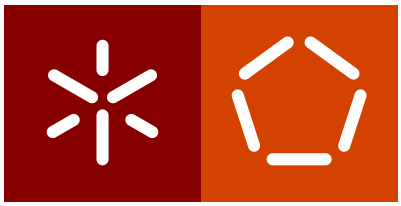
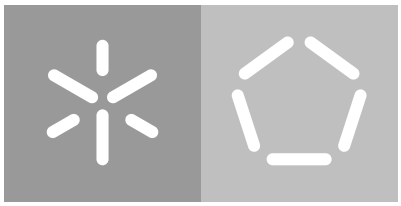


**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Bernardo Braga Bastos Mota

# Tagus: an IoT data ingestion pipeline for MonetDB

July 2021



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Bernardo Braga Bastos Mota

# Tagus: an IoT data ingestion pipeline for MonetDB

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
**Dr. Jose Orlando Roque Nascimento Pereira**
**Dr. Ying Zhang**

July 2021

## COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

# ACKNOWLEDGEMENTS

I want to express my gratitude to my supervisor, Doctor Ying Zhang, for giving me the opportunity to work with the MonetDB Solutions team and for making the transition from an academic to an enterprise environment that much easier. Without your unceasing help and patience, this thesis would not be possible.

Thank you to everyone the MonetDB Solutions personnel for welcoming me into the team and for always being ready to lend a hand.

I want to thank everyone in the University of Minho who made me grow in knowledge and as a person in the last 5 years, with a special thanks to my academic supervisor, Doctor José Orlando Pereira. I appreciate all the support and knowledge you have given me, and the opportunity to do an internship abroad.

Most of all, I want to show my gratitude to my family, who has always given me everything I need to be happy and to grow into what I am today. Thank you for all the patience and opportunities, I will never forget what you did and continue to do for me.

I also want to show my appreciation to all the friends I've made along the way. You have given me some of my best times in life, are always present for support and continue to shape me into someone better. My life wouldn't be the same without you, so thank you to every friend I have made in university and before.

Finally, I want to thank Mariana for all the love and support you have given me.

## STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

## ABSTRACT

In this project, we design and implement an IoT streaming data ingestion pipeline for MonetDB, using the distributed message queueing platform Apache Kafka. Our objective is to leverage MonetDB's analytical power for IoT data, expanding its ingestion capabilities to improve reliability and performance. The ingestion pipeline is put to the test with the real-world maritime tracking system AIS. We also evaluate MonetDB's current IoT processing engine and compare it to other state-of-the-art engines, to appraise its functionalities and identify possible future improvements.



**Keywords:** *Databases, Streaming Ingestion, Stream Processing, AIS, Maritime Informatics, Internet of Things, Continuous Queries, Apache Kafka, Columnar Database*

## RESUMO

Neste projecto, conceptualizamos e implementamos uma pipeline para ingestão de dados streaming para o sistema de base de dados MonetDB, utilizando o Apache Kafka, uma plataforma para message queueing distribuído. O nosso objectivo é utilizar o poder analítico do MonetDB para dados IoT, expandindo as suas capacidades e melhorando a confiabilidade e desempenho na ingestão de dados streaming. A pipeline é posta à prova com o sistema de tracking marítimo AIS, demonstrando a sua aplicabilidade no mundo real. As funcionalidades de processamento de dados IoT do MonetDB são avaliadas e comparadas com outras plataformas de última geração para identificar desenvolvimentos futuros.



**Palavras-Chave:** *Bases de Dados, Ingestão Streaming, Processamento de Streams, AIS, Informática Marítima, Internet das Coisas, Queries Contínuas, Apache Kafka, Bases de Dados Columnar*

# CONTENTS


1 INTRODUCTION 1
- 1.1 Background 1
- 1.2 IoT data processing challenges 2
- 1.3 Hypothesis: we can adapt a modern RDBMS for IoT data analysis 4
- 1.4 Research questions and contributions 6
  - 1.4.1 Q1: How to cope with the large volumes and high diversity of streaming data 6
  - 1.4.2 Q2: How to ingest streaming data into an RDBMS efficiently 7
- 1.5 Structure of the document 8

2 IOT ANALYTICS PIPELINE 9
- 2.1 Background: Streaming data processing in the IoT era 9
- 2.2 Example IoT application: the AIS global ship tracking system 13
  - 2.2.1 AIS use cases 13
  - 2.2.2 AIS message formats 14
  - 2.2.3 AIS data in this project 15
- 2.3 IoT platform requirements 16
- 2.4 The Tagus platform 18
  - 2.4.1 Replay 19
  - 2.4.2 Message collector 20
  - 2.4.3 Transformer 20
  - 2.4.4 Streaming Processor 20

3 DATA INGESTION 22
- 3.1 AIS replay 22
- 3.2 Message collector 24
- 3.3 Transformer 26
  - 3.3.1 Transforming data 27
  - 3.3.2 Loading data 28
  - 3.3.3 Data reliability 28
  - 3.3.4 Concurrent loading 29
- 3.4 MonetDB data loading 29

4 DATA PROCESSING 31
- 4.1 Continuous Query Engine in MonetDB 31
  - 4.1.1 MonetDB Implementation Architecture 31
  - 4.1.2 Continuous Query Engine 32

4.2 Stream processing AIS data 33
4.2.1 Example query: Query 11 35
5 EVALUATION 37
5.1 Tagus ingestion pipeline evaluation 37
5.2 Tagus platform functional evaluation 40
6 CONCLUSION 43
6.1 Summary 43
6.2 Future work 44
A APPENDIXES 47
A.1 AIS benchmark data schemas 47
A.2 AIS benchmark queries 47
A.2.1 Query 1: Find currently anchored ships 47
A.2.2 Query 2: Get the speed of ships 48
A.2.3 Query 3: Track the movements of a ship S 48
A.2.4 Query 4: Calculate number of distinct ships 48
A.2.5 Query 5: To each voyage message, add the current position of ship 48
A.2.6 Query 6: Find ships anchored at base station 49
A.2.7 Query 7: Find ships within a kilometer radius from a base station 49
A.2.8 Query 8: For every ship, find the closest neighbor ship 50
A.2.9 Query 9: Calculate average speed observed per ship over time 50
A.2.10 Query 10: Calculate number of ships under-way in hour windows 51
A.2.11 Query 11: Calculate average and maximum speed of moving ships in hour windows 51
A.3 Auxiliary queries 53
A.3.1 Distinct vessels 53
A.3.2 Distance calculation 53

# LIST OF FIGURES


| | | |
|---|---|---|
| Figure 1 | General IoT application stack (Source [11]) | 2 |
| Figure 2 | The MonetDB based Tagus architecture sketch | 4 |
| Figure 3 | AIS encoded single-line message example | 15 |
| Figure 4 | Overview of the Tagus platform | 18 |
| Figure 5 | The Tagus ingestion and processing pipeline | 19 |
| Figure 6 | AIS replay component control flow | 23 |
| Figure 7 | Message collector data flow | 25 |
| Figure 8 | Transformer component control flow | 26 |
| Figure 9 | Data representation in row stores and column stores | 32 |
| Figure 10 | Average ingestion latency across various replaying periods. Latency is measured in milliseconds. | 38 |
| Figure 11 | Average ingestion latency for a 12 hour replay period, with different batch sizes for database insertion. Latency is measured in milliseconds. | 39 |

# LIST OF TABLES


| | | |
|---|---|---|
| Table 1 | Differences between AIS sensors and our replay component | 22 |
| Table 2 | AIS benchmark queries and their tested features | 34 |
| Table 3 | Test environment hardware specification | 37 |
| Table 4 | Functional comparisons between MonetDB CQE and other modern stream processing systems | 41 |
| Table 5 | Data schemas and attributes used in this project, derived from AIS messages of type 1-5 | 47 |

# ACRONYMS

**A**

**AIS** Automatic Identification System.

**C**

**CQ** Continuous Query.

**CQE** Continuous Query Engine.

**D**

**DBMS** Database Management System.

**E**

**ELT** Extract, Load and Transform.

**ETL** Extract, Transform and Load.

**I**

**IOT** Internet of Things.

**J**

**JDBC** Java Database Connectivity.

**M**

**MAL** MonetDB Assembly Language.

**MEI** Mestrado em Engenharia Informática.

**R**

**RDBMS** Relational Database Management System.

**U**

**UM** Universidade do Minho.

# 1

# INTRODUCTION

## 1.1 BACKGROUND

Data is flowing everywhere around us nowadays: through phones, social media, credit cards and an increasing number of previously unconnected devices such as home appliances and sensor-equipped buildings. Our world is operating more and more in the present moment, and businesses need to have valuable insights over the data flying around in real-time to stay competitive [12]. At the consumer side, users want real-time updates about the world around them. Not only has there been an increase in the number and types of devices connected to the Internet, but also an increase in demand for systems that provide ever-faster analysis over the data as it happens.

The Internet of Things (IoT) can be seen as the integration of previously offline objects and sensors into business infrastructures to provide value-added services. These newly connected objects provide new data sources from which businesses can take valuable insights to improve service quality, reduce downtime, and increase sales. IoT infrastructures also allow new types of services for consumers to be developed: from smart homes to Internet-connected cars, emerging IoT applications promise to bring more automation and convenience into consumers' lives.

Figure 1 shows a general architecture for the modern IoT applications, divided into four layers [11]. At the bottom, the device layer encompasses the spectrum of data sources (sensors, RFID, cameras) that capture measurements and produce information to the platform. Above that, the data collection layer acts as a buffering system and aggregates data from multiple devices. This data is then analysed by the data processing layers where business logic is applied. Processing results are then stored and made available to IoT client applications by the application service layer, which acts the platform's interface.

The services and business insights this paradigm provides are made possible by analysing flows of data from a potentially large number of heterogeneous connected devices, which are subsequently filtered, aggregated and have business logic applied to them. This analysis needs to be continuous, as data is made available over time, and low latency, to meet the real-time constraints businesses require. The increasingly common need for unbounded

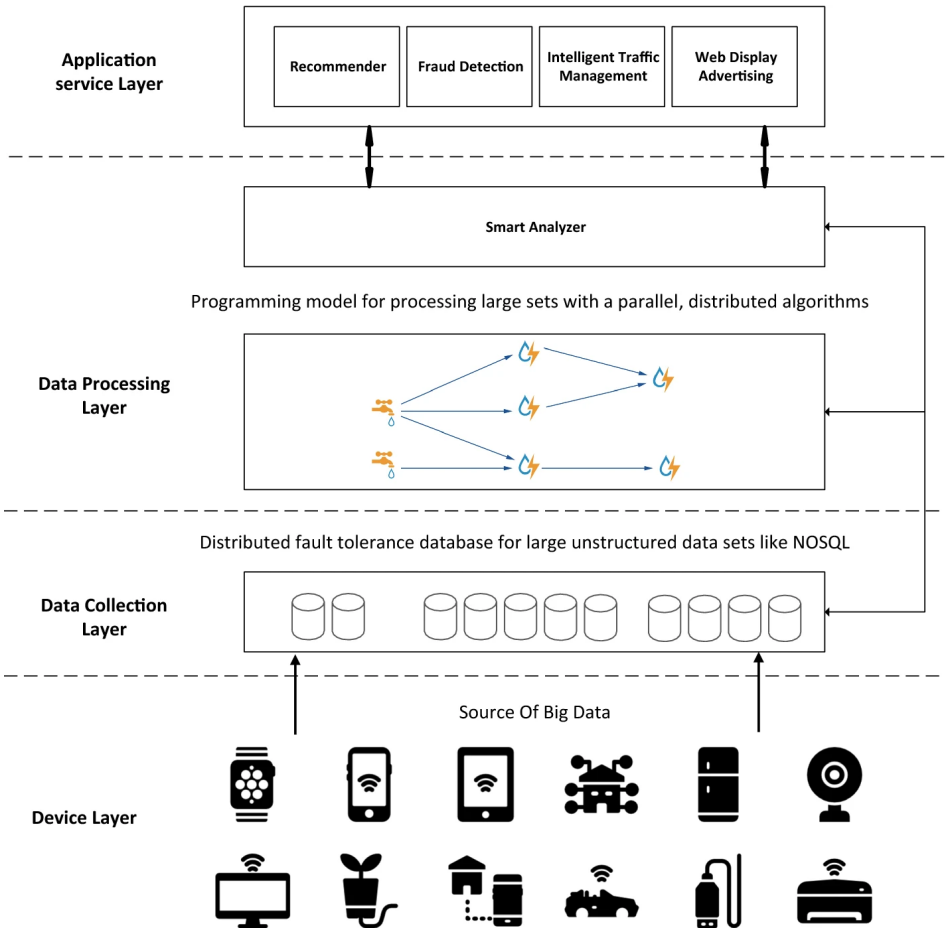



Figure 1: General IoT application stack (Source [11])

data processing is pushing the limits of traditional batch processing engines such as the relational database management systems (RDBMSs).

## 1.2 IoT data processing challenges

IoT data sources are characterised by many heterogeneous devices that continuously send data without a definite end. Sources differ in many aspects: data schemas, data rates and requirements they impose on applications. There are many different data formats in the IoT world, which can change over time to fit business needs. Devices often produce massive volumes of data, but data rates can vary between sources. The regularity of the rate can also differ, with some devices having highly variable data rates.

A data event does not have much meaning by itself, but we can get useful information by applying business logic over various events from multiple sources. IoT applications may require low-latency processing of events, as real-time data can have increased value. Data reliability can also be a concern, as data loss may be unacceptable in some applications.

While batch processing systems and DBMSs have historically dealt with growing volumes of data by horizontal scaling, they struggle to meet the strict temporal constraints, questioning the storage and batching of data before applying the relevant operations [10]. The one-time, user-issued query model of RDBMSs is being challenged by IoT applications that require automatic recomputing of queries and emission of results as new data comes in. The static and unified data formats of RDBMSs contrast with the varied and ever-changing schemas found in the IoT domain. Furthermore, query optimisation in RDBMSs is focused on big batch analytics (e.g. generating annual reports), not on real-time processing.

The IoT ecosystem's growth increasingly drives data analytics platforms to pursue real-time analytics over continuous data streams and tackle the challenges this paradigm poses. Most of today's database systems are not prepared for the data ingestion/storage and query processing requirements that streaming data imposes, leading to RDBMSs only being used as offline storage in IoT architectures. Nevertheless, given the wide adoption and understanding of RDBMSs, it would be beneficial to provide stream-oriented processing as a part of an RDBMS [4].

