache Software Foundation. *IPC File Format*. Accessed December, 2022. 2022. URL: https://arrow.apache.org/docs/format/Columnar.html#ipc-file-format.

[10] iPerf authors. *iPerf*. Accessed January, 2023. 2023. URL: https://iperf.fr/.

[11] Maurice J. Bach. *The design of the UNIX operating system*. Prentice-Hall, 1986, p. 471. ISBN: 0132017997. URL: https://archive.org/details/designofunixoper00bach/page/128.

[12] Rudolf Bayer. "Symmetric Binary B-Trees: Data Structure and Algorithms for Random and Sequential Information Processing". In: *Acta Informatica*. 1971.

[13] Jadeed Beita et al. "Thermoacoustic Instability Considerations for High Hydrogen Combustion in Lean Premixed Gas Turbine Combustors: A Review". In: *Hydrogen* 2.1 (Jan. 2021), pp. 33–57. DOI: 10.3390/hydrogen2010003.

[14] Michael Borkowski, Christoph Hochreiner, and Stefan Schulte. "Minimizing Cost by Reducing Scaling Operations in Distributed Stream Processing". In: *Proc. VLDB Endow.* 12.7 (Mar. 2019), pp. 724–737. ISSN: 2150-8097. DOI: 10.14778/3317315.3317316. URL: https://doi.org/10.14778/3317315.3317316.

[15] Norman Bourassa et al. "Operational Data Analytics". In: *Workshop Proceedings of the 48th International Conference on Parallel Processing*. ACM, Aug. 2019. DOI: 10.1145/3339186.3339210.

[16] Andrea Bruno et al. "TSXor: A Simple Time Series Compression Algorithm". In: *String Processing and Information Retrieval*. Ed. by Thierry Lecroq and Hélène Touzet. Cham: Springer International Publishing, 2021, pp. 217–223. ISBN: 978-3-030-86692-1.

[17] Martin Burtscher and Paruj Ratanaworabhan. "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data". In: *IEEE Transactions on Computers* 58.1 (Jan. 2009), pp. 18–31. DOI: 10.1109/TC.2008.131.

[18] Burtscher, Martin. *Scientific IEEE 754 32-Bit Single-Precision Floating-Point Datasets*. Accessed June, 2020. 2020. URL: https://userweb.cs.txstate.edu/~burtscher/research/datasets/FPsingle/.

[19] Burtscher, Martin. *Scientific IEEE 754 64-Bit Double-Precision Floating-Point Datasets*. Accessed June, 2020. 2020. URL: https://userweb.cs.txstate.edu/~burtscher/research/datasets/FPdouble/.

[20] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. "The information visualizer, an information workspace". In: *Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '91*. ACM Press, 1991. DOI: 10.1145/108844.108874.

[21] Ugur Cayoglu et al. "Data Encoding in Lossless Prediction-Based Compression Algorithms". In: *2019 15th International Conference on eScience (eScience)*. 2019, pp. 226–234. DOI: 10.1109/eScience.2019.00032.

[22] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. “Approximate Query Processing: No Silver Bullet”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, May 2017. DOI: 10.1145/3035918.3056097.

[23] Steven Claggett, Sahar Azimi, and Martin Burtscher. “SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data”. In: *2018 Data Compression Conference*. IEEE. IEEE, Mar. 2018, pp. 335–344. DOI: 10.1109/DCC.2018.00042.

[24] Collet, Yann. *Zstandard-Fast real-time compression algorithm*. Accessed April 2020. 2020. URL: https://facebook.github.io/zstd/.

[25] *COMMISSION IMPLEMENTING DECISION of 10 February 2012 laying down rules concerning the transitional national plans referred to in Directive 2010/75/EU of the European Parliament and of the Council on industrial emissions*. Accessed Feburary 2023. Feb. 10, 2012. URL: https://eur-lex.europa.eu/eli/dec_impl/2012/115.

