A Framework for Integrating IoT Streaming Data from Multiple Sources

by

Doan Quang Tu
ORCID: https://orcid.org/0000-0002-1399-2585

Master of IT

A thesis submitted in total fulfillment for the
degree of Doctor of Philosophy

in the
Department of Computer Science and Information Technology
School of Engineering and Mathematical Sciences
College of Science, Health and Engineering
La Trobe University Vic, Australia

August 2021

Abstract

The Internet of Things (IoT) has recently attracted considerable interest due to the development of smart technologies in today's interconnected world. With the rapid advancement in Internet technologies and the proliferation of IoT sensors, myriad systems and applications generate data of a massive volume, variety and velocity which traditional databases and systems are unable to manage effectively. Many organizations need to deal with these massive datasets that are of different data types, typically from multiple sources with variations in the types of data sources (e.g., IoT streaming data, static data) and data formats (e.g., structured, semi-structured) from multiple sources. Traditionally, several data integration mechanisms have been designed to process mostly static data. These techniques are not able to deal with IoT streaming datasets from multiple sources effectively. In addition, with the proliferation of data generated by these IoT technologies and sensors, there are emerging challenges in integrating time-series data from multiples sources, and indexing and managing the resulting integrated data for the purpose of optimizing the storage space as well as the information retrieve process. Various researchers have addressed these data integration issues by developing IoT streaming data processing frameworks with specific techniques for data compression techniques and data indexing; however, they have not focused on several crucial issues such as data de-duplication, time alignment and optimised information retrieval.

To address the aforementioned issues, this thesis introduces, in successive developments, a framework to facilitate the integration of IoT streaming data from multiple sources. It makes several important research contributions, including a new windowing technique for streaming data integration, a mechanism to optimise the storage of integrated data with a lossless compression technique for IoT streaming data and an index-based optimisation scheme. **Firstly**, this thesis identifies the challenges of integrating IoT streaming data from multiple sources and presents a formal approach for the real-time integration of IoT streaming datasets, which addresses important issues concerning timing conflict/alignment. A generic window-based framework, called **I**oT **S**treaming **D**ata **I**ntegration (**ISDI**), is then proposed to deal with IoT data in different formats and relevant algorithms are developed to integrate IoT streaming data from multiple sources. In

particular, a basic windowing algorithm is extended for real-time data integration and to deal with the timing alignment issue. A de-duplication algorithm is also introduced to deal with data redundancy and to demonstrate the useful fragments of the integrated data. Several sets of experiments are conducted to quantify the performance of the proposed window-based approach. The local experiment results are compared with a real setup for streaming data, using Apache Spark. The results of the experiments, which are performed on several IoT datasets, show the efficiency of our proposed solution in terms of processing time, and they are used to provide an integrated data view to the users. **Secondly**, as a significant improvement of the first one, a second framework called the IoT data **C**ompression framework (**ISDI-C**) is proposed, in which a lossless compression for floating point time-series data is developed and incorporated. An index based on the timestamp is built for the compressed data. The experiment results on the IoT datasets show a reduction in storage compared with existing compression techniques. The experimental study also demonstrates the capability of time-series data indexing and integration in real time. **Thirdly**, a variety of queries from multiple IoT scenarios is identified as a motivation basis for query optimization and an **I**ndexing framework (**ISDI-CI**) is developed to optimise the way to access and retrieve the integrated data for a wide range of user queries. The optimisation is evaluated by conducting experiments on the response of each query with different indexing schemes.

Overall, the IoT streaming data integration framework contributes to both academia and industry in dealing with the issues of IoT data integration and plays a vital role in today's interconnected environment. As potential practical applications, it can be used to support data-driven decisions to improve the customer experience, minimize fraud, and optimise operations and resource utilization. In addition, the proposed streaming data concepts and techniques can also be incorporated to leverage next-generation infrastructures such as cloud, advanced analytics/machine learning, real-time applications, and IoT analytics.

Declaration of Authorship

I, Quang Tu Doan, declare that this thesis titled, 'A Framework for Integrating IoT Streaming Data from Multiple Sources' and the work presented in it are my own. I confirm that:

"Except where reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis accepted for the award of any other degree or diploma. No other person's work has been used without due acknowledgment in the main text of the thesis. This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution."

Signed: Doan Quang Tu

Date: 09/08/2021

Acknowledgements

Writing a PhD is like being on a long journey without knowing when the destination will be reached. During this journey, there were a lot of obstacles and distractions, which called for a great deal of perseverance and determination. Fortunately, this was not a solo journey. Throughout my PhD candidature, I had generous networks of support from different people without whom I would never have found ways to reach the destination. I would like to take the opportunity to thank a number of people whose immense support has shaped this thesis.

First, I am deeply grateful for the extensive support and inspiration I have had the privilege of enjoying from all my supervisors. This thesis is a tribute to my principal supervisor Professor Wenny Rahayu, who introduced me to the scientific research on streaming data integration systems. Without her unconditional support which ignited my passion for this PhD project, this thesis would never have reached its final form. Although I suspect that at times I gave her moments of frustration, I hope that she enjoyed working with me and that our collaboration has had an effect on our respective future careers. Professor Wenny Rahayu, I am proud to have had a supervisor like you and I cannot thank you enough for being a more-than-wonderful supervisor who shared all the joys and anguish of my PhD journey. I would also like to express my deepest gratitude to Dr A. S. M. Kayes, my second supervisor and my 'big brother', who helped me to understand how to solve problems and write a thesis and academic papers. His insightful suggestions, generous encouragement and considerate instruction helped me during my PhD candidature. I learned many things from him which helped me become an independent researcher in my field. Over countless hours and with endless patience, he taught me how to conduct research from finding a problem to the later stages of writing a paper and finally building a solution and completing this thesis. I am also deeply thankful to Dr Kinh Nguyen for the many fruitful and inspiring discussions that influenced every contribution of this thesis. I always felt encouraged and extremely privileged to have a supervisor like you. I will never forget the times when you excitedly presented your ideas and reviewed my points on the white board in your office. I would also like to express my thanks to Dr Eric Pardede, Chair of the Progress Committee for his kindness and compassion as well as his critical comments on my PhD progress.

Second, I would like to thank La Trobe University for giving me so many memorable and wonderful experiences during my six years in Australia pursuing a master's and PhD degrees. I am also deeply indebted for the generous financial support from La Trobe University that made this PhD candidature possible, and **this work was supported by a La Trobe University Postgraduate Research Scholarship**. I am very grateful to the many great professional staff in the Department of Computer Science: Dr Fei Liu,

Dzung Le, Renuka Eliezer, Michele Mooney and others. Thank you again, Michele, for proofreading this thesis. All of you have made my time here memorable.

