



University of Bamberg  
Chair of Mobile Systems



## Master Thesis

in the degree programme International Software Systems Science  
at the Faculty of Information Systems and Applied Computer Sciences,  
University of Bamberg

Topic:

# Benchmarks of quality-aware Data Stream Processing in DSMS

Author:

Md Saiful Ambia Chowdhury

Reviewer:

Prof. Dr. Daniela Nicklas

Date of submission:

16.05.2025

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                         | <b>1</b>  |
| 1.1      | Background and Context . . . . .            | 1         |
| 1.2      | Motivation . . . . .                        | 1         |
| 1.3      | Problem Statement . . . . .                 | 2         |
| 1.4      | Research Question . . . . .                 | 2         |
| 1.5      | Research Methodology Overview . . . . .     | 3         |
| 1.6      | Structure of the Thesis . . . . .           | 3         |
| <b>2</b> | <b>Literature Review</b>                    | <b>5</b>  |
| 2.1      | Benchmarking . . . . .                      | 5         |
| 2.2      | Data Stream Management System . . . . .     | 5         |
| 2.3      | Data Quality in Stream Processing . . . . . | 7         |
| 2.4      | Data Quality Dimensions . . . . .           | 8         |
| 2.4.1    | Accuracy . . . . .                          | 8         |
| 2.4.2    | Completeness . . . . .                      | 8         |
| 2.4.3    | Consistency . . . . .                       | 9         |
| 2.4.4    | Timeliness . . . . .                        | 9         |
| 2.4.5    | Uniqueness . . . . .                        | 9         |
| 2.4.6    | Validity . . . . .                          | 9         |
| 2.5      | Odysseus DSMS . . . . .                     | 10        |
| 2.6      | Procedural Query Language (PQL) . . . . .   | 11        |
| 2.7      | Mean Absolute Deviation (MAD) . . . . .     | 12        |
| <b>3</b> | <b>Related Work</b>                         | <b>14</b> |
| 3.1      | Existing DSMS Benchmarks . . . . .          | 14        |

|          |   |           |
|----------|---|-----------|
| 3.1.1    | Linear Road Benchmark (LRB)                       | 14        |
| 3.1.2    | Yahoo Streaming Benchmark (YSB)                   | 14        |
| 3.1.3    | DSPBench  | 15        |
| 3.1.4    | ESPBench  | 15        |
| 3.2      | Quality-Aware Stream Processing Approaches        | 16        |
| 3.3      | Comparison with Our Approach                      | 17        |
| <b>4</b> | <b>Methodology</b>                                | <b>19</b> |
| 4.1      | Research Design & Approach                        | 19        |
| 4.2      | Selection of Data Quality Dimensions              | 19        |
| 4.3      | Determination of Data Quality Calculation Methods | 20        |
| 4.3.1    | Accuracy Calculation                              | 20        |
| 4.3.2    | Completeness Calculation                          | 22        |
| 4.3.3    | Timeliness Calculation                            | 22        |
| 4.4      | Dataset Selection & Preprocessing                 | 23        |
| 4.4.1    | Original Dataset Overview                         | 23        |
| 4.4.2    | Reason for Selection                              | 24        |
| 4.4.3    | Data Pre-processing                               | 24        |
| 4.4.4    | Selection of Odysseus as the DSMS                 | 26        |
| 4.4.5    | Justification for Using PQL                       | 26        |
| 4.4.6    | Experiment Setup & Performance Analysis           | 28        |
| <b>5</b> | <b>Implementation</b>                             | <b>30</b> |
| 5.1      | Overview of Benchmarking System                   | 30        |
| 5.2      | Bench-Tool CLI Program                            | 30        |
| 5.2.1    | Dataset Preprocessing                             | 32        |
| 5.2.2    | Dataset Analysis and Statistics                   | 34        |

|          |   |           |
|----------|---|-----------|
| 5.2.3    | Data Quality Computation and Verification . . . . .                         | 34        |
| 5.2.4    | Performance Analysis and Comparison . . . . .                               | 35        |
| 5.3      | PQL Implementation . . . . .  | 36        |
| 5.3.1    | Accuracy Measurement Query . . . . .  | 36        |
| 5.3.2    | Completeness Measurement Query . . . . .                                    | 37        |
| 5.3.3    | Timeliness Measurement Query . . . . .                                      | 38        |
| 5.3.4    | Combined Data Quality Query . . . . .                                       | 38        |
| 5.3.5    | Data Stream Input Using the ACCESS Operator . . . . .                       | 39        |
| 5.3.6    | Result Output Using CSVFileSink . . . . .                                   | 40        |
| <b>6</b> | <b>Experiments &amp; Results</b>  | <b>42</b> |
| 6.1      | Experimental Environment . . . . .  | 42        |
| 6.2      | Experiment 1: Cost of Quality Dimensions . . . . .                          | 43        |
| 6.3      | Experiment 2: Effect of Input Rate on Performance . . . . .                 | 45        |
| 6.4      | Experiment 3: Performance Cost of different Quality-Aware Queries . . . . . | 47        |
| 6.5      | Experiment 4: Impact of Parallel Stream Processing . . . . .                | 49        |
| 6.6      | Summary of Findings . . . . .   | 51        |
| <b>7</b> | <b>Conclusion &amp; Future Work</b>   | <b>53</b> |
| 7.1      | Summary of Contributions . . . . .  | 53        |
| 7.2      | Limitations . . . . .   | 53        |
| 7.3      | Future Research Directions . . . . .  | 54        |
|          | <b>References</b>   | <b>56</b> |

## List of Figures

|    |  |    |
|----|--|----|
| 1  | Architecture of Data Stream Management System (DSMS)[Sad19]                        | 6  |
| 2  | Key Features of Odysseus <sup>1</sup>  | 10 |
| 3  | An architectural overview of Odysseus <sup>2</sup>                                 | 11 |
| 4  | Visualization of how MAD-based approach is used to detect inaccurate values[BOD22] | 12 |
| 5  | Flowchart of the research approach.  | 20 |
| 6  | Pre-processing steps to prepare test dataset                                       | 24 |
| 7  | The User Interface of Odysseus Studio <sup>3</sup>                                 | 27 |
| 8  | Architecture of the Benchmarking Framework   | 31 |
| 9  | Structure of the groups and commands of Bench-Tool                                 | 31 |
| 10 | Example of the dataset file before and after preprocessing                         | 32 |
| 11 | Example output of <i>show-stats</i> command  | 34 |
| 12 | Sample output of <i>verify</i> command   | 35 |
| 13 | Operator graph of the Accuracy measurement query.                                  | 36 |
| 14 | Operator graph of the Completeness measurement query.                              | 37 |
| 15 | Operator graph of the Timeliness measurement query.                                | 38 |
| 16 | Operator graph of the Timeliness measurement query.                                | 39 |
| 17 | Effect of Outliers on Latency and Throughput                                       | 44 |
| 18 | Effect of Missing Values on Latency and Throughput                                 | 44 |
| 19 | Effect of Outdated Tuples on Latency and Throughput                                | 45 |
| 20 | Performance of individual sensors under varying ingestion rates                    | 46 |
| 21 | Latency vs. Ingestion Rate across Different Memory Allocations                     | 47 |
| 22 | Average Latency and Throughput for Individual and Combined Quality-Aware Queries   | 49 |
| 23 | Latency Comparison Across Windows for Each DQ Query                                | 49 |

|    |   |    |
|----|---|----|
| 24 | Latency and throughput of sensor_6127 under increasing number of concurrent streams . . . . . | 50 |
| 25 | Window-level latency trend of three parallel queries . . . . .                                | 51 |
| 26 | Initial latency spikes across multiple windows for different sensor query executions          | 52 |

## List of Tables

|   |   |    |
|---|---|----|
| 1 | Comparison between DSMS and traditional Database Management System (DBMS) . . . . . | 6  |
| 2 | Summary of some notable existing Data Stream Processing Benchmarks . . . .          | 16 |
| 3 | Overview of Selected Sensors for the Benchmark . . . . .                            | 42 |
| 4 | Average Latency (ms) for Different Ingestion Rates and Heap Sizes . . . . .         | 47 |

## Listings

|   |  |    |
|---|--|----|
| 1 | Example YAML Configuration . . . . .                     | 33 |
| 2 | MAD-based Accuracy Computation . . . . .                 | 36 |
| 3 | Pseudocode of Completeness Computation . . . . .         | 37 |
| 4 | Pseudocode of Timeliness Computation . . . . .           | 38 |
| 5 | PQL Source Configuration Using ACCESS Operator . . . . . | 39 |
| 6 | PQL Output Configuration Using CSVFileSink . . . . .     | 40 |



## Abbreviations

|             |                                 |
|-------------|---------------------------------|
| <b>AWS</b>  | Amazon Web Service              |
| <b>CEP</b>  | Complex Event Processing        |
| <b>CLI</b>  | Command Line Interface          |
| <b>CQL</b>  | Continuous Query Language       |
| <b>CQs</b>  | Continuous Queries              |
| <b>CSV</b>  | Comma-Separated Values          |
| <b>DBMS</b> | Database Management System      |
| <b>DQ</b>   | Data Quality                    |
| <b>DSMS</b> | Data Stream Management System   |
| <b>GMMs</b> | Gaussian Mixture Models         |
| <b>IoT</b>  | Internet of Things              |
| <b>LRB</b>  | Linear Road Benchmark           |
| <b>MAD</b>  | Mean Absolute Deviation         |
| <b>ML</b>   | Machine Learning                |
| <b>OSGi</b> | Open Service Gateway Initiative |
| <b>PQL</b>  | Procedural Query Language       |
| <b>QPM</b>  | Quality Propagation Model       |
| <b>YSB</b>  | Yahoo Streaming Benchmark       |

# 1 Introduction

## 1.1 Background and Context

Data streams are continuously generated information flows that occur in real-time and typically at high velocity. Unlike conventional data stored in static databases, this type of data does not sit still, it keeps arriving, and to make use of it effectively, it needs to be processed on the fly as it comes in. Modern innovations, including Internet of Things (IoT) enabled devices, environmental sensing tools, financial transaction systems, and social media platforms, generate massive amounts of data streams every second[SS17]. Effectively handling and understanding ongoing streams of data is crucial for supporting timely and informed decisions across many practical applications. Whether it is monitoring a patient's health in a smart healthcare system, tracking environmental changes, performing predictive maintenance on machinery, or analyzing financial trends as they occur quickly and efficiently makes all the difference[ZDL<sup>+</sup>12]. As data streams become more diverse and complex, making sure their quality is maintained has become more important than ever. When data quality is lacking, it becomes difficult to trust the outcomes or make confident, well-informed decisions.

To effectively manage and process these continuous, high-velocity data streams, specialized systems known as Data Stream Management System (DSMS) have emerged. Unlike traditional relational databases, DSMS are designed to handle live or nearly live data analysis without the need for prior data storage. This continuous processing capability makes DSMS particularly valuable for applications requiring instant analysis, immediate decision-making, and timely responses. DSMS efficiently handles multiple queries concurrently, leveraging incremental data processing techniques, window-based computations, and sophisticated stream operators. Prominent DSMS platforms, like Odysseus[ody], Apache Flink[fli], Spark Streaming[spa], and Storm[sto], exemplify diverse approaches to address the challenges posed by streaming environments.

## 1.2 Motivation

Live processing of data streams is gaining significance across various industries, including healthcare, financial services, smart urban infrastructure, automated manufacturing, and Internet of Things IoT ecosystems. In these areas, decisions often depend on having data that is not only fast but also accurate and complete, otherwise, the insights just are not reliable.[AJS18]. Simultaneously, the ever-changing and uninterrupted nature of data streams introduces unique challenges, particularly in maintaining data quality standards. These streams are often affected by problems such as inaccurate data, delays, missing entries, and inconsistencies [PSAJ14]. Such problems may greatly affect how well dependent applications function and how consis-

tently they operate. While studies have been carried out on benchmarking the performance of Data Stream Management Systems, a majority of existing benchmarks tend to focus on factors such as throughput, latency, scalability, and fault tolerance, often overlooking data quality as a core evaluation metric. Consequently, there is a notable gap in comprehending how factors like accuracy, completeness, and timeliness—key dimensions of data quality—impact the performance of DSMS. This work seeks to bridge this gap by presenting a structured model for evaluating the performance of DSMS in quality-aware stream processing. The goal is to support more informed decision-making when it comes to balancing system performance with data reliability.

### 1.3 Problem Statement

Despite the growing importance of data quality in stream processing environments, existing benchmarks primarily measure traditional DSMS performance metrics such as throughput, latency, scalability, and fault tolerance. However, they do not adequately address how the explicit computation of data quality dimensions affects overall system performance. There is currently a significant gap in benchmarking DSMS performance when it comes to executing quality-aware queries. Because of this, researchers and developers often do not have enough information to understand the trade-offs between keeping data quality high and system performance. This thesis tries to fill that gap by creating a clear and structured benchmark that looks at how including quality checks in streaming tasks affects system performance.

### 1.4 Research Question

To thoroughly explore the research problem, the following research questions guide this study:

**Research Question 1** *What are the benchmarking practices used for DSMS?*

This question focuses on identifying how DSMS are currently evaluated, including the common techniques, tools, and performance indicators used in different domains.

**Research Question 2** *How can they be adapted for data quality processing purposes?*

This involves determining how traditional benchmarking frameworks can be improved or adapted to explicitly incorporate the evaluation of key data quality aspects, including accuracy, completeness, and timeliness.

**Research Question 3** *How do we create benchmarks for evaluating DSMS performance in data quality processing tasks?*

This question is about designing a clear and structured benchmark that helps evaluate how quality-aware queries affect the performance of DSMS. The goal is to make sure the evaluation is both relevant to real-world use cases and follows a solid, reliable method.

## 1.5 Research Methodology Overview

This research takes a structured and step-by-step approach to build a benchmarking framework that focuses on data quality in DSMS. As a starting point, existing benchmarking methods were reviewed to understand the common practices, tools, and performance metrics used to evaluate these systems. Based on insights from this analysis, I chose key data quality aspects (such as accuracy, completeness, and timeliness) that are vital for efficient real-time stream processing. Subsequently, I reviewed different computational methods and identified suitable statistical techniques to accurately compute these dimensions within streaming environments. A realistic dataset was then selected and preprocessed to represent practical streaming scenarios by introducing intentional inaccuracies, missing values, and simulated delays. Following dataset preparation, I developed and executed specialized PQL queries in the Odysseus DSMS to compute the chosen quality dimensions in real-time. The query outputs were verified using a reference Python implementation to ensure correctness. Finally, experiments were carried out, measuring performance metrics like latency and throughput, and analyzing the results to assess how incorporating quality-aware stream processing impacts the performance of DSMS.

## 1.6 Structure of the Thesis

The structure of the rest of this thesis is organized as follows:

- **Chapter 2 (Literature Review)** presents foundational concepts critical to this research, including Data Stream Management System (DSMS), data quality dimensions, query languages, and an overview of the Odysseus platform.
- **Chapter 3 (Related Work)** explores existing benchmarks for DSMS, reviews prior studies related to data quality in streaming environments, and identifies gaps addressed by this research.
- **Chapter 4 (Methodology)** details the research methods used, including the rationale for selecting the dataset, data preprocessing strategies, and the steps involved in developing the benchmarking framework.

- **Chapter 5 (Implementation)** elaborates on the development of the benchmarking system, including the Bench-Tool Command Line Interface (CLI), Odysseus DSMS setup, Procedural Query Language (PQL) query implementation, and configuration of stream inputs and outputs.
- **Chapter 6 (Experiments and Results)** presents the experimental design, performance evaluation, and analysis of how different data quality metrics, input rates, and system configurations affect latency and throughput.
- **Chapter 7 – (Conclusion and Future Work)** provides a summary of the contributions of this study, addresses its limitations, and suggests potential avenues for future research and improvements to the benchmark.

## 2 Literature Review

### 2.1 Benchmarking

Benchmarking refers to the process of evaluating the performance of systems, applications, or processes by comparing them to predefined standards or specific criteria[DJ03]. It is a valuable tool that helps organizations see how efficient and effective their operations are, pinpoint areas that could use improvement, and make more informed decisions regarding technology investments. In today's fast-changing tech world, benchmarking is crucial because it ensures that systems not only meet the required performance levels and industry standards but also stay competitive in the market.[HH22]

In the field of research, an organized approach to benchmarking involves several crucial steps for evaluating and enhancing performance. Initially, researchers set clear benchmarking objectives and select appropriate alternatives for comparison, which might include leading studies or established methodologies. Next, they determine the relevant metrics and gather necessary data by different experiments [BKZS12]. Finally, the data is analyzed to pinpoint performance gaps and uncover best practices, which can guide strategic improvements[AZF10].

### 2.2 Data Stream Management System

With the rapid advancements in today's digital world, data streams from sensors, IoT devices, social media feeds, financial transactions, etc., are generated continuously[AM23]. This surge makes it crucial to have systems that can process and manage data in real-time. DSMS are designed to handle this continuous flow, enabling real-time applications like fraud detection, network monitoring, smart home automation, and many more [JH20]. As data volume and velocity increases, traditional databases became unable to handle these streams of data, making DSMS crucial for efficient data management [AJS18]. Leading platforms like Apache Flink, Apache Kafka, and StreamSets offer low-latency, high-throughput solutions with features such as Complex Event Processing (CEP) to detect patterns or anomalies in real time[FGD<sup>+</sup>17]. With built-in fault tolerance and scalability, DSMS ensures reliable performance even under variable workloads, making them indispensable for data-driven research and decision-making [KRK<sup>+</sup>18].