Stream processing systems have emerged as a new type of processing software to tackle the data processing challenges in IoT applications. Their goal is to provide low-latency, scalable, highly available and continuous analysis over massive amounts of heterogeneous data from multiple sources, meeting the processing requirements imposed by IoT and other real-time applications.

Over the years, various stream processing solutions with different complexity and use case focus have emerged, bringing new ideas on how to provide fast and reliable processing in this new data landscape. Some stream processing engines are designed and built from scratch to deal with the unbounded data scenario (e.g. Apache Flink[1]). In contrast, others extend existing batch-oriented systems to the streaming data paradigm (e.g. TimescaleDB[2]). Among those stream processing engines, some deal with reliably storing and ingesting data on their own, while others use distributed message queues to decouple the challenges of ingesting and processing streaming data.

Architecture-wise, we can divide the emerging streaming processing systems into platforms with a dataflow graph model, on which computation is done as a series of interdependent function calls, and platforms which consist of a collection of independent processes (i.e. queries and updates) against a shared backboard. Streaming processing systems that adapt an RDBMS usually belong to the second type of platforms.

1 Apache Flink: https://flink.apache.org/
2 TimescaleDB (built on top of PostgreSQL): https://timescale.com/

## 1.3 Hypothesis: we can adapt a modern RDBMS for IoT data analysis

IoT applications have a growing need for data analytics to gain insights from the data streamed by devices. Therefore, even though the RDBMSs do not handle the dynamicity and variability of streaming data well out-of-the-box, their support for a declarative query language (SQL) and highly optimised query processing engines form a strong basis for a streaming data analysis platform.

By adding the necessary features into a mature high-performance RDBMS, we can leverage their highly optimised operations for bounded data to deal with the unbounded case. This extension effort can save time and resources by maximally reusing existing features, leading to a faster adoption by developers familiar with SQL and RDBMSs. This extension approach also has the advantage of seamlessly integrating both streaming and historical data, leading to a more straightforward and less costly platform that can query both types of data.

So, we hypothesise that we can adapt a modern RDBMS for IoT data processing platform by extending it with several vital components to deal with the characteristics of the new application area. Figure 2 shows a sketch of one such IoT platform using MonetDB as the core data processing engine. The platform receives incoming streams from end devices, processes them according to the application logic and then produces new streams, which can be visualised by users.

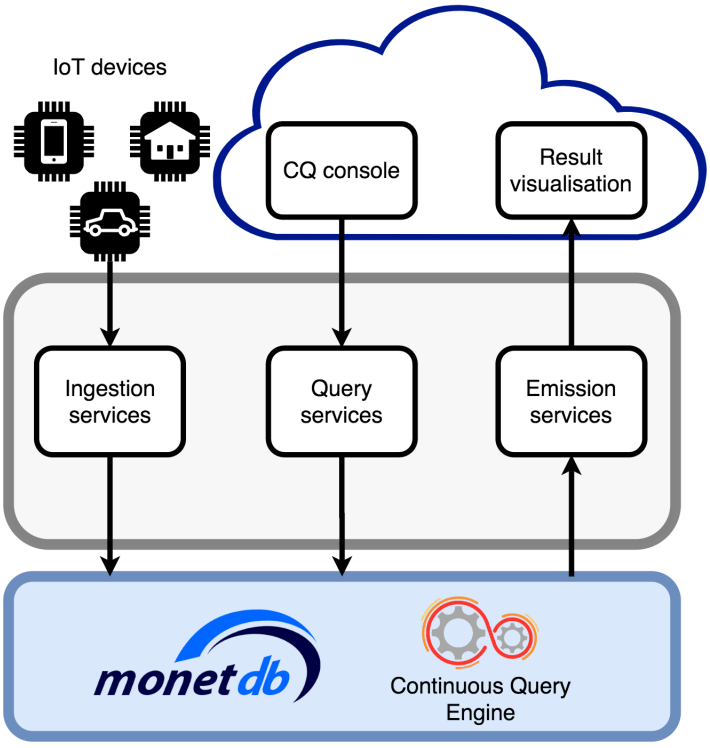



Figure 2: The MonetDB based Tagus architecture sketch

MonetDB is an open-source columnar RDBMS designed to provide high performance in processing complex analytical queries against large databases. It uses a storage model based on vertical fragmentation (column-store), a modern CPU-tuned query execution architecture and a modular software architecture [8]. Given the performance focus of existing data operators and the extensibility made possible by the modular architecture, MonetDB is a good fit for expansion into the unbounded data processing landscape.

To adapt MonetDB to the IoT data processing scenario, the following additional components are necessary:

- an ingestion pipeline for IoT data
  - The ingestion of IoT data presents various problems to MonetDB: the insertion of a large number of small data pieces in an analytical database, less optimised for constantly updating data; dealing with unstructured data and evolving schemas, which contrasts with the structured nature of relational data; or maintaining a significant number of connections to end-devices.
  - We will design and implement an ingestion pipeline to tackle these challenges, abstracting them from the stream processor.
- a continuous query engine,
  - To extend MonetDB to the stream processing field, we must worry about getting data into the system and correctly processing it, fulfilling this processing paradigm's requirements. There are challenges related to performance, like real-time processing of big data flows, and analytical challenges, like correctly handling time when analysing an unordered stream, that the processing engine must address.
  - While the ingestion components were developed for this project, we will reuse the continuous query engine (CQE) extension to MonetDB, designed and implemented in a previous project [7]. This will allow us to focus on developing the ingestion domain and evaluating the work previously done on the streaming engine. For this evaluation, we will assess the correctness and timeliness of the analytical results from a purpose-built benchmark, which uses a real-world IoT use case, and judge the functionalities the engine provides, comparing it to other modern streaming engines. Our evaluation will give us a better picture of how capable the DBMS is in handling this use case and the limitations of the current implementation. We will also identify features that need to be improved or added in future efforts.
- an emission pipeline for query results, and

  - Delivering the results of processing to the user is another aspect where stream processing differs from traditional databases. Traditional SQL queries operate synchronously, meaning that the user waits for the query results, and results are expected to be the final answer to the query. Continuous queries, on the other hand, operate asynchronously, meaning that the user registers the query but does not wait for the results. Query results are produced over time and have to be stored for posterior visualisation because the user is not waiting to capture the results. The results for continuous queries can be accessed through a dashboard like Grafana, which can plot results over time and deliver notifications when certain result conditions are verified.
  - There are three different strategies for storing CQ results in our proposed platform. We will focus on storing results in traditional SQL tables, which stores results persistently, and stream tables, for results that will be processed further. Storing CQ results in the message queue component is also possible, but will be left for future work.

- an administrative system to manage the IoT data producers and subscribers.
  - Users can register their IoT streaming devices and publish data to the platform, making it publicly available. Users can also register their interest in IoT streams, receiving updates and notifications from their subscribed sources.

## 1.4 Research questions and contributions

In this project, our primary goal is to design and develop an IoT data ingestion pipeline for our envisioned Tagus platform. Therefore, we need to research and answer the following two questions.

### 1.4.1 *Q1: How to cope with the large volumes and high diversity of streaming data*

Getting unbounded data into the processing component can be a challenge when faced with the heterogeneity and other issues of the IoT landscape: platforms must handle both high-volume streams and a large number of tiny streams, cope with varied and ever-changing data formats and provide data reliability in environments with different error rates and regularity of data production.

A large number of producers periodically send their information to the platform, and the rate at which events are produced fluctuates over time. Because of the pipeline's real-time requirements, ingestion must be low-latency and has to scale as the number of devices and the data rates increase. The possibility to pre-process data before the stream analytics engine

is also valuable, as it allows us to deal with heterogeneous data formats and errors in the stream.

MonetDB's capabilities for data insertion are not suited to IoT streams' requirements, as MonetDB is not optimised for small concurrent inserts from a large number of producers. Following a common strategy in stream processing, we propose using a distributed message queuing component as the entry point for data in the pipeline. Modern distributed message queues allow for low-latency concurrent ingestion of high-volumes of events, with the necessary reliability and scalability of producers and consumers. Using a message queue as the ingestion layer frees us from having to persist data in the processing layer, where it is more efficient to do in-memory processing, while not losing data in case of failures. It also allows us to batch and transform the raw event streams for more efficient insertion into MonetDB. The concurrency and rate of insertion into the DBMS can both be configured to scale ingestion horizontally and avoid overwhelming the database in periods of higher data rates. Finally, using a message queue allows us to effortlessly add more data sources (IoT or otherwise) and more components to consume the data, decoupling the devices from the processing engines.

We will study strategies for ingesting reliably and efficiently from IoT devices into the message queue and from the queue to the DBMS. We wish to evaluate the current ingestion capabilities, find out if using a message queue is beneficial, and identify what can be improved in the insertion of data into the DBMS to better suit it to the stream processing use case.

### 1.4.2 *Q2: How to ingest streaming data into an RDBMS efficiently*

Given the importance of real-time results and low-latency processing in streaming data, our ingestion strategy should not only provide reliability but also minimise the overhead in data insertion. We consider different strategies for transforming and loading the data into the processor, focusing on where to place the transform step in the pipeline.

We can follow an ELT approach, i.e. we Extract and Load the raw data into the RDBMS first, and only then Transform the necessary data inside the database. This approach has gained more interest in recent years. Advances in technology and the shift to cloud-based platforms has allowed developers to dynamically scale their computational needs and avoid impacting other processing work.[3] Using Kafka Connect[4] as a consumer, we can load raw data into the RDBMS with little effort. However, this approach is not always applicable. Some sensor data are produced in binary format, in which a message can be split into multiple

3 Article: Zero to Snowflake: ETL vs ELT: https://interworks.com/blog/chastie/2019/11/12/zero-to-snowflake-etl-or-elt/

4 Kafka Connect: https://docs.confluent.io/5.5.0/connect/index.html

transmissions. In such cases, no meaningful processing is possible without decoding the events beforehand. For these types of IoT data, one has to resort to the ETL approach.

Alternatively, one could use the traditional ETL approach, i.e. we Extract and Transform the data into a desirable format before Loading it into the RDBMS. Transforming raw streams before the database allows us to filter messages and correct or expand their information before insertion. With this strategy, we can optimise data loading and save database resources, which are only used for processing the transformed data. Furthermore, the ETL approach decouples data processing from its format, making integrating new downstream components easier. However, this approach requires more implementation effort than the ELT method, and the implementation is data format specific.

## 1.5 Structure of the document

Chapter 2 starts by introducing the challenges imposed by emerging IoT applications and overviews some of the most popular stream processing engines currently used to tackle IoT stream processing. Afterwards, we present our architecture for the Tagus ingestion pipeline, designed and implemented according to common requirements for IoT applications. We also introduce our example application, the AIS maritime tracking system. The ingestion pipeline's implementation is detailed in Chapter 3, where each component's behaviour and design choices are presented. Chapter 4 summarises the characteristics of the MonetDB continuous query engine while also presenting the continuous query benchmark we developed for our example application. Chapter 5 concerns the evaluation of the current state of the Tagus platform. We assess the performance of our ingestion pipeline implementation and the current state of the processing engine, while also comparing the functionalities the engine offers compared to other state-of-the-art streaming engines.

# 2

# IOT ANALYTICS PIPELINE

This chapter starts with an introduction of the IoT and stream processing paradigms, summarising the challenges that emerging IoT applications bring to traditional data processing and how state-of-the-art stream processing applications tackle these challenges.

We then outline the data ingestion requirements for the Tagus platform, designed to address the previously mentioned challenges. Afterwards, we present our design for the Tagus platform and give a brief overview of the flow of data through the pipeline. Finally, we introduce our example application for this project, the AIS global ship tracking system.

## 2.1 BACKGROUND: STREAMING DATA PROCESSING IN THE IOT ERA

The topic of streaming data processing has a broad scope and long history. In this section, we will limit our discussion to recent developments with special attention to the challenges posed by the emerging IoT applications.

The challenges and techniques involved in all aspects of streaming data ingestion and processing have been described and resulted in a systematic approach and model for stream processing architectures [9, 12]. In this context, the key challenges and advances in this area are the following:

- Networking and scaling challenges posed by IoT lead to novel networking infrastructures [6]
- The characteristics of streaming IoT data fostered the evolution of streaming processors and the identification of new challenges [10]
- The need to timeliness, reliability, and order in stream processing paved the way for a new generation of streaming engines [1, 2, 13]

While the idea of devices with embedded sensors that can automatically share sensed information and change their state based on received data was already conceptualised in 1999, it has only gained considerable traction in the last decade, benefiting from the pervasiveness of computers and the reduction of their size [5]. Nowadays, we use the term

Internet of Things to refer to a network of interconnected objects with computing capabilities that can transfer data between themselves or other systems without human interaction.

The IoT revolution's primary motivation is the ability to integrate an ever-growing number of previously unconnected devices into business processes. These devices allow for more efficient and automated workflows and provide valuable data for more informed decisions. Beyond improving business processes, the IoT promises to bring new services to make customers' lives more convenient and connected, from their home to their transportation mode.

Despite promising results, the implementation of a comprehensive IoT application presents many challenges.

- **Networking challenges**: Due to the hardware limitations most devices and connections on the edge of the network have, networking can become an issue when reporting data to the IoT platform's data collection tier. It would be cost-prohibitive to provision all devices with enough computing power to support the infrastructure of the IP protocol, given that these capabilities are not necessary for the device's prime function [6].

- **Data challenges**: IoT data fits into the big data model, characterised by its volume, velocity, variety and variability, while also being defined by its low result latency requirements. Many sensors and devices produce massive volumes of information in real-time, leading to high-velocity data streams. There are many different implementations of IoT devices, which leads to a variety of data formats and schemas. The speed at which devices produce data varies highly due to variable connectivity issues. Devices may buffer data while offline and suddenly dump it all when reconnected, leading to bursty data.

- **Processing challenges**: The flows of data must also be analysed in real-time to maximise the results' business value, which may diminish over time. Users want a useful and current picture of their environment, and analysis results must change as soon as new data becomes available.

- **Data storage challenges**: Due to its volume and indefinite end, storing all information from an IoT stream is neither practical nor desired. Most IoT applications want to access updating analysis results and more compact views over the data, like aggregations over time. However, reliably storing the most recent event data may be necessary for fault-tolerance concerns. If there is no persistent event storage, we might lose data after a processing engine crash or if the production rate of events is higher than their processing rate, which is not unusual during peak times. By using a data collection layer, we can decouple the production of events from their processing.