Third, my list of support also extends to my other great friends. I particularly thank Syed for being my best friend at La Trobe University. Thank you for giving me positive thoughts during the difficult days of my PhD candidature. I would like to extend my thankfulness to my many other great friends both in Australia and Vietnam: Dong (Mr.Cu), Khoa (Kevin Phan), Tommy, Long Truong, Frankie, Duc, Loan, Chien, Thuy Bac, Nhan Tinh, Dao Tung, Anne Tieu, Nga Duc, Anh Thanh, Trammoo, Dan and many others. Enjoying life outside academic circles has been so joyful with all of you! I would like to extend my thankfulness to my very old (secondary and high school) friend Dr Mai Vu for your support in the latter stages of completing this thesis. My special thanks to Jenna, who came to me just in time to help me to overcome my personal issues.

Last but not least, I would like to express my special thanks to my family for their love, advice and encouragement over these past few years. I cannot adequately express my gratitude to my parents for raising me in a happy family and for their unconditional support which allowed me to choose any pathway I wanted to take. Though they do not know anything about research or academia, my parents may now feel relieved as they do not have to keep asking me "when will you finish your study?" anymore. I thank Mum and Dad for giving me the wings to fly in freedom. I also thank Thu Le for caring support she gave to our children which allowed me the time to focus on this thesis. Lastly, my heartfelt thanks go to my little sons, Ken and Ka, who accompanied me to Australia for this further study when they were twenty-three months old and eight months old, respectively. They are the best companions and give me cuddles whenever I need them, as well as giving me great happiness which helped me throughout this PhD candidature. This thesis is dedicated to Ken and Ka.

Contents

List of Figures

List of Tables

List of Publications

- Doan Quang Tu, A. S. M. Kayes, Wenny Rahayu, Kinh Nguyen: (2020) "IoT Streaming Data Integration from Multiple Sources", Computing, 102(10), 2299-2329, Springer (Q2 journal, Impact Factor: 2.063, H Index: 60, **published**).

- Doan Quang Tu, A. S. M. Kayes, Wenny Rahayu, Kinh Nguyen: (2020) "Integration of IoT Streaming Data with Efficient Indexing and Storage Optimization", IEEE Access (Q1 journal, Impact Factor: 4.098, H Index: 127, **published**).

- Doan Quang Tu, A. S. M. Kayes, Wenny Rahayu, Kinh Nguyen: (2019) "ISDI: A New Window-Based Framework for Integrating IoT Streaming Data from Multiple Sources", AINA 2019: 498-511 (**Received Best Paper Award**, CORE B Ranked, **published**).

- Doan Quang Tu, A. S. M. Kayes, Wenny Rahayu, Kinh Nguyen: (2021) "A Framework for IoT Streaming Data Indexing and Query Optimisation", IEEE Internet of Things Journal (Q1 journal, Impact Factor: 9.47, H Index: 97, **submitted**).

- Doan Quang Tu, A. S. M. Kayes, Wenny Rahayu, Kinh Nguyen: (2021) "IoT Streaming Data Indexing and Integration from Multiple Sensors: A Survey and Future Research Directions", IEEE Sensors Journal (Q1 journal, Impact Factor: 3.30, H Index: 121, **submitted**).

Chapter 1

Introduction

The convergence of physical-digital systems, as the globally ground-breaking driving force of the fourth industrial revolution (Industry 4.0), has highlighted the essential role the Internet of Things (IoT) has come to play in daily lives. A report from Juniper Research has revealed that the total number of connected IoT sensors and devices is set to exceed 50 billion by 2022, which is a double increase compared to 2016. It is also predicted that the growth will be equivalent to 140% over the next 4 years [1]. Due to the widespread popularity of IoT devices, the increase in IoT streaming data is unprecedented and it is an enormous challenge to collect and integrate data from these IoT sources. As a result, IoT streaming data analytics which deals with the processing and analysis of large data volumes generated by connected devices is a critical area. Companies can derive a number of benefits from streaming data by optimizing their operations, controlling processes automatically, predicting faults in relation to maintenance in the manufacturing industry and so on. The combination of IoT and data analytics has already proven to be beneficial in retail, healthcare, telematics, manufacturing, and smart cities.

To shed light on concerns about IoT streaming data and its integration from multiple sources, in the remaining sections of this chapter, a background of the connected world and IoT data is discussed (Section 1.1), and the term *streaming data* is introduced (Section 1.2). An overview of streaming data integration is discussed in Section 1.3. Finally, in Section 1.4, the thesis outline is presented to illustrate its structure.

1.1 The Connected World and IoT Data

The introduction of the Internet of Things (IoT) has brought about a revolution in the data industry. A range of sensors and other personal devices, including security systems, smart TVs, smart appliances, and wearable health devices, which collect data from everywhere and are connected over the Internet, detect, measure, and send data in several forms. There are also various commercial IoT devices, such as traffic monitoring devices, commercial security systems, and weather tracking systems that continuously send and receive. This data is collected by the IoT and provides real-time valuable insights to save time, money and energy. For example, in the commercial real estate industry, there are several types of IoT data, namely equipment data, environmental data and sub-meter data. Equipment data enables real-time fault detection, runtime-based schedules and predictive maintenance thus saving energy cost, increasing productivity and extending equipment life. Taking the advantage of environmental data, IoT sensors are deployed to track a range of data streams within buildings, for example, temperature, air quality, people flow, moisture, and movement. These datasets are mainly used to predict potential issues to avoid disaster scenarios such as leaks and floods. Similarly, sub-meter data support vendors by automating the utility sub-metering process, eliminating errors and generating bills as soon as the billing period ends. In health care, IoT applications such as connected thermal cameras, contact tracing devices and health-monitoring wearables provide the critical data needed to help fight diseases. In the manufacturing industry, when sensors collect data from a connected device, the sensor data can be used to update a "digital twin" copy of the device's state in real time [2, 3]. Many other IoT data use cases also offer valuable insights such as medical data (e.g., heartbeat, blood pressure, etc.), educational data (e.g., attendance and learning), location data (e.g., traffic congestion) and agriculture data (e.g., weather and soil information).

To sum up, as the emerging paradigm of IoT enables communication between electronic devices and sensors through the Internet, IoT data has become very critical in daily life. Overall, IoT is an innovation that combines an extensive variety of smart systems, frameworks and intelligent devices and sensors which generate an unprecedented volume of IoT data which benefit human lives in many domains. Several examples of IoT streaming data usage in different domains are illustrated in Figure 1.1.