[26] Graham Cormode et al. “Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches”. In: *Foundations and Trends in Databases* 4.1-3 (2011), pp. 1–294. DOI: 10.1561/1900000004.

[27] Intel Corporation. *Intel oneAPI DPC++/C++ Compiler. A Standards-Based, Cross-architecture Compiler*. 2023. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html (visited on 04/20/2023).

[28] cppreference.com. *Release-Acquire Ordering*. Accessed December, 2022. 2022. URL: https://en.cppreference.com/w/cpp/atomic/memory_order#Release-Acquire_ordering.

[29] D. Cutting and J. Pedersen. “Optimization for dynamic inverted index maintenance”. In: *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR ’90*. ACM Press, 1990. DOI: 10.1145/96749.98245.

[30] Sheng Di and Franck Cappello. “Fast Error-Bounded Lossy HPC Data Compression with SZ”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2016, pp. 730–739. DOI: 10.1109/IPDPS.2016.11.

[31] Sheng Di and Franck Cappello. “Optimization of Error-Bounded Lossy Compression for Hard-to-Compress HPC Data”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.1 (Jan. 2018), pp. 129–143. DOI: 10.1109/TPDS.2017.2749300.

[32] Avrilia Floratou et al. “Dhalion: Self-Regulating Stream Processing in Heron”. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1825–1836. ISSN: 2150-8097.

[33] Valentin Forch et al. “Are 100 ms Fast Enough? Characterizing Latency Perception Thresholds in Mouse-Based Interaction”. In: *Engineering Psychology and Cognitive Ergonomics: Cognition and Design*. Springer International Publishing, 2017, pp. 45–56. DOI: 10.1007/978-3-319-58475-1_4.

[34] Free Software Foundation, Inc. *GNU Gzip*. Accessed Mai 2023. 2020. URL: https://www.gnu.org/software/gzip/.

[35] General Electric. *9E gas turbine*. Accessed April 2023. 2023. URL: https://www.ge.com/gas-power/products/gas-turbines/9e.

[36] Jongmin Gim and Youjip Won. “Extract and infer quickly: Obtaining sector geometry of modern hard disk drives”. In: *ACM Transactions on Storage* 6.2 (July 2010), pp. 1–26. DOI: 10.1145/1807060.1807063.

[37] B. Goeman, H. Vandierendonck, and K. de Bosschere. “Differential FCM: increasing value prediction accuracy by improving table usage efficiency”. In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 2001, pp. 207–216. DOI: 10.1109/HPCA.2001.903264.

[38] Adrián Gómez-Brandón et al. “Lossless compression of industrial time series with direct access”. In: *Computers in Industry* 132 (Nov. 2021), p. 103503. DOI: 10.1016/j.compind.2021.103503.

[39] Google. *Snappy, a fast compressor/decompressor*. Accessed Mai 2023. 2022. URL: https://github.com/google/snappy/blob/main/README.md.

[40] Google LLC. *Protocol Buffers Documentation*. Accessed April 2023. 2023. URL: https://developers.google.com/protocol-buffers.

[41] Torbjorn Granlund et al. *du*. Accessed February 2023. 2023. URL: https://man7.org/linux/man-pages/man1/du.1.html.

[42] gRPC Authors. *Core concepts, architecture and lifecycle*. Accessed April 2023. 2022. URL: https://grpc.io/docs/what-is-grpc/core-concepts/.

[43] gRPC Authors. *gRPC*. Accessed April 2023. 2023. URL: https://grpc.io/.

[44] John L. Gustafson and Isaac T. Yonemoto. “Beating Floating Point at its Own Game: Posit Arithmetic”. In: *Supercomputing Frontiers and Innovations* 4.2 (June 2017), pp. 71–86. ISSN: 2409-6008. DOI: 10.14529/jsfi170206. URL: https://doi.org/10.14529/jsfi170206.