Additionally, DSMS are now often paired with machine learning and big data tools, allowing organizations to quickly turn raw data streams into practical, actionable insights [DRK23]. For instance, platforms like Apache Flink[fli] provide native support for stateful stream processing, windowing, and event-time semantics, enabling fine-grained control over data freshness and time-sensitive computations[ABC<sup>+</sup>15]. In contrast to traditional DBMS, which operate on

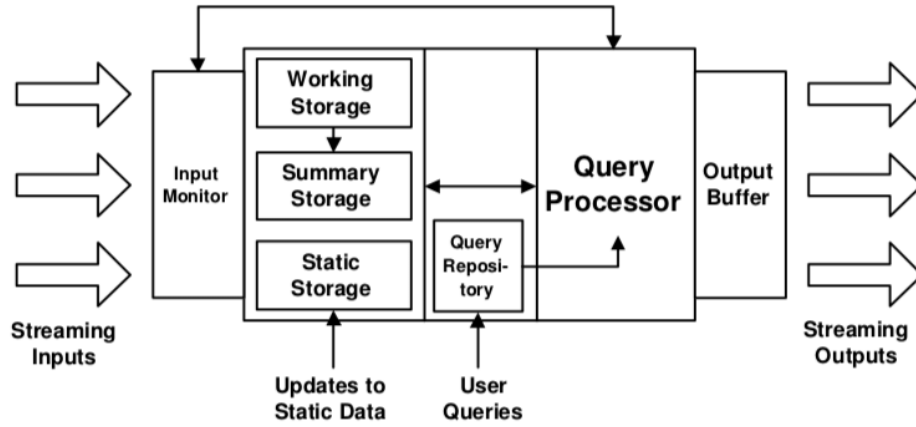


Figure 1: Architecture of DSMS[Sad19]

persisted, finite datasets, DSMS process unbounded data streams using continuous queries that remain active indefinitely[BBD<sup>+</sup>02, ABW03]. Unlike DBMS that rely on pull-based execution, DSMS adopt a push-based model in which data actively triggers computation as it arrives.

This fundamental distinction introduces additional architectural and operational challenges. DSMS require specialized components such as stream ingestion layers, window managers, continuous query schedulers, and state stores that are optimized for low-latency, high-throughput data processing[CKE<sup>+</sup>15, ACL13]. Many modern systems incorporate features like event-time watermarking, fault-tolerant checkpoints, and operator chaining to address latency and correctness under high-volume workloads.

To further clarify the differences between DSMS and traditional DBMS, Table 1 summarizes key architectural and operational contrasts.

Table 1: Comparison between DSMS and traditional DBMS

| Aspect                     | DSMS  | DBMS                                |
|----------------------------|---|-------------------------------------|
| <b>Data Model</b>          | Unbounded, continuous streams                       | Bounded, persisted datasets         |
| <b>Query Type</b>          | Continuous queries (CQs)                            | One-time ad hoc queries             |
| <b>Execution Model</b>     | Push-based (data-driven)                            | Pull-based (query-driven)           |
| <b>Storage</b>             | Typically memory-first, optional persistence        | Disk-based, persistent storage      |
| <b>Latency Sensitivity</b> | Requires millisecond-level response                 | Can tolerate higher latency         |
| <b>Windowing Support</b>   | Native support (sliding, tumbling, session windows) | Not applicable                      |
| <b>Fault Tolerance</b>     | Checkpointing, replay-based recovery                | ACID transactions, write-ahead logs |

When processing continuous data streams, DSMS encounter a number of major challenges that affect their effectiveness and productivity[CJ09]. Unlike traditional DBMS, DSMS processes unbounded, rapidly generated data processed in real time. The execution of Continuous Queries (CQs) can become complicated by variations of stream properties, such as fluctuating data arrival

rates and changes in system resources[FHA10]. Moreover, the need to ensure high throughput with low latency, manage volatile and bursty workloads, and scale efficiently to handle increasing data volumes further increases the complexity. In order to manage these large data ingestion rates without sacrificing performance, a DSMS must balance memory, computation, and I/O resources[Alz23a]. Many platforms like Apache Flink or Odysseus address this by using internal state stores, operator fusion, and adaptive scheduling strategies[CKE<sup>+</sup>15, oO24].

Additionally, managing data streams becomes even more challenging when combining diverse data sources and maintaining data quality during real-time processing[SWWF18]. This is particularly true in heterogeneous environments where sensors, devices, or systems use differing formats or produce asynchronous data. To mitigate the resulting issues in consistency and reliability, some systems now integrate metadata tracking, schema evolution tools, or quality-aware operators for error propagation and correction[HH20b].

## 2.3 Data Quality in Stream Processing

Data quality plays a central role in data management and has a direct impact on how well data-driven decisions are made across various domains. The term Data Quality (DQ) describes how well a dataset meets the needs and expectations of the people or systems using it. It is commonly assessed using several key dimensions—such as accuracy, timeliness, completeness, consistency, and relevance—which help define whether the data is qualified [BRSV15, VPV<sup>+</sup>19].

To meet these standards, organizations often adopt data quality management practices that include data cleansing, validation, and monitoring [ATP24]. These practices are important because what counts as "high quality" can change depending on the context. For example, a financial application may need extremely accurate data updated every second, while a reporting dashboard might tolerate minor delays or occasional gaps [VPV<sup>+</sup>19]. In other words, data quality is not a fixed standard but something that depends on how the data is going to be used.

High-quality data is especially important in real-time applications where decisions must be made instantly. As more companies use Data Stream Management System (DSMS) to handle live data from sensors, user devices, or other real-time sources, keeping data quality high becomes more challenging—but also more critical. Streaming data is constantly moving and often cannot be stored or reviewed before it is processed. This makes it harder to catch errors like missing values, noise, or inconsistencies as they occur [ATP24][GG10].

Take sensor networks, for instance. In systems that monitor things like traffic, weather, or patient health, data comes in continuously and must be cleaned and checked on the fly. Techniques like filtering out noise, filling in missing values, or removing outliers are often used in real time to improve the quality of the data before it is analyzed[AJS18]. These methods also need to be fast and lightweight so they do not slow down the overall system.



Another big challenge in stream processing is combining data from different sources. Each source might use different formats, standards, or update rates, which can lead to inconsistencies or conflicts. Managing this kind of integration requires careful planning and adaptive methods that can adjust to the nature of the incoming data [AJS18]. Some modern stream processing platforms now include built-in tools to help detect and fix these issues automatically, based on the patterns they observe in the data over time.

## 2.4 Data Quality Dimensions

Data Quality Dimensions refer to the key characteristics that determine how well a dataset—or in this context, a data stream—supports meaningful, accurate, and timely decision-making [SPA<sup>+</sup>12]. These dimensions are often used as benchmarks to assess whether data is useful, reliable, and fit for purpose. Commonly recognized dimensions include accuracy, completeness, consistency, timeliness, uniqueness, and validity [PLW02].

In stream processing settings, where data is generated and handled on-the-fly, maintaining these data quality dimensions becomes especially difficult. Data quality management frameworks like the Quality Propagation Model (QPM) are often employed to monitor and maintain these attributes in fast-moving data streams [KL09]. Without them, quality issues can arise rapidly, especially in domains like sensor networks, finance, or health monitoring systems, where the cost of low-quality data can be significant.

### 2.4.1 Accuracy

Accuracy refers to how well data captures the phenomena that occur in the actual world it aims to represent. In the context of streaming data, accuracy is crucial because decisions are made on the fly and often without the chance to retrospectively verify the data [KL09]. In environments such as environmental monitoring or healthcare analytics, sensor readings may be impacted by hardware faults, environmental noise, or temporary signal loss. These issues can distort the truth unless validation mechanisms are in place to filter out erroneous readings. Strategies such as smoothing techniques or anomaly detection algorithms can help detect and correct misleading data points before they propagate through the system [WS96].

### 2.4.2 Completeness

Completeness assesses whether all necessary data is present. In data stream systems, missing values can result from network interruptions, sensor downtime, or processing delays. These gaps are particularly problematic when decisions depend on continuous observation—for example,

in health monitoring or predictive maintenance scenarios. Approaches like imputation (filling missing values) or redundancy through multiple sensor nodes can help improve completeness in real-time settings [KL09, BS06].

### 2.4.3 Consistency

Consistency deals with maintaining coherent data across different time frames and data sources. In streaming systems where updates arrive from multiple channels, inconsistency can arise if similar data points present conflicting values. For instance, if two sensors monitoring the same environment report significantly different temperatures at the same timestamp, the reliability of the analysis may be compromised. Ensuring consistency often involves using synchronization protocols, version control, and data fusion techniques [MCCW09]. Moreover, as noted by Panahy et al., problems in one dimension—such as accuracy—can negatively affect others like consistency, illustrating the interdependent nature of data quality dimensions [PSAJ14].

### 2.4.4 Timeliness

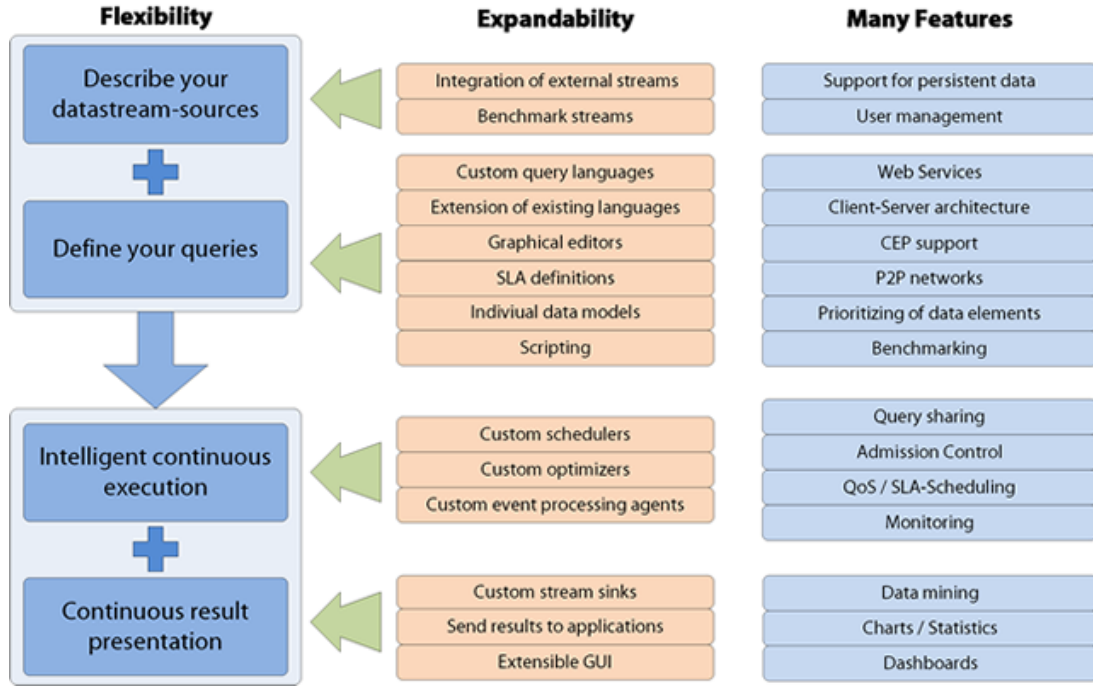
Timeliness indicates how current and available the data is at the time of use. In fast-moving domains like finance, logistics, or medical diagnostics, even slight delays can render otherwise accurate data useless. The value of data can decay rapidly, so stream management systems typically prioritize time-sensitive processing to avoid latency. Techniques such as windowing, prioritization queues, and deadline-aware schedulers are common in improving timeliness in these systems [KL09, BBD<sup>+</sup>02].

### 2.4.5 Uniqueness

Uniqueness refers to the absence of duplicate entries in a dataset. In continuous data streams, duplicates may occur due to retransmissions, system retries, or synchronization issues. Duplicate data not only increases storage and processing costs but can also skew analysis. Deduplication mechanisms often rely on hashing, bloom filters, or fingerprinting methods to efficiently track and eliminate repeated data entries in real time [GMM<sup>+</sup>03, Red96].

### 2.4.6 Validity

Validity checks whether data conforms to predefined rules, formats, or value ranges. For streaming data, these constraints might involve type checks, range limits, or domain-specific conditions. For example, a body temperature reading of 200°C would clearly violate medi-

Figure 2: Key Features of Odysseus<sup>4</sup>

cal validity thresholds. Enforcing validity through lightweight rule-based filters or machine learning-based classification helps maintain overall stream integrity [BH06, PLW02].

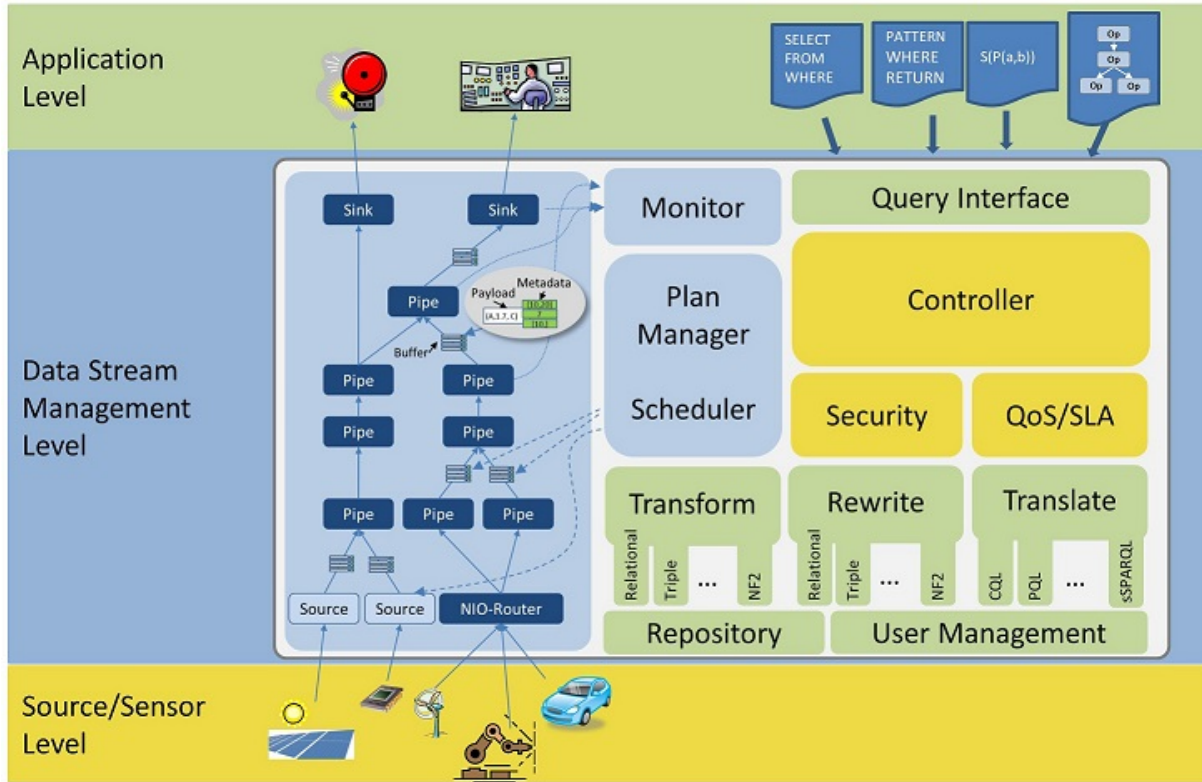
## 2.5 Odysseus DSMS

Odysseus is an advanced Data Stream Management System (DSMS) made to effectively manage, process, and analyze continuous streams of data, especially in settings where data inputs are high in volume and velocity. As part of continuing research at the University of Oldenburg, Odysseus was created to enable a range of applications that require handling of data streams in real-time. This enables decision-makers to immediately extract insights from the data[GG12]. Figure 2 shows the many key features of Odysseus.

One of the defining features of Odysseus is its ability to manage and analyze uncertain data through probabilistic modeling. According to [KN14], the system processes these measurements using operators created especially for this purpose and models such uncertainties using Gaussian Mixture Models (GMMs). By incorporating probabilistic reasoning, Odysseus is better equipped to handle the difficulties of processing streams in real time, leading to more accurate and trustworthy evaluations.

Odysseus’s wide range of operators and metadata support, which facilitate sophisticated data processing workflows, is one of its main advantages. Researchers have the freedom to expand its capabilities for particular use cases by integrating custom operators and information. Because of

<sup>4</sup>Source: <https://odysseus.informatik.uni-oldenburg.de/about/about/>

Figure 3: An architectural overview of Odysseus<sup>5</sup>

this, Odysseus is a perfect platform for scholarly research on real-time decision-making systems, IoT analytics, and stream data quality assessment.

The Herakles project is one noteworthy example of an Odysseus application, using the system to analyze sports information in real time via distributed processing [MBCA15]. Herakles gives coaches access to vital game analytics on mobile devices through the use of OdysseusP2P, a peer-to-peer version of the main system, facilitating quick tactical alterations during live games. Research by Kim et al. presents the integration of Odysseus with Database Management System and distributed file systems, showing how this combination improves transaction management and data handling [KWKS14].

## 2.6 Procedural Query Language (PQL)

The Procedural Query Language (PQL) is an integral component of the Odysseus Data Stream Management System (DSMS), facilitating the specification of complex data processing workflows through a procedural and functional paradigm. Unlike declarative query languages, such as the Continuous Query Language (CQL), PQL enables users to explicitly define the sequence of operations in data stream processing, offering greater control over the execution flow.

PQL's design is particularly advantageous in scenarios requiring the integration of specialized

<sup>5</sup>Source: [https://odysseus.informatik.uni-oldenburg.de/about/odysseus\\_server/](https://odysseus.informatik.uni-oldenburg.de/about/odysseus_server/)

or custom operators. In Odysseus, adding new operators is typically achieved by providing an additional OSGi bundle. The flexibility of PQL allows these custom operators to be seamlessly incorporated into data processing workflows without necessitating modifications to the core parser or existing system components[pql].

This extensibility makes PQL a powerful tool for researchers and practitioners aiming to implement tailored data processing strategies within the Odysseus framework. By allowing precise control over the sequence and combination of operations, PQL facilitates the development of sophisticated data stream applications that can adapt to specific requirements and evolving data landscapes.

## 2.7 Mean Absolute Deviation (MAD)

The Mean Absolute Deviation (MAD) approach is method is commonly employed to assess the accuracy of data streams, especially in IoT and sensor networks[Aji23][BGBC23]. Unlike traditional methods that require a golden ground truth, MAD quantifies accuracy by measuring deviations from the median, making it well-suited for dynamic environments where data is continuously generated [BGBC23]. This adaptability is essential for real-time applications like environmental monitoring and smart agriculture, where sensor readings frequently fluctuate [BJ19].

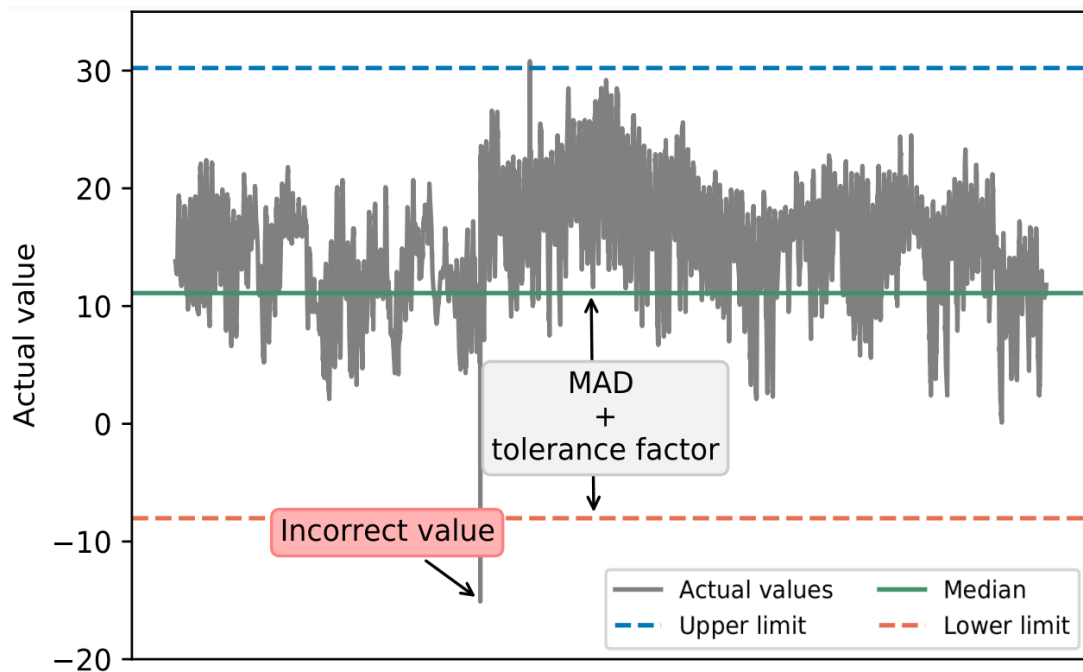


Figure 4: Visualization of how MAD-based approach is used to detect inaccurate values[BOD22]

One of MAD’s key advantages is its robustness against noise, a common issue in sensor data that can distort accuracy if left unfiltered [BJ19]. Figure 4 illustrates how MAD is employed to

identify erroneous values in a data stream by establishing upper and lower thresholds derived from the MAD value and tolerance factor. This method is particularly advantageous as it is resistant to outliers, unlike standard deviation-based approaches. By operating in the same units as the data, MAD simplifies interpretation and real-time decision-making, making it effective in IoT applications that rely on continuous data streams[MHA24]. Additionally, MAD plays a crucial role in anomaly detection and drift identification, helping systems adjust algorithms to maintain data reliability [Fu22]. For instance, integrating MAD within control charts allows for process stability monitoring in sensor-based data collection [MHA24]. MAD's straightforward computation also makes it a useful benchmark for model performance, particularly in machine learning-based streaming systems [Fu22].

## 3 Related Work

Evaluating Data Stream Management System (DSMS) has become increasingly important because of the large volumes of data generated in real time across various applications, including financial monitoring, sensor networks, and IoT frameworks[Alz23b]. Benchmarks are crucial instruments for evaluating these systems' performance, scalability, and reliability. The assessment of DSMS has conventionally focused on performance indicators including resource utilization, latency, and throughput. Various benchmarking frameworks have been developed to evaluate how well streaming engines handle continuous data streams, such as the linear road[ACG<sup>+</sup>04], Yahoo streaming[SC15], and ESPBench[HMP<sup>+</sup>21] benchmarks. Several studies have evaluated the quality of data in stream processing, offering strategies for handling missing data, managing late arrivals, and detecting inaccuracies. This section explores a number of noteworthy stream processing benchmarks, their approaches, and their distinctive contributions to the larger comprehension of DSMS capabilities as well as the approaches of measuring DQ in DSMS.

### 3.1 Existing DSMS Benchmarks

This subsection reviews key benchmarking efforts aimed at evaluating the performance of Data Stream Management Systems (DSMS), highlighting their methodologies, metrics, and areas of application.

#### 3.1.1 Linear Road Benchmark (LRB)

The Linear Road Benchmark (LRB)[ACG<sup>+</sup>04], is a seminal benchmark used to evaluate how well stream processing systems handle real-time traffic data. It simulates a highway tolling scenario by generating continuous streams that emulate vehicle movements across a road network[KW07]. Widely adopted for its realism, LRB captures practical use cases that DSMS are likely to encounter. Its main objective is to quantify a system's capability to maintain correct and timely responses under load, measured by the L-rating—defined as the number of expressways a DSMS can support while meeting predefined latency and accuracy constraints[ACG<sup>+</sup>04]. Additionally, it enables comparative analysis between DSMS platforms and traditional relational database systems, evaluating both real-time and historical query handling while maintaining low-latency responsiveness[Pat22].

#### 3.1.2 Yahoo Streaming Benchmark (YSB)

The Yahoo Streaming Benchmark (Yahoo Streaming Benchmark (YSB))[SC15] was introduced to assess the practical performance of leading DSMS frameworks such as Apache Flink, Apache

Spark, and Apache Storm. Designed around a simulated advertising campaign pipeline, it evaluates a system’s ability to manage real-time event streams. The processing flow includes the operations: Parse  $\rightarrow$  Filter  $\rightarrow$  Project  $\rightarrow$  Aggregate  $\rightarrow$  Store. Performance is typically measured over a 30-minute execution window, focusing on two core metrics—latency and throughput. YSB has evolved over time with various extensions. For instance, Farhat et al.[FDQ21] adapted YSB to evaluate Apache Flink under optimized scheduling techniques, showcasing improvements in throughput and reduced latency. Similarly, Mostafaei et al.[MAA21] extended YSB to a geo-distributed setting, introducing a network-aware analytics framework that enhanced resource placement efficiency and system performance.

### 3.1.3 DSPBench

DSPBench[BGM<sup>+</sup>20] is a comprehensive benchmarking suite specifically designed to evaluate the scalability and performance of distributed DSMS platforms. It addresses limitations in earlier benchmarks by introducing diverse and realistic workloads reflective of real-world applications—such as log processing, network traffic analysis, machine learning pipelines, and word count tasks. DSPBench examines a broad range of performance metrics including latency, throughput, fault tolerance, scalability, and resource utilization. It supports benchmarking across widely used frameworks including Apache Storm, Apache Flink, and Apache Spark Streaming, thereby promoting standardized evaluation across platforms[BGM<sup>+</sup>20].

### 3.1.4 ESPBench

ESPBench[HMP<sup>+</sup>21] is a recent addition tailored for benchmarking enterprise-grade DSMS. Its primary goal is to provide a reliable framework for evaluating enterprise-level stream processing systems in terms of scalability, message processing rate, and latency. ESPBench leverages Apache Beam to define and execute benchmark queries, allowing seamless deployment across a variety of modern DSMS backends such as Apache Flink, Apache Spark, and Hazelcast Jet. This cross-platform compatibility enhances its relevance for industry-grade evaluations where flexibility and system diversity are key considerations.

In addition to the aforementioned benchmarks, several other efforts have contributed to the standardization of DSMS evaluation. StreamBench[LWXH14], for example, focuses on latency and fault tolerance across systems like Apache Spark and Apache Storm under various synthetic workloads. Karimov et al.[KRK<sup>+</sup>18] proposed a benchmarking methodology that emphasizes sustainable throughput and latency under window-based stream operations, carefully separating benchmark control logic from the system under test. Dongen et al.[vDVdP20], in their comparative evaluation, assessed multiple frameworks under varying workload conditions—constant-rate, bursty, and high-throughput—providing valuable insights into their performance in both



Table 2: Summary of some notable existing Data Stream Processing Benchmarks

| Benchmark   | DSMS  | Performance Metric  | Data Stream  |
|---|---|---|--|
| <i>LinearRoad</i>                                       | Aurora,<br>Traditional-RDBMS                    | L-rating, Response time, Accuracy   | Traffic tolling system in a highway                        |
| <i>StreamBench</i>                                      | Apache Spark,<br>Apache Storm                   | Latency, Throughput, Fault tolerance  | Synthetic logs, network monitoring, financial transactions |
| Yahoo Streaming Benchmark                               | Apache Flink,<br>Apache Spark,<br>Apache Storm  | Latency, Throughput   | Advertising campaign simulation                            |
| Benchmarking Distributed Stream Data Processing Systems | Apache Flink,<br>Apache Spark,<br>Apache Storm  | Sustainable Throughput, Event-time latency, Process-time latency                    | Synthetic IoT event-based workload                         |
| Evaluation of Stream Processing Frameworks              | Apache Flink,<br>Apache Spark,<br>Kafka Streams | Stateful Processing, Stateless Processing, Latency, Throughput, Different workloads | Financial transactions, social media analytics             |
| <i>DSPBench</i>   | Apache Storm,<br>Apache Flink,<br>Apache Spark  | Latency, Throughput, Fault tolerance  | Network traffic, financial transactions, sensor data       |
| <i>ESPBench</i>   | -   | Message processing rate, Latency, Scalability                                       | Enterprise IT event logs, business transactions            |

stateful and stateless stream processing scenarios.

### 3.2 Quality-Aware Stream Processing Approaches

Data quality measurement within data streams has emerged as a vital area of research, driven by the increasing reliance on real-time data in various applications, such as environmental monitoring, IoT, and smart city infrastructures [ACSV18]. Ensuring data quality, which includes aspects like accuracy, completeness, consistency, timeliness, and validity, has become essential for developing reliable and efficient data stream processing systems[KL09]. This section synthesizes recent studies that address the challenges and methodologies for measuring data quality in streaming contexts.

Klein's work on "Representing Data Quality for Streaming and Static Data" provides a comprehensive framework for evaluating data quality measures across both static and dynamic environments. The authors recognize that sensor data, widely used in smart environments and industrial automation, often suffers from inaccuracies, missing values, and inconsistencies due to sensor limitations and failures[KL09]. To address this, the paper presents a versatile model that enables the effective propagation and storage of data quality information. The study proposes the use of jumping windows to manage and propagate DQ efficiently in stream processing

systems, reducing the computational burden[KL09]. This allows real-time applications to make better decisions by relying on the trustworthiness of the data.

"Incorporating Quality Aspects in Sensor Data Streams" is another notable work from the same author that further explores the challenges of ensuring data quality in sensor networks, particularly in the context of variable data rates and noise inherent in sensor readings. Focusing on a systematic approach to capturing, propagating, and managing data quality in sensor streams, a quality propagation model (QPM) is developed to track the impact of stream processing operations, such as sampling, aggregation, and filtering on data quality dimensions like accuracy, completeness, and confidence[Kle07].

In "End-to-End Data Quality Assessment Using Trust for Data Shared IoT Deployments", Byeabazeire et al. expands upon the idea of data quality by introducing trustworthiness as a critical factor in evaluating data quality in IoT contexts[BOD22]. This work emphasizes that trust and data quality are interrelated, proposing a model that incorporates trust implications for data accuracy and reliability. The proposed approach incorporates statistical and heuristic methods to assess and propagate data trustworthiness across IoT networks. By integrating trust scores and metadata tracking, the framework enables better decision-making in IoT applications, ensuring reliable and high-quality data streams[BOD22].

Other studies have similarly focused on specific aspects of data quality measurement in data streams. Kuka and Nicklas Kuka utilize Gaussian Mixture Models (GMMs) to model uncertainties in sensor readings, contributing to a probabilistic framework for managing data quality in streaming environments. Their work reveals that integrating probabilistic approaches can significantly improve data quality assessments under uncertainty in sensor data[KN14].

Moreover, Henning and Hasselbring [HH20a] discuss scalable and reliable data aggregation techniques for multi-dimensional sensor streams, underscoring the need for reliable processing architectures that prioritize data quality in real-time analyses. Their findings suggest that as data grows in volume and complexity, robust aggregation methods must be implemented to maintain high data quality levels[HH20a].

### 3.3 Comparison with Our Approach

While numerous benchmarks exist for evaluating DSMS, each focusing on different aspects of performance evaluation. Similarly, a broad range of studies on DQ assessment in data streams concentrate on dimensions like accuracy, completeness, timeliness, and consistency. However, there remains a notable gap at the intersection of these two areas: the performance impact of executing quality-aware queries within stream processing systems has not been sufficiently explored.

Existing DSMS benchmarks typically overlook the computational overhead introduced by real-time data quality assessments. Conversely, data quality research often assumes an idealized system environment, without evaluating how such assessments affect processing efficiency under practical workloads. This separation creates a blind spot in understanding the trade-offs between maintaining high data quality and sustaining optimal system performance.

The approach presented in this work addresses this gap by explicitly analyzing how integrating quality-aware queries influences core DSMS performance. Our benchmarking model systematically evaluates both data quality metrics and traditional performance indicators, providing a unified framework for assessing real-world feasibility. This dual-focus design allows practitioners and researchers to better understand the cost-benefit balance of quality-aware stream processing, ultimately guiding more informed system design decisions for applications where both speed and reliability are critical.

## 4 Methodology

This section outlines the methodology used to develop and evaluate a benchmark for quality-aware stream processing in DSMS. This benchmark aims to assess the impact of quality-aware stream processing queries on system performance, considering key quality dimensions, such as accuracy, completeness, and timeliness. This methodology follows a structured approach, starting from data quality dimension selection, dataset preprocessing, and query design to the experimental setup and evaluation.

### 4.1 Research Design & Approach

The research follows an experimental approach in which various quality-aware queries are executed on a DSMS to analyze their impact on performance metrics such as latency and throughput. The key phases of the methodology are as follows:

- Identify relevant DQ dimensions that will be used in the experiment.
- Choosing mathematical formulations to compute the selected DQ metrics.
- Selecting an appropriate dataset and preparing it for the experiment.
- Implementing PQL queries to compute DQ metrics.
- Developing a Python script to validate DSMS results.
- Running experiments under different workloads.
- Evaluating performance.

### 4.2 Selection of Data Quality Dimensions

Data quality is generally categorized as accuracy, timeliness, completeness, consistency, and validity. Some researchers [JGDW18][ARV22][PBR<sup>+</sup>24] further expanded this classification to include dimensions such as trustworthiness, uniqueness, readability, and usefulness, depending on the specific application and context. However, for the purpose of this experiment, selecting the most relevant data quality (DQ) dimensions is imperative to secure both the feasibility and reliability of the benchmarking process.

To maintain a manageable level of complexity in the experiments while still capturing the essential aspects of data quality in stream processing, this study focused on accuracy, completeness,

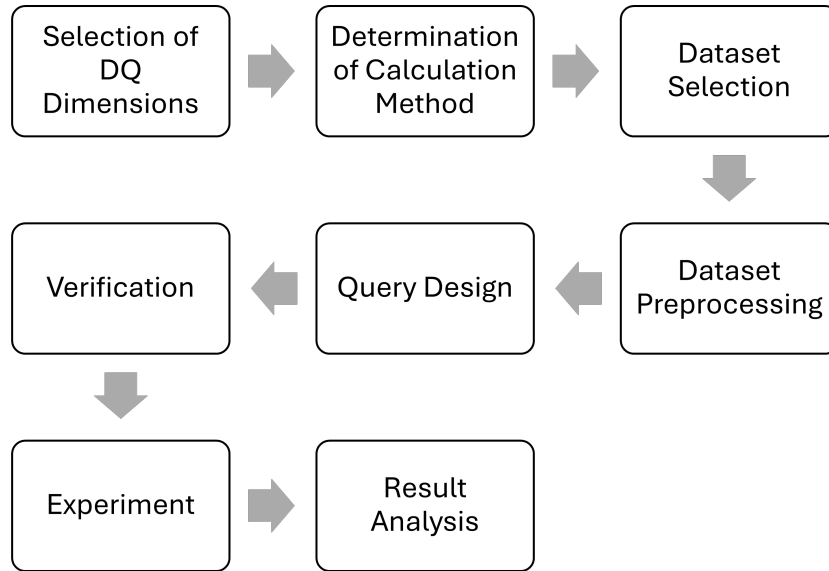


Figure 5: Flowchart of the research approach.

and timeliness. Accuracy and completeness are fundamental to any form of data as they determine the correctness and completeness of the dataset. Additionally, in data stream processing, timeliness plays a critical role, as delayed or outdated data can significantly affect real-time decision-making. By selecting these three key dimensions, this research ensured a balanced assessment of data quality while keeping the experiment structured and practical.

### 4.3 Determination of Data Quality Calculation Methods

To effectively evaluate data quality in a streaming environment, I employ a jumping window approach, which allows me to compute aggregated data quality metrics over fixed intervals. This method ensures that I can continuously monitor changes in data quality as the stream progresses. In this experiment, I focus on computing accuracy, completeness and timeliness, each using an appropriate calculation method that aligns with real-time data constraints.

#### 4.3.1 Accuracy Calculation

For accuracy measurement, I use the Mean Absolute Deviation (MAD)[KK05] approach to determine the number of tuples containing incorrect values ( $V_T$ ). MAD is a well-known statistical method for detecting deviations in data, particularly useful in streaming scenarios where outliers may distort traditional accuracy measures[KK05]. The choice of the MAD approach is motivated by the fact that, in real-world streaming scenarios—such as IoT and sensor data processing—there is often no golden ground truth value available to determine accuracy. Unlike traditional accuracy measures that require a reference dataset, the MAD approach relies on statistical deviation to assess inaccuracies, making it well-suited for scenarios where data values fluctuate dynamically.

The calculation follows this formula[Hub96]:

(1)

where:

- $x_i$  represents the individual data values in the stream,
- $M_j$  refers to the dataset median,
- $\alpha$  is a normalization constant, expressed as:

$$\alpha = \frac{1}{Q(0.75)} \quad (2)$$

where  $Q(0.75)$  is the 0.75 quantile of the dataset distribution[RC93]. This ensures the calculation is resilient to outliers, making it more effective in dynamic stream processing scenarios.

After computing the MAD value, I use it to assess whether a data point is accurate. A data value  $x_i$  is considered inaccurate if its deviation from the median exceeds a predefined threshold based on the MAD:

$$|x_i - M_j| > k \cdot MAD \quad (3)$$

where  $k$  is a threshold parameter that defines the acceptable deviation. Any data point that exceeds this threshold is classified as inaccurate, contributing to the total count of  $V_T$ , which is then used to calculate the accuracy score:

$$Accuracy = 1 - \frac{V_T}{N_A} \quad (4)$$

where:

- $V_T$  is the number of tuples in the stream containing incorrect values,
- $N_A$  represents the total count of tuples within the window.

This method allows for a robust assessment of data accuracy, ensuring that errors in the stream are detected efficiently while minimizing the impact of noise.

### 4.3.2 Completeness Calculation

Completeness is an essential data quality dimension that measures whether all expected data is present in a given stream. Missing or incomplete data can lead to biased or incorrect decisions, particularly in real-time applications. To assess completeness, I classify tuples as either complete or incomplete, depending on the presence of required attributes.

The completeness of a given data stream window is computed as:

$$Completeness = 1 - \frac{N_M}{N_A} \quad (5)$$

where:

- $N_M$  represents the number of tuples with missing or incomplete attributes.
- $N_A$  represents the total count of tuples within the window.

### 4.3.3 Timeliness Calculation

Timeliness is a crucial metric in stream processing since delayed data can significantly impact decision-making. To measure timeliness, I use a two-step approach:

1. **Timeliness Calculation for Each Tuple:** Each tuple's timeliness is determined using the formula:

$$Timeliness_{tuple} = \max \left( 1 - \frac{Currency}{Volatility}, 0 \right) \quad (6)$$

where:

- Currency refers to the delay between data generation and processing[BWPT98].
  - Volatility represents the period for which the data remains valid before becoming outdated[BWPT98].
2. **Timeliness Calculation for the Window:** The overall timeliness for a window is calculated as the average of all tuple timeliness values:

$$Timeliness_{window} = \frac{\sum_{i=1}^N Timeliness_{tuple_i}}{N} \quad (7)$$

where  $N$  represents the total count of tuples within the window.

By applying this method, I ensure that the timeliness metric accurately reflects the real-world challenges of processing streaming data. This approach accounts for both lateness in data arrival

and its validity period, helping to better understand the performance impact of quality-aware queries.

Through these methods, I aim to provide a structured and practical way of assessing accuracy, completeness, and timeliness in stream data processing. These calculations will be integrated into PQL queries and independently validated using a Python-based verification script to ensure consistency and correctness across different test scenarios.

## 4.4 Dataset Selection & Preprocessing

The dataset used for this research was obtained from a publicly available multi-sensor dataset collected in a smart home environment[CAPL21]. This dataset was selected due to its rich variety of sensor data and its real-world applicability in evaluating data quality dimensions in streaming environments. This section offers a brief overview of the original dataset and explain why this dataset is chosen for this research.

### 4.4.1 Original Dataset Overview

The original dataset[CAPL21] comprises sensor readings from 23 different sensors deployed in a residential environment, including motion sensors, pressure sensors, temperature, light sensors and smart plug sensors, contact sensors, and humidity sensors. These sensors were strategically placed in various locations within the home, such as the living room, kitchen, bedroom, bathroom, and corridors, to monitor real-time human activities. The data was continuously recorded at a frequency of 1 Hz, ensuring high temporal granularity[CAPL21]. The key features of the dataset are given below:

- **Total Duration:** Six months of continuous data collection.
- **Sensor Types:**
  - **Motion Sensors (Passive Infrared - PIR):** Detect movement in different rooms.
  - **Pressure Sensors:** Measure interactions with furniture such as beds and couches.
  - **Light Sensors:** Detect changes in ambient light, including appliance usage.
  - **Temperature and Humidity Sensors:** Monitor environmental conditions.
  - **Contact Sensors (Reed Switches):** Track the opening and closing of doors and cabinets.
  - **Smart Plug Sensors:** Measure power consumption of appliances such as microwaves, washing machines, and coffee makers.



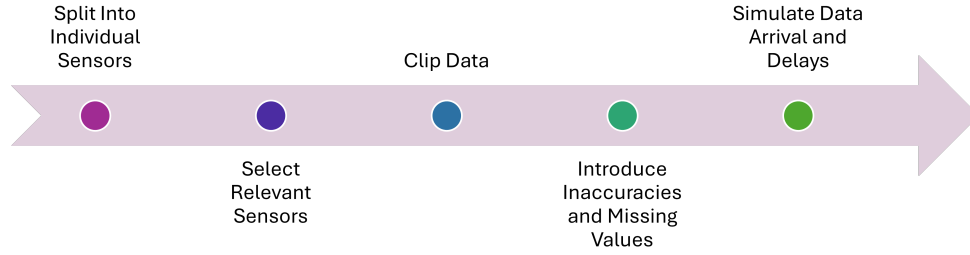


Figure 6: Pre-processing steps to prepare test dataset

- **Data Structure:**

- Each record consists of a **timestamp**, **sensor ID**, **value**, and **sensor name**.
- **Binary sensors** (e.g., motion, contact) report **0** (inactive) or **1** (active).
- **Analog sensors** (e.g., temperature, power consumption) provide continuous values.

#### 4.4.2 Reason for Selection

This dataset was chosen for benchmarking due to its diverse sensor types and real-time streaming characteristics, which align well with the objectives of evaluating data quality dimensions such as accuracy, completeness, and timeliness. Furthermore, the presence of multiple sensor modalities enables an extensive analysis of data inconsistencies, missing values, and real-time processing challenges, which are crucial for quality-aware stream processing. The high-resolution time-series nature of the dataset makes it a suitable candidate for real-time stream processing evaluation, allowing an investigation into how different data quality attributes impact the performance of DSMS.

#### 4.4.3 Data Pre-processing

To ensure the dataset aligns with the requirements of my benchmark, I applied several pre-processing steps to structure the data appropriately for streaming and data quality assessment. These steps were necessary to manage data volume, sensor selection, and controlled quality variations, allowing for a meaningful evaluation of accuracy, completeness, and timeliness. 6 illustrates the transformation of raw sensor data into a structured, benchmark-ready format.

The preprocessing pipeline was implemented in Python due to its extensive ecosystem for data handling and processing. Specifically, I utilized the following tools:

- **Pandas:** This library was used extensively for reading, transforming, and manipulating data. It supports reading large CSV files in chunks, which is particularly useful for handling datasets that exceed available memory[McK10].

- **NumPy**[HMvdW<sup>+</sup>20]: Used alongside Pandas for efficient numerical operations and random sampling, particularly in the introduction of inaccuracies and missing values.
- **Click**: A Python package for creating command-line interfaces. It was used to develop a modular and reusable CLI tool for running different preprocessing steps[Ron].
- **Matplotlib**[Hun07]: Employed to generate visualizations of sensor activity, value distributions, and quality transformations, providing insight into data structure and behavior during preprocessing.

The key preprocessing steps are described below:

**Step 1: Splitting Sensor Data** The dataset originally contained all sensor readings of a particular data type in a single file, making it impractical for independent sensor analysis. To facilitate per-sensor processing, I split the data into individual files for each sensor.

**Step 2: Selecting Relevant Sensors** Given the large number of sensors, only a subset was selected based on value distribution, variability, and type diversity. This step ensured that the benchmark covers a representative range of real-world streaming scenarios while reducing unnecessary computational overhead.

**Step 3: Reducing Dataset Duration** Since the original dataset spans six months, processing the entire dataset would be inefficient. To maintain a practical and representative sample, I selected a 30-day window, ensuring that typical sensor activity patterns are preserved while optimizing computational feasibility.

**Step 4: Introducing Data Inaccuracies and Missing Values** To assess accuracy and completeness, I introduced controlled inaccuracies by modifying selected sensor values, simulating real-world sensor deviations. Additionally, missing values were introduced to reflect sensor failures or data transmission issues, enabling a robust completeness evaluation.

**Step 5: Simulating Data Delays** To evaluate timeliness, I introduced timestamp offsets, simulating real-world latency in data transmission. This step allows for assessing how delayed data impacts DSMS performance and its ability to maintain real-time processing efficiency.

In summary, the dataset provides an ideal testbed for conducting quality-aware stream processing experiments, offering real-world sensor data that closely simulates IoT-based streaming applications. Its structured format, diverse sensor readings, and long-term collection period allow for a comprehensive benchmarking approach, ensuring robust evaluation of data quality dimensions within a streaming context.

#### 4.4.4 Selection of Odysseus as the DSMS

Following dataset selection and preprocessing, the next step was to develop a benchmarking framework capable of evaluating quality-aware stream processing. This required selecting an optimal DSMS. The choice of DSMS significantly impacts the performance, scalability, and flexibility of stream processing tasks. Odysseus was selected as the DSMS for this benchmark due to its modular architecture, extensibility, and strong support for real-time data stream processing. The essential features of Odysseus that make it the most suitable choice for this experiment are[ody]:

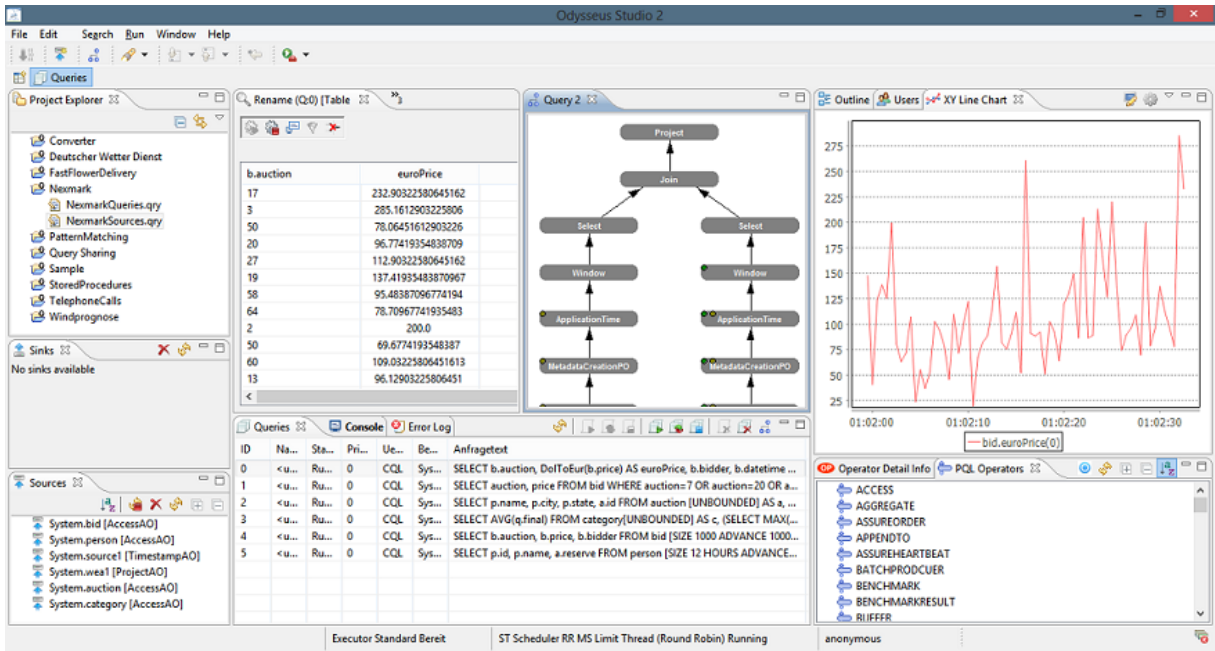
- **Multi-language Support:** Odysseus allows queries to be written in StreamSQL, PQL, CQL, and StreamingSPARQL, providing flexibility in query design. This variety of options enables me to select the most suitable language for the experiment[ody].
- **Custom Operator Integration:** The system supports custom operator development, allowing for the implementation of data quality-aware processing beyond standard streaming operations. Although this study utilizes basic operators to implement queries, this flexibility leaves the door open for the development of robust quality-aware operators in future research based on this benchmark[ody].
- **Built-in Optimization Strategies:** Odysseus includes query optimization techniques that enhance performance, ensuring efficient real-time data handling. These optimizations can further refine and improve the efficiency of current query plans [ody].
- **Client-Server Architecture:** Odysseus consists of two main components:
  - **Odysseus Server:** Manages query execution, stream processing, and data transformation.
  - **Odysseus Studio:** A graphical user interface for query development, debugging, and system administration.
- **Extensibility & Configurability:** Odysseus supports multiple data sources and stream formats, making it adaptable to diverse benchmarking requirements[ody].

#### 4.4.5 Justification for Using PQL

Odysseus supports multiple query languages, including SQL-based (StreamSQL) and graph-based (StreamingSPARQL) languages[ody]. However, for this benchmark, Procedural Query Language (PQL) was chosen due to its procedural nature, differing from declarative languages

---

<sup>6</sup>Source: [https://odysseus.informatik.uni-oldenburg.de/about/odysseus\\_studio/](https://odysseus.informatik.uni-oldenburg.de/about/odysseus_studio/)

Figure 7: The User Interface of Odysseus Studio<sup>6</sup>

like StreamSQL, which allows for step-by-step data transformation—a critical requirement for computing data quality dimensions[pql]. The following advantages of PQL motivated me to use it to develop the queries for this experiment.

- **Procedural Execution:** Unlike declarative SQL-like queries, PQL explicitly defines the sequence of data transformations and computations.
- **Custom Processing Pipelines:** PQL supports complex stream processing logic, making it ideal for computing accuracy (MAD-based deviation), completeness (missing value analysis), and timeliness (latency assessment)
- **Seamless Integration with Odysseus:** Since PQL is natively supported by Odysseus, it can directly utilize built-in streaming operators, ensuring efficient execution.
- **Flexibility:** PQL allows for dynamic adjustments to query logic, making it adaptable to different experimental setups.

In PQL, queries are constructed using chaining operators, each representing a logical processing step. An operator is defined using its name, followed by the parameters and input operators as follows:

OPERATORNAME(parameter, operator, operator, ...)

- **Parameters:** Configure the operator’s behavior [pql].
- **Input Operators:** Specify preceding operators that supply data to the current operator[pql].

Operators can be nested to create complex processing pipelines:

```
OPERATOR1(parameter1, OPERATOR2(parameter2, OPERATOR3(...)))
```

To enhance readability and reusability, PQL allows the assignment of intermediate results to names using the '=' symbol:

```
Result2 = OPERATOR2(parameter2, OPERATOR3(...))
```

These named results can be reused as input for other operators:

```
Result1 = OPERATOR1(parameter1, ...)
Result2 = OPERATOR2(parameter2, Result1)
```

Although intermediate names are local to the query, PQL also supports the definition of global views and sources:

**Views:** Defined using ':=', they are stored in the data dictionary and accessible across different queries.

```
ViewName := OPERATOR2(parameter2, OPERATOR3(...))
```

**Sources:** Defined using '::=', they are also stored globally and can be referenced in various queries.

```
SourceName ::= OPERATOR2(parameter2, OPERATOR3(...))
```

This flexibility in defining intermediate results, views, and sources enhances modularity and maintainability of complex queries.

This benchmark requires precise control over data transformations to accurately compute the quality metrics for each window of streaming data. PQL's procedural execution flow provides this level of control, allowing each quality dimension to be computed incrementally as new data arrives in real-time.

#### 4.4.6 Experiment Setup & Performance Analysis

To evaluate the impact of quality-aware queries on DSMS performance, a structured experiment was conducted. Each preprocessed dataset was streamed individually into Odysseus, and multiple trials were performed at different data arrival rates to assess system performance under varying workloads. The PQL queries for accuracy, completeness, and timeliness were executed on each dataset, and the query outputs were exported as Comma-Separated Values (CSV) files for further analysis.

A Python-based verification script was used to validate the correctness of the query results

by comparing them with independently computed data-quality metrics. Additionally, another Python script measured latency and throughput, capturing the system's processing efficiency. The collected results were analyzed to observe how different query workloads impacted system performance, particularly in terms of latency and throughput.

## 5 Implementation

### 5.1 Overview of Benchmarking System

The benchmarking framework developed for this research is composed of two core components: the custom-developed Python command-line interface (Bench-Tool) and the Odysseus Data Stream Management System (DSMS), which executes procedural queries designed in PQL to measure data quality dimensions. Figure 8 illustrates the complete system architecture and the interaction between its primary components.

The Bench-Tool is a versatile CLI-based program specifically developed to streamline the benchmarking process. It facilitates crucial preprocessing tasks, including splitting large datasets into sensor-specific files, extracting relevant data segments, and deliberately introducing controlled inaccuracies and delays to simulate real-world streaming scenarios. Additionally, Bench-Tool provides functionality for statistical data analysis, manual computation of data quality metrics (accuracy, completeness, timeliness), and verification of query outputs generated by Odysseus. It further supports performance evaluation by calculating critical metrics such as throughput and latency and offering intuitive visual comparisons across multiple experimental runs.

On the other hand, the Odysseus DSMS forms the backbone of the streaming data processing workflow. Chosen for its flexibility, modularity, and robust operator support, Odysseus efficiently executes quality-aware queries implemented through its native Procedural Query Language (PQL). The queries designed for this benchmark evaluate accuracy, completeness, and timeliness of incoming data streams in real-time, either individually or in an integrated manner.

In the subsequent sections of this chapter, each component of this benchmarking system, along with their specific implementations, will be discussed in detail.

### 5.2 Bench-Tool CLI Program

To practically implement the benchmarking workflow, a Python-based command-line tool named Bench-Tool was developed. This tool was designed with a modular command-group structure, each targeting specific tasks within the overall benchmarking procedure. Figure 9 illustrates the structure and commands of the Bench-Tool clearly, highlighting their roles within the benchmarking pipeline.

The Bench-Tool simplifies key experimental steps, making dataset preparation, data-quality verification, and performance analysis straightforward and reproducible. Its primary command groups are:

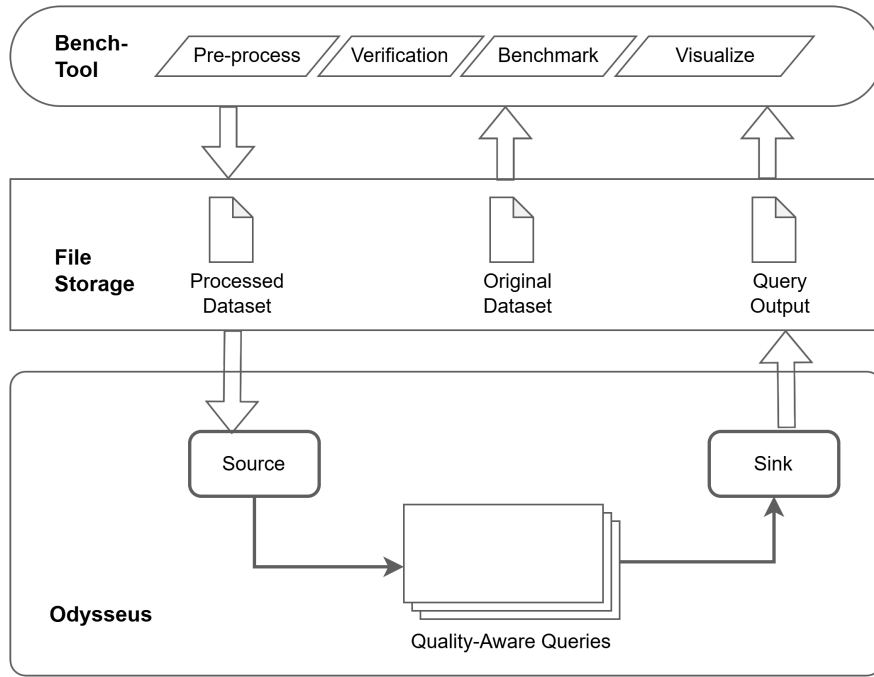


Figure 8: Architecture of the Benchmarking Framework

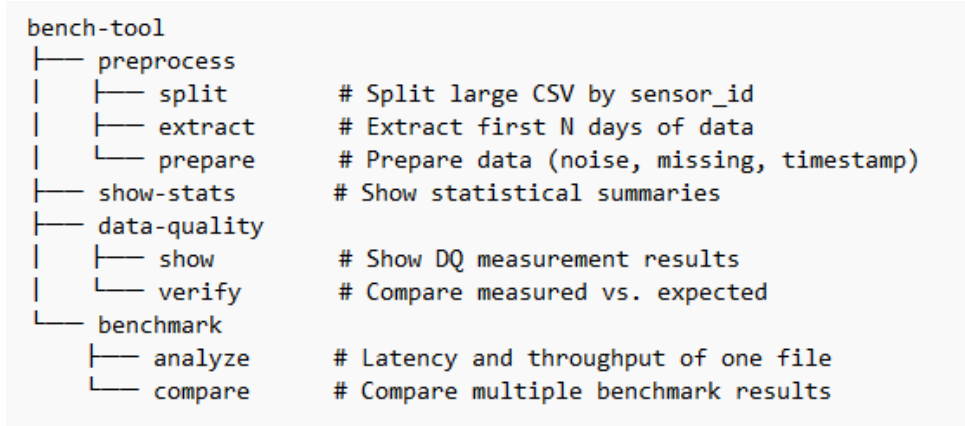


Figure 9: Structure of the groups and commands of Bench-Tool

- **preprocess:** Facilitates initial dataset preparation tasks, including dataset splitting, extraction, quality simulation, and dataset statistics.
- **data-quality:** Handles the manual computation and validation of data quality metrics—accuracy, completeness, and timeliness.
- **benchmark:** Enables performance evaluations by measuring throughput and latency, providing tools for detailed comparative analysis of experiment outcomes.

In the following subsections, each command group and their associated functionalities are discussed in detail.



### 5.2.1 Dataset Preprocessing

The *preprocess* command group of the Bench-Tool handles the essential preprocessing tasks necessary for preparing the dataset for the benchmarking process. It includes three commands: *split*, *extract*, and *prepare*.

The initial raw dataset was provided in a large CSV file containing mixed data from multiple sensors over a six-month period, making direct processing inefficient and memory-intensive. To handle this efficiently, the Bench-Tool implements a chunk-wise reading strategy that reads and processes the file in smaller segments, avoiding memory overflow and ensuring smooth preprocessing even on hardware with limited resources.

The first preprocessing step uses the *split* command, which splits the original multi-sensor CSV file into individual sensor-specific files. This simplifies further processing, analysis, and quality assessments for each sensor independently.

The next step, performed by the *extract* command, extracts a subset of data corresponding to the initial period (for example, the first 30 or 60 days), depending on the requirements of the experimental scenario. This targeted extraction allows efficient experimentation on manageable yet sufficiently large datasets.

The critical step of the preprocessing pipeline is executed using the *prepare* command, designed

|    | A        | B         | C                          | D     |    | A        | B         | C             | D     | E              |
|----|----------|-----------|----------------------------|-------|----|----------|-----------|---------------|-------|----------------|
| 1  | value_id | sensor_id | timestamp                  | value | 1  | value_id | sensor_id | timestamp     | value | available_time |
| 2  | 18730546 | 5896      | 2020-02-26 00:00:00.371814 | 652   | 2  | 18730546 | 5896      | 1582675200371 | 656   | 1582675200521  |
| 3  | 18730564 | 5896      | 2020-02-26 00:00:01.445061 | 652   | 3  | 18730564 | 5896      | 1582675201445 | 663   | 1582675202205  |
| 4  | 18730585 | 5896      | 2020-02-26 00:00:02.527703 | 652   | 4  | 18730585 | 5896      | 1582675202527 | 652   | 1582675203594  |
| 5  | 18730605 | 5896      | 2020-02-26 00:00:03.587275 | 652   | 5  | 18730605 | 5896      | 1582675203587 | 649   | 1582675203925  |
| 6  | 18730625 | 5896      | 2020-02-26 00:00:04.654731 | 652   | 6  | 18730625 | 5896      | 1582675204654 | 638   | 1582675206646  |
| 7  | 18730646 | 5896      | 2020-02-26 00:00:05.714195 | 652   | 7  | 18730646 | 5896      | 1582675205714 | 656   | 1582675209129  |
| 8  | 18730666 | 5896      | 2020-02-26 00:00:06.780699 | 652   | 8  | 18730666 | 5896      | 1582675206780 | 657   | 1582675207007  |
| 9  | 18730685 | 5896      | 2020-02-26 00:00:07.863465 | 652   | 9  | 18730685 | 5896      | 1582675207863 |       | 1582675208951  |
| 10 | 18730704 | 5896      | 2020-02-26 00:00:08.921110 | 652   | 10 | 18730704 | 5896      | 1582675208921 | 642   | 1582675209920  |
| 11 | 18730723 | 5896      | 2020-02-26 00:00:10.011118 | 652   | 11 | 18730723 | 5896      | 1582675210011 | 654   | 1582675210369  |
| 12 | 18730743 | 5896      | 2020-02-26 00:00:11.087082 | 652   | 12 | 18730743 | 5896      | 1582675211087 | 644   | 1582675216490  |
| 13 | 18730763 | 5896      | 2020-02-26 00:00:12.153497 | 652   | 13 | 18730763 | 5896      | 1582675212153 | 647   | 1582675212494  |
| 14 | 18730782 | 5896      | 2020-02-26 00:00:13.216585 | 652   | 14 | 18730782 | 5896      | 1582675213216 | 645   | 1582675213880  |
| 15 | 18730802 | 5896      | 2020-02-26 00:00:14.314472 | 652   | 15 | 18730802 | 5896      | 1582675214314 | 652   | 1582675214317  |
| 16 | 18730859 | 5896      | 2020-02-26 00:00:17.509050 | 652   | 16 | 18730859 | 5896      | 1582675217509 | 648   | 1582675219206  |
| 17 | 18730879 | 5896      | 2020-02-26 00:00:18.578411 | 652   | 17 | 18730879 | 5896      | 1582675218578 |       | 1582675222363  |
| 18 | 18730899 | 5896      | 2020-02-26 00:00:19.640769 | 652   | 18 | 18730899 | 5896      | 1582675219640 | 649   | 1582675220896  |
| 19 | 18730918 | 5896      | 2020-02-26 00:00:20.696765 | 652   | 19 | 18730918 | 5896      | 1582675220696 | 657   | 1582675220844  |
| 20 | 18730956 | 5896      | 2020-02-26 00:00:22.857274 | 652   | 20 | 18730956 | 5896      | 1582675222857 | 639   | 1582675223924  |
| 21 | 18730974 | 5896      | 2020-02-26 00:00:23.925318 | 652   | 21 | 18730974 | 5896      | 1582675223925 | 658   | 1582675224344  |
| 22 | 18730994 | 5896      | 2020-02-26 00:00:24.998579 | 652   | 22 | 18730994 | 5896      | 1582675224998 | 657   | 1582675226754  |
| 23 | 18731014 | 5896      | 2020-02-26 00:00:26.075312 | 652   | 23 | 18731014 | 5896      | 1582675226075 | 659   | 1582675229295  |
| 24 | 18731035 | 5896      | 2020-02-26 00:00:27.143561 | 652   | 24 | 18731035 | 5896      | 1582675227143 | 652   | 1582675228215  |
| 25 | 18731053 | 5896      | 2020-02-26 00:00:28.221351 | 652   | 25 | 18731053 | 5896      | 1582675228221 | 650   | 1582675228720  |

(a) Original dataset

(b) Processed dataset

Figure 10: Example of the dataset file before and after preprocessing

to introduce controlled data quality challenges—such as inaccuracies, missing values, and outdated data, to simulate realistic scenarios encountered in streaming data. The command accepts an optional configuration file (*config.yaml*) that explicitly defines parameters such as:

- Inaccuracy Parameters:
  - deviation: Defines standard deviation of Gaussian noise added to sensor values to simulate small inaccuracies.
  - outlier\_percentage: Percentage of tuples marked as outliers.
  - outlier\_factor: A multiplier to significantly distort the value, positively or negatively, for selected outliers.
- Completeness Parameters:
  - missing\_percentage: Defines the percentage of data tuples for which values are intentionally replaced with empty or missing entries to test completeness handling.
- Timeliness Parameters:
  - volatility: Specifies the random time volatility added to the original timestamps to simulate latency in data availability.
  - outdated\_percentage: Percentage of tuples marked as outdated by adding delays exceeding the defined volatility.

If no configuration file is provided, the Bench-Tool utilizes a set of predefined default parameters. Below is a representative example of the *config.yaml* used to configure these preprocessing parameters:

```

1 DEVIATION: 0.05           # percentage of noise to be added
2 OUTLIER_FACTOR: 2         # factor to determine the outlier threshold
3 OUTLIER_PERCENTAGE: 0.05 # percentage of outliers to be added
4 MISSING_PERCENTAGE: 0.1   # percentage of missing values to be added
5 VOLATILITY: 2000          # time of availability in milliseconds
6 OUTDATED_PERCENTAGE: 0.1  # percentage of outdated values to be added
7 CHUNK_SIZE: 35000        # Number of rows per chunk

```

Listing 1: Example YAML Configuration

The *prepare* command uses randomized selection to introduce these anomalies at varying positions within the data, meaning each execution with identical parameters and input datasets yields different anomaly locations. As a result, while the data quality dimensions (accuracy, completeness, timeliness) remain statistically consistent across executions, the specific positions of anomalies differ, leading to datasets that are not strictly reproducible on a tuple-by-tuple basis. This randomized approach better reflects real-world scenarios, where data quality issues naturally

occur unpredictably, and ensures that the benchmarking experiments accurately capture DSMS performance under realistic and varied conditions. Figure 10 depicts an example of the before and after preprocessing situation of the dataset file.

### 5.2.2 Dataset Analysis and Statistics

After preprocessing, basic statistical analysis was conducted to gain insights into dataset characteristics and confirm its suitability for the benchmarking experiments. The *show-stats* command was developed specifically to obtain key statistical insights, including minimum and maximum values, number of unique values, and total row counts. Figure 11 illustrates an example output, showing a clear distribution of sensor values along with essential statistical information.

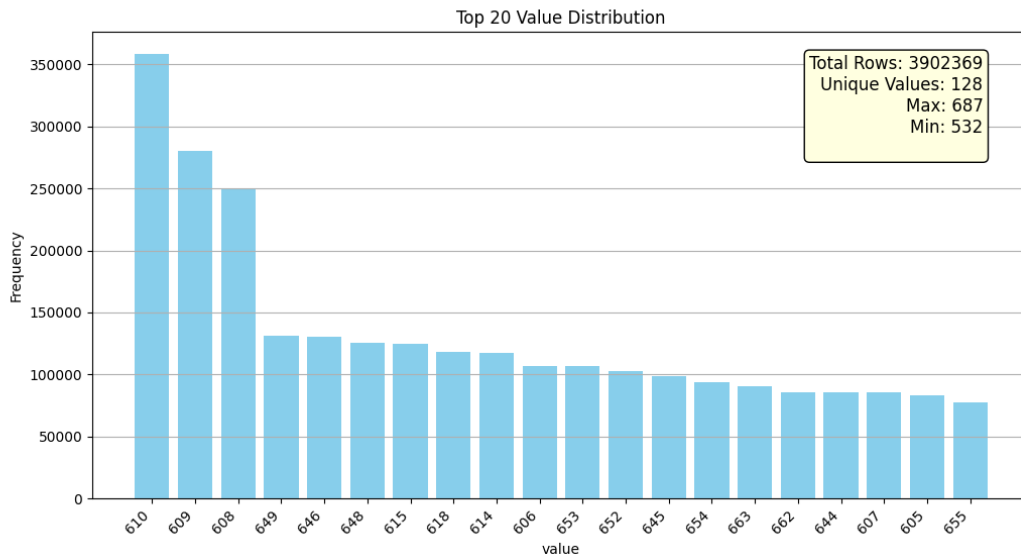


Figure 11: Example output of *show-stats* command

These statistics helped identify suitable sensors for experiments, verify preprocessing effectiveness, and ensure that introduced anomalies realistically reflect real-world streaming scenarios. Such insights were essential for confidently conducting subsequent quality and performance assessments.

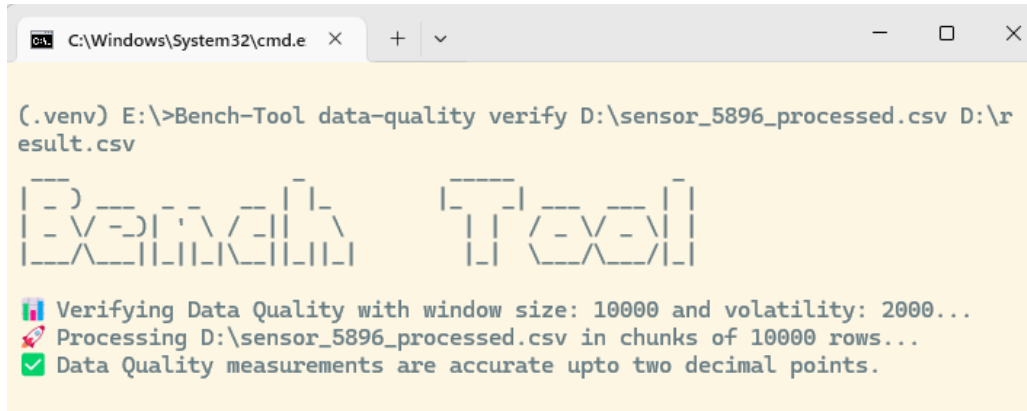
### 5.2.3 Data Quality Computation and Verification

To independently measure data quality dimensions (accuracy, completeness, and timeliness) and verify Odysseus query results, I developed the *data-quality* command group within the Bench-Tool. This command group includes two primary commands: *show* and *verify*.

The *show* command was designed to manually compute data quality metrics directly from the preprocessed dataset without using DSMS queries. It calculates accuracy using the Mean

Absolute Deviation (MAD) method, completeness by counting tuples with missing values, and timeliness by evaluating tuple delays against predefined volatility parameters. This manual calculation served as a baseline for validation.

The *verify* command then compares the output from Odysseus' PQL queries to these manually calculated metrics. This ensured query correctness and reliability, providing confidence in the DSMS's data quality processing capabilities of DSMS. Figure 12 shows a sample output of the *verify* command.



```

C:\Windows\System32\cmd.e X + v
(.venv) E:\>Bench-Tool data-quality verify D:\sensor_5896_processed.csv D:\result.csv

Bench-Tool

Verifying Data Quality with window size: 10000 and volatility: 2000...
Processing D:\sensor_5896_processed.csv in chunks of 10000 rows...
Data Quality measurements are accurate upto two decimal points.

```

Figure 12: Sample output of *verify* command

By integrating this two-step approach, manual computation followed by automated verification, I was able to systematically confirm the accuracy and effectiveness of the quality-aware queries implemented within Odysseus.

#### 5.2.4 Performance Analysis and Comparison

To evaluate how quality-aware queries affect DSMS performance, I implemented the *benchmark* command group in the Bench-Tool, comprising two main commands: *analyze* and *compare*.

The *analyze* command was developed to calculate average latency and throughput. Here, latency is defined as the time span between the arrival of the first tuple in a window and the processing of the last tuple within the same window. Throughput, on the other hand, measures how quickly the system processes each window.

The *compare* command further facilitates detailed analysis by comparing performance metrics across multiple experiments. It provides intuitive visualizations, enabling quick identification of performance variations, trends, and potential bottlenecks across different query workloads or data-quality conditions.

By systematically measuring and visualizing these metrics, I effectively assessed the performance trade-offs introduced by incorporating data quality computations into DSMS processing pipelines.

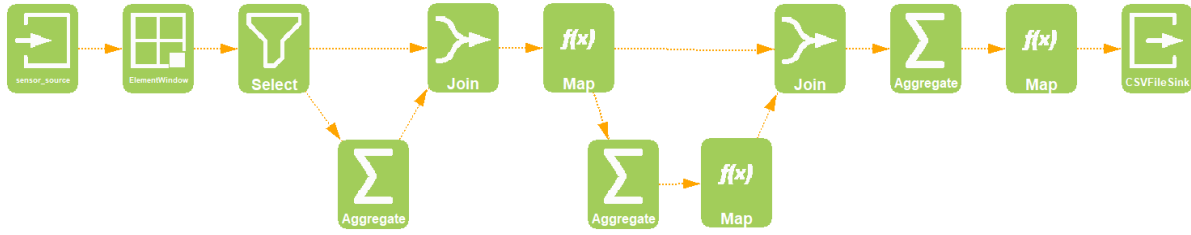


Figure 13: Operator graph of the Accuracy measurement query.

## 5.3 PQL Implementation

In this section, the design and implementation of individual PQL queries tailored to measure each selected data quality dimension independently are described in detail, followed by the implementation of an integrated query that simultaneously assesses accuracy, completeness, and timeliness. Each subsection clearly highlights key operators, query logic, and processing details, demonstrating the practical implementation steps undertaken within Odysseus.

### 5.3.1 Accuracy Measurement Query

The accuracy measurement query was implemented using Odysseus PQL, guided by a statistical approach based on Mean Absolute Deviation (MAD). This method is particularly advantageous as it does not require an explicit ground truth, making it well-suited for real-world IoT and sensor data streams.

Figure 13 illustrates the operator graph generated by Odysseus for this accuracy computation query. The query's logic and execution flow are clearly visualized, showing the sequential and parallel operations performed to achieve the desired accuracy assessment.

The query execution logic is summarized in the following pseudocode steps:

1. Window the incoming stream by size = window\_size
2. Remove all tuples where value IS NULL
3. Compute median of values in the window
4. For each tuple, calculate:  
 $\text{absolute\_deviation} = |\text{value} - \text{median}|$
5. Compute  $\text{MAD} = \text{median}(\text{absolute\_deviation})$
6. Compute  $\text{threshold} = 3 * \text{MAD} * 1.4826$
7. Mark tuples as correct if:  
 $\text{absolute\_deviation} \leq \text{threshold}$

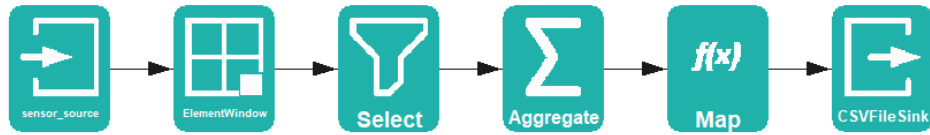


Figure 14: Operator graph of the Completeness measurement query.

```

8. Count number of correct tuples (vt)

9. Calculate accuracy:
    accuracy = 1 - (window_size - vt) / window_size

10. Output accuracy and related metadata to CSV

```

Listing 2: MAD-based Accuracy Computation

The resulting accuracy values, along with relevant metadata (e.g. window timestamps), were exported to a CSV file for subsequent analysis and verification.

### 5.3.2 Completeness Measurement Query

The completeness measurement query was developed using Odysseus PQL to quantify the presence or absence of expected sensor values within data streams. This dimension helps assess how consistently data points are transmitted without missing entries.

Figure 14 depicts the operator graph generated by Odysseus, clearly visualizing each step involved in computing completeness.

The logic of the completeness query can be clearly summarized in the following pseudocode:

```

1. Window the incoming stream by size = window_size

2. Select tuples where value IS NULL

3. Count number of tuples with missing values (vm)

4. Compute completeness using the formula: completeness = 1 - (vm / window_size)

5. Output completeness and relevant metadata to CSV

```

Listing 3: Pseudocode of Completeness Computation

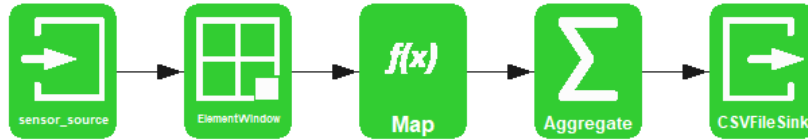


Figure 15: Operator graph of the Timeliness measurement query.

By systematically identifying and quantifying missing values, this query reliably assesses completeness, providing essential insights into data integrity within streaming environments.

### 5.3.3 Timeliness Measurement Query

Timeliness measures how promptly data tuples become available for processing in relation to their actual occurrence. To assess this dimension, a PQL query was implemented within Odysseus to determine the timeliness score of each data window.

Figure 15 shows the operator graph generated by Odysseus, outlining the sequential execution steps for computing timeliness clearly.

The query logic implemented for timeliness is described below:

1. Window the incoming stream by size = window\_size
2. For each tuple, calculate:  $\text{currency} = \text{processing\_time} - \text{event\_time}$   
 $\text{tuple\_timeliness} = \max(1 - (\text{currency} / \text{volatility}), 0)$
3. Calculate average timeliness for the window:  $\text{timeliness} = \text{avg}(\text{tuple\_timeliness})$
4. Output window timeliness and metadata to CSV

Listing 4: Pseudocode of Timeliness Computation

This approach systematically quantifies how quickly tuples are processed relative to their original timestamps, providing clear insight into the responsiveness of the DSMS under quality-aware processing constraints.

### 5.3.4 Combined Data Quality Query

In practical streaming environments, it is often beneficial to measure multiple data quality dimensions simultaneously to reduce redundant computations and efficiently assess the overall

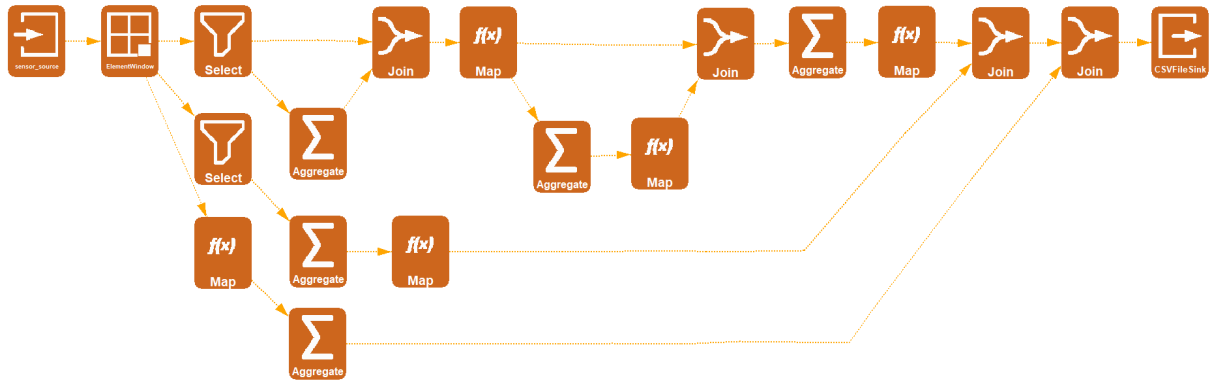


Figure 16: Operator graph of the Timeliness measurement query.

quality of the data stream. To address this, a combined quality query was implemented in Odyssey PQL, simultaneously evaluating accuracy, completeness, and timeliness within the same processing workflow. The operator graph shown in Figure 16 clearly illustrates the comprehensive query implementation.

### 5.3.5 Data Stream Input Using the ACCESS Operator

To read the sensor stream from the preprocessed dataset, I used the ACCESS operator in Odyssey, which is designed to integrate external data sources into the query pipeline. Since my dataset is stored in CSV format, the operator was configured accordingly to simulate a real-time data stream.

For this purpose, I used the GenericPull wrapper, which allows pulling data tuple by tuple, making it ideal for controlled experiments. The transport was set to File, and the protocol was set to CSV to match the dataset format. A slight delay between tuples was introduced using the `delayeach` parameter, which helps simulate different input stream rates for performance testing.

Here is the PQL configuration used to define the source stream:

```

1 sensor_source ::= ACCESS([
2     source = 'sensor_source',
3     wrapper = 'GenericPull',
4     transport = 'File',
5     protocol = 'CSV',
6     datahandler = 'Tuple',
7     options = [
8         ['filename', ${file_path}],
9         ['separator', ','],
10        ['delay', '1'],
11        ['delayeach', 100000],

```



```

12         ['skipFirstLines', '1']
13     ],
14     schema = [
15         ['value_id', 'Integer'],
16         ['sensor_id', 'Integer'],
17         ['timestamp', 'Timestamp'],
18         ['value', 'Double'],
19         ['available_time', 'Timestamp']
20     ]
21 ])
```

Listing 5: PQL Source Configuration Using ACCESS Operator

The `options` section configures file reading behavior, such as the file path, field separator, and whether to skip header lines. The schema defines the structure of the incoming data tuples, which includes the sensor ID, timestamp, original value, and the corresponding available time.

This setup allowed me to stream sensor data in a controlled and repeatable way, which was necessary for running the benchmark experiments with different workloads.

### 5.3.6 Result Output Using CSVFileSink

To store the results of each query for further analysis and verification, I used the `CSVFileSink` operator in *Odysseus*. This operator writes the output stream into a structured CSV file, making it easy to process the results later using Python or other analysis tools.

The sink is configured to include metadata, column headers, and a specific floating-point format to ensure consistency across all experimental outputs. The following snippet shows the PQL definition of the sink:

```

1 out = CSVFILESINK({
2     sink = 'output',
3     filename = ${file_path},
4     writemetadata = true,
5     options = [
6         ['csv.writeheading', 'true'],
7         ['csv.floatingformatter', '#.##'],
8         ['decimalseparator', '.']
9     ]
10 },
11 some_operator)
```

Listing 6: PQL Output Configuration Using CSVFileSink

Here, `filename` sets the path of the output CSV file, and `writemetadata = true` includes timestamp information in the output. The option `csv.writeheading = true` ensures that the first row contains column names, which is particularly useful during post-processing. The floating-point formatter rounds numeric values to two decimal places, and the decimal separator is explicitly set to avoid locale-based inconsistencies.

This output configuration made it convenient to verify the data quality results against manually computed values and allowed for consistent measurement of latency and throughput across experiments.

## 6 Experiments & Results

### 6.1 Experimental Environment

All experiments were performed on a personal computer with the configuration outlined below:

- **Processor:** Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (Base Frequency: 1.80GHz)
- **RAM:** 16 GB
- **Operating System:** Windows 11 Home, Version 24H2
- **Architecture:** 64-bit OS, x64-based processor

The stream processing tasks were executed using the **Odysseus Data Stream Management System**. The system was set up with the monolithic version of Odysseus Studio 2, downloaded on **28 January 2025**, from the official distribution portal<sup>7</sup>. The installation includes both the Odysseus Server and the Odysseus Studio interface required for query execution and visualization.

In addition to Odysseus, a custom command-line tool named **Bench-Tool** was developed in **Python 3.10.7** to support benchmarking tasks such as dataset preprocessing, injection of data quality anomalies, data quality verification, and performance analysis. The source code for the Bench-Tool can be found at github<sup>8</sup>.

The dataset used in this experiment is publicly available and described in detail in Chapter 4, Section 4.4.1, originally published by Makonin et al.<sup>9</sup>. This dataset contains real-world sensor data collected from a smart home environment. For this benchmark, four sensors were selected based on their relevance and variability in data distribution. Each sensor's data was extracted for a duration of 60 days.

Table 3: Overview of Selected Sensors for the Benchmark

| Sensor ID | Sensor Type | Room        | Placement             | Tuples (60 days) |
|-----------|-------------|-------------|-----------------------|------------------|
| 5896      | Pressure    | Bedroom     | On Bed                | 3,902,369        |
| 6127      | Light       | Living Room | Near TV               | 4,978,677        |
| 6223      | Temperature | Bathroom    | Wall-mounted          | 1,089,150        |
| 6896      | Current     | Kitchen     | Plug-in Power Monitor | 3,206,612        |

<sup>7</sup>[https://odysseus.informatik.uni-oldenburg.de/download/products/monolithic/202501/202501280048\\_14218/](https://odysseus.informatik.uni-oldenburg.de/download/products/monolithic/202501/202501280048_14218/)

<sup>8</sup>[https://github.com/sas2505/MS\\_Thesis/tree/main/Codes/bench\\_tool](https://github.com/sas2505/MS_Thesis/tree/main/Codes/bench_tool)

<sup>9</sup><https://www.sciencedirect.com/science/article/pii/S2352340920315122>

The preprocessing steps and configuration parameters used to prepare the dataset for the benchmark experiments are described in Chapter 5, Section 5.2.1. This includes details on how inaccuracies, missing values, and delayed data were introduced to simulate real-world data quality issues in sensor streams.

## 6.2 Experiment 1: Cost of Quality Dimensions

### Objective

The aim of this experiment is to analyze how different types of data quality issues—namely, inaccuracies (outliers), incompleteness (missing values), and untimeliness (outdated tuples)—impact the performance of stream queries in terms of latency and throughput.

### Setup

Three separate sets of experiments were designed to individually assess the impact of each data quality dimension. For each dimension, datasets were preprocessed with increasing levels of quality deterioration from `sensor_6127`. A fixed window size and input rate were maintained across all tests to isolate the impact of each factor. The benchmark tool was used to measure both average latency and throughput across these different conditions.

### Results and Analysis

**Impact of Inaccuracy (Outliers):** The first set of experiments used datasets containing 1%, 5%, 10%, and 45% artificially introduced outliers. The results, shown in Figure 17, indicate that the latency decreases as the number of outliers increases. This may appear counterintuitive, but it is explained by the internal mechanism of the accuracy measurement query. After calculating the threshold using the MAD formula, a `SELECT` operator filters the valid tuples. As the outlier percentage increases, more tuples are discarded early, reducing the amount of data passed on to subsequent operators. As a result, computation is reduced, leading to improved throughput and lower latency.

**Impact of Incompleteness (Missing Values):** The second experiment involved datasets with 1%, 10%, 50%, and 80% missing values. As illustrated in Figure 18, the results demonstrate that higher proportions of missing values significantly increase the average latency. In the completeness query, missing values are filtered after the windowing step. With more missing values present, the `SELECT` operator produces larger intermediate results that need to be further

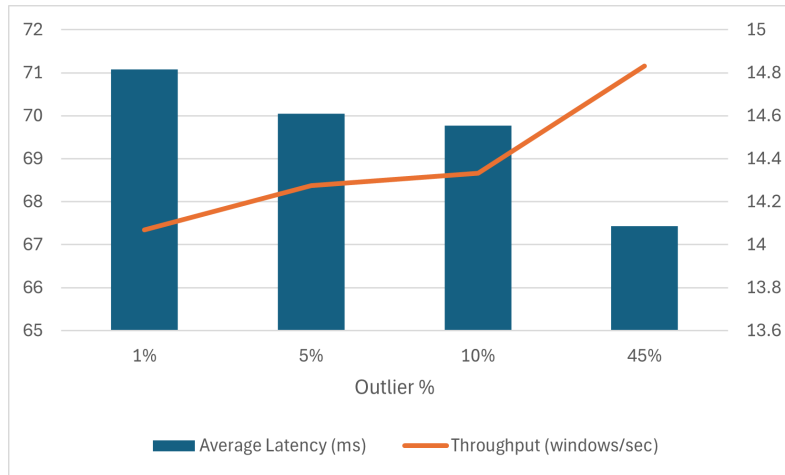


Figure 17: Effect of Outliers on Latency and Throughput

processed. This leads to increased computational overhead and delays in processing, resulting in higher latency and reduced throughput.

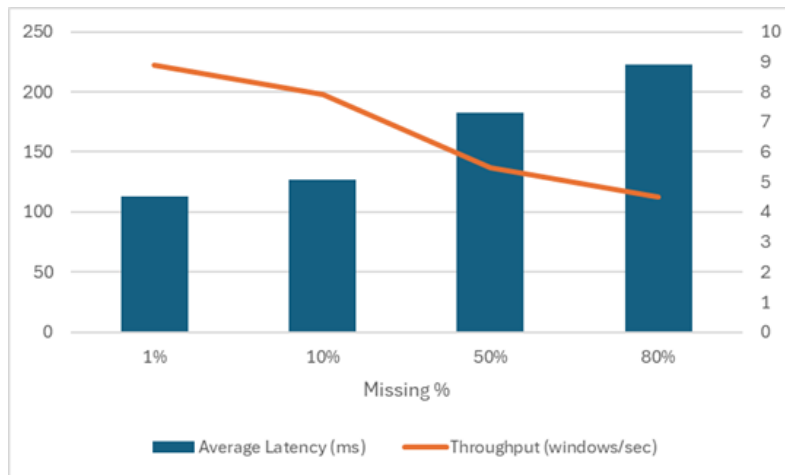


Figure 18: Effect of Missing Values on Latency and Throughput

**Impact of Untimeliness (Outdated Tuples):** The third experiment tested timeliness degradation using datasets with 1%, 10%, 50%, and 80% outdated tuples. The latency and throughput outcomes are plotted in Figure 19. As the percentage of outdated tuples increases, the system requires additional time to compare and validate each tuple's currency. Since timeliness is computed per tuple using the currency and volatility values, processing larger portions of outdated data increases computation time, which is reflected in both higher latency and lower throughput.

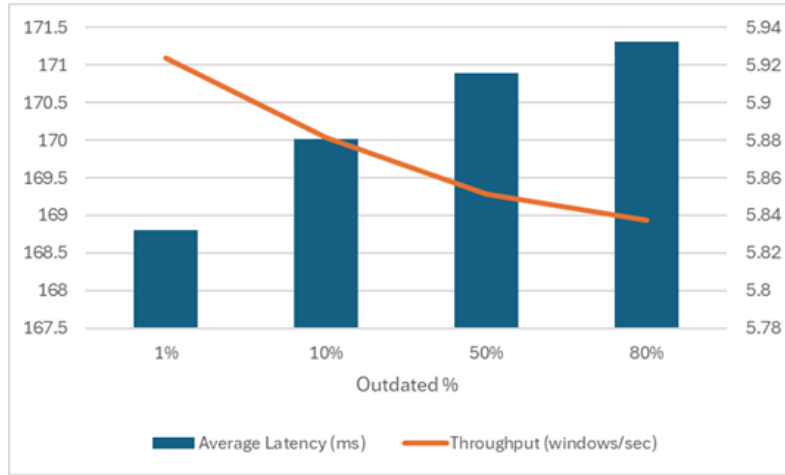


Figure 19: Effect of Outdated Tuples on Latency and Throughput

### 6.3 Experiment 2: Effect of Input Rate on Performance

#### Objective

The objective of this experiment was to investigate how varying the ingestion rate of streaming data affects the query performance in terms of latency and throughput. This helps us understand the responsiveness of the system under different load conditions and identify whether the ingestion rate could become a performance bottleneck.

#### Setup

To conduct this experiment, all four selected sensor streams (`sensor_5896`, `sensor_6127`, `sensor_6223`, and `sensor_6896`) were used. For each sensor, the same data was replayed multiple times with different ingestion rates. Specifically, the ingestion delay (the pause between tuples arriving at the system) was varied to simulate different data arrival scenarios. The rates tested were: 1K, 5K, 10K, 20K, 50K, 80K, 100K, 500K tuples/ms, and one final run with no delay, meaning the entire dataset was made available at once.

#### Results and Analysis

Figure 20 shows the latency and throughput across different ingestion rates for each of the selected sensors.

Across all four sensors, the system exhibited a clear trend—higher ingestion rates led to lower latencies. This can be attributed to the time it takes for a window to be filled: when the delay is higher, the system waits longer for enough data to fill a window, increasing latency. Conversely, with faster ingestion, the windows are populated more quickly, and processing can proceed



Figure 20: Performance of individual sensors under varying ingestion rates

without delays.

Interestingly, the latency decreases substantially up to around 50K/ms, after which the benefits become marginal. This suggests that the DSMS (Odysseus) handles incoming streams through an internal buffer mechanism. Once this buffer is saturated, further increases in ingestion rate do not significantly impact latency.

### Extended Analysis: Impact of Memory Configuration on Ingestion Behavior

To further explore the buffering behavior and investigate potential bottlenecks, we conducted an extended analysis by adjusting the memory allocation available to Odysseus. Specifically, we modified the `Xms` and `Xmx` parameters in the `studio.ini` file to set maximum heap sizes to 100M, 200M, and 300M, and repeated the ingestion rate experiment using the same dataset (sensor\_5896).

The results, summarized in Table 4 and visualized in Figure 21, show that memory plays a key role in ingestion handling. With smaller heap sizes, the buffer gets saturated faster, leading to slightly higher latencies at all ingestion rates. As memory allocation increases, the system handles the same load more efficiently and maintains lower latency.

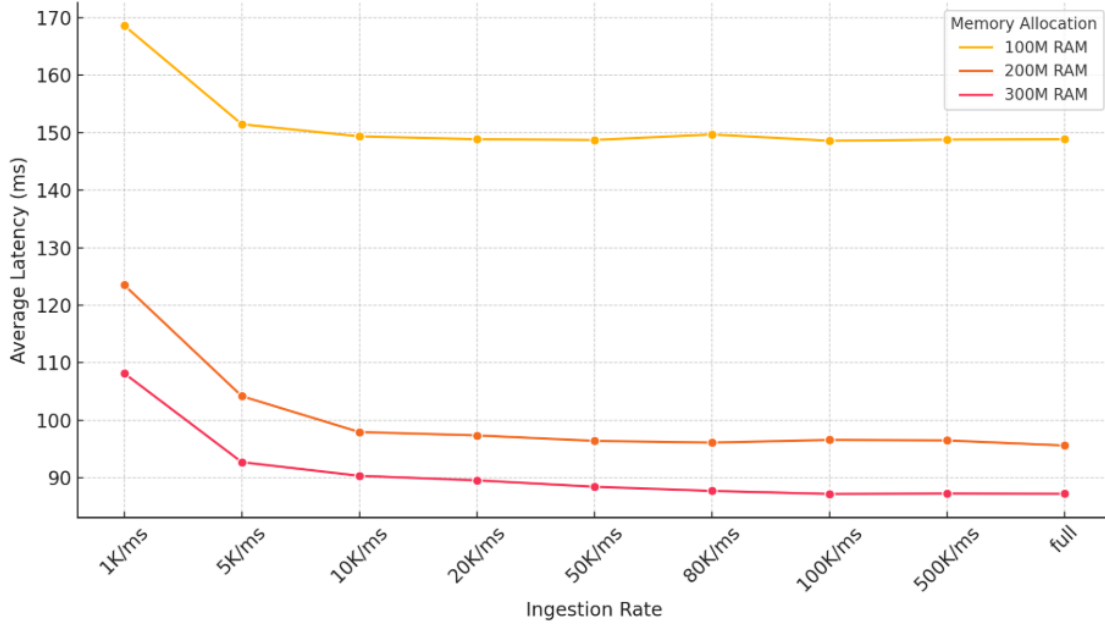


Figure 21: Latency vs. Ingestion Rate across Different Memory Allocations

This behavior further supports the idea that Odysseus utilizes buffering to manage incoming stream loads, helping to avoid bottlenecks. The system intelligently balances ingestion and processing using available memory, ensuring stable throughput beyond a certain rate.

Table 4: Average Latency (ms) for Different Ingestion Rates and Heap Sizes

| Ingestion Rate | 100M RAM | 200M RAM | 300M RAM |
|----------------|----------|----------|----------|
| 1K/ms          | 168.63   | 123.52   | 108.17   |
| 5K/ms          | 151.49   | 104.20   | 92.72    |
| 10K/ms         | 149.37   | 97.96    | 90.35    |
| 20K/ms         | 148.88   | 97.37    | 89.56    |
| 50K/ms         | 148.75   | 96.42    | 88.44    |
| 80K/ms         | 149.70   | 96.12    | 87.72    |
| 100K/ms        | 148.61   | 96.59    | 87.20    |
| 500K/ms        | 148.81   | 96.50    | 87.26    |
| Full           | 148.89   | 95.62    | 87.22    |

## 6.4 Experiment 3: Performance Cost of different Quality-Aware Queries

### Objective

This experiment investigates how the complexity of different quality-aware queries impact the performance of the DSMS when measured individually. The goal is to identify the computational cost associated with each DQ measurement query —accuracy, completeness, and timeliness—when executed separately, and to compare them with the performance of a combined



query that computes all three dimensions simultaneously.

## Setup

In this experiment, separate PQL queries were executed for each DQ dimension: *accuracy*, *completeness*, and *timeliness*. Each query was designed to compute the corresponding metric using the preprocessed data stream with a moderate rate of ingestion. The queries were applied to the same dataset (`sensor_5896`) using a fixed window size and configuration. An additional *combined query* was developed, which incorporates the logic for computing all three DQ metrics within a single query execution.

## Result Analysis

The experiment results confirm that different DQ queries place different computational loads on the DSMS. As shown in Figure 22, the completeness query achieved the lowest average latency and the highest throughput. This is primarily due to the use of a `SELECT` operator that filters out missing tuples early in the pipeline (see Figure 14), effectively reducing the number of records processed in subsequent stages.

The accuracy query, on the other hand, had the highest latency among the individual queries. This is attributed to its more complex structure (see Figure 13), which includes multiple `AGGREGATE`, `MAP`, and `JOIN` operations needed to calculate the MAD and determine correctness of each tuple.

Timeliness falls between the two in terms of complexity and latency. It involves computing currency and volatility per tuple and aggregating their ratios, which is computationally less expensive than accuracy but more involved than completeness.

The combined query, which integrates all three DQ computations into a single stream, exhibited the highest latency and lowest throughput overall. This is expected as it merges the full logic of all three queries, amplifying the operator cost and processing time.

As seen in Figure 23, the latency trends for each query over multiple windows remain consistent. The combined query consistently maintains the highest latency, while completeness shows a relatively stable and low processing time across all windows.

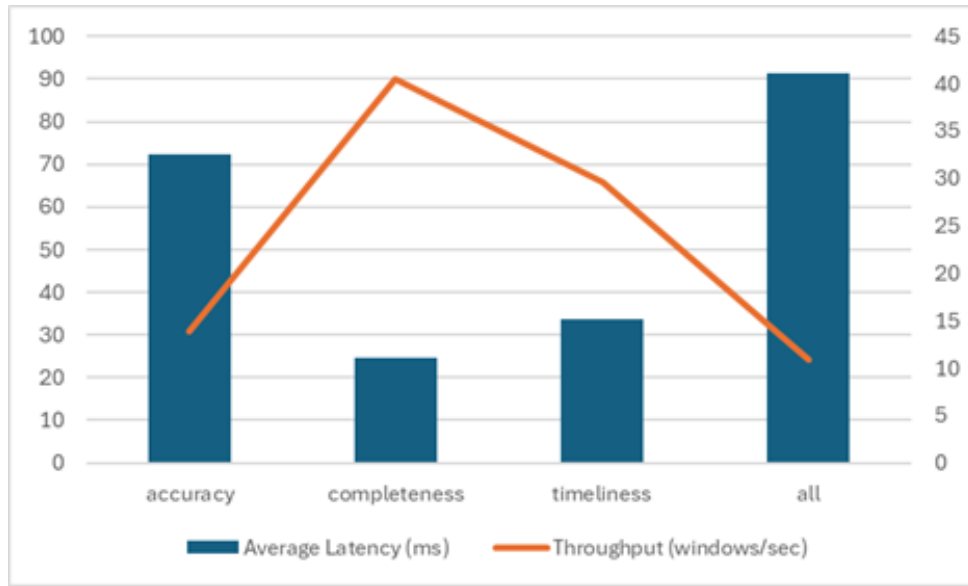


Figure 22: Average Latency and Throughput for Individual and Combined Quality-Aware Queries

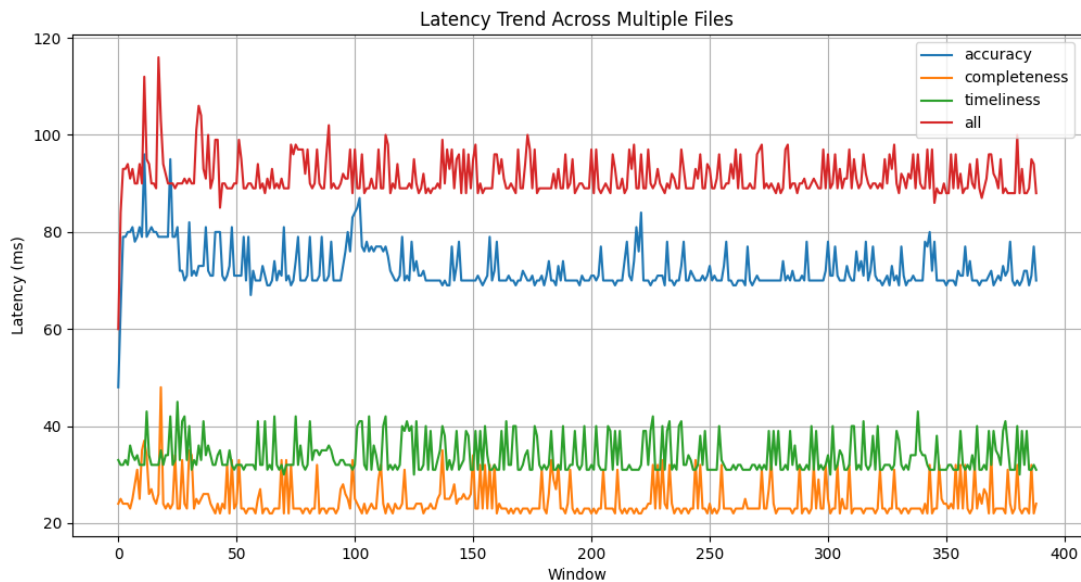


Figure 23: Latency Comparison Across Windows for Each DQ Query

## 6.5 Experiment 4: Impact of Parallel Stream Processing

### Objective

In real-world IoT or sensor-based deployments, stream processing systems rarely handle a single source of input. Instead, they often need to deal with a variety of concurrent streams coming from different sensors or subsystems. Each stream may have independent quality-aware processing requirements. The primary goal of this experiment is to understand how the performance of a DSMS is affected when it processes multiple data streams in parallel.

## Setup

This experiment evaluated performance when running multiple queries in parallel over different streams. The number of active sensor streams varied from 1 to 4. For each run, each stream executed the combined quality-aware query independently. `sensor_6127` was used as the main reference stream, while `sensor_5896`, `sensor_6223`, and `sensor_6896` were added incrementally in different runs.

## Result Analysis

Figure 24 illustrates how the latency and throughput of `sensor_6127` changed as the number of concurrently running streams increased. The results show a clear trend: increasing the number of parallel streams led to higher latency and lower throughput. This is expected, as Odysseus needs to allocate shared system resources (CPU, memory, threads) across all active queries, resulting in longer processing time per window.

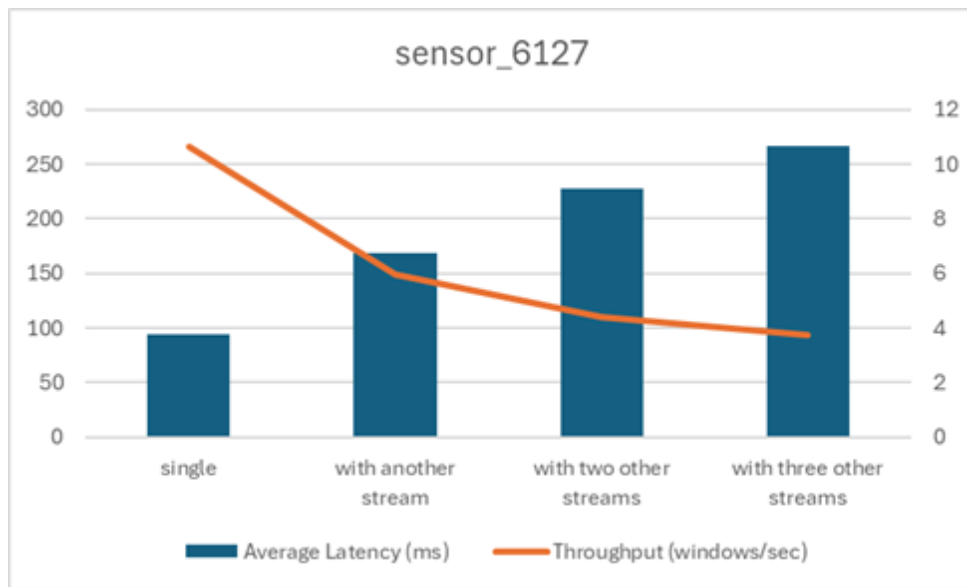


Figure 24: Latency and throughput of `sensor_6127` under increasing number of concurrent streams

To further understand runtime behavior, Figure 25 shows a window-by-window latency trend of three streams running in parallel. Initially, all three streams exhibit similar latency due to shared contention. However, as one or more streams complete execution, remaining streams show reduced latency. This suggests that the system dynamically reclaims and reallocates freed resources, improving performance for the remaining workloads.

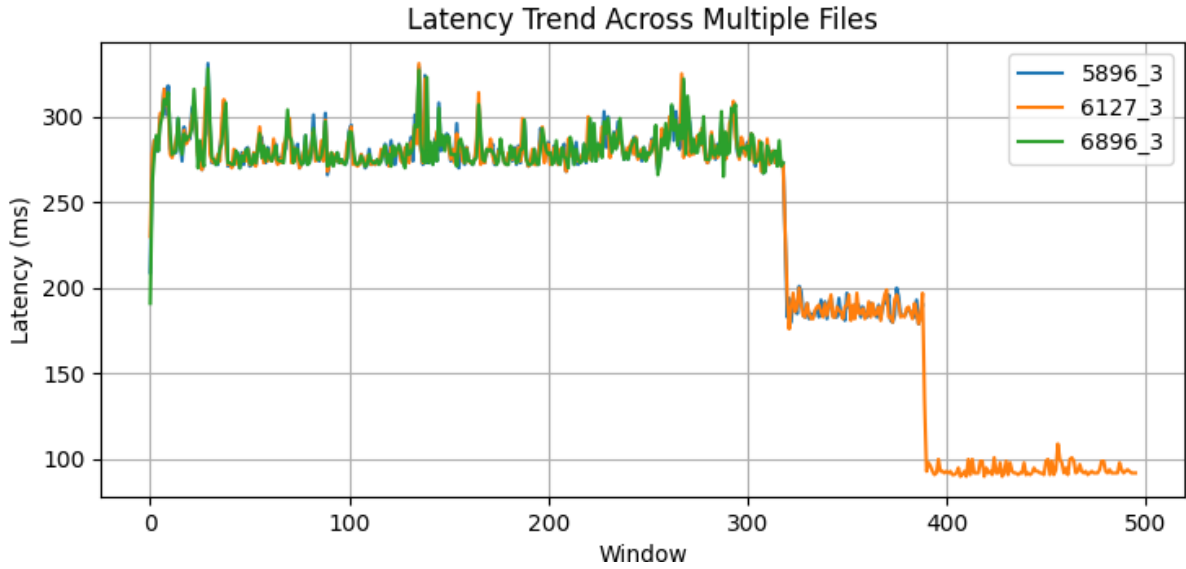


Figure 25: Window-level latency trend of three parallel queries

## 6.6 Summary of Findings

The benchmarking experiments conducted in this study provide important insights into how various factors influence the performance of a DSMS in quality-aware processing environments. The observations below summarize the key outcomes from all four experiments and highlight some general patterns that emerged across setups.

**Effect of Data Quality Dimensions.** The experiments demonstrated that different quality dimensions have varying impacts on system performance. Accuracy measurement resulted in higher latency due to its computational complexity, while completeness queries were the least expensive as they filtered out missing data early. Timeliness processing fell in between, and the combined query, as expected, imposed the highest performance cost. This reveals a clear trade-off: richer quality assessments come at the expense of throughput and latency.

**Effect of Ingestion Rate.** Across all sensors, increasing the data ingestion rate led to reduced latency and increased throughput—up to a certain threshold. Beyond this threshold, the performance stabilized, likely due to internal buffering and memory allocation mechanisms in Odysseus. Additional tests under different memory limits further confirmed that heap size affects how soon the buffer stabilizes. However, Odysseus efficiently avoids performance bottlenecks even with large volumes.

**Effect of Parallel Streams.** When multiple queries were executed in parallel, the system experienced a clear increase in latency. This result is consistent with the expectation that system resources are shared among concurrent tasks. However, visual analysis of latency trends also showed that as one query completed, the system gradually recovered and offered improved performance for the remaining queries.

**General Trend: Initial Latency Spikes.** A common pattern observed across all experiments was a spike in latency for the initial few windows. This could be attributed to system warm-up phases, query initialization time, or internal buffering mechanisms. It suggests that the early-stage performance of DSMS queries may not be indicative of their stable-state behavior and should be treated accordingly in evaluation. This trend is visually represented in Figure 26, which captures the latency profile across multiple windows for three separate result files. The first 20–30 windows clearly show elevated latency, which gradually stabilizes over time.

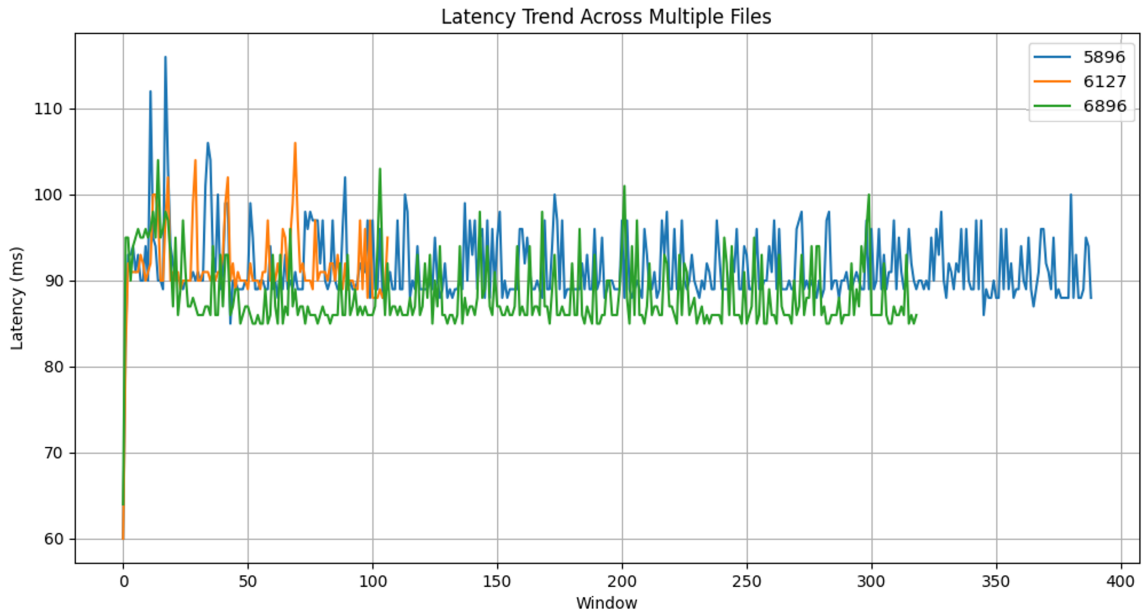


Figure 26: Initial latency spikes across multiple windows for different sensor query executions

**Performance vs. Quality Trade-offs.** From a broader perspective, the experiments reinforce a critical takeaway: increasing data quality computations introduces overhead. Users and developers must balance their need for quality guarantees with the operational limits of the stream processing system. This benchmark provides a foundation for making such decisions in a principled and quantitative way.

## 7 Conclusion & Future Work

### 7.1 Summary of Contributions

This thesis addressed the critical need for performance benchmarking of DSMS under data quality-aware processing tasks. Existing benchmarks largely focus on system-centric metrics such as latency and throughput, often overlooking the additional computational complexity introduced by quality-aware queries. This research fills that gap by proposing a complete benchmarking framework that evaluates the impact of computing data quality dimensions in real-time data stream environments.

The core contributions of this work include the design and implementation of a flexible benchmarking framework for evaluating Data Stream Management System (DSMS) performance under data quality-aware queries. A formal definition of mathematical models for computing key data quality dimensions—*Accuracy*, *Completeness*, and *Timeliness*, has been presented. Furthermore, an end-to-end benchmarking system was developed using the Odysseus DSMS and its Procedural Query Language (PQL), which enabled flexible operator-based query design. To support this, a CLI-based tool named `Bench-Tool` was built to preprocess datasets, simulate data quality issues, execute queries, and measure both system performance and data quality metrics. Several experiments were conducted to evaluate how data quality variations and system parameters (e.g., ingestion rate, window size) affect latency and throughput. Through these contributions, the thesis provides a foundation for future researchers to explore the interplay between data quality and DSMS performance.

### 7.2 Limitations

Despite the completeness of the proposed framework, certain limitations were encountered during the course of the research. First, the accuracy measurement in this benchmark relies on the MAD method. While MAD is robust to outliers, it fails when more than 50% of the data is corrupted, when the dataset contains systematic bias, or when the data is highly random—situations where the median becomes unreliable and deviation values lose meaning. MAD is also inherently unsuitable for boolean or categorical data, which limits the generalizability of the approach.

Another limitation lies in the nature of the dataset used. The benchmark only supports numerical sensor data, and textual or categorical streams were not considered. As a result, data quality dimensions such as consistency or correctness in text attributes were beyond the scope of this work.

The experiments were executed entirely on a personal computer with limited hardware resources. Consequently, performance metrics like latency and throughput might be affected by

uncontrolled variables such as OS-level scheduling, background processes, and garbage collection behavior in the Java Virtual Machine. These factors can cause slight variances in the results and reduce the accuracy of the measurements in a production-grade environment.

Furthermore, although the benchmark leverages the flexibility of PQL and the extensibility of Odysseus, the implementation was restricted to fundamental operators such as MAP, AGGREGATE, and JOIN. Custom user-defined operators, which could provide richer ways to analyze data quality, were not developed or tested. In addition, the benchmark framework does not include adaptive behaviors that react to changing data conditions or system loads, limiting its utility in highly dynamic environments.

### 7.3 Future Research Directions

Building on the foundation laid in this thesis, several research directions can be pursued in the future. One important direction is extending the benchmarking framework to support multiple DSMSs beyond Odysseus, such as Apache Flink or Apache Spark. This would allow for comparative evaluations and could lead to standardization of quality-aware stream benchmarking practices.

Future work should also focus on expanding the benchmark to support heterogeneous data types. Currently limited to numerical values, the framework could be enhanced to include categorical, boolean, or textual data—enabling broader applicability across domains such as social media analysis or customer feedback streams. This would also necessitate redefining some of the quality metrics and processing logic.

Another promising direction involves incorporating more diverse and complex workloads. For example, simulating bursty data arrival patterns, intermittent sensor failure, or interleaved queries can help evaluate the robustness of DSMSs in real-world scenarios. Integration of data fusion and conflict resolution strategies would also be essential for properly assessing the consistency dimension when dealing with multiple data sources.

Machine Learning (ML) techniques could be leveraged to optimize data quality processing. For instance, models could be trained to predict optimal window sizes or adaptive thresholds for detecting inaccuracies. ML-based optimization could also improve query planning and reduce computation overhead dynamically.

Lastly, scaling the benchmark to cloud or distributed computing environments such as Kubernetes or Amazon Web Service (AWS) would be beneficial. This would allow for better resource isolation, elastic scalability, and reproducibility under controlled configurations. A production-grade benchmark deployed in such environments would be a valuable contribution to the research and development of high-performance, quality-aware DSMSs.

## Final Remarks

In conclusion, this thesis contributes to the emerging research space of quality-aware stream processing by proposing a complete benchmarking framework that integrates both system performance metrics and data quality dimensions. The development of the benchmark, its implementation using Odysseus and PQL, and the comprehensive experiments carried out, collectively demonstrate the feasibility and importance of incorporating data quality into the evaluation of DSMS performance. As real-time systems become increasingly complex and scale, the need for such quality-aware benchmarking approaches will only become more critical. It is hoped that this work not only serves as a foundation for further academic research but also inspires practical applications in quality-critical domains such as IoT, smart homes, and sensor networks.



## References

- [ABC<sup>+</sup>15] Tyler Akidau, Alex Bradshaw, Craig Chambers, Slava Chernyak, Raul Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry Nordstrom, Eric Whittle, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.
- [ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford InfoLab, 2003.
- [ACG<sup>+</sup>04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, page 480–491. VLDB Endowment, 2004.
- [ACL13] Tyler Akidau, Slava Chernyak, and Reuven Lax. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.
- [ACSV18] Danilo Ardagna, Cinzia Cappiello, Walter Samá, and Monica Vitali. Context-aware data quality assessment for big data. *Future Generation Computer Systems*, 89:548–562, 2018.
- [Aji23] A. Ajiono. Comparison of three time series forecasting methods on linear regression, exponential smoothing and weighted moving average. *IJIS International Journal of Informatics and Information Systems*, 6(2):89–102, 2023.
- [AJS18] Admirim Aliti, Edmond Jajaga, and Kozeta Sevrani. A need for an integrative security model for semantic stream reasoning systems. *International Journal of Business & Technology*, 2018.
- [Alz23a] A. Alzghoul. Monitoring big data streams using data stream management systems: industrial needs, challenges, and improvements. *Advances in Operations Research*, 2023:1–12, 2023.
- [Alz23b] Ahmad Alzghoul. Monitoring big data streams using data stream management systems: Industrial needs, challenges, and improvements. *Advances in Operations Research*, 2023(1):2596069, 2023.

- [AM23] Hanane Alloui and Youssef Mourdi. Exploring the full potentials of iot for better financial growth and stability: A comprehensive survey. *Sensors*, 23(19):8015, 2023.
- [ARV22] Franco Arolfo, Kevin Rodriguez, and Alejandro Vaisman. Analyzing the quality of twitter data streams. *Information Systems Frontiers*, 02 2022.
- [ATP24] O. Alotaibi, S. Tomy, and E. Pardede. A framework for cleaning streaming data in healthcare: a context and user-supported approach. *Computers*, 13:175, 2024.
- [AZF10] Tutie Asrofah, Suhaiza Zailani, and Yudi Fernando. Best practices for the effectiveness of benchmarking in the indonesian manufacturing companies. *Benchmarking an International Journal*, 2010.
- [BBD<sup>+</sup>02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.
- [BGBC23] S. Bakhshi, P. Ghahramanian, H. Bonab, and F. Can. A broad ensemble learning system for drifting stream classification. *IEEE Access*, 11:89315–89330, 2023.
- [BGM<sup>+</sup>20] M. Bordin, D. Griebler, G. Mencagli, C. Geyer, and L. Fernandes. Dspbench: a suite of benchmark applications for distributed data stream processing systems. *Ieee Access*, 8:222900–222917, 2020.
- [BH06] J. Beringer and E. Hüllermeier. Online clustering of parallel data streams. *Data Amp; Knowledge Engineering*, 58:180–204, 2006.
- [BJ19] I. Bae and U. Ji. Outlier detection and smoothing process for water level data measured by ultrasonic sensor in stream flows. *Water*, 11(5):951, 2019.
- [BKZS12] Can Başaran, Kyoung-Don Kang, Yan Zhou, and Mehmet Hadi Süzer. Adaptive load shedding via fuzzy control in data stream management systems. 2012.
- [BOD22] John Byabazaire, Gregory M.P. O’Hare, and Declan T. Delaney. End-to-end data quality assessment using trust for data shared iot deployments. *IEEE Sensors Journal*, pages 19995–20009, 2022.
- [BRSV15] C. Batini, A. Rula, M. Scannapieco, and G. Viscusi. From data quality to big data quality. *Journal of Database Management*, 26:60–82, 2015.
- [BS06] Carlo Batini and Monica Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

- [BWPT98] Donald Ballou, Richard Wang, Harold Pazer, and Giri Tayi. Modeling information manufacturing systems to determine information product quality. *Management Science*, 44:462–484, 04 1998.
- [CAPL21] Gibson Chimamiwa, Marjan Alirezaie, Federico Pecora, and Amy Loutfi. Multi-sensor dataset of human activities in a smart home environment. *Data in Brief*, 34:106632, 2021.
- [CJ09] Sharma Chakravarthy and Qingchun Jiang. Dsms challenges. In *Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing*, pages 23–31. Springer, 2009.
- [CKE<sup>+</sup>15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.
- [DJ03] R Dattakumar and R Jagadeesh. A review of literature on benchmarking. *Benchmarking: An International Journal*, 10(3):176–209, 2003.
- [DRK23] B. Deepthi, K. Rani, and P. Krishna. Smeras -state management with efficient resource allocation and scheduling in big data stream processing systems. *International Journal of Computer Engineering in Research Trends*, 10:150–154, 2023.
- [FDQ21] O. Farhat, K. Daudjee, and L. Querzoni. Klink: progress-aware scheduling for streaming data systems. pages 485–498, 2021.
- [FGD<sup>+</sup>17] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software*, 127:217–236, 2017.
- [FHA10] F. Farag, M. Hammad, and R. Alhadjj. Adaptive query processing in data stream management systems under limited memory resources. pages 9–16, 2010.
- [fli] flink.apache.org. Apache flink. <https://flink.apache.org/what-is-flink/flink-architecture/>.
- [Fu22] Y. Fu. Mad: self-supervised masked anomaly detection task for multivariate time series. 2022.
- [GG10] João Gama and Mohamed Medhat Gaber. Knowledge discovery from data streams. *Chapman and Hall/CRC*, 2010.

- [GG12] Dennis Geesen and Marco Grawunder. Odysseus as platform to solve grand challenges: Debs grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 359–364, 2012.
- [GMM<sup>+</sup>03] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O’Callaghan. Clustering data streams: theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15:515–528, 2003.
- [HH20a] S. Henning and W. Hasselbring. Scalable and reliable multi-dimensional sensor data aggregation in data streaming architectures. *Data-Enabled Discovery and Applications*, 4, 2020.
- [HH20b] Stefan Henning and Wilhelm Hasselbring. Scalable and reliable data aggregation for multi-dimensional sensor streams. *Journal of Systems and Software*, 169:110703, 2020.
- [HH22] Sören Henning and Wilhelm Hasselbring. A configurable method for benchmarking scalability of cloud-native applications, 2022.
- [HMP<sup>+</sup>21] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, and H. Plattner. Espbench: the enterprise stream processing benchmark. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 201–212, 2021.
- [HMvdW<sup>+</sup>20] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.
- [Hub96] Peter J. Huber. *Robust Statistical Procedures*. Society for Industrial and Applied Mathematics, 1996.
- [Hun07] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
- [JGDW18] Suraj Juddoo, Carlisle George, Penny Duquenoy, and David Windridge. Data governance in the health industry: Investigating data quality dimensions within a big data context. 1, 11 2018.
- [JH20] Johan Jansson and Ismo Hakala. Managing sensor data streams in a smart home application. *International Journal of Sensor Networks*, 2020.
- [KK05] Hiroshi Konno and Tomoyuki Koshizuka. Mean-absolute deviation model. *Iie Transactions*, 37(10):893–900, 2005.
- [KL09] A. Klein and W. Lehner. Representing data quality in sensor data streaming environments. *Journal of Data and Information Quality*, 1:1–28, 2009.

- [Kle07] Anja Klein. Incorporating quality aspects in sensor data streams. In *Proceedings of the ACM First Ph.D. Workshop in CIKM*, page 77–84, New York, NY, USA, 2007. Association for Computing Machinery.
- [KN14] C. Kuka and D. Nicklas. Supporting quality-aware pervasive applications by probabilistic data stream management. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 330–333, 2014.
- [KRK<sup>+</sup>18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. 2018.
- [KW07] Hiroyuki Kitagawa and Yousuke Watanabe. Stream data management based on integration of a stream processing engine and databases. In *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, pages 18–22. IEEE, 2007.
- [KWKS14] J. Kim, K. Whang, H. Kwon, and I. Song. Odysseus/dfs: integration of dbms and distributed file system for transaction processing of big data. 2014.
- [LWXH14] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: towards benchmarking modern distributed stream computing frameworks. *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 69–78, 2014.
- [MAA21] H. Mostafaei, S. Afridi, and J. Abawajy. Snr: network-aware geo-distributed stream analytics. *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 820–827, 2021.
- [MBCA15] T. Michelsen, M. Brand, C. Cordes, and H. Appelrath. Herakles. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 356–359, 2015.
- [MCCW09] K. Mehmood, S. S. Cherfi, and I. Comyn-Wattiau. Data quality through model quality. *Proceedings of the First International Workshop on Model Driven Service Engineering and Data Quality and Security*, pages 29–32, 2009.
- [McK10] Wes McKinney. Data structures for statistical computing in python. *Proceedings of the 9th Python in Science Conference*, 445:51–56, 2010.
- [MHA24] M. Mayashari, E. Herdiani, and A. Anisa. Comparison of control chart x based on median absolute deviation with s. *Barekeng Jurnal Ilmu Matematika Dan Terapan*, 18(2):0737–0750, 2024.
- [ody] About Odysseus — [odysseus.informatik.uni-oldenburg.de/about/about/](https://odysseus.informatik.uni-oldenburg.de/about/about/). [Accessed 10-03-2025].

- [oO24] University of Oldenburg. Odysseus stream management system. <https://odysseus.informatik.uni-oldenburg.de/>, 2024. Accessed May 2025.
- [Pat22] Deepayan Patra. *High-Performance Database Management System Design for Efficient Query Scheduling*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2022.
- [PBR<sup>+</sup>24] Yordan Penev, Timothy Buchanan, Matthew Ruppert, Michelle Liu, Ramin Shekouhi, Ziyuan Guan, Jeremy Balch, Tezcan Ozrazgat Baslanti, Benjamin Shickel, Tyler Loftus, and Azra Bihorac. Electronic health record data quality and performance assessments: A scoping review (preprint). *JMIR Medical Informatics*, 12, 03 2024.
- [PLW02] Leo L Pipino, Yang W Lee, and Richard Y Wang. Data quality assessment. *Communications of the ACM*, 45(4):211–218, 2002.
- [pq] Procedural Query Language (PQL) - Odysseus - Odysseus Wiki — [wiki.odysseus.informatik.uni-oldenburg.de](https://wiki.odysseus.informatik.uni-oldenburg.de/spaces/ODYSSEUS/pages/4587829/Procedural+Query+Language+PQL). <https://wiki.odysseus.informatik.uni-oldenburg.de/spaces/ODYSSEUS/pages/4587829/Procedural+Query+Language+PQL>. [Accessed 08-03-2025].
- [PSAJ14] P. H. S. Panahy, F. Sidi, L. S. Affendey, and M. A. Jabar. The impact of data quality dimensions on business process improvement. *2014 4th World Congress on Information and Communication Technologies (WICT 2014)*, 2014.
- [RC93] Peter J. Rousseeuw and Christophe Croux. Alternatives to the median absolute deviation. *Journal of the American Statistical Association*, 88(424):1273–1283, 1993.
- [Red96] Thomas C Redman. *Data Quality for the Information Age*. Artech House, 1996.
- [Ron] Armin Ronacher. Click documentation. <https://click.palletsprojects.com/>. Accessed: 2025-05-09.
- [Sad19] Kaushik Sadhu. Load shedding via window drop in stream processing systems, 03 2019.
- [SC15] Bobby Evans Reza Farivar Tom Graves Mark Holderbaugh Zhuo Liu Kyle Nussbaum Kishorkumar Patil Boyang Jerry Peng Paul Poulosky Sanket Chintapalli, Derek Dagit. Benchmarking streaming computation engines at yahoo!, 2015. Accessed: 2025-01-09.
- [spa] [spark.apache.org](https://spark.apache.org). Apache spark - a unified engine for large-scale data analytics. <https://spark.apache.org/docs/latest/index.html>.

- [SPA<sup>+</sup>12] Fatimah Sidi, Payam Hassany Shariat Panahy, Lilly Suriani Affendey, Marzanah A Jabar, Hamidah Ibrahim, and Aida Mustapha. Data quality: A survey of data quality dimensions. In *2012 International Conference on Information Retrieval & Knowledge Management*, pages 300–304. IEEE, 2012.
- [SS17] A. Shukla and Y. Simmhan. Benchmarking distributed stream processing platforms for iot applications. *Performance Evaluation and Benchmarking. Traditional - Big Data - Interest of Things*, pages 90–106, 2017.
- [sto] Apache storm. <https://storm.apache.org/index.html>.
- [SWWF18] M. J. Sax, G. Wang, M. Weidlich, and J. Freytag. Streams and tables. *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, 2018.
- [vDVdP20] Giselle van Dongen and Dirk Van den Poel. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1845–1858, 2020.
- [VPV<sup>+</sup>19] Á. Valencia-Parra, L. Parody, Á. J. Varela-Vaca, I. Caballero, and M. T. Gómez-López. Dmn for data quality measurement and assessment. *Lecture Notes in Business Information Processing*, pages 362–374, 2019.
- [WS96] Richard Y Wang and Diane M Strong. Beyond accuracy: What data quality means to data consumers. *Journal of Management Information Systems*, 12(4):5–33, 1996.
- [ZDL<sup>+</sup>12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: a fault-tolerant model for scalable stream processing. 2012.

In accordance with § 9 Para. 12 APO, I hereby declare that I wrote the preceding master's thesis independently and did not use any sources or aids other than those indicated. Furthermore, I declare that the digital version corresponds without exception in content and wording to the printed copy of the master's thesis and that I am aware that this digital version may be subjected to a software-aided, anonymised plagiarism inspection.

Bamberg, 16.05.2025

Md Saiful Ambia Chowdhury