In recent years, a new generation[5] of streaming data processing systems has emerged which have the IoT requirement embedded into their initial design. These systems' architectures vary from simple DBMS extensions that continuously update materialised views to full-on event-by-event stream processing engines. The choice for a streaming engine is determined by the use case with its particular requirements. Below we highlight several new systems that have gained much popularity in the IoT community: two streaming databases (TimescaleDB and InfluxDB) and two dataflow-graph streaming processors (Apache Flink and ksqlDB).

**TimescaleDB**[6] is a time-series database built on top of PostgreSQL, adding time-oriented features and improving ingestion and query performance. TimescaleDB adds support for continuous aggregates, stream windows, time-related analytics operators and retention policies. Stream events are partitioned according to time and other key attributes and stored in disjoint SQL tables (chunks), improving ingestion and query performance.[7]

TimescaleDB follows the batch database model of streaming data processing, maintaining the underlying RDBMS' execution model. It provides continuous aggregates by extending the materialised view abstraction, automatically calculating and materialising updated results whenever new data is ingested. Real-time aggregates combine the pre-calculated continuous aggregate with the most recent data to give an up-to-date answer.[8] Furthermore, TimescaleDB supports event-time windows, stream-to-table joins and data compression.

**The TICK stack**[9] is a time-series processing platform with a modular architecture composed of a collection agent (Telegraf), a purpose-built time-series database (InfluxDB), an interface for visualising results and submitting queries (Chronograf) and a real-time stream processing engine (Kapacitor).

**InfluxDB** is a high-performance data store for time-series data, optimised for large data sets where a single event is not as important as aggregate results. It features data downsampling and compression capabilities and privileges scalability and performance over strong consistency and atomicity.[10] Data can be inserted through an HTTP POST request to the API or natively ingested through the Telegraf collection agent. While InfluxDB has some querying capabilities, **Kapacitor**[11] offers real-time streaming data processing, allowing for pre-processing data before ingestion and analysing and acting on events present in InfluxDB. Tasks run user-defined logic over a stream periodically, enabling continuous data

5 As opposed to the generation of the Data Streaming Management Systems, e.g. CQL ([3]) and STREAM ([4]), which have mainly focused on developing continuous query languages and engines.
6 TimescaleDB: https://timescale.com/
7 Documentation: TimescaleDB vs Postgres: https://docs.timescale.com/latest/introduction/timescaledb-vs-postgres
8 Documentation: Continuous Aggregates: https://docs.timescale.com/latest/using-timescaledb/continuous-aggregates
9 InfluxDB/TICK stack: https://influxdata.com/time-series-platform
10 Documentation: InfluxDB design insights and tradeoffs: https://docs.influxdata.com/influxdb/v1.8/concepts/insights_tradeoffs/
11 Kapacitor: https://www.influxdata.com/time-series-platform/kapacitor/

transformations (i.e. continuous queries) and real-time alerts. When an alert is triggered, event handlers are used to define actions to respond to the alert. There are two types of tasks: batch tasks, i.e. periodically run on slices of the stream pulled from InfluxDB, and stream tasks, i.e. events are read and processed event-by-event as they are ingested into InfluxDB.[12]

**Apache Flink** [13] takes a different approach to stream processing, using dataflow graphs to model its streaming data operations. Applications are composed of a graph of user-defined data operators that transform data from one or more sources into data sinks. Flink offers scalable stateful computations over bounded and unbounded data, featuring powerful state management optimised for in-memory processing. Exactly-once state consistency is guaranteed by asynchronously and incrementally checkpointing the local in-memory state to a durable state back-end. Users can write programs with the Java streams style DataStream API for higher expressiveness, or with the Table/SQL API for more conciseness and ease-of-use.[14]

Unlike the two previous systems, Flink processes data event-by-event, following the pure streaming model. It supports event-time processing, uses watermarks to track event-time progress and allows the user to define an allowed lateness period for tuples that arrive after the watermark. We can define tumbling, sliding and session windows, which can be either keyed, i.e. partitioned over an attribute and eligible for parallelisation, or non-keyed, i.e. not parallelised. The processing of a window can be triggered event-by-event, by event-time watermarks or by processing-time, i.e. wall clock.[15] Data sources and sinks can be easily integrated through connectors, which handle both the logic of reading/writing data from the external system and data mapping.

**ksqlDB**[16] is a streaming database built on top of Kafka Streams, a robust stream processing framework which uses the dataflow-graph model. ksqlDB aims to abstract away the complex programming needed for real-time operations on streams of data. It uses Apache Kafka for storing events and processed data, leveraging Kafka's high-performance for low-latency storage and making data source and sink integration easier through Kafka's extensive connector libraries.

ksqlDB uses the concept of Stream/Table duality as a basis for its stream processing model. The main idea is to represent the result of an operator, captured at a point in time by the relational notion of a table, as a stream of successive updates. A stream is immutable and represents the historical sequence of events, while a table models change over time, representing what is true as of now. This duality between streams and tables allows for better

12 Kapacitor Documentation: https://docs.influxdata.com/kapacitor/v1.5/guides/continuous_queries/
13 Apache Flink: https://flink.apache.org/
14 Documentation: Flink Applications and APIs: https://flink.apache.org/flink-applications.html
15 Documentation: Flink Timely Processing: https://ci.apache.org/projects/flink/flink-docs-release-1.12/concepts/timely-stream-processing.html
16 ksqlDB: https://ksqldb.io/

reasoning over time inconsistencies and a more unified model. A table represents a view over the data for a point in time that is continuously updated as more streaming updates arrive, dispensing with notions of operator completeness towards constantly evolving results. This approach also simplifies fault-tolerance, as the changelog stream for a table can be replayed to return to the last stable state [13]. Furthermore, ksqlDB provides materialised views, i.e. tables which evaluate queries on the changes only.[17]

ksqlDB supports tumbling, sliding and session windows. Windowed aggregations are updated continuously, emitting freshly computed results as new data arrives, and do not depend on any notion of completeness (e.g. watermarks). A window retention parameter can be used to balance the tradeoff between result completeness and storage cost, avoiding that aggregation result tables grow indefinitely [13]. ksqlDB has two methods for querying data: pull queries, which retrieve information at a point in time, and push queries, which act as subscriptions to the query output and deliver results continuously as they are calculated.[18] This analysis of the challenges of IoT applications and the techniques used by state-of-the-art tools to tackle them will allow us to define the requirements and goals for our platform, to expand the capabilities of MonetDB to the IoT paradigm.

## 2.2 Example IoT application: the AIS global ship tracking system

The automatic identification system (AIS) is an example of an application area for IoT systems for maritime tracking. In this project, it serves as our example application, using it both as a dataset and as a use case example.

### 2.2.1 *AIS use cases*

AIS has been used as an automatic tracking system in the maritime world since the 1990s, allowing the exchange of navigational information between AIS-equipped terminals. AIS base stations can use their AIS receivers to track vessels equipped with AIS transceivers. AIS was designed to allow ships to view marine traffic in their area, assist the decision-making in collision avoidance cases, and allow maritime authorities to monitor vessel movements and identify specific ships, improving security and control over sea traffic. It is also used for aids to navigation (AtoN), search-and-rescue (SAR) operations, fleet and cargo tracking and accident investigation.

Over the years, as the reach of AIS went global with satellite transmission and as AIS data became publicly available on the Internet, applications started to integrate AIS processing capabilities to provide marine traffic visualisation for their users. Global, real-time positional

17 Documentation: ksqlDB Materialized Views: https://docs.ksqldb.io/en/latest/concepts/materialized-views/
18 Documentation: ksqlDB Queries Overview: https://docs.ksqldb.io/en/latest/concepts/queries/

data collected by coastal stations can be viewed on many online devices, allowing commercial and recreational users to easily visualise marine traffic and track ships.

**MarineTraffic Live Map**[19] features a live map that displays public tracking data from ships around the world. The data is collected by AIS-receiving stations and sent to Marine Traffic's central database, where it is decoded and processed. The information in the database can then be visualised through a dashboard.

**The Marine Exchange of Alaska (MXAK)**[20] operates the only terrestrial AIS network in Alaska. It uses its network to transmit weather information as AIS application-specific messages (ASM).[21] The project aims to enhance maritime safety in Alaska by providing real-time environmental information to vessels over AIS, one of the most reliable means of communication in coastal waters due to its broad adoption.

**AIS transceivers** automatically broadcast dynamic and static information, such as the position and ship name, at regular intervals. Dynamic data is collected through its GPS (Global Positioning System) receiver, which collects the vessel's position and movement details, and a gyrocompass. In the IoT paradigm, these are the sensors through which the IoT device continually collects its data. Dynamic data is sent every 2 to 10 seconds, depending on the vessel's speed, and every 3 minutes while the vessel is anchored.[22] Broadcasted information can then be received by other vessels, base stations or satellites, and with the use of software, can be processed and visualised.

### 2.2.2 *AIS message formats*

There are several types of AIS messages with different purposes. The most common ones are the position report messages (types 1, 2 and 3), which are sent by ships to report information related to navigation: coordinate positions (longitude and latitude), speed, heading and the ship's navigational status (anchored, underway). Other common message types include base station reports (type 4), which are sent by coastal bases to indicate their presence, and voyage messages (type 5), which report static data about the ship, such as ship name, and data about the current trip, such as the intended destination. Application-specific messages (ASM) allow for the exchange of environmental data such as weather, waves and water level information.

AIS messages are encoded with a two-layer protocol, as shown in Figure 3. The outer layer encoding[23] contains information about the protocol and message fragmentation, while the

19 MarineTraffic Live Map: https://www.marinetraffic.com
20 MXAK: https://www.mxak.org/
21 See section "AIS message formats" for more information about the ASM.
22 AIS reporting interval: https://arundaleais.github.io/docs/ais/ais_reporting_rates.html
23 Outer layer encoding (NMEA 0183): https://en.wikipedia.org/wiki/NMEA_0183

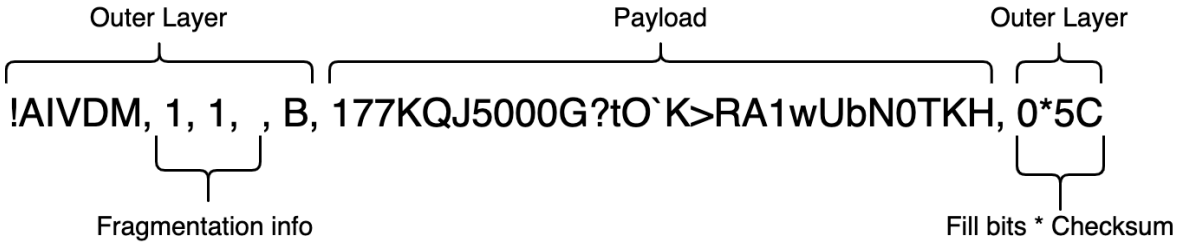



Figure 3: AIS encoded single-line message example

payload contains the actual AIS message data, encoded as an ASCII 6-bit character string[24]. Encoded payloads do not contain any parseable information and need to be decoded before any data can be extracted from the message.

Messages larger than the 82 character limit need to be split into multiple lines and need to be put together before decoding. The fragmentation info section of the outer layer contains: the fragment number (from 1 to fragment count), the total number of fragments and a multi-message ID, used to identify message fragments from the same message. The message example shown in Figure 3 is a single-line message, with fragment number and fragment count of 1 and no multi-message ID. At the end of the message, the outer layer also includes the number of fill bits required to pad the data payload to a 6-bit boundary and a data-integrity checksum, calculated as the XOR of all of the bytes in the message.

The AIS data processing use case fits the IoT paradigm: many ships and bases periodically send collected data to some receiving stations, which can then analyse it to get useful real-time results. AIS was not initially designed for real-time ordered use, so there is no full timestamp present in standard AIS messages. To take full advantage of the possibilities of IoT analytics over AIS data, receiving stations can add a timestamp to each message as it is received.

### 2.2.3 *AIS data in this project*

From the 27 different AIS message types, we chose three types for our case study: The position report messages, which make up the majority of the data set and are the best choice for stream processing due to their periodicity Base station report messages, which are more suited to be aggregated into a relational table The voyage messages Position report and voyage messages will be considered streaming data, while base station messages will be considered static data.

Position report messages are the main focus of our study, but the other message types can be used in conjunction with ship position data to answer more complicated queries.

24 Inner layer encoding (ASCII encoded bit-vector): https://gpsd.gitlab.io/gpsd/AIVDM.html#_aivdmaivdo_payload_armoring

For example, we can query ship position data to find out how many vessels are currently anchored. However, using base station data, we can also determine at which bases they are anchored.

Instead of using live AIS data, this project used a data set containing captured AIS messages. The data set features messages gathered by our AIS transceiver, which tracks ships around Amsterdam, during January 2019. The receiving station added a UNIX epoch timestamp to each captured message, representing event time.

There are 36 million lines of AIS encoded messages in the log files, translating to around 34 million decoded messages. This difference is due to some messages being multi-line, and some errors present in the received encoded messages. Most of the decoded messages are types 1, 2 and 3 (vessel position report messages), accounting for more than 32 million messages. There are 200,000 base station report messages and 650,000 voyage messages. All other types of messages, which we will not consider for this case study, account for less than 1% of the data set's messages. The captured data features 17,718 different vessels and 309 different bases.

Given our choice to replay a data set instead of using live streaming data, our pipeline's device stage has some differences from how the AIS messages were originally generated. Without decoding the messages, we cannot know which multi-line messages belong to each other and which messages were from the same sensor.

Messages are read from files and sent to the collector in the encoded format, similarly to how a real-world AIS device would transmit its data. The compressed encoded format benefits ingestion performance, as less data is sent from end-devices to the message collector. However, messages in the encoded format are not ready to be processed, as all the useful information about events is not accessible. This means that messages need to be decoded prior to analysis.

For the AIS use case, it would be exceptionally difficult to decode the messages inside the streaming processor (i.e. the ELT approach). Hence, we chose to follow the ETL method, i.e. using the Transformer component to decode and prepare the data for analysis. We selected several attributes for each type of messages to make up the general schema for that particular type. The schemas for the messages we use in this project, along with a description of their attributes, can be viewed in Appendix A.1.