[45] Jakob Hermann. *Anregungsmechanismen und aktive Dämpfung (AIC) selbsterregter Verbrennungsschwingungen in Flüssigkraftstoffsystemen*. German. In: *Fortschrittsberichte VDI*. Energietechnik. Vol. 6.364. 1997. ISBN: 3183364069.

[46] Jakob Hermann and Stefan Hoffmann. “Implementation of Active Control in a Full-Scale Gas-Turbine Combustor”. In: *Combustion Instabilities in Gas Turbine Engines: Operational Experience, Fundamental Mechanisms, and Modeling*. Ed. by Timothy C. Lieuwen and Vigor Yang. Vol. 210. American Institute of Aeronautics and Astronautics, Jan. 2006. Chap. 19, pp. 611–634. ISBN: 978-1-56347-669-3. URL: https://www.researchgate.net/publication/294428184.

[47] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. “Megaphone: Latency-Conscious State Migration for Distributed Streaming Dataflows”. In: *Proc. VLDB Endow.* 12.9 (May 2019), pp. 1002–1015. ISSN: 2150-8097. DOI: 10.14778/3329772.3329777. URL: https://doi.org/10.14778/3329772.3329777.

[48] Qun Huang and Patrick P. C. Lee. “Toward High-Performance Distributed Stream Processing via Approximate Fault Tolerance”. In: *Proc. VLDB Endow.* 10.3 (Nov. 2016), pp. 73–84. ISSN: 2150-8097. DOI: 10.14778/3021924.3021925. URL: https://doi.org/10.14778/3021924.3021925.

[49] “IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems”. In: *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)* (2020), pp. 1–499. DOI: 10.1109/IEEESTD.2020.9120376.

[50] “IEEE Standard for Binary Floating-Point Arithmetic”. In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

[51] IfTA GmbH. *Schwingungsmessung mit Argusaugen*. Accessed April 2023. 2022. URL: https://www.ifta.com/produkte/systemloesungen/argusomds-diagnose-schutz.html.

[52] Thomas Ilsche. “Energy Measurements of High Performance Computing Systems: From Instrumentation to Analysis”. PhD thesis. Dresden University of Technology, Germany, 2020. URL: https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-716000.

[53] Thomas Ilsche et al. “MetricQ: A Scalable Infrastructure for Processing High-Resolution Time Series Data”. In: *2019 IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control (DAAC)*. IEEE, Nov. 2019, pp. 7–12. DOI: 10.1109/DAAC49578.2019.00007.

[54] InfluxData Inc. *InfluxDB*. Accessed Juli 2021. 2021. URL: https://docs.influxdata.com/.

[55] InfluxData Inc. *InfluxDB file system layout*. Accessed December 2022. 2022. URL: https://docs.influxdata.com/influxdb/v2.5/reference/internals/file-system-layout/.

[56] InfluxData Inc. *InfluxDB storage engine*. Accessed December 2022. 2022. URL: https://docs.influxdata.com/influxdb/v2.5/reference/internals/storage-engine/.

[57] Intel Corporation. *Intel Intrinsics Guide*. Accessed Mai 2023. 2023. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[58] ISO. *ISO/IEC 14882:2020 Programming languages — C++*. Sixth Edition. Geneva, Switzerland: International Organization for Standardization, Dec. 2020. URL: https://www.iso.org/standard/79358.html.

[59] *ISO/IEC 21778:2017 - The JSON data interchange syntax*. Tech. rep. ECMA, Dec. 2017. URL: https://www.iso.org/standard/71616.html.

[60] Daniel Jäger. "Improvement of the Time Accuracy of a Vibration Measurement System". In: *Bachelor Thesis* (2019).

[61] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. "Time Series Management Systems: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 29.11 (Nov. 2017), pp. 2581–2600. DOI: 10.1109/TKDE.2017.2740932.