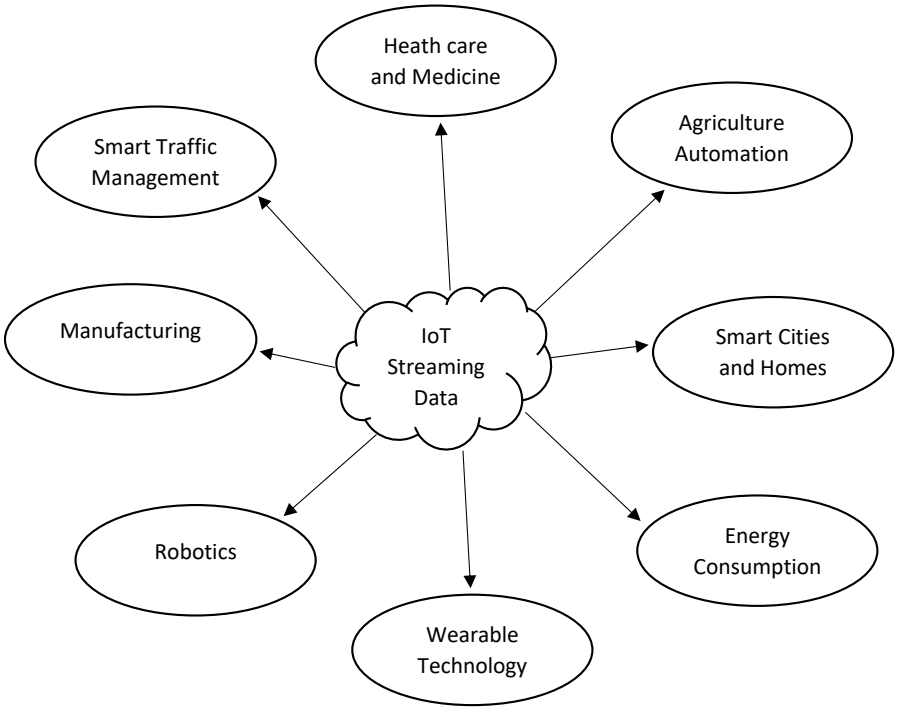


FIGURE 1.1: IoT Streaming Data Usage In Different Domains.

To gain insight from data, the first step is to collect multiple streams from which information arrives in numerous sources and formats. As discussed, the data for analysis may come from a data warehouse, data mart, data lake, or IoT devices. In these cases, the data may be an extract from a production system, for example, an e-commerce application. In addition, increasingly these days, the data for machine learning projects come from a variety of source including unstructured sources such as social media. In this scenario, IoT data collected from multiple sources must be processed efficiently, and it needs to perform several sub-processes, which collect, extract, gather and aggregate data. These processes are part of IoT data integration, which combines data from different sources into a single, unified view with the purpose of producing effective, actionable intelligence. Nowadays, IoT data integration is in high demand to support the diverse analytical and operational requirements for data. However, there is an urgent need to look at the nature of IoT data, which is the flow of information or data streams, before turning it into insights. Hence, in the following section, a background of IoT streaming data and an overview of streaming data integration to obtain insights from IoT data are presented.

1.2 Streaming Data

IoT sources or connected communicating devices not only produce big data but also fast data which arrive at a speed that is not perceptible to the human eye and which can seem fast to obtain value from it. This kind of IoT streaming data, and machine-generated data in particular, requires technology options like event stream processing to ensure optimum processing and insight. Data coming from the sensors of connected devices are the key source of streaming data now and in the future.

Valuable information is collected from a massive amount of streaming data that will be used to improve applications and business processes, so the first step in IoT streaming data is to collect and extract data from sources. IoT streaming data collection is the process of using sensors to track the conditions of physical things. Devices and technology connected over the IoT can monitor and measure data in real time. IoT data is unstructured and different IoT sources may generate different data formats. It is not easy to transform the data to a uniform format and to ensure that format is compatible with applications or clients' requirements. There are various types of data generated by IoT devices and it is essential to apply integrating tools to handle this data.

The second step is to extract useful information from IoT data and store it efficiently. Two commonly asked questions are: what are IoT data attributes and which information should be extracted? In practice, IoT data is highly dependent on sensors, processors, and other technical equipment. For instance, real-time GPS asset tracking data attributes are the position of objects and maps; energy and environment monitoring data attributes are temperature, pollution levels, and air-quality index; and health monitoring data attributes are pulse rates, blood pressure, and body temperature. However, all streaming data have a common attribute which is a time-stamp. Hence, when data is integrated, it is challenging to gain insights data from multiple sources within the choice of attributes for the insights depending on applications.

The third step is to access integrated IoT streaming data from its optimized storage. To deal with commercial pressure to utilise data in IoT solutions, proficiency in data integration becomes critical. However, there is a need to coordinate these solutions. A lack of access to, and reuse of, data to support shared goals and evolving business needs

will quickly constrict synergistic use of IoT technologies and information capabilities. The benefit of quick IoT data access is as important as data integration in real-time.

In the following section, an overview of IoT streaming data integration is discussed in relation to the topic of real-time processing and integration with IoT streaming data storage and encryption and IoT data access with indexing.

1.3 IoT Streaming Data Integration

Due to the rapid advancement of big data platforms, the need to improve access to data from multiple sources through data analysis and decision support systems has grown significantly over the last few years. However, with the unprecedented expansion of data in business, decision makers and researchers find it difficult to access the necessary data for a comprehensive and in-depth analysis. On the one hand, it is critical to be able to react and respond to queries from clients accurately in a quick and timely manner. On the other hand, it is necessary to integrate data coming from multiple sources as data is generated from different streams and is infinite. Continuously generated data is called streaming data [4].

In both research and industry arenas, it is necessary to obtain useful data and information in real time, but streaming data may come from multiple sources in a way that it seems infeasible to access as a whole. We are currently experiencing the era of IoT streaming data which arrive in different structures or even semi/non-structures. As such, capturing and/or transferring these heterogeneous data in different formats (e.g., structured, unstructured and semi-structured) into a unified form, which is suitable for analysis, is a challenging task [5]. Furthermore, the high volume of streaming data from each source may increase dramatically and dynamically, making it difficult to manage and store when integrating data from multiple sources in federated storage. Although there are several tools (e.g., storage resource management [6], and the Hadoop distributed system [7]) which satisfy the requirements of data storage management from distributed sources, they cannot handle dynamic changes and the need to integrate IoT streaming data with timing alignment and de-duplication issues. Hence, another issue revolves around how to integrate IoT streaming data from multiple sources on the fly in real time.