## 2.3 IoT platform requirements

The main requirement for analysing IoT data is real-time processing, which impacts data access, result emission, scalability and even the processing model, as the delays inherent in batching data for traditional processing may be unacceptable. Based on the needs of IoT stream data processing, we outlined several requirements for the Tagus platform.

The following requirements were taken into consideration when designing the pipeline:

1. The platform must ingest events from a large number of IoT devices with low latency, to adhere to the real-time constraints. Furthermore, it must be reliable and highly available.
2. We expect a large amount of heterogeneous IoT devices and event formats, hence, parsing the relevant information into a device-agnostic format is necessary before loading it into the processor.
3. The platform should minimise the latency between data arrival and result emission to levels required by its upper layer applications.
4. Data processing should not block the collection of new data. Furthermore, the processor should be able to analyse data at its own rate without data loss.
5. Frequent emission of updated results and continuous analysis of new information is necessary for giving end applications an evolving view of the data.
6. If an application requires the platform to process streams in order, events should include some timestamp information to serve as the ordering attribute. If there is no information about when an event happened, the system should order according to when it ingested the events.
7. The ingestion and processing of events must scale to increasing workloads and new data sources and sinks. Distributed execution of partitioned streams and concurrent production and consumption of events may be necessary for scaling event processing.
8. The platform must provide at-least-once processing guarantees and aims to provide deduplication methods for exactly-once processing.
9. In most use cases, storing individual IoT events after being processed is not necessary or valuable. The platform should have configurable limits for data retention, deleting old data to save storage.

While all requirements can be achieved with our platform's design, not all of them are covered by the current implementation. Requirement 1 is only partially fulfilled because our current implementation does not allow for testing many event producers. Requirement 2 is also partially fulfilled, as the pipeline integrates schema transformation, but only for a small subset of IoT event formats. Requirement 7 is conceptually possible but could not be implemented due to dataset limitations and time constraints. Our prototype pipeline covers all other requirements.

The platform requirements lead to the decoupling of the platform's collection and processing components, by having a message queue component to deal with the challenges of

ingesting data into the system and a processing component to deal with the challenges of in-flight data analysis. Distributed message queues are a good choice for the data collection component, providing efficient, durable storage that meets all the requirements listed above.

Using a message queue with pull-based consumption as a buffer between devices and the processor, we allow the processor to consume collected data at its rate and rewind the consumption whenever needed [11]. This makes the use of in-memory processing possible, as data reliability is not an issue for the processor. This approach also allows data to be transformed before being loaded for analysis.

We can see the architecture for a streaming analytics platform as a pipeline: data flows from IoT devices into the system, through all defined transformations, before arriving at the end-users in the form of query results. We can divide this pipeline into four main stages: the device stage, the ingestion stage, the processing stage and the result emission stage. Our work mostly focuses on the ingestion and processing stages, with limited attention given to the emission of results and edge devices.

## 2.4 THE TAGUS PLATFORM

Figure 4 shows a high-level view of our proposed Tagus platform, featuring the three main components, i.e. message collector, transformer and streaming processor, alongside the device and emission stages. The message collector and the transformer make up the ingestion stage, while the streaming processor represents the processing stage. IoT edge devices are external to the Tagus platform but typically would run a lightweight producer process to send data to the platform.

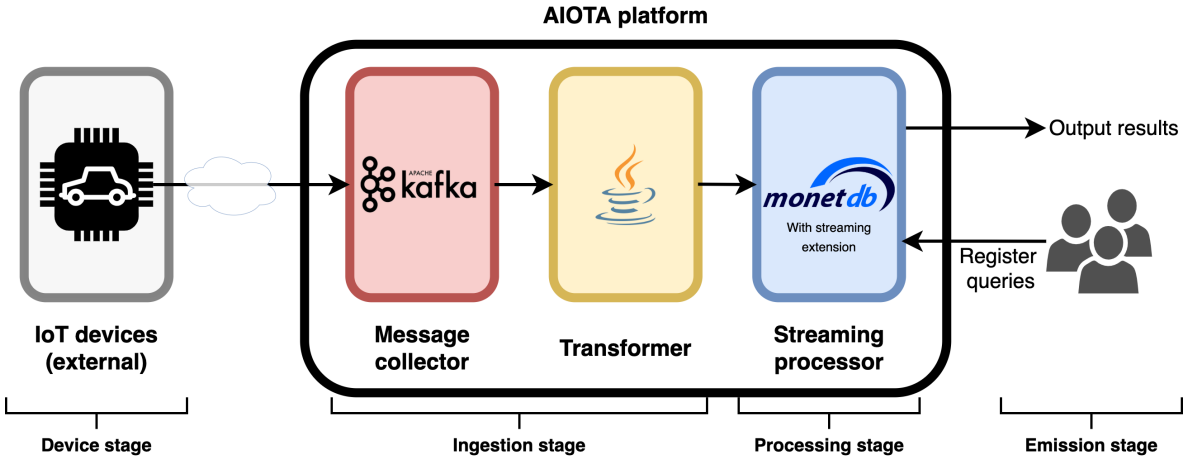



Figure 4: Overview of the Tagus platform

We chose the distributed message queuing system Apache Kafka[25] as the core ingestion component, as it supports several features which are essential for our IoT platform. Compared to other message queueing solutions, e.g. RabbitMQ[26] and Apache ActiveMQ[27], Kafka is better suited for the stream processing use case. It provides exactly-once semantics, maintains strict ordering over streams, has better persistence and scaling capabilities than other message queues and has pull-based consumption. Kafka's persistence capabilities and pull-based model allow for both replaying streams in case of a crash and decoupling the consumption rates from data production.

Figure 5 shows a detailed view of the ingestion and processing pipeline. In our pipeline implementation, the Replay component simulates IoT end-devices that produce data to the platform. The architecture is designed with reliability and scalability in mind, guaranteeing that no data is lost and that the system can handle growing data volumes. It also privileges modularity and decoupling of responsibilities, complementing how streaming data flows through the pipeline and allowing components to scale independently.

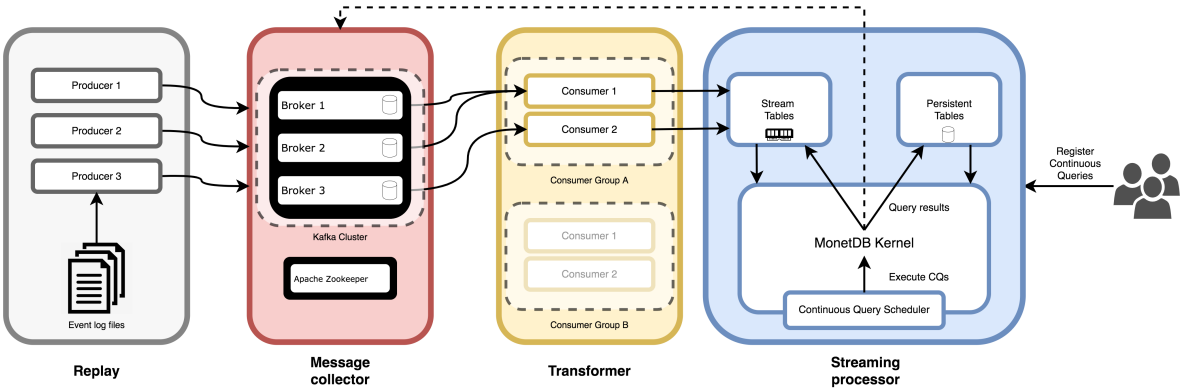



Figure 5: The Tagus ingestion and processing pipeline

Data flows through the pipeline as follows:

### 2.4.1 *Replay*

The **replay** component reads previously captured messages from event log files. It sends them to the platform in the original order in which the messages have been produced, therefore replaying the captured stream. This component has the role of simulating the production of events by end-devices and has a configurable production rate for testing and stressing the pipeline.

25 Apache Kafka: https://kafka.apache.org/
26 RabbitMQ: https://rabbitmq.com/
27 Apache ActiveMQ: http://activemq.apache.org/

### 2.4.2 *Message collector*

The **message collector** receives stream events over time, durably storing them and allowing their consumption downstream. It acts as the source of truth in case of failures, allowing consumers to replay events for recovery.

A Kafka broker is used as the message collector component in the current implementation, serving as the buffer for incoming streams and reliability. Data sources can use Kafka's producer library to send data to the platform, and the processing components downstream can use the consumer library to fetch ingested data. Not only the brokers but also consumers and producers can scale-out and are easily configurable, giving a fair amount of control to optimise this layer, aiming for low-latency. Furthermore, IoT devices that use low requirements network protocol MQTT can be easily integrated. Fault-tolerance, failover and horizontal scaling capabilities are also available if the standalone broker is upgraded to a Kafka cluster.

### 2.4.3 *Transformer*

The **transformer** consumes raw event streams, pre-processes them and loads the enriched stream into the streaming processor. This component filters corrupt and unwanted messages and transforms the various heterogeneous data schemas into a general structure, only containing useful information. After extracting the useful information, the transformer loads the data into the processor in the format it expects.

There are two main approaches for loading data into the stream processor: we could either load the encoded data and transform it inside the streaming database (ELT) or transform it into a parse-able format before loading it (ETL). As referenced in section 1.4.2, both approaches have their pros and cons.

Due to the example application chosen for our prototype pipeline, our choice is limited to ETL. Raw data needs to be decoded before being parseable, and the algorithm for decoding is non-trivial to implement in a DBMS. Despite this, the transformer currently supports both approaches.

### 2.4.4 *Streaming Processor*

The **streaming processor** consumes the stream and keeps the tuples in in-memory stream tables. Users can analyse data flows by registering continuous queries and their triggering conditions and data sources. The continuous query scheduler continually verifies if any CQ is ready to be executed, as data is made available over time. When the scheduler triggers a query, the kernel calculates results using relevant data from stream tables. These can be

stored persistently in the message buffer or the streaming engine, or be used for further processing.

For the processing component, we use the columnar database MonetDB and its stream processing extension. The streaming extension offers us low-latency storage of event data in stream tables, a continuous query scheduler (CQE), windowed execution of queries and an extended SQL catalogue to manage continuous queries and stream tables. Next to these stream-specific features, the streaming processor leverages MonetDB's analytical power for high-performance query execution.

---

All the components can be run on the same machine or in different nodes, as the scalability concerns start to arise with higher data rates. Both the Kafka brokers and the MonetDB servers can scale out to clusters, and the transformer component can be run concurrently to increase performance.

The replay and transformer components were developed using Java and the appropriate Kafka library, while the Kafka message collector and MonetDB streaming processor are open-source software.

While the Tagus pipeline is designed for general use, our implementation also aimed to fully accommodate the processing of data from an example application. We chose the AIS ship tracking system as our example application, to test and demonstrate the capabilities of our pipeline implementation.

# 3

# DATA INGESTION

Getting data from edge devices into the streaming platform with the lowest latency possible is a crucial step for a real-time streaming pipeline. Devices may produce data at much higher rates than the processing layer can consume, leading to data loss if there is no buffering of events before analysis [12]. Using a distributed message queue such as Kafka, we can persistently store streams as they are produced, allowing consumers to read at their own rates and recover from failures. Based on this decoupling, we have designed the data ingestion portion of our data pipeline (see Figure 5). In this chapter, we elaborate the decisions and observations made during the development of the ingestion components.

## 3.1 AIS REPLAY

This component acts as the IoT device layer for our case study, simulating the continuous sending of data over time. This component uses Kafka's producer Java API to write data to the message collector. An exact replay of the live AIS messages is out of the scope of this project. However, in our simulation, we have captured the relevant properties to the best of our knowledge. Table 1 shows where the replay component implementation diverges from live AIS sensors.

| **AIS sensor** | **AIS replay** |
|---|---|
| Real-time data | Logged data |
| Each device sends its own data | Produces data from many devices |
| Multiple connections | One connection |
| Low-fidelity networks with delays | No networking |
| Unpredictable workloads | Configurable workloads |
| Data is sent over a long period of time | Data spanning long time interval is sent much faster |

Table 1: Differences between AIS sensors and our replay component

The dataset contains data sent by various devices, meaning that one replay component represents multiple IoT sources. This approach does not consider the overhead of many

sources sending their information over the network, which can lead to issues if the collector cannot scale producer connections well enough. The rate at which events are produced is configurable, which allows us to simulate different workloads.

We considered dividing the dataset between various replay threads and waiting between sending messages to simulate a device layer better. This approach would allow us to have concurrent producers without sending repeated data and to replay messages in the time scale they were initially sent. However, the data rate was too slow to stress the message collector. We decided that each replay thread would send the entirety of the data set, allowing us to further scale the data rate.

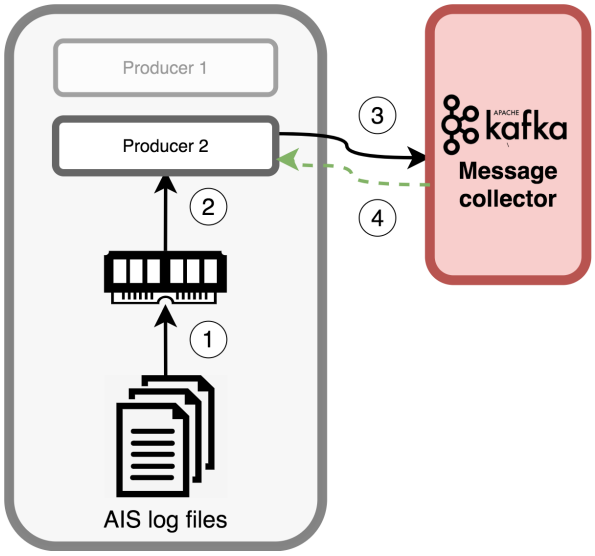



Figure 6: AIS replay component control flow

Figure 6 shows the behaviour of the AIS replay component:

1. The component begins by reading all AIS log files to memory to avoid the I/O overhead while sending data to the message collector. Messages are stored in the order of event timestamps to keep time consistency while replaying events.

2. A Kafka producer is then configured to send all recorded messages, acting as multiple vessels.

3. The producer sends messages to the message collector one-by-one in time order.

4. The collector may send an acknowledgement to the producer, synchronously or asynchronously, depending on the configuration.

All messages are sent to the same Kafka topic, which represents the stream. Records do not feature a key, as no relevant identifying information is accessible when messages are in their encoded form. Encoded messages are serialised as a string in the record payload.