[62] Junekey Jeon. *Dragonbox: A New Floating-Point Binary-to-Decimal Conversion Algorithm*. 2022. URL: https://raw.githubusercontent.com/jk-jeon/dragonbox/master/other_files/Dragonbox.pdf.

[63] Jeroen T. Vermeulen. *libpqxx: pqxx::stream_to Class Reference*. Accessed October 2022. 2021. URL: https://libpqxx.readthedocs.io/en/6.4/a01211.html#details.

[64] Samuel C. Johnson et al. "Evaluating rotational inertia as a component of grid reliability with high penetrations of variable renewable energy". In: *Energy* 180 (Aug. 2019), pp. 258–271. DOI: 10.1016/j.energy.2019.04.216.

[65] Roman Karlstetter, Robert Widhopf-Fenk, and Martin Schulz. "Querying Distributed Sensor Streams in the Edge-to-Cloud Continuum". In: *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*. IEEE, July 2022. DOI: 10.1109/edge55608.2022.00035.

[66] Roman Karlstetter et al. "Living on the Edge: Efficient Handling of Large Scale Sensor Data". In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE Computer Society, May 2021, pp. 1–10. DOI: 10.1109/CCGrid51090.2021.00010. URL: https://doi.org/10.1109/CCGrid51090.2021.00010.

[67] Roman Karlstetter et al. "Turning Dynamic Sensor Measurements From Gas Turbines Into Insights: A Big Data Approach". In: vol. Volume 6: Ceramics; Controls, Diagnostics, and Instrumentation; Education; Manufacturing Materials and Metallurgy. Turbo Expo: Power for Land, Sea, and Air. V006T05A021. June 2019. DOI: 10.1115/GT2019-91259.

[68] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. “A Holistic View of Stream Partitioning Costs”. In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), pp. 1286–1297. ISSN: 2150-8097.

[69] Daniel Lemire and Leonid Boytsov. “Decoding billions of integers per second through vectorization”. In: *Software: Practice and Experience* 45.1 (May 2013), pp. 1–29. DOI: 10.1002/spe.2203.

[70] Kaiyu Li and Guoliang Li. “Approximate Query Processing: What is New and Where to Go?” In: *Data Science and Engineering* 3.4 (Sept. 2018), pp. 379–397. DOI: 10.1007/s41019-018-0074-4.

[71] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. “Chimp: Efficient Lossless Floating Point Compression for Time Series Databases”. In: *Proc. VLDB Endow.* 15.11 (Sept. 2022), pp. 3058–3070. ISSN: 2150-8097. DOI: 10.14778/3551793.3551852. URL: https://doi.org/10.14778/3551793.3551852.

[72] Xin Liang et al. “SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors”. In: *IEEE Transactions on Big Data* (2022), pp. 1–14. DOI: 10.1109/TBDATA.2022.3201176.

[73] Peter Lindstrom. “Fixed-Rate Compressed Floating-Point Arrays”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 2674–2683. DOI: 10.1109/TVCG.2014.2346458.

[74] Peter Lindstrom and Martin Isenburg. “Fast and Efficient Compression of Floating-Point Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 1245–1250. DOI: 10.1109/TVCG.2006.143.

[75] Linux. *dd(1) — Linux manual page*. Last Accessed March 2023. 2022. URL: https://man7.org/linux/man-pages/man1/dd.1.html.

[76] Linux. *madvise(2) - Linux manual page*. Last Accessed February 2023. 2021. URL: https://man7.org/linux/man-pages/man2/madvise.2.html.

[77] LZ4 Authors. *LZ4*. Accessed Mai 2022. 2022. URL: https://lz4.github.io/lz4/.

[78] Matthias Maiterth. “A reference model for integrated energy and power management of HPC systems”. en. PhD thesis. 2021. DOI: 10.5282/EDOC.28625.

[79] Meta Platforms, Inc. *Velox*. Accessed April 2023. 2023. URL: https://velox-lib.io/.