Regarding the last step of IoT streaming data integration, efficient access to data for a comprehensive and in-depth analysis has become more critical. This is due to the nature of data being non-static and continuously generated, which is even more challenging to access and store. This kind of data is referred to as streaming data or time series data and in the context of IoT data, it is a sequence of real numbers in time order. It is critical to be able to react and respond to queries from clients accurately in a timely manner by accessing time-series data. Factors that should be considered in improving data access are: how the data is accessed and how it is processed in real time from multiple sources. Hence, to adopt the idea of quick response queries from streaming data sources, there should be a mechanism for pre-processing streaming data including storage efficiency and efficient indexing. A characteristic of streaming data is that it is potentially unbounded in size, so there is a need to improve data compression in relation to storage. Also, it is necessary to index from the compressed time-series data without decompression, which facilitates much better performance in queries. Hence, there is a need to develop a framework to integrate time-series data from multiple IoT sources using compression and indexing techniques for streaming data.

Data compression is a reduction in number of bits that represent the data, saving storage capacity, speeding up file transfer, and decreasing the cost of storage hardware and network bandwidth. Compression techniques were developed last century but since the expansion of IoT data, many researchers again focused on compressing time-series data techniques, for example, Balck et al. with Sprintz [8], Wegener et al. with signal data compression [9] and Diffenderfer et al. with ZFP [10]. However, these techniques are merely compression approaches, which only focus on saving storage capacity. In this thesis, an improvement to compression is introduced, not only for storage saving but for the ability to index from compressed time-series data.

In recent years, a large body of research has been conducted on similarity searches and the subsequent data access and indexing [11–13]. In the context of time-series data indexing, an example query related to a similarity search can include finding past days in which the temperature recording is similar to today's pattern. From a different aspect, in our research, we propose an indexing framework that features easy-to-find results based on timing requirements but which are not based on similarity patterns. In particular, it is observed that clients not only focus on finding a trend (up or down) or a similar pattern in time-series data in a period of time, they also expect to obtain summarized

information on such time series. The term '*summarized information*' referred to in this thesis is not like "summarizations" proposed in [13], which are representations of time-series data segments. The term in this thesis refers to summarized outcomes extracted from a segment of data by relevant user-defined functions. For example, with the ability to continuously track time-series data based on time-stamps, the solution framework in this thesis can summarize information such as the average, maximum or minimum temperatures in a certain period of time.

1.4 Thesis Outline

This thesis introduces a framework to integrate time-series data from multiple IoT sources to facilitate further data analysis by integrating IoT streaming data in real-time, offering optimized storage with useful information and optimizing data access performance using indexing schemes.

This thesis is structured into seven chapters. The first three chapters are the introduction (Chapter 1), a literature review (Chapter 2) to provide the background knowledge required for this thesis, the research motivation, problem statement and a solution framework (Chapter 3) to provide the general requirements and formal specifications of the IoT streaming data integration (SDI) approach. The three following chapters present the key research contributions of this thesis: IoT streaming data processing using the windowing technique (Chapter 4), IoT streaming data compression and storage (Chapter 5) and IoT streaming data indexing and query optimization (chapter 6). Finally, the last chapter presents the conclusion, recapping the contributions of this thesis and related topics for future investigation (Chapter 7).

- Chapter 2 reviews the existing approaches to the issues of IoT SDI. The chapter focuses on the topic of integrating approaches and how to access and process data from multiple sources. It also investigates a mechanism to optimize the storage for IoT streaming data by reviewing the existing streaming data compression techniques and data access optimization with indexing. Finally, data integration features are discussed and the techniques regarding theses features are compared.
- Chapter 3 provides a generic scenario of SDI, involving several issues such as timing conflict, data redundancy and real-time integration issues. Based on these

challenges, the general requirements for SDI are discussed, and then the problem statement of this thesis is outlined. In this chapter, a solution framework is proposed with three main research contributions, namely streaming data processing using the windowing technique, IoT streaming data optimized storage and optimized data access with an index scheme.

- Chapter 4 presents a formal approach for the real-time integration of IoT streaming datasets. In this chapter, the thesis addresses one of the important issues of timing conflict/alignment among streaming data coming from multiple sources. A generic window-based ISDI approach is proposed to deal with IoT data in different formats and introduces the algorithms to integrate the IoT streaming data obtained from multiple sources. In particular, a basic windowing algorithm for real-time data integration is extended to deal with the timing alignment issue. A de-duplication algorithm is introduced to deal with data redundancies and to identify the useful fragments of the integrated data. Several sets of experiments are conducted to quantify the performance of the proposed window-based ISDI approach.

- Chapter 5 solves the issues of information retrieval by constructing a compression framework (ISDI-C) within a lossless compression technique for floating point time-series data, where an index is based on the time-stamp from the compressed data that facilitates the search for data without full decompression. Several sets of experiments are conducted to quantify the performance of our proposed approach. The experimental results, performed on IoT datasets, show a reduction in storage compared with the existing compression techniques. The experimental study also demonstrates the capability of time-series data indexing and integration in real time.

- Chapter 6 presents a variety of queries from the identified scenarios and then it proposes a framework (ISDI-CI) to optimize the way to access and retrieve data per the users' queries. The optimization is evaluated by conducting an experiment to illustrate the ability of the framework to respond to queries using different indexing schemes.

- Chapter 7 concludes this research by summarizing the key contributions of this thesis and discussing some interesting research directions for future investigation.

Chapter 2

Literature Review

In today's interconnected world of technologies, modern data comes from a large variety of sources such as hardware sensors, servers, mobile devices, applications and web browsers. Data generated continuously in this way is known as streaming data. Streaming data are continuously generated by multiple data sources, e.g. sensors, mobile devices, etc., and are sent simultaneously to the relevant application program to be processed in real time in a continuous and timely fashion.

Streaming data processing is beneficial in most industries and big data use cases where dynamic data is generated on a continual basis. Companies generally begin with simple applications such as collecting system logs and rudimentary processing like rolling min-max computations. Then, these applications evolve to more sophisticated near-real-time processing. Initially, applications may process data streams to produce simple reports, and perform simple actions in response, such as emitting alarms when key measures exceed certain thresholds. However, these applications perform more sophisticated forms of data analysis, like applying machine learning algorithms to extract deeper insights from data. Complex, stream and event processing algorithms, such as time-decaying windows [14, 15] to find the most recent popular movies, have been developed. These processes depend on how unprecedented amounts of data are accessed from mobile devices, IoT sensors, social media, and other databases that simply did not exist a decade or two ago. This increase in information sources leads to the need for streaming data integration. Streaming data integration is a process where data sources are integrated in real time to

provide up-to-the-minute information and makes use of data streaming, which enables organizations to collect and analyse information in real time.