Given our use of the Kafka producer library to send messages to the collector, most possible optimisations for the replay component are on Kafka's side. Depending on our needs for reliability, performance and disk use, we can configure the Kafka producer to fit our requirements. Most non-essential configurations were not explored in this project, as our implementation pipeline only used one broker, and our focus was not to fine-tune a Kafka ecosystem[28].

However, one configuration we do use is the method for sending data to the collector. Kafka producers can send data synchronously or asynchronously. With synchronous sending, the producer actively waits for an acknowledgement from the broker before sending another message. With the asynchronous method, the producer does not wait for an acknowledgement before resuming. The producer receives the acknowledgement through a callback function. Although the synchronous method simplifies the control flow and allows production latency to be measured, the performance impact that it has is significant. Sending all data in the AIS log files takes less than one minute with asynchronous sending, while the synchronous method took more than an hour.

Kafka can provide exactly-once and ordering guarantees with the asynchronous method. It automatically retries to send unacknowledged messages for a number of times, before raising an exception through the callback function. Messages that were re-sent after a recoverable error are guaranteed to be in order if the enable.idempotence producer configuration is true. We decided to use asynchronous sending for this project, given the synchronous method's enormous performance overhead.

## 3.2 Message collector

The message collector is the broker layer of the ingestion pipeline. Producers and consumers connect to this layer to write or read data to or from topics, in which streams are temporarily stored. Figure 7 shows the Kafka ecosystem in our pipeline, with the message collector broker in the middle.

Using Kafka as our primary message storage instead of a database has several advantages. Kafka's connection overhead is much smaller than an RDBMS, which would struggle to maintain a large number of open connections for IoT data sources. Kafka has better scalability than a database, providing more parallelisation and load balancing features than traditional RDBMS. Kafka supports easy integration of many data sources and sinks, i.e. MQTT data source and JDBC data sink, giving more flexibility for future expansion.

The message collector consists of a Kafka broker (or cluster of brokers) and an Apache Zookeeper instance used to store metadata and coordinate broker clusters. Both brokers

28 Producer configurations that could be explored for future optimizations include: number of broker acknowledgments (durability vs latency trade-off), batch size (latency vs throughput trade-off) and compression.

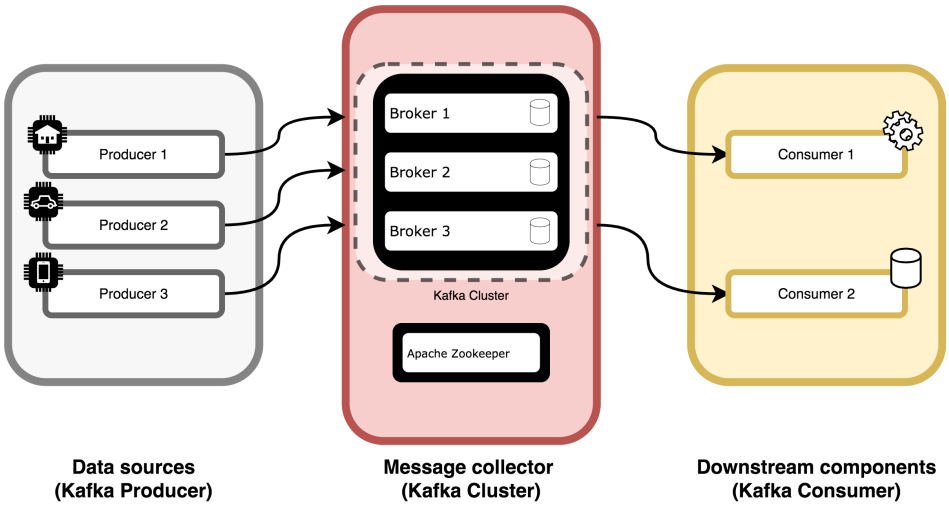



Figure 7: Message collector data flow

and topics are configurable, allowing us to adapt the server's behaviour to our needs. Our pipeline implementation only features one broker, as the objective was only to get a working prototype for an IoT pipeline. A cluster of brokers should be used in future implementations, guaranteeing high availability and better load balancing.

When creating a stream's Kafka topic, we define the number of partitions and the replication factor. Partitions allow for more concurrency when reading from the message collector, as various consumers in a consumer group can fetch data from different partitions within a topic simultaneously. They also provide more scalability on the broker side, as a cluster of brokers can divide partitions from a topic between them. The replication factor defines each partition's redundancy in Kafka clusters. Although these configurations were not used in our prototype pipeline, which only features one broker, they are important considerations for future scaling of the pipeline.

Given that this layer is the source of truth in our pipeline, data durability configurations are essential.[29] However, due to the unbounded size of streaming data, it would not be possible to store all stream events forever. Kafka allows us to set a time or a size limit for the event log, after which the oldest segments are deleted (or compacted). We can use the retention period configuration to limit how far behind a consumer may be before unprocessed data is lost in a future implementation.

29 Broker configurations that could be explored for future optimizations include: log flush policy and the minimum number of replicas that must acknowledge a write (durability vs latency trade-off).

## 3.3 TRANSFORMER

The transformer component has the role of transferring data made available by the message collector into the streaming processor. While the current implementation supports both ETL and ELT approaches to loading data, our prototype pipeline is limited to ETL.

Raw streams fetched from the message collector are filtered and transformed before being inserted into the streaming processor. Binary AIS messages are decoded into their full-text format, giving us access to their attributes for filtering and insertion purposes. As the transformer pre-processes events in the stream, it delivers them to the processing stage.

This component uses Kafka's consumer Java API to fetch data from the broker, the *AisLib* library to decode binary AIS messages into their full-length format and MonetDB's Java library to load data into the streaming processor. MonetDB's library includes both the JDBC interface and the *MAPI* library, a lower-level API for communication with the database. The *MAPI* library powers the JDBC library and allows access to non-JDBC specific functionalities, such as the COPY INTO statement for efficient bulk data loading into MonetDB.

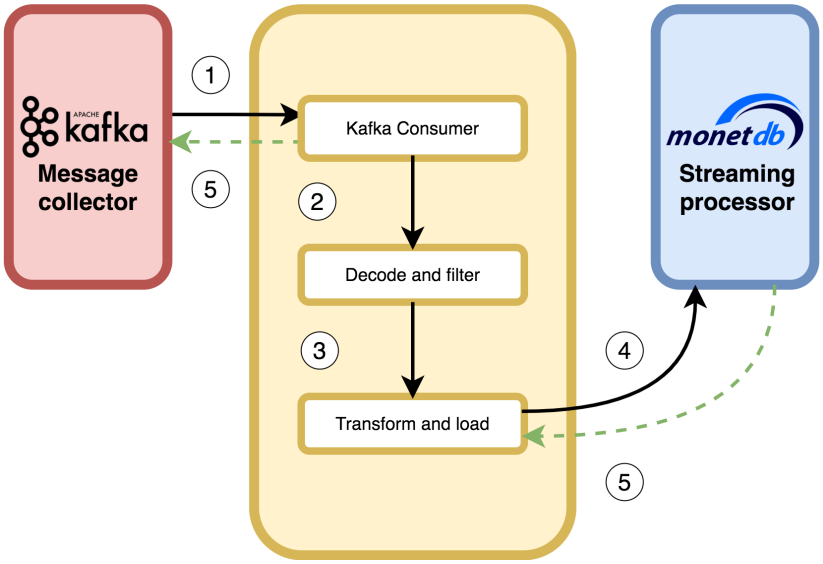



Figure 8: Transformer component control flow

Figure 8 shows the behaviour of the transformer component, after having connected to both the message broker and the streaming processor:

1. The transformer fetches a batch of AIS messages from the message collection layer, using Kafka's consumer library.

2. Fetched AIS messages are decoded into full-text, and corrupt or unwanted events are filtered out of the stream.

3. Decoded events are converted into a general data schema known by the processor, only containing the relevant information.

4. The pre-processed stream slice is loaded into MonetDB.

5. The streaming processor replies with the success of the data insertion operation. If the load was successful, the consumer commits the consumption offset to Kafka.

Given the unbounded nature of streams, the transformer tries to fetch new batches indefinitely. Each batch of data is fully pre-processed and loaded before the transformer requests more data from the broker.

### 3.3.1 *Transforming data*

The first step in pre-processing raw AIS streams is decoding the binary format they are sent in. Developing an AIS decoder is out of this thesis's scope, so we decided to use one of the various decoding libraries available. Two Java decoding libraries, *AisLib*[30] and *aismessages*[31], were compared using our use case data, to identify the fastest one. *AisLib* consistently decoded messages in less time, as the input rate was scaled. We chose it as our decoding library, given that it also satisfies our requirements for a multi-line message decoder.

For each batch, the transformer decodes binary AIS lines in the order of their timestamps. Decoding is done one message at a time, except when a multi-line message is detected. In that case, the transformer stores all binary lines corresponding to a fragmented message, to decode when the complete message is available. Corrupted or incorrect messages are discarded from the stream, with the possibility of logging these occurrences to file.

The next step is filtering, whereby the transformer discards messages we are not interested in processing. By removing these events before insertion into the database, we avoid their impact on the streaming processor. Given that our case study focuses on AIS messages of type 1 through 5, all other message types are not inserted into the database. Other filtering work that the transformer could do include discarding position reports with invalid coordinates or speed, or only inserting messages from a given geographical region.

Before loading the filtered stream into the processor, the transformer converts the raw event format into a general format. This projection removes the data transfer overhead caused by sending data not used for processing. Converting into a general format before the processing layer also future-proofs our solution, by defining how to transform each different data schema into a format that our processing component knows. The transformer also changes corrupted or missing attributes in messages to default values and escapes strings to avoid data integrity constraints and parsing issues in the database.

30 AisLib GitHub page: https://github.com/dma-ais/AisLib
31 aismessages GitHub page: https://github.com/tbsalling/aismessages

### 3.3.2 *Loading data*

There are two methods for loading data into the database: issuing an SQL INSERT INTO command through a JDBC connection or sending data after a COPY INTO command using MonetDB's underlying communication library MAPI. The INSERT INTO command contains decoded messages in full-text form, while the COPY INTO command uses a buffered writer interface to send rows of data. For large amounts of data, the COPY INTO method is faster than INSERT INTO, due to its better memory allocation. COPY INTO was chosen as our primary method for loading data in the current implementation and used for testing the pipeline in Chapter 5.

The most crucial configuration to consider when loading data is the number of pre-processed records to batch before inserting into the processor. MonetDB is optimised for OLAP workloads, meaning that a small number of larger update transactions perform better than a larger number of small updates. However, batching more data before loading it increases the pipeline's ingestion latency. By trying out different batch sizes, we can reach an optimal configuration for our case and improve our pipeline performance.

### 3.3.3 *Data reliability*

One of the most critical roles of the transformer component is to ensure that stream data is correctly inserted into the streaming database. Data insertion follows a synchronous model, where the transformer waits for an acknowledgement from the database before fetching more tuples. If the transformer could not insert a batch of decoded data, it will retry for a specified number of tries. After several unsuccessful retries, the transformer alerts the user and does not commit the offsets from the latest batch fetched from Kafka. This strategy ensures that data is only marked as read when the component successfully moves it to the database.

Given that the streaming database only provides in-memory storage for streams and the delay between a tuple being inserted and used for processing, we need further reliability mechanisms to deal with database crashes. Whenever the transformer detects a database crash, we must ensure that all data not used for processing is re-sent.

After losing connection to the processor, the transformer will retry connecting for a specified number of tries. If a successful connection is made, the stream consumption offset is rewound. If we only need at-least-once processing guarantees, the stream consumption offset is reset to 0, restarting the fetching process at the beginning of the stream. If we need exactly-once guarantees, i.e. processed events cannot be re-sent, the transformer must know the offset of the last tuple that was processed. For this reason, the streaming database registers the last offset of each processed window after execution. The transformer can then

read the latest processed offset and reset the consumption offset to the first tuple that was not processed.

### 3.3.4 *Concurrent loading*

The transformer can be run as a single consumer or a multithreaded consumer group. Consumers in a group share the work of fetching data from a topic, allowing for concurrent reads from the message broker. Each consumer runs on its thread and gets assigned some topic partitions to read from exclusively. Using consumer groups increases read throughput and allows for horizontal scaling. Running the transformer in multiple threads can also improve the throughput of the pre-processing work, as multiple threads are decoding and transforming simultaneously. We could also scale data ingestion into the streaming database in a multithreaded scenario if the database has reliable concurrent insertion capabilities.

However, due to our case study's limitations, concurrent consumption was not possible in our pipeline implementation. Concurrent reads require the stream to be partitioned, which makes sense in some application scenarios, but it is hard to achieve with our use case. AIS data is represented in a binary-encoded format throughout the pipeline before being decoded for database loading. This format provides no information about the event that we could use as a partitioning key. Because we cannot partition the stream in a meaningful way, the transformer component is limited to run as a single consumer.

## 3.4 MONETDB DATA LOADING

The streaming processor is the last step in the ingestion pipeline, where data arrives to be processed. Delivered events require immediate processing and do not need to be persisted. The message collector component already persists raw streams and provides mechanisms for replaying lost data in case of crashes. Redoing this work in the streaming database would be unnecessary and a performance drain.

For that reason, the streaming extension of MonetDB provides stream tables, a lightweight variation of a regular relational table to store event data temporarily. Stream tables are kept in-memory and are not subject to the standard transaction management of MonetDB. They are designed to be the end-points to deliver IoT events and do not offer reliability guarantees. Streaming tables can be created with the following SQL syntax:

```
CREATE STREAM TABLE table_name (table_columns)
  [SET WINDOW positive_number [STRIDE positive_number]];
```

The WINDOW parameter defines a tuple count window, denoting the minimum number of tuples present for a continuous query to be triggered on the stream table. If this parameter

is not set, any continuous query using this table will be triggered by a time-based window instead, which can be defined when registering the query. If a tuple count window is used, the STRIDE parameter determines how many tuples should be removed from the stream table at the end of a continuous query invocation. If STRIDE is set to a positive number, the oldest N tuples are removed. If STRIDE is set to 0, all tuples are kept. The default action is to delete all tuples used for a CQ invocation.

Each stream corresponds to a stream table, characterised by its schema and windowing properties. As the transformer layer pre-processes a given stream, it inserts decoded events in its stream table through an INSERT INTO or a COPY INTO command. To simplify the flow of data and guarantee at-least-once processing, the transformer blocks after loading a batch until the database acknowledges the insertion.