[80] Mike Freedman. *Building columnar compression in a row-oriented database*. Accessed April 2023. 2019. URL: https://www.timescale.com/blog/building-columnar-compression-in-a-row-oriented-database/.

[81] Robert B. Miller. “Response time in man-computer conversational transactions”. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS ’68 (Fall, part I)*. ACM Press, 1968. DOI: 10.1145/1476589.1476628.

[82] David L. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. June 2010. DOI: 10.17487/RFC5905. URL: https://www.rfc-editor.org/info/rfc5905.

[83] NERC. *Reliability Guideline. Primary Frequency Control*. Tech. rep. Version 2.0. NERC, May 2019. 34 pp. URL: https://www.nerc.com/comm/OC/RS_GOP_Survey_DL/PFC_Reliability_Guideline_rev20190501_v2_final.pdf (visited on 04/20/2023).

[84] Alessio Netti. “Holistic and Portable Operational Data Analytics on Production HPC Systems”. Dissertation. München: Technische Universität München, 2022.

[85] J. von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. DOI: 10.1109/85.238389.

[86] Shadi A. Noghabi et al. “Samza: Stateful Scalable Stream Processing at LinkedIn”. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1634–1645. ISSN: 2150-8097.

[87] Oracle. *ByteBuffer*. Accessed January 2023. 2022. URL: https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.

[88] Paul Dix, InfluxData Inc. *Announcing InfluxDB IOx - The Future Core of InfluxDB Built with Rust and Arrow*. Accessed December 2020. 2020. URL: https://www.influxdata.com/blog/announcing-influxdb-iox/.

[89] Pedro Pedreira et al. “Velox: Meta’s Unified Execution Engine”. In: *Proc. VLDB Endow.* 15.12 (Aug. 2022), pp. 3372–3384. ISSN: 2150-8097. DOI: 10.14778/3554821.3554829. URL: https://doi.org/10.14778/3554821.3554829.

[90] Tuomas Pelkonen et al. “Gorilla: A Fast, Scalable, in-Memory Time Series Database”. In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1816–1827. ISSN: 2150-8097. DOI: 10.14778/2824032.2824078. URL: https://doi.org/10.14778/2824032.2824078.

[91] Kasun S. Perera et al. “Efficient Approximate OLAP Querying Over Time Series”. In: *Proceedings of the 20th International Database Engineering & Applications Symposium on - IDEAS’16*. ACM Press, 2016. DOI: 10.1145/2938503.2938526.

[92] LLVM Project. *Clang: a C language family frontend for LLVM*. 2023. URL: https://clang.llvm.org/ (visited on 04/20/2023).

[93] QuestDB. *QuestDB*. Accessed November 2022. 2022. URL: https://questdb.io/.

[94] QuestDB. *QuestDB Root directory structure*. Accessed December 2022. 2022. URL: https://questdb.io/docs/concept/root-directory-structure/.

[95] QuestDB. *QuestDB Storage Model*. Accessed November 2022. 2022. URL: https://questdb.io/docs/concept/storage-model/.

[96] Mark Raasveldt and Hannes Mühleisen. "Don't hold my data hostage: a case for client protocol redesign". In: *Proceedings of the VLDB Endowment* 10.10 (June 2017), pp. 1022–1033. DOI: 10.14778/3115404.3115408.

[97] Xiangnan Ren et al. "Strider: An Adaptive, Inference-Enabled Distributed RDF Stream Processing Engine". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1905–1908. ISSN: 2150-8097. DOI: 10.14778/3137765.3137805. URL: https://doi.org/10.14778/3137765.3137805.

[98] Y. Sazeides and J.E. Smith. "The predictability of data values". In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. 1997, pp. 248–258. DOI: 10.1109/MICRO.1997.645815.