In comparison to traditional data processing, the processing of streaming data – with the requirement of handling incoming data in real time and the need to cope with data generated by different sources, potentially with variations among them – presents a number of additional and challenging issues.

First, synchronizing information and ensuring it is consistent despite sudden changes in the data flow from sources is challenging. In traditional data integration, data is transferred to a staging area, where it is synchronized as information sets and processed for loading into the target system. During the real-time data integration process, there is no staging area, rather information is brought together instantly, so there is no method to ensure it is synchronized.

In general, streaming data processing involves two main functions, storage and processing. The storage function needs to support record ordering and to have strong consistency to enable the fast and inexpensive processing of large streams of data. The processing function is responsible for consuming data from the storage layer, running computations on that data, and making decisions on whether to keep all or partial data in the persistence layer of the data storage. It is necessary to plan for scalability, data durability, and fault tolerance in both the storage and processing functions.

Taking a broader view, the issues of streaming data processing can be grouped into the following general tasks:

- Data integration. How are we to process data that are generated continuously from multiple sources, with potential differences in generation frequencies and data formats?
- Storage optimisation. Streaming data tend to be generated and accumulated in a large volume over time. How should we optimize the storage of integrated data to cope with such a large volume?
- Data access mechanism. The stored integrated data are to be extracted and used by application programs to serve the users' interests. It is necessary to have a data access mechanism to facilitate efficient query performance. The term "querying"

is commonly used to signify the extraction of the stored data for such purposes. What can be done, as generally applicable techniques, to have this kind of data access?

To complete the aforementioned general tasks, the literature review is organized as follows. Section 2.1 covers the related work on integrating streaming data from multiple sources. Section 2.2 reviews the related work that involves storage optimisation with streaming data compression. Section 2.3 covers a review of some of the indexing techniques which can be applied to facilitate query optimisation.

2.1 Integrating Streaming Data from Multiple Sources

As an overall picture, the statistics on the number of research studies on Google Scholar which relate to data streaming data in general processing and streaming data integration in particular are presented in Figure 2.1. The figure is based on the results of the searches on the two key terms “streaming data processing” and “streaming data integration”. The term “streaming data processing” was introduced in academic research about two decades ago. It is used to refer to the overall process of handling streaming data, appropriate to some purpose and typically up to the point of saving the processed data to long-term storage for further use. In contrast, the term “streaming data integration” refers to a sub-process of streaming data processing with an emphasis on how to deal with the potential complications caused by the fact that the data are generated continuously by multiple sources with possible diverse characteristics.

As shown in Figure 2.1, with only 4 articles on data streaming processing in the year 2000, the number of research studies related to this topic has been increasing steadily and significantly in subsequent years. The number of articles on streaming data processing in 2020 was triple the number of research studies on this topic in 2015, which was 1,560 and 515, respectively. Likewise, although the articles with the keyword “streaming data integration" were modest in quantity (51), the number of articles in 2020 were two-and-a-half times more in comparison with those in 2015 (124).

In this section, we first provide a short overview of the state-of-the-art big data processing and integration approaches as the area related to our research. The overview

includes integrating data from multiple sources, data integration features and accessing and processing data from multiple sources.

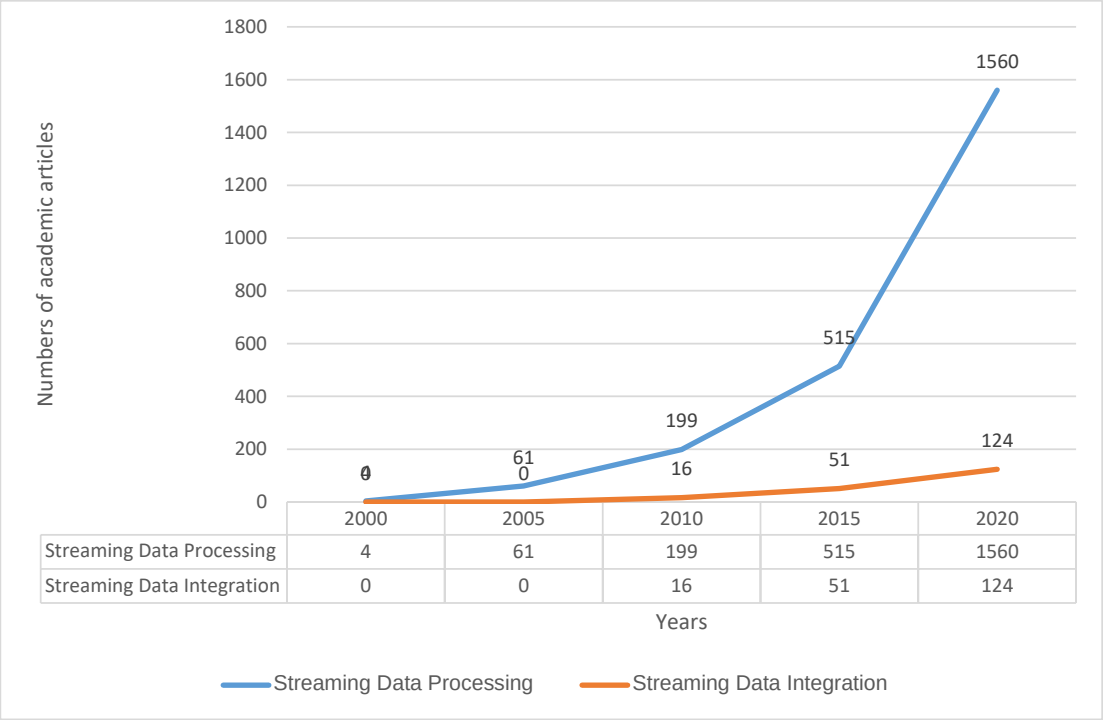


FIGURE 2.1: Number of related articles from 2000 to 2020, according to Google Scholar

2.1.1 Integrating Approaches

In this subsection, integration techniques are categorised into three types, including structure-based integration, semantic-based integration and time-based integration. While structure-based integration is a traditional technique which considers data structure and schema to map data from sources, the semantic-based techniques focus on building domain-based knowledge to share a global schema for different sources. The time-based techniques are reviewed as a close work to this research regarding streaming data integration.