Concurrent loading of events into the database could improve the throughput and scaling potential of loading the data into MonetDB, but this requires that we partition the stream. As previously explained, this is not possible in the current implementation due to limitations of the AIS dataset. The only safe mechanism to load data concurrently into MonetDB is to insert data into multiple stream tables, with each transformer instance having exclusive access to a table. The full stream table is the union of all individual stream tables, each representing a partition.

# 4

# DATA PROCESSING

This section will discuss the data processing stage of our pipeline, the MonetDB streaming processor. We begin with an overview of the MonetDB database and its Continuous Query Engine (CQE) extension. Then, we present our AIS stream processing benchmark, while also discussing our implementations of event-time windows and failure recovery in the current CQE.

## 4.1 CONTINUOUS QUERY ENGINE IN MONETDB

### 4.1.1 *MonetDB Implementation Architecture*

MonetDB is an open-source columnar RDBMS that originated from the national research institute for mathematics and computer science in the Netherlands (CWI) in 1993. It is designed for applications in data warehousing, business intelligence and business analytics. MonetDB focuses on read-intensive workloads with updates in large chunks. Since going open-source in 2004, MonetDB has evolved to include support for various programming languages, a kernel-level columnar approach, and a lightweight embedded version, presenting an SQLite alternative.

One of the most significant differences between MonetDB and traditional DBMSs is how data is stored and accessed. In traditional DBMSs, all tuple attributes are stored in one record (i.e. row-store), meaning the data is fragmented horizontally (Figure 9). MonetDB fragments its data vertically (i.e. column-store), meaning that each attribute column is stored separately. Each column is stored in a C array, where the OID maps to an index of the array, calculated with a base offset. Columns of the same relation are aligned, simplifying tuple reconstruction (i.e. translation into row format).

MonetDB manipulates data through MonetDB Assembly Language (MAL), which consists of low-level two-column relational algebra operations on BATs. MAL plans are executed in an operator-at-a-time manner, meaning that each operator is evaluated over the entire data before moving on to the next operation. This bulk processing model has performance advantages over the traditional tuple-at-a-time paradigm, such as creating high instruction

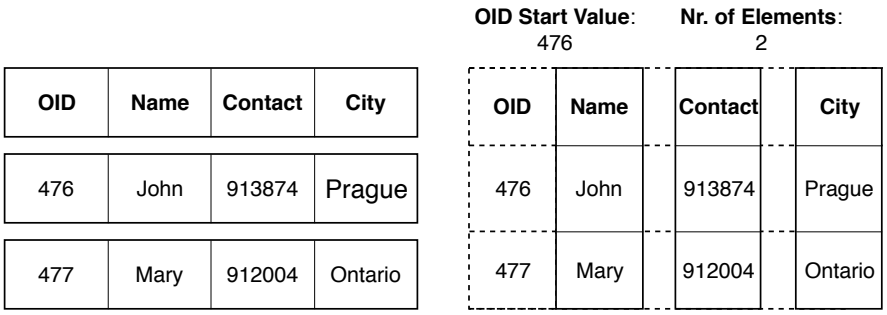



Figure 9: Data representation in row stores and column stores

locality and being more receptive to compiler optimisation. All intermediate results are stored in a columnar format to take full advantage of optimised vector operations. Row-wise tuples are only constructed before sending the final result to the client.

MonetDB's query processing architecture is composed of three main layers. The front-end layer translates high-level SQL queries into MAL plans and features strategic optimisers to exploit SQL and relational algebra semantics. The back-end further optimises the MAL plan, using a pipeline of optimiser modules that target specific performance improvements. The kernel layer provides the implementation for the columnar data structure and the binary relational algebra operators.

### 4.1.2 *Continuous Query Engine*

In 2016, a basic streaming engine was added to MonetDB.[32] To implement stream processing functionalities on top of the columnar DBMS, a new set of MAL operators was created, along with a continuous query execution scheduler and several optimisation policies. The engine uses the batch processing programming model, leveraging MonetDB's vectorised operations and being closer to the traditional RDBMS batch model. It uses SQL as the programming language, extending it to include the stream table, window and continuous query abstractions [7]. A description of stream tables can be found in the Section 3.4.

Continuous queries (CQs) are a special type of SQL user-defined procedures/functions that use stream tables as input and execute when their window triggering condition is fulfilled. When users register CQs to the continuous query scheduler, they also define when the query should be executed, i.e. tuple or time window boundaries. The scheduler initiates a continuous query when all of its trigger conditions are satisfied, similarly to the Petri-net model.[33] Continuous queries can be registered with the following SQL syntax:

32 IoT and Streaming in MonetDB: https://monetdb.org/blog/IoT-and-streaming-in-MonetDB
33 Petri-net model overview: https://en.wikipedia.org/wiki/Petri_net

START CONTINUOUS – PROCEDURE — FUNCTION ″
function˙name ( arguments )
[WITH [HEARTBEAT positive˙number] [CLOCK timestamp]
[CYCLES positive˙number]] [AS tagname];

The HEARTBEAT parameter defines a time-based window, indicating the number of milliseconds between continuous query executions. If it is not set, the scheduler will use tuple count windows. If no tuple count windows are defined, the CQ will be triggered at every execution round. The CYCLES parameter limits the number of times a continuous query will run before being removed by the scheduler. If it is not set, the continuous query will run forever. The CLOCK parameter specifies a wall-clock time for the continuous query to start. If it is not set, the CQ will start immediately upon registration. The tagname parameter is used to identify a continuous query, allowing a procedure with different arguments to be registered as different continuous queries.

After registering a continuous query, it is possible to pause and resume execution or stop it completely. The status of registered CQs, including errors and the last execution time, can be obtained with a *cquery.status()* call. All continuous queries are stopped once the MonetDB server shuts down and must be manually restarted.

## 4.2 STREAM PROCESSING AIS DATA

To get useful business insights out of stream processing, we need to process the data with the appropriate queries and data operations. We developed a benchmark containing analytical queries which are useful for AIS applications. Our objective was to demonstrate the engine's capabilities in processing AIS data into valuable results and test and stress the underlying stream processor. We are interested in appraising what operations and functionalities the engine provides, as well as its performance and correctness.

We aim to test all available streaming operators, along with some additional features. The streaming operators include projections, selections, aggregations, stream-relational joins and stream-stream joins. Due to event-time windows not being available in the continuous query engine and tuple-count windows not fitting our use case, processing-time tumbling windows present the most appropriate window type. We also implemented queries that simulate event-time windowing in their results, despite using processing-time tumbling windows for scheduling.

We also want to test the output of processing results, which can be a stream table, if the data is to be processed further, or a relational table for persistent storage. Moreover, a query can store results in relational tables by appending the last window's values or continuously updating the previous results.

The queries that we developed for the benchmark are described in Table 2, alongside the stream processing features we test in each query.

| | **Query description** | **Tested features** |
|---|---|---|
| 1 | Find currently anchored ships | Selection |
| 2 | Get the speed of ships | Projection, output to stream |
| 3 | Track the movements of a ship | Projection, output to stream |
| 4 | Calculate the number of distinct ships | Aggregation |
| 5 | To each voyage message, add the current position of ship | Stream-stream join |
| 6 | Find ships anchored at a base station | Stream-relational join, using a processed stream |
| 7 | Find ships within a kilometre radius from a base station | Stream-relational join, distance calculation, selection |
| 8 | For every ship, find the closest neighbour ship | Stream-stream self join, distance calculation |
| 9 | Calculate average speed observed per ship over time | Continuous update of results on a relational table |
| 10 | Calculate the number of ships under-way in hour windows | Event-time windowing |
| 11 | Calculate the average and the maximum speed of moving ships in hour windows | Event-time windowing |

Table 2: AIS benchmark queries and their tested features

After conceptualising queries that tested all the desired features, we implemented them in MonetDB as continuous procedures. Full query implementations can be found in appendix A.2. For each continuous procedure, we created an output table to store results. The type of table (relational or stream) used for storing results depends on the query.

We also implemented an auxiliary function to calculate the distance between two ships and an auxiliary view that returns each ship's latest information. The distance function uses an approximation of the Haversine formula to calculate the distance in kilometres between two points in the geographic coordinate system, defined by their latitude and longitude (Appendix A.3.2). The view that presents the latest message for each ship uses the event timestamp for ordering and is useful for queries only interested in each vessel's most recent information.

Providing exactly-once processing guarantees not only requires the transformer to re-send events after a MonetDB crash but also that previously processed events are not loaded again. This requires the streaming processor to keep track of the broker offset for the last event that was processed. For this purpose, we created a table for registering the offset of the last event seen by each continuous query. At the end of each CQ invocation, the maximum offset in

the window is appended to the offset table, along with the query name. After a crash, the transformer reads from the offset table to rewind the stream to the first unprocessed event.

Event-time processing, i.e. processing events according to when they happened and not when they were ingested, is a crucial feature for any IoT application with correctness requirements. The MonetDB CQE assumes ordered streams and, therefore, only implements processing-time windows. However, it is possible and useful to program queries which approximate the results of event-time windowing.

The event-time queries are triggered over a processing-time window, but aggregate events by their event timestamp. The result for each event-time window is then updated, ensuring that out-of-order tuples are taken into account, with no distinction between calculating late-arriving and on-time events.

### 4.2.1 *Example query: Query 11*

Query 11 is an example of an event-time query, where the average and maximum speed of ships are calculated in one-hour windows. Because event-time and processing-time can diverge and continuous queries can only be triggered on processing-time, partial aggregates for a given event-time period may have to be calculated over multiple calls to a continuous query. This means that we have to consider every message that has arrived between continuous query calls, and not only the ones with a timestamp within the current processing-time window.

To ensure that out-of-order messages are included in the final aggregations, this query features two steps: first, a continuous query computes partial averages and maximums grouped in one hour periods of event time, for all the vessel messages not yet processed; then, a view is used to aggregate all the calculated averages and maximums for an event-time period.

```
1  -- Save partial results to table
2  INSERT INTO r11
3  SELECT
4      -- These operands remove the minutes and seconds value from the timestamp,
       resulting in the lower boundary of the one one hour window
5      str_to_timestamp(
6          timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
7          '%Y-%m-%d %H:%M:%S'
8      ) as window_start,
9      -- Similar to the above operation, but adding one hour to the timestamp, the
       higher boundary of the window
10     str_to_timestamp(
11         timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
12         '%Y-%m-%d %H:%M:%S'
```

```
13      ) + INTERVAL '1' HOUR as window_end,
14      -- Partial aggregates for the event-time window and the number of tuples for
        the window computed in this call
15      avg(sog) as partial_average,
16      max(sog) as partial_maximum,
17      count(*) as partial_window_count
18  FROM
19      vessels
20  -- Messages are grouped by the lower boundary of the one hour window, e.g. a
        message with timestamp 13:40:56 would belong to the 13:00:00-14:00:00 window
21  GROUP BY
22      str_to_timestamp(
23          timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
24          '%Y-%m-%d %H:%M:%S'
25      );
```

While the continuous query above calculates partial aggregates for both in time and late-arriving messages across multiple calls, the view below can be used to join the partial results into the final aggregations.

```
1   CREATE VIEW q11_totals AS
2       SELECT
3           -- Window boundaries
4           window_start,
5           window_start + INTERVAL '1' HOUR as window_end,
6           -- Complete averages for a given event time period
7           avg(average) as final_average,
8           max(maximum) as final_maximum,
9           -- Total number of tuples in the window
10          sum(count) as window_count
11  -- Uses output table from the partial results CQ
12      FROM
13          r11
14  -- Final results are aggregated according to the start of the window, which can
        have multiple partial results to join
15      GROUP BY
16          window_start;
```

# 5

# EVALUATION

This chapter will assess the current implementation of the Tagus platform, evaluating both the ingestion pipeline and the processing stage. We start by assessing the performance of the ingestion pipeline and impact of the message collector layer by measuring the latency between the production of an event and the ingestion of that event into the stream processor.

Then, we judge the current state of the stream processing engine through a functional evaluation. This evaluation will inform us about the shortcomings and successes of the current implementation, and the features and designs that the next engine should have.

## 5.1 TAGUS INGESTION PIPELINE EVALUATION

Our main objective in building the Tagus ingestion pipeline is to provide low-latency and reliable streaming data ingestion. After implementing the pipeline, we did several tests to assess if the pipeline fits our latency requirements. Data streamed from the replay component should arrive at the streaming processor with low latency to make real-time streaming data processing a possibility.

Our test environment includes all the ingestion components, with no processing of events. The replay component was configured to send messages one-at-a-time to simulate a real-world environment. The transformer component was configured to batch 100 messages before inserting them into the streaming database.

This evaluation was conducted on a machine with the following characteristics:

| | |
|---|---|
| CPU Manufacturer | Intel |
| CPU Model | Core i7-8559U |
| # Cores | 8 |
| CPU Frequency | 2.7 GHz |
| RAM Memory | 32GB |
| Disk | 1024 GB SSD |
| Operating System | Fedora 32 x86_64 (Linux Kernel 5.10.8) |

Table 3: Test environment hardware specification

Time measurements are taken at each step of the pipeline, allowing us to determine the biggest latency bottlenecks. Timestamp information is recorded for each individual message. The pipeline latency is calculated as the difference between when an encoded event is sent to the platform by the replay component and when the corresponding decoded event is present in the streaming processor.

The replay component produces events from our AIS dataset over a given period, allowing us to speed up the replaying of a month's worth of AIS messages. The lower the replaying time is, the higher the data rate the platform will have to handle. Given that the original data rate is much lower than what the system should be able to handle, our tests start with a replay over a 96 hour period, which means that the data rate is around seven times higher than the original rate. We performed tests up to a data rate sixty times higher than the original, which corresponds to a 12 hour replay period.