[99] J. R. Seume et al. "Application of Active Combustion Instability Control to a Heavy Duty Gas Turbine". In: *Journal of Engineering for Gas Turbines and Power* 120.4 (Oct. 1998), pp. 721–726. ISSN: 0742-4795. DOI: 10.1115/1.2818459. URL: https://doi.org/10.1115/1.2818459.

[100] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[101] solidIT. *DB-Engines Ranking - Trend of Time Series DBMS Popularity*. Accessed December 2022. 2022. URL: https://db-engines.com/en/ranking_trend/time+series+dbms.

[102] Stadtwerke München GmbH. *Der neue Energiestandort Süd – Wandel zur neuen Energiewelt*. Accessed Mai 2023. 2022. URL: https://www.swm.de/magazin/energie/energiestandort-sued.

[103] Daniel Stenberg. *Everything curl*. 2018. ISBN: 978-91-639-6501-2.

[104] Steve Klabnik, Carol Nichols, and the Rust Community. *The Rust Programming Language*. Accessed Mai 2023. 2023. URL: https://doc.rust-lang.org/book/.

[105] M. Stonebraker and U. Cetintemel. ""One size fits all": an idea whose time has come and gone". In: *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 2005. DOI: 10.1109/icde.2005.1.

[106] NERC Resources Subcommittee. *Balancing and Frequency Control. Reference Document*. Tech. rep. NERC, May 2021. URL: https://www.nerc.com/comm/OC/ReferenceDocumentsDL/Reference_Document_NERC_Balancing_and_Frequency_Control.pdf (visited on 04/20/2023).

[107] Mohammadreza Tahan et al. “Performance-based health monitoring, diagnostics and prognostics for condition-based maintenance of gas turbines: A review”. In: *Applied Energy* 198 (July 2017), pp. 122–144. DOI: 10.1016/j.apenergy.2017.04.048.

[108] Nisha Talagala, Remzi H. Arpaci-Dusseau, and D. Patterson. *Microbenchmark-based Extraction of Local and Global Disk Characteristics*. Tech. rep. UCB/CSD-99-1063. EECS Department, University of California, Berkeley, 1999. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1999/6275.html.

[109] Dingwen Tao et al. “Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2017. DOI: 10.1109/ipdps.2017.115. URL: https://doi.org/10.1109/ipdps.2017.115.

[110] Taylor-Davies, Raphael and Lamb, Andrew. *Querying Parquet with Millisecond Latency*. Accessed December 2022. 2022. URL: https://www.influxdata.com/blog/querying-parquet-millisecond-latency/.

[111] GCC Team. *GCC 11 Release Series*. Apr. 21, 2022. URL: https://gcc.gnu.org/gcc-11/ (visited on 04/20/2023).

[112] The Apache Software Foundation. *Apache Arrow*. Accessed April 2020. 2020. URL: https://arrow.apache.org/.

[113] The Apache Software Foundation. *Apache Arrow DataFusion*. Accessed April 2023. 2022. URL: https://arrow.apache.org/datafusion/.

[114] The Apache Software Foundation. *Apache Parquet Data Pages*. Accessed April 2023. 2022. URL: https://parquet.apache.org/docs/file-format/data-pages/.

[115] The Apache Software Foundation. *Apache Parquet Encodings*. Accessed Mai 2022. 2022. URL: https://parquet.apache.org/docs/file-format/data-pages/encodings/.

[116] The Apache Software Foundation. *Apache Parquet File Format*. Accessed April 2023. 2022. URL: https://parquet.apache.org/docs/file-format/.

[117] The Apache Software Foundation. *Arrow Flight SQL*. Accessed April 2023. 2023. URL: https://arrow.apache.org/docs/format/FlightSql.html.

[118] The Apache Software Foundation. *Parquet compression definitions*. Accessed February 2023. 2023. URL: https://github.com/apache/parquet-format/blob/apache-parquet-format-2.9.0/Compression.md.

[119] The Blosc Developers. *What Is Blosc?* Accessed December 2020. 2020. URL: https://blosc.org/pages/blosc-in-depth/.