In the context of IoT streaming data integration, the main challenges can be categorized as follows: i) dealing with various types of data in different structures [16], ii) dynamic streaming data acquisition from multiple sources and dealing with transformation towards modeling a unified schema [17]; and iii) other concerns related to data privacy and security issues. In this thesis, the research work is to address the first two challenges.

2.1.1.1 Structure-Based Integration

IoT streaming data integration is a promising approach to address the challenges and issues that traditional data integration approaches cannot deal with [18–22]. To address the streaming data integration issues, in [16], the authors list some popular models and techniques such as schema mapping, record linkage, and data fusion, which are potential solutions for traditional big data challenges. The earlier research works on record linkage [23–26] utilize blocking techniques to combine identical data from different sources. These works deal with a huge amount of data, but not the issue related to timing alignment, where aligning streaming data with identical or different timestamps is one of the critical concerns.

2.1.1.2 Semantic-Based Integration

Similar to record linkage research, the schema matching and mapping [27] and global schema [28] research works are not adequate to handle the same issue of timing alignment. Also, these existing works do not resolve the data redundancy issue while dealing with identical or different timestamps from multiple data sources. In terms of IoT streaming data integration, there are several ontology-based research works in the literature. In [29], an ontology-based data integration approach with respect to collating streaming data from different sources has been introduced. The integration task in this work [29] is based on ontology-based access to relational data sources. In this perspective, in [30–34], the authors have proposed an approach to OBDM (Ontology-Based Data Management) in order to provide a shared or unified vision to process/integrate data from different sources. This ontology-based shared vision is used as a global schema for all data sources. Typically, in these research works, the OBDM model is proposed based on three main layers: the *ontology layer* for the representation of the conceptual domain, the *mapping layer* for the correspondence between the local data sources and the general concepts, and the *source layer* for producing data from different environments. The ontology can be considered as a global schema for mapping different local schema into a unified data model, however, similar to other data integration solutions, these ontology-based solutions are not adequate to address the issue of timing conflicts (i.e., aligning different timing frequencies) while integrating time-series data from multiple sources.

A source description contains a source schema that describes the content of the source, and a mapping between the corresponding elements of the source schema and the mediated schema. In later work on data integration, techniques for automating the schema and tasks as much as possible are needed to simplify and speed up the development, maintenance and use of metadata-intensive applications. As such, ontology matching was developed as powerful schema matching prototypes and applied to a large variety of matching problems [35–37].

2.1.1.3 Time-Based Integration

In the existing research related to window-based techniques to process streaming data, the basic windowing algorithms proposed in [38–40] focus on the machine learning and data mining techniques through windows of data with fixed sizes. A time-based window is a batch of data and is considered as a training set for such learning algorithms and the window is revised if the mining rule is not sufficient [38]. In [41], the authors utilize the basic windowing technique to identify the observed average of data elements in a window and adjust its size to derive an efficient performance variations. The research work in [42] is also relevant to the windowing technique for processing time-series data. However, this work mainly focuses on concept drift when collecting data in sequences. These existing windowing approaches are not sufficient to integrate IoT streaming data from multiple sources in real time.

Pareek et al. [43] introduced a streaming analytics platform (the Striim engine) for real-time data integration with respect to structured data from multiple sources. Recently, Pareek et al. [44] proposed several adapters for the Striim engine to extract such data, to transform SQL-based data and for continuous data validation. This Striim engine can integrate time-series data with other structured data. In particular, it collects datasets from different sources and transforms them for aggregation without considering identical or different timestamps. As such, Striim is not adequate to handle timing alignment among different streaming data from multiple sources. In addition, it is also not enough to resolve data duplication issue while integrating data with various timestamps.

2.1.2 Discussion on Data Integration Features

Streaming data integration from multiple sources requires processing in real time, and the data from these sources needs to be consistent at arrival time and synchronized with an unified scheme at the integrated stage. Therefore, in this subsection, several features which are used to address streaming data processing and integration issues are described, including timing alignment, de-duplication, (nearly) real-time processes and window-based integration. Apart from reviewing the existing work on these features, a comparative assessment of these works is discussed compared to the proposed approach of this thesis.

2.1.2.1 Time-based

Alignment-related issues and timing synchronization are among the top concerns of many always-on real-time applications. In the context of streaming data integration, this is called a timing conflict of data coming from different sources, and timing alignment is a solution to address this issue. For example, IoT sensor data obtained from multiple sources have different timing orders and all the instances in the integrated data need to have the same timestamp. This idea of dealing with different timing conflicts is discussed in [45]. In industry, IoT data obtained from multiple sources can arrive at different times, e.g., data from sensors A and B arrive at different timestamps t1 and t2, whereas t2 = (t1 + n), respectively. This requires the streaming data to be processed in order. Furthermore, data from multiple sources can be generated in different frequencies so the timestamps of this data must be aligned and sorted before being integrated.

2.1.2.2 De-duplication

In general, many existing streaming processing approaches do not consider the issue of timing conflict. Some use de-duplication to eliminate duplicated or redundant information. Data de-duplication is a step in database record linkage referring to the task of finding entries that refer to the same entity in different sources. There are several types of research on data de-duplication work.

In this research, de-duplication is the process of merging instances that have the same timestamp, an instance key, and aggregate the identical attributes of different instances

which have the same keys. In particular, the frequency of timestamps (e.g., seconds, minutes or hours) of the different data instances may be different, and more than one data instance may contain the same timestamp. Those with the same timestamp contain identical attributes and non-identical attributes, so the raw IoT data from many sensors are usually not ready for any kind of processing or integration. Therefore, there is a need to deal with data redundancy and duplication issues by merging or aggregating data instances into one with the associated attributes of such instances.

Many existing works have addressed the issue of duplication and data redundancy; however, they only focused on specific scenarios and lead to the loss of data integrity [46]. For example, the work in [47] introduced a timing-based de-duplication with inline de-duplication and post-process de-duplication. The work in [48] deals with data redundancy efficiently but uses a de-duplication tool, ZFS [49], to support the de-duplication functionality for virtual machine disk images. Similarly, the schema matching and mapping techniques in [27] focus on specific data types to reduce and de-duplicate identical records. Not all of these works can handle the duplication issues comprehensively on both timestamps and identical non-timestamp attributes at the same time.

2.1.2.3 Window-based Integration

Data integration is the process of combining data from different sources into a single, unified view. Some existing works make use of sliding windows in the streaming process [50–52] to integrate data. At the stage of integration, integrated windows begin the ingestion process which includes steps such as cleansing, mapping, and transformation. To form an integrated window with a starting timestamp at t and a period of p (e.g., a minute, a few minutes or an hour), each window extracted from a single source is also created at time t and the data volume of windows with different time frequencies are also different. The technique to synchronize the time is time alignment. In addition, the integrated data windows need to be de-duplicated and the redundancy removed. These processes are also processed in real time. These features are discussed above. None of those approaches focus on all the features and their issues.

Some existing works aggregate and integrate data at analytical layers by integrated windows, but they do not satisfy the requirement of windows integration. The approach

TABLE 2.1: Comparative Analysis of the Existing Data Integration and Processing Solutions

	Handling IoT Streaming Data		
Approaches	**Timing alignment**	**De-duplication**	**Window-based integration**
Data Linkage [24] [26]	*N/A*	*N/A*	*N/A*
Schema Matching and Global Schema [27] [28]	*N/A*	*P/A*	*N/A*
Ontology-Based Solutions [29] [32] [33]	*N/A*	*P/A*	*N/A*
Window-Based Solutions [38] [41] [42]	*N/A*	*N/A*	*P/A*
Striim [43] [44]	*N/A*	*P/A*	*N/A*
Hadoop and Access Control [55] [56] [57]	*N/A*	*N/A*	*N/A*
The ISDI Approach	*YES*	*YES*	*YES*

in [53] introduced a framework building feature vectors for machine learning from heterogeneous streaming data sources online to enable the easy configuration of streaming data fusion and modeling pipelines. The model integrates data by windows aggregations but does not focus on the issues of timing conflict and duplication. The works in [29, 32, 33, 54] make use of streaming queries to aggregate data windows, but the integration merely depends on query functions.

2.1.2.4 Comparative Assessment

Table 2.1 shows the results of comparative studies in which we use "YES" when a feature is available, "N/A" when a feature is not available, and "P/A" when a feature is partially available. In our comparative study, we consider the following aspects of our proposed IoT streaming data integration approach.

Several research studies address the traditional challenges of data integration, such as data linkage [24, 26], schema matching [27], global schema [28] and ontology-based

approaches [29, 32, 33]. However, these approaches does not consider timing conflict (by handling different ***timing alignments*** and data redundancy (by handling ***de-duplication***) issues while integrating streaming data from multiple sources. Several window-based techniques [38, 41, 42] have been introduced in the literature on processing dynamic streaming data in real time based on the windows from different data sources. On the other hand, recently, the Striim platform [43, 44] has been introduced and it can also process and integrate time-series data. However, these existing solutions using the traditional windowing technique and the Striim engine are not adequate to deal with the identified issues of timing conflicts and data redundancy. The existing approaches to access and process data from different sources [55–57] have been considered ***IoT data integration in real time***. To address this issue, in this research a new ***window-based integration*** approach is introduced to handle IoT streaming data from multiple sources.

Different from the aforementioned data integration and processing research, in this thesis, a formal model for integrating time-series data from multiple sources is introduced, including underlying concepts through preliminary definitions and formal notations. In addition, a generic integrator model is proposed to apply in different environments to integrate multiple time-series data.

2.1.3 Accessing and Processing Data from Multiple Sources

Ahad and Biswas [55] have proposed a distributed Hadoop-based architecture for combining unstructured data from different small files (e.g., text, document, pdf, and so on). They have introduced a dynamic merging mechanism with the goal of collating data from different types of files. These Hadoop-based research works are useful to process and manage data from multiple environments. Similar to Hadoop-based research, our group's earlier works in the area of context-sensitive access control can be applicable to capture data from a single centralized source [12, 56, 58, 59] or from multiple decentralized sources [57, 60]. However, these research works do not focus on the key issue of timing alignment of multiple time-series data from different sources. Based on our experience in this research, the streaming or time-series data need to be processed and integrated in real time by dealing with timing conflicts from multiple data sources.

There are many ways to address the issue of data processing. Traditionally, in many cases, historical and archived data are processed and time issues are not critical. It can

take a few hours, or even a few days for a response to a monthly/yearly report. However, with the need for a quick response on decision making and data analysis, processing tasks are more crucial in a timely manner, and the answers need to be delivered within seconds or shorter. This is real-time processing, which requires continual input, constant processing, and a steady output of data. In some cases, near real-time processing, processing time in minutes is acceptable in lieu of seconds, can be considered as real-time processing. A significant example of real-time processing is data streaming, radar systems, customer service systems, and bank ATMs, where immediate processing is crucial to ensure the system works properly. Apache Spark [61–63] is a valuable tool to use for real-time processing, and Striim [43] is an example of a platform that build continuous, streaming data pipelines, including change data capture (CDC), to power real-time cloud integration, log correlation, edge processing, and streaming analytics.

Topic streaming data processing in this research is discussed in relation to the process of extracting data from sources by using windows and issues such as timing alignment and de-duplication for streaming data integration, are addressed. Much of the research on streaming data processing and integration for the purposes of data analytics focuses on specific tasks in real time, but not on these issues. For instance, the authors of [64] introduced the architectural concept of the Apache Storm based real-time data processing topology to process streams with their data analytics task, a task building up predictive model for programming technologies trends, from social networks. The framework processes data by adapters between the data sources and real-time data processing infrastructure. Each adapter is responsible for a specific data stream which enables tuples to be produced from streams with different protocols and data formats. To pursue a similar purpose for a real-time data analytics framework to analyse Twitter data, the work in [65] performed basic processing tasks and proposed an infrastructure to perform analytics on the streaming data.

2.2 Storage optimisation

In the previous section, our review work focuses on the models of time series data integration; however, to make use of the integration, there is a need to store integrated data efficiently while still allowing effective user queries. Therefore, there should be a

mechanism for pre-processing streaming data including efficient data storage which facilitate data access for future purposes. In this section, storage optimisation based on compression and patterns are reviewed and discussed as follows.

2.2.1 Compression based Storage optimisation

Time-series data has a special structure, which takes into account the gaps between the values of the two adjacent timestamps. For example, in financial time series data, the price of the WOOLWORTHS GROUP LIMITED at time T is very close to its price at time T+1. This structure can be exploited by many floating-point compression techniques. These approaches are very popular when analysing floating-point representations with three main components, namely sign, exponent and mantissa. Wegener and Albert [9] proposed a typical floating-point compression and decompression method by removing the least significant bits (LSBs) of a component (mantissa) based on similar consecutive floating-point values and grouping values into blocks to facilitate the compression. An important process for this method is to create a quantization function before encoding the data. [66] used blocks differently and invented blocks of 4^d values (d is the number of dimensions). In this work, the lossy compression, ZFP, groups values into a block and converts floating-point values to a fixed-point representation. It then de-correlates the values by applying an orthogonal block transform and encodes the ordered transform coefficients. Also, based on the binary representations of components, [10] improved ZFP by establishing a bound which is a well-known limitation of ZFP.

2.2.2 Pattern based Storage optimisation

In addition, compression techniques rely on the small difference between consecutive values and make predictions for the next values. For example, the approach in [67] takes advantage of the correlation bewteen the subsequent data and earlier data. Recurring difference patterns are identified and then recorded in a hash table which supports the predictions of the next similar patterns. It compresses values by encoding the differences between the predicted and the true values. Similarly, FPC [68] compresses data in sequences by predicting the next value in the sequence and using hash tables as predictors. Lindstrom et al. [69] also provide a method based on coding prediction within a plug-in scheme. However, this work performs a floating-point quantization process before

encoding integer data. Similar to [9], Sprintz [8] removes LSBs to reduce the number of redundant bits to store values. This work focuses only on compressing the integer data, and it recommends the compression of floating-point data using floating point quantization.

The work in this thesis extends time-series integer data compression which also exploits the nature of time-series data, the similarity between consecutive values, and greatly reduces storage requirements.

2.3 Data Access with Indexing Techniques

This section focuses mainly on indexing techniques which optimize the way to access data and provides the ability of respond to queries. It also investigates big data indices in general and explores some techniques to index streaming data.

2.3.1 Key-value Pair Indexing

In terms of data indexing, SmallClient[70] improves query execution and search performance for big datasets and minimises the overhead incurred by indexing. The framework is implementable on any distributed file system. Basically, the main part of SmallClient consists of three processes, namely block creation, index creation and query execution. The systems create blocks so that no records are broken and then they use <key,value> pairs as the content of records and the location of a data block to add in a BTree structure. Also, based on <key,value> pairs to make a basic structure, Elsayed et al.[71] proposed a framework to address document similarity problems. They used MapReduce because it has same-structure tasks which perform a computation on a chunk of data to obtain partial results and then is aggregated to obtain the last outcome. The indexing mechanism of the framework is used as a mapper, taking <key, value> pairs as inputs to generate intermediate ones. The reducer produces the output based on all the values associated with the same key. In particular, each term and its weight (the importance of a term in a document) is associated with a document (docid) so that the term is considered as the key, and a tuple containing docid and term weight are values. The reducer is responsible for summing all the scores of the compared individuals. Likewise, Lee et al.[72] applied indexing methods and MapReduce to the area of digital forensics

which requires big data processing. They proposed the distributed text processing system (DTPS) for searching which can support the identification of relevant evidence in a trial from very large-scale data in a quick and accurate manner. The index method used in the system is the document indexer and MapReduce is used to manipulate the <key,value> based on <docId, term>. Hadoop is applied to solve problems involving massive amounts of incoming data as its inputs. Several comments have noted that the authors need to improve the accuracy of this in the future.

2.3.2 Criteria-based Indexing

According to Mamta[73], indexing splits data into fragments so that they can be in a query, based on certain criteria. An example of a popular indexing technique is the Cracking Database (Selection cracking). Indexes in Hadoop include Hadoop ++, HAIL (Hadoop Aggressive Indexing Library), and LIAH (Lazy Indexing and Adaptivity in Hadoop). Mamta summarised the challenges of big data from a different perspective. These challenges are representation, redundancy, storage, heterogeneity and scalability; process challenges include acquisition, alignment (ER), transforming and filtering, modeling, understandable output and visualizing data; management challenges are privacy, ethics, security and legal. In another survey, the authors of [74] identified the 6V requirements for big data indexing, namely volume, velocity, variety, veracity, variability and value. They categorized indexing techniques into three methods, namely non-AI, AI and collaborative AI. Non-AI methods are traditional indexing techniques (index construction and query responses). These methods are mostly based on bitmap, hashing, B-Tree and R-Tree. All the data/patterns in these methods are known and implemented following rule-based techniques. The AI methods are reviewed in the following subsection.

2.3.3 AI-based Indexing

AI methods use a knowledge base to index a large number of moving objects. The data in this case is variable, so the index needs to be updated frequently; whereas, collaborative AI methods improve accuracy and search efficiency by collaborative AI (collaborative ML and knowledge representation and reasoning methods). For example, it can adopt multiple indexing algorithms along with knowledge representation and reasoning to achieve a high detection rate for the prediction of missing user preferences.

In a survey of indexing on big data [75], the authors review two popular AI indexing approaches, namely Latent Semantic Indexing (LSI) [76–80], and Hidden Markov Model (HMM) [81–85]. For the purpose of indexing, the two methods make use of pattern recognition and relationship between data, such as Resource Description Framework (RDF), to make predictions of future states of an item/data or to support decision making by using contextual matching. This is not aligned with the thesis's scenario because the query predicate in this work is to filter data base on the data set from streaming sensor devices which is more suitable for indexing approaches such as tree-based indexing and hash indexing strategies.

2.3.4 Partition-based Indexing

To address the challenges associated with streams, some studies focus on optimising the ways data are processed in many aspects [86–88]. The research in [86] detects the reusable parts, which are common resource usage that can be run by different applications and optimize the streaming applications based on analysing the meta-store that captures and exposes those applications. The work in [87] splits the workflow into suitable partitions to reduce the cost of the inter-partition communication cost. Each partition is mapped to an execution node that offers minimum execution time. Since each partition is mapped on one execution node, the communication cost within the same execution node is negligible. The optimized partitions provide minimum inter-partition data movement, which improves the overall performance of the application. In this work, data parallelism is applied to the most compute-intensive task of each partition which reduces the latency. The work in [88] trace individual input records so that they can identify outliers in a web crawling and document processing system and use the insights to define URL filtering rules. Then they identify heavy keys, such as NULL, which should be filtered before processing. In terms of optimisation they improve the key-based partitioning mechanisms and measure the limits of over-partitioning if heavy thread-unsafe libraries are imported. In contrast to these optimisations, the proposed framework focuses on how to get access to data and obtain a quick response in nearly real time using indexing schemes. Hence, in this subsection, some existing work on both indexing approaches are reviewed.

In [89], the authors discuss indexing heterogeneous data streams, which is nearly the same as our topic. They introduce an index structure using bitmap based techniques

to support the space-efficient lossless archiving of the data stream and then develop an optimisation framework to adapt the changes of streaming data for index. In detail, their index is on the segmented sections of an incoming records stream, which