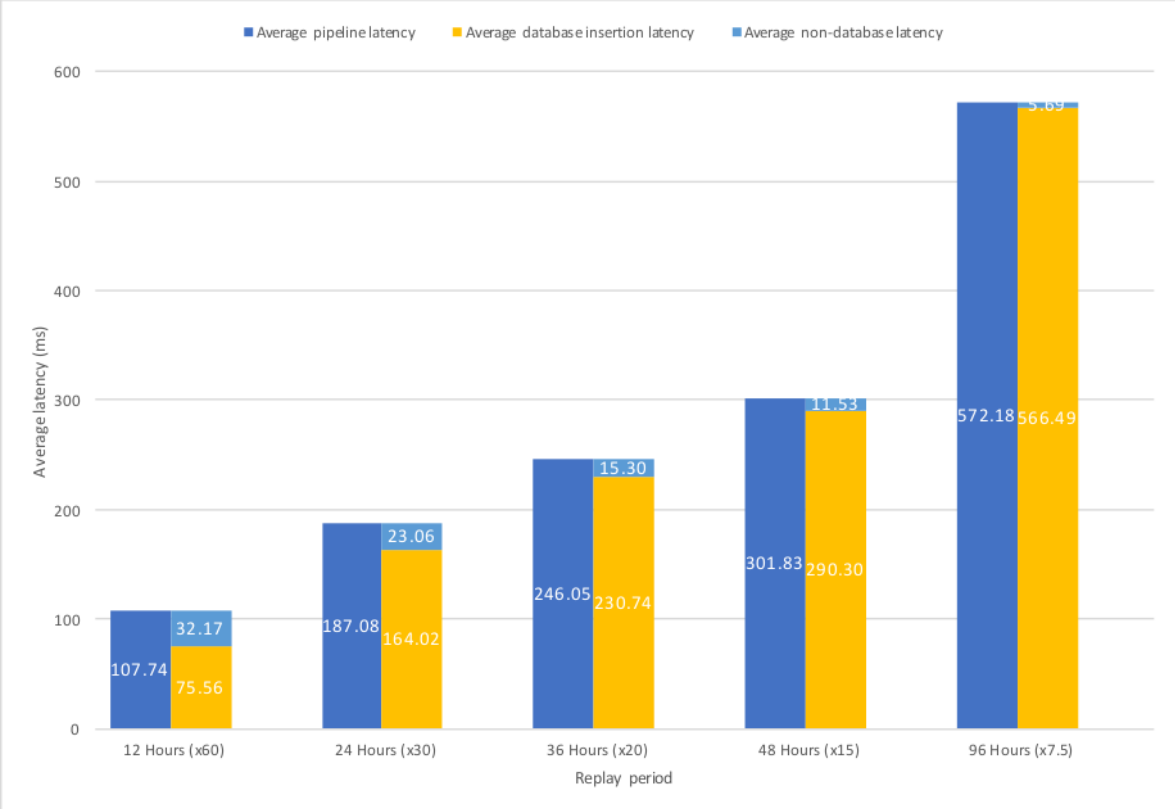



Figure 10: Average ingestion latency across various replaying periods. Latency is measured in milliseconds.

From the results presented in figure 10, we can conclude that the step between data reaching the transformer and being loaded into the streaming database is the biggest bottleneck in the current implementation of the pipeline. In this step, data is batched before being loaded, which introduces additional latency.

Tests with more extended replay periods present higher average pipeline latency due to the increased time the transformer spends in batching the data before loading it into the database. Longer replay periods mean that the replay component sends data less frequently, which increases the time necessary for a batch to be complete and subsequently loaded, given that all executions used the same batch size (100 tuples).

This means that the number of messages that should be batched before insertion should be adapted to the given scenario's data rate, striking a balance between too many small insertions and waiting too long before loading the data. To calculate the optimal batch size for a 12 hour replay period, we repeated the ingestion benchmark with different batch sizes and recorded each size's latency impact.

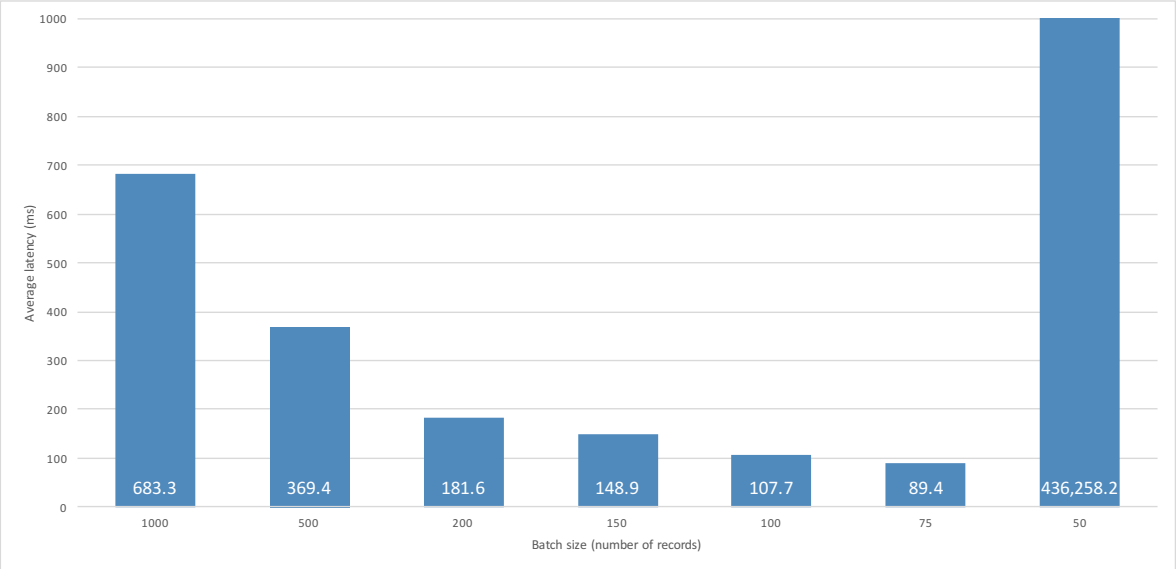



Figure 11: Average ingestion latency for a 12 hour replay period, with different batch sizes for database insertion. Latency is measured in milliseconds.

Figure 11 shows the average latency for batch sizes between 50 and 1000. As expected, the latency decreases with smaller batch sizes, given that the transformer waits less time before inserting data. However, we can also identify a sharp rise in latency when the batch size is 50 records. While smaller batch sizes result in less latency, they also mean more overhead for database insertion, as MonetDB is not optimized for small inserts. If the batch size is too small, the database cannot keep up with records being produced, and latency continually increases throughout the execution as the database lags more and more behind. For a 12 hour replay period, the optimal batch size is around 75 records. Larger batch sizes lead to higher latency, but lower batch sizes are too small for the database to keep up.

This evaluation concludes that most of the latency comes from the transformer-to-database pipeline stage, both from batching the data for loading and the load itself. The latency overhead from using Kafka in the pipeline is much smaller than the database loading latency,

meaning that our use of the message queue to provide reliability does not come with a very high cost. We can also conclude that we can reduce the database loading stage's latency by configuring an appropriate load batch size for a particular data rate. The insertion into the database is the biggest bottleneck in the pipeline, and its impact on latency should be addressed, but this work is out of this thesis's scope.

## 5.2 TAGUS PLATFORM FUNCTIONAL EVALUATION

We continue the Tagus platform's evaluation with a functional evaluation of MonetDB's continuous query engine's current implementation. While changing the CQE implementation is out of this project's scope, assessing its features and comparing it to other state-of-the-art systems will highlight possible future improvements.

Table 4 summarises the functional comparison between MonetDB CQE and other stream processing systems in the market. TimescaleDB and InfluxDB extend RDBMS functionalities to provide stream processing, therefore being closer to the CQE, while Flink and ksqlDB follow the dataflow-graph approach for more complex functionalities.

MonetDB's CQE follows an architectural and processing model similar to other streaming engines adapted from RDBMS. Various processes (updates and queries) work against a shared data environment in batches. While batching data might introduce some latency to when the query results will be available compared to systems with a pure stream processing model (e.g. Apache Flink), the current approach is generally regarded as more suited for extending an existing RDBMS.

While it is possible to construct both processing and event time tumbling windows, the assignment of tuples to their window must be implemented by the user with regular SQL. This contrasts with systems, such as TimescaleDB, which feature a special operator for defining event-time windows with a given time interval. Adding a window construct should be a priority for the future work on the CQE, as window assignment is a common operation that the user should not have to implement manually. The continuous query scheduler allows the user to define trigger conditions for CQs according to tuple count and processing time. Support for more complex triggers requires significant changes to the database model, such as pure stream processing and watermarks, and is usually not present in systems which extend an existing RDBMS.

There is no automatic mechanism for recalculating the result of a given window or delaying query execution if a late tuple arrives. Adding the capability of automatically updating the results of older windows, similarly to TimescaleDB's continuous aggregates, would be worthwhile. Currently, the user must explicitly implement the updates in the query.

| | **MonetDB CQE** | **TimescaleDB** | **InfluxDB** | **Flink** | **ksqlDB** |
|---|---|---|---|---|---|
| Architectural model | Database model | Database model | Database model | Dataflow model | Dataflow model |
| Processing model | Batch | Batch | Batch | Stream | Stream |
| Windowing | No time window construct | Tumbling event time | Tumbling event time | Tumbling, sliding and session event time | Tumbling, sliding and session event time |
| Query triggering | Tuple count or processing time | Processing time | Processing time | Event-by-event, processing and event time | Event-by-event |
| State | In-memory state, persistent log with ingestion pipeline | Disk | Disk | In-memory state, persistent log | In-memory state, persistent log |
| Joins | S-2-T and S-2-S (only on current batch) | S-2-T | S-2-T | S-2-S and S-2-T | S-2-S and S-2-T |
| Event time processing | No | Possible, results are always recalculated | No, but possible to delay window execution | Yes | Yes |
| Processing semantics | None by default, at-least-once with ingestion pipeline | — | — | Exactly-once | Exactly-once |

Table 4: Functional comparisons between MonetDB CQE and other modern stream processing systems

Transient state (e.g. input streams) is stored in-memory to avoid disk I/O overhead, while persistent results can be stored in regular SQL tables, persisted to disk. Data reliability is not ensured by the CQE, as all in-memory state is deleted when the database server crashes. If the CQE is used with the Tagus ingestion pipeline, transient data is persisted by the message collector component. In the case of a crash, the transformer component can rewind the consumption of the stream from the message collector and redeliver lost data. Stream processing systems with a focus on low-latency results also adopt this approach, while systems like InfluxDB persist all data to disk, increasing latency but also simplifying the data ingestion step. The current implementation of the CQE features a simple data retention mechanism, whereby tuples used in processing are deleted from the input stream table. However, this mechanism is not supported if a processing-time trigger is being used. Further implementation is needed to expand the data retention functionality.

By default, the CQE provides no processing guarantees, due to stream data being lost upon shutdown. If the CQE is used with the Tagus ingestion pipeline, at-least-once processing can be achieved by re-sending lost stream data stored in the message collector. Exactly-once processing requires that the stream processor keeps track of its position in the stream. The transformer component can then determine where it should rewind in the stream to guarantee that no event is loaded twice. This can currently be achieved in a limited way, but the user's queries must manually manage the stream position.

Joins between streams are only supported on the current batch, meaning that merging data with the CQE is very limited. Join capabilities could be extended by adding a user-defined join window for stream-to-stream joins, which could differ from the continuous query windows.

While MonetDB natively supports timestamp data types and operations, no event or processing timestamps are automatically added, leaving it to the user to add time information to the stream schema explicitly. Although it is possible to register more than once a continuous query per stream, a stream can be consumed by maximally one continuous query during a round.

# 6

# CONCLUSION

## 6.1 SUMMARY

This project proposed an approach to adapting MonetDB, a modern columnar RDBMS, for IoT data analysis, focusing on the ingestion/storage layer of the IoT application stack. Our goals were to learn how to handle large volumes of heterogeneous streaming data and efficiently and reliably ingest this data into the database. Furthermore, we assessed the capabilities of the MonetDB continuous query engine, developed for a previous project, to delineate future development work.

We designed an ingestion and processing pipeline that aimed to fulfil standard IoT application requirements. This pipeline uses Apache Kafka as the core ingestion component, which provides data durability and reliability, and MonetDB as the processing component, which offers continuous queries over in-memory tuples. The pipeline also features a replay component, which simulates IoT end-devices producing real-time data, and a transformer component, which consumes data from Kafka, transforms the data and loads it into the MonetDB.

This project used the AIS maritime tracking system as its example application: we used logs of collected AIS streams as the dataset, implemented our pipeline design to ingest and process this data, and modelled our stream processing benchmark according to useful real-world maritime queries. Our choice of AIS as the example application raised some limitations on what could be implemented in our pipeline. Due to AIS messages being transmitted in an encoded format with no identifying information, we could not pursue the ELT strategy when ingesting data or partition the stream according to a stream key.

After implementing our proposed pipeline to handle AIS data, we assessed its performance and reliability by replaying an AIS stream into our platform. Our pipeline was able to comfortably ingest events replayed at a much higher data rate than the original dataset, leading us to conclude that the current implementation can handle the AIS use case. Moreover, we evaluated MonetDB's CQE by comparing its functionalities to other modern

stream processing systems. This assessment provided us with a blueprint for improvements in the processing layer, paving future work on the component.

## 6.2 Future work

This project mostly focused on the ingestion layer of the Tagus platform, meaning that the processing, emission and administrative layers still leave much to be improved in the future. Even within the ingestion layer, not all the work that could be done falls under this project's scope due to time constraints. Below we first elaborate several main topics we plan to address shortly before identifying more relevant topics for future work.

**Integration with Apache Avro -** Integrating a data serialisation system such as Apache Avro[34] may be necessary when dealing with other use cases given the data format heterogeneity in the IoT field. Avro provides fast serialisation to a compact binary format that directly maps to JSON while having much less overhead. Using Avro to help exchange data between the pipeline components brings robustness and clarity by ensuring data producers send correctly formatted data and allowing consumers to clearly understand what the data is. Its rich schema-definition language and schema evolution features simplify schema management and reduce the overhead of changing data formats, common in IoT.[35] Kafka has extensive support for Avro and MonetDB has JSON support, so this integration would be possible.

**Integration with MQTT -** Another requirement when upgrading our platform to real-world use is integrating IoT devices as data sources, i.e. real devices sending information to the message collector through a communication protocol. MQTT[36] is a popular choice for IoT protocol, offering lightweight, low requirements and scalable message passing between devices and applications. End devices publish their data to an MQTT broker, which passes it to subscribing applications. We can connect the MQTT broker to our message collector layer (Apache Kafka) through Kafka Connect, which offers bidirectional communication between the two brokers, or directly integrate MQTT clients into the Kafka broker using Confluent MQTT proxy[37] or Waterstream.[38]

**Kafka clustering and stream partitioning -** Our current pipeline implementation does not leverage Kafka's fault-tolerance, failover and horizontal scaling features, because our message collector layer only features one Kafka broker. Using a cluster of Kafka brokers means that data is replicated across various machines, which guarantees data reliability and availability in case of node failures. Furthermore, the work of reading and writing from

34 Apache Avro: https://avro.apache.org/
35 Article: Why Avro for Kafka Data? https://www.confluent.io/blog/avro-kafka-data/
36 MQTT: https://mqtt.org/
37 Article: Apache Kafka Native MQTT at Scale: https://www.confluent.io/blog/iot-streaming-use-cases-with-kafka-mqtt-confluent-and-waterstream/
38 Waterstream (MQTT-Kafka platform): https://waterstream.io/

the message collector can be balanced across cluster nodes, allowing our ingestion pipeline to scale to higher data rates. Parallelising reading and writing to brokers requires that the stream topic is partitioned, meaning that a stream is divided into distinct, independent sub-streams. Messages can be balanced across partitions according to their key (if the messages have one) or in a round-robin way. Order is only guaranteed within a partition, meaning that applications which require ordering of events according to some attribute, e.g. ordered events within a geographical zone, must use it as the stream partitioning key. Our AIS dataset does not allow for keyed partitioning and order is a requirement, meaning that we could not pursue this in the current project.

**Automatic window assignment and update of continuous aggregates -** An essential feature to add to our stream processor would be support for an event-time windowing operator, similar to TimescaleDB's time_bucket, which could abstract the logic of window assignment from the user. This operator would calculate the window to which an event belongs, given an event timestamp and a window size. Currently, if the user wants to do a continuous windowed aggregation on a stream, he must manually implement the mechanism to update results from older windows (e.g. when a tuple is late). To simplify this typical operation, we could implement a continuously updating materialised view for windowed aggregate queries, which would automatically recalculate its results as new data comes in.

At a longer term, there are many more features to explore and add. Just to name a few:

- **Expand the pipeline benchmarks -** Test our pipeline with higher data rates, more varied data and more producers. Furthermore, testing the effect of networking on the pipeline when components are in different machines.

- **Exploring more Kafka configurations -** Number of broker acknowledgements (durability vs latency), producer batch size (latency vs throughput), broker retention period (durability vs storage) and broker log flush policy.

- **Improve CQE data retention -** Add new retention strategy decoupled from processing to bring more flexibility and control. Data older than a user-defined interval could be automatically removed or compacted by a dedicated thread.

- **Allow for multiple continuous queries to process data from one stream table -** Currently, MonetDB's stream tables can only be safely consumed by one continuous query, requiring users to duplicate the stream for concurrent use [7].

- **Automatic logging of last processed event -** During failure recovery, the transformer component needs to access the last event used in processing to rewind the stream loading with no duplicates. Currently, this must be done manually by the user.

- **Explore alternatives to store processed data for client applications to access -** Results from query processing could be stored in permanent tables in MonetDB or produced into Kafka.
- **Add an administrative component -** An interface for users to register IoT data sources (producers) and continuous queries (consumers).

# A

# APPENDIXES

## A.1 AIS BENCHMARK DATA SCHEMAS

| Vessels | |
|---|---|
| id | Ship ID |
| latitude | Position |
| longitude | Position |
| navigation_status | Current ship activity |
| speed | Speed in knots |
| turn | Turn angle in degrees |
| timestamp | Event timestamp |
| insertion_time | Ingestion timestamp |

| Base | |
|---|---|
| id | Base ID |
| latitude | Position |
| longitude | Position |
| timestamp | Event timestamp |
| insertion_time | Internal timestamp |

| Voyage | |
|---|---|
| id | Ship ID |
| name | Ship name |
| ship_type | Code for type of vessel |
| draught | Draught in meters |
| destination | Trip destination |
| timestamp | Event timestamp |
| insertion_time | Internal timestamp |

Table 5: Data schemas and attributes used in this project, derived from AIS messages of type 1-5

## A.2 AIS BENCHMARK QUERIES

### A.2.1 *Query 1: Find currently anchored ships*

```
1 CREATE PROCEDURE q1()
2 BEGIN
3     INSERT INTO r1
4     SELECT current_timestamp AS calc_time, * FROM distinct_vessels
```

```
5       WHERE nav_status = 1;
6   END;
```

### A.2.2 *Query 2: Get the speed of ships*

```
1   CREATE PROCEDURE q2()
2   BEGIN
3       INSERT INTO r2
4       SELECT current_timestamp AS calc_time, id, sog, timestamp, k_offset FROM
        vessels;
5   END;
```

### A.2.3 *Query 3: Track the movements of a ship S*

```
1   CREATE PROCEDURE q3(ship integer)
2   BEGIN
3       INSERT INTO r3
4       SELECT current_timestamp AS calc_time, lat, log, nav_status FROM vessels
5       WHERE id = ship;
6   END;
```

### A.2.4 *Query 4: Calculate number of distinct ships*

```
1   CREATE PROCEDURE q4()
2   BEGIN
3       INSERT INTO r4
4       WITH vessel_count AS (
5           SELECT count(*) AS distinct_count FROM distinct_vessels
6       )
7       SELECT current_timestamp AS calc_time, distinct_count FROM vessel_count;
8   END;
```

### A.2.5 *Query 5: To each voyage message, add the current position of ship*

```
1   CREATE PROCEDURE q5()
2   BEGIN
3       INSERT INTO r5
```

```
4       SELECT current_timestamp AS calc_time, voyage.id, vessels.lat, vessels.log,
        vessels.sog, voyage.name, voyage.ship_type, voyage.draught, voyage.dest,
        voyage.timestamp as voyage_timestamp, vessels.timestamp as vessels_timestamp
5       FROM voyage
6       JOIN distinct_vessels AS vessels
7       ON vessels.id = voyage.id
8       AND vessels.timestamp <= voyage.timestamp;
9   END;
```

### A.2.6 *Query 6: Find ships anchored at base station*

```
1   CREATE PROCEDURE q6()
2   BEGIN
3       INSERT INTO r6
4       SELECT current_timestamp as calc_time, r1.id as ship, base.id as station
5       FROM r1
6       JOIN base_downsampled AS base ON base.log = r1.log AND base.lat = r1.lat;
7   END;
```

### A.2.7 *Query 7: Find ships within a kilometer radius from a base station*

```
1   CREATE PROCEDURE q7(distance integer)
2   BEGIN
3       INSERT INTO r7
4       WITH base_not_null AS (
5           SELECT *
6           FROM base_downsampled
7           WHERE lat <> 0 AND log <> 0
8       ),
9       vessels_not_null AS (
10          SELECT *
11          FROM vessels
12          WHERE lat <> 0 AND log <> 0
13      )
14      SELECT current_timestamp as calc_time, vessels.id as ship, base.id as station
        , vessels.timestamp as vessel_timestamp, base.timestamp as base_timestamp,
        distance_in_km(base.lat, base.log, vessels.lat, vessels.log) as distance
15      FROM vessels_not_null as vessels
16      JOIN base_not_null as base ON distance_in_km(base.lat, base.log, vessels.lat,
         vessels.log) <= distance
17      AND distance_in_km(base.lat, base.log, vessels.lat, vessels.log) > 0;
18  END;
```

### A.2.8 *Query 8: For every ship, find the closest neighbor ship*

```
CREATE PROCEDURE q8()
BEGIN
    INSERT INTO r8
    WITH distance AS (
        SELECT v1.id AS ship, v2.id AS neighbour_ship, distance_in_km(v1.lat,v1.log,v2.lat,v2.log) AS distance_in_km
        FROM distinct_vessels AS v1
        JOIN distinct_vessels AS v2 ON v1.id <> v2.id
        AND v1.lat <> 0 AND v1.log <> 0
        AND v2.lat <> 0 AND v2.log <> 0
    )
    SELECT current_timestamp AS calc_time, ship, neighbour_ship, distance_in_km
    FROM distance
    WHERE (ship, distance_in_km) IN (SELECT ship, min(distance_in_km) FROM distance GROUP BY ship);
END;
```

### A.2.9 *Query 9: Calculate average speed observed per ship over time*

```
CREATE PROCEDURE q9()
BEGIN
    UPDATE r9 SET
        calc_time = current_timestamp,
        speed_sum = speed_sum + update_vessels.sum,
        speed_count = speed_count + update_vessels.count
    FROM
        (SELECT vessels.id, COUNT(sog) AS count, SUM(sog) AS sum FROM vessels
        JOIN r9 ON vessels.id = r9.id
        GROUP BY vessels.id) AS update_vessels
    WHERE
        r9.id = update_vessels.id;

    INSERT INTO r9
    WITH new_inserts AS
        (SELECT id, SUM(sog) AS sum_speed, COUNT(*) AS count_speed
        FROM vessels
        WHERE id NOT IN (SELECT id FROM r9)
        GROUP BY id)
    SELECT current_timestamp AS calc_time, id, sum_speed, count_speed
```

```
21      FROM new_inserts;
22  END;
```

### A.2.10 *Query 10: Calculate number of ships under-way in hour windows*

```
1   CREATE PROCEDURE q10()
2   BEGIN
3       INSERT INTO r10
4       SELECT
5           current_timestamp AS calc_time,
6           str_to_timestamp(timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
        '%Y-%m-%d %H:%M:%S') as window_start,
7           str_to_timestamp(timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
        '%Y-%m-%d %H:%M:%S') + INTERVAL '1' HOUR as window_end,
8           min(timestamp) as actual_window_start,
9           max(timestamp) as actual_window_end,
10          count(*) as count
11      FROM
12          vessels
13      WHERE
14          vessels.nav_status = 0
15      GROUP BY
16          str_to_timestamp(timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
        '%Y-%m-%d %H:%M:%S');
17  END;
18
19  CREATE VIEW q10_totals AS
20      SELECT
21          window_start,
22          window_start + INTERVAL '1' HOUR as window_end,
23          min(actual_window_start) as actual_window_start,
24          max(actual_window_end) as actual_window_end,
25          sum(count) as count
26      FROM
27          r10
28      GROUP BY
29          window_start;
```

### A.2.11 *Query 11: Calculate average and maximum speed of moving ships in hour windows*

```
1   CREATE PROCEDURE q11 ()
2   BEGIN
3       INSERT INTO r11
```

```
4       SELECT
5           current_timestamp as calc_time,
6           str_to_timestamp(timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
        '%Y-%m-%d %H:%M:%S') as window_start,
7           str_to_timestamp(timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
        '%Y-%m-%d %H:%M:%S') + INTERVAL '1' HOUR as window_end,
8           min(timestamp) as actual_window_start,
9           max(timestamp) as actual_window_end,
10          avg(sog),
11          max(sog),
12          count(*)
13      FROM
14          vessels
15      WHERE
16          sog <> 0 AND
17          sog <> 1023
18      GROUP BY
19          str_to_timestamp(timestamp_to_str(timestamp, '%Y-%m-%d %H') || ':00:00',
        '%Y-%m-%d %H:%M:%S');
20  END;
21
22  CREATE VIEW q11_totals AS
23      SELECT
24          window_start,
25          window_start + INTERVAL '1' HOUR as window_end,
26          min(actual_window_start) as actual_window_start,
27          max(actual_window_end) as actual_window_end,
28          avg(average) as average,
29          max(maximum) as maximum,
30          sum(count) as count
31      FROM
32          r11
33      GROUP BY
34          window_start;
```

## A.3 AUXILIARY QUERIES

### A.3.1 *Distinct vessels*

```
1  CREATE VIEW distinct_vessels AS
2      SELECT id, lat, log, nav_status, sog, rot, timestamp, insertion_time,
       k_offset from vessels
3      WHERE (timestamp, id) IN (SELECT max(timestamp), id FROM vessels GROUP BY id)
       ;
```

### A.3.2 *Distance calculation*

Approximation of the Haversine formula SQL implementation (distance in km):

```
1  CREATE FUNCTION distance_in_km (p1_lat REAL, p1_log REAL, p2_lat REAL, p2_log
       REAL)
2  RETURNS REAL
3  BEGIN
4      RETURN 111.319 *
5      SQRT(
6          (p2_lat-p1_lat) *
7          (p2_lat-p1_lat) +
8          ((p2_log-p1_log) * cos((p2_lat+p1_lat)*0.00872664626)) *
9          ((p2_log-p1_log) * cos((p2_lat+p1_lat)*0.00872664626))
10     );
11 END;
```

Full Haversine formula SQL implementation (distance in km):

```
1  CREATE FUNCTION distance_in_km_full (p1_lat REAL, p1_log REAL, p2_lat REAL,
       p2_log REAL)
2  RETURNS REAL
3  BEGIN
4      RETURN 111.111 * DEGREES(ACOS(LEAST(1.0, COS(RADIANS(p1_lat))
5            * COS(RADIANS(p2_lat))
6            * COS(RADIANS(p1_log - p2_log))
7            + SIN(RADIANS(p1_lat))
8            * SIN(RADIANS(p2_lat)))));
9  END;
```

# BIBLIOGRAPHY


[1] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). Millwheel: Fault-tolerant stream processing at internet scale. 6(11):1033–1044.

[2] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803.

[3] Arasu, A., Babu, S., and Widom, J. (2003). Cql: A language for continuous queries over streams and relations. pages 1–19.

[4] Babu, S. and Widom, J. (2001). Continuous queries over data streams. *SIGMOD Record*, 30:109–120.

[5] Chin, J., Callaghan, V., and Allouch, S. B. (2019). The internet-of-things: Reflections on the past, present and future from a user-centered and smart environment perspective. *Journal of Ambient Intelligence and Smart Environments, vol. 11, no. 1, pp. 45-69, 2019*.

[6] daCosta, F. (2013). *Rethinking the Internet of Things*. Apress.

[7] Ferreira, P. E. S. (2016). Aiota: an iot platform on monetdb.

[8] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, S., and Kersten, M. (2012). Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35.

[9] Kleppmann, M. (2016). *Making Sense of Stream Processing*. O'Reilly Media.

[10] Liu, X., Dastjerdi, A., and Buyya, R. (2016). *Stream processing in IoT: Foundations, state-of-the-art, and future directions*, pages 145–161.

[11] Nasiri, H., Nasehi, S., and Goudarzi, M. (2019). Evaluation of distributed stream processing frameworks for iot applications in smart cities. *Journal of Big Data*, 6(1):52.

[12] Psaltis, A. G. (2017). *Streaming Data*. Manning.

[13] Sax, M. J., Wang, G., Weidlich, M., and Freytag, J.-C. (2018). Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business*

*Intelligence and Analytics*, BIRTE '18, New York, NY, USA. Association for Computing Machinery.