[120] The fmtlib Authors. *{fmt}, fmtlib*. Accessed November 2022. 2022. URL: https://github.com/fmtlib/fmt.

[121] The HDF Group. *HDF5 User's Guide*. Accessed December 2020. 2020. URL: https://support.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide.pdf.

[122] The kernel development community. *ext4 Data Structures and Algorithms*. Last Accessed March 2023. URL: https://www.kernel.org/doc/html/latest/filesystems/ext4/.

[123] The OpenTSDB Authors. *OpenTSDB*. Accessed April 2020. 2020. URL: http://opentsdb.net/.

[124] The PostgreSQL Global Development Group. *Frontend/Backend Protocol*. Accessed October 2022. 2022. URL: https://www.postgresql.org/docs/15/protocol.html.

[125] Timescale Inc. *About compression*. Accessed December 2022. 2022. URL: https://docs.timescale.com/timescaledb/latest/how-to-guides/compression/about-compression/.

[126] Timescale Inc. *Timescale Docs*. Accessed Juli 2021. 2021. URL: https://docs.timescale.com/timescaledb/latest/.

[127] Tobias Oetiker. *About RRDtool*. Accessed December 2022. 2022. URL: https://oss.oetiker.ch/rrdtool/.

[128] van Riel, Rik and Peter W. Morreale. *Linux Kernel Documentation for /proc/sys /vm/**. Last Accessed February 2023. 2008. URL: https://docs.kernel.org/admin-guide/sysctl/vm.html.

[129] Deepak Vohra. "Apache Parquet". In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, pp. 325–335. ISBN: 978-1-4842-2199-0. DOI: 10.1007/978-1-4842-2199-0_8. URL: https://doi.org/10.1007/978-1-4842-2199-0_8.

[130] Allan J. Volponi. "Gas Turbine Engine Health Management: Past, Present, and Future Trends". In: *Journal of Engineering for Gas Turbines and Power* 136.5 (Jan. 2014). DOI: 10.1115/1.4026126.

[131] Jianguo Wang et al. "An Experimental Study of Bitmap Compression vs. Inverted List Compression". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, May 2017. DOI: 10.1145/3035918.3064007.

[132] Stefan K. Weber et al. "Data Acquisition System of the CLOUD Experiment at CERN". In: *IEEE Transactions on Instrumentation and Measurement* 70 (2021), pp. 1–13. DOI: 10.1109/tim.2020.3023210.

[133] Kexiang Wei et al. "A review on ice detection technology and ice elimination technology for wind turbine". In: *Wind Energy* 23.3 (Dec. 2019), pp. 433–457. DOI: 10.1002/we.2427.

[134] Sage A. Weil et al. “Ceph: A Scalable, High-Performance Distributed File System”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI ’06. Seattle, Washington: USENIX Association, 2006, pp. 307–320. ISBN: 1931971471.

[135] Henry Wong. *Discovering Hard Disk Physical Geometry through Microbenchmarking*. Sept. 2019. URL: http://blog.stuffedcow.net/2019/09/hard-disk-geometry-microbenchmarking/ (visited on 03/03/2023).

[136] Xiaodong Yu et al. “Ultrafast Error-Bounded Lossy Compression for Scientific Datasets”. In: *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’22. Minneapolis, MN, USA: Association for Computing Machinery, 2022, pp. 159–171. ISBN: 9781450391993. DOI: 10.1145/3502181.3531473. URL: https://doi.org/10.1145/3502181.3531473.

[137] Mingming Zhang et al. “CarStream: an industrial system of big data processing for internet-of-vehicles”. In: *Proc. VLDB Endow.* 10 (Aug. 2017), pp. 1766–1777. DOI: 10.14778/3137765.3137781.

[138] Kai Zhao et al. “Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation”. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 2021, pp. 1643–1654. DOI: 10.1109/ICDE51399.2021.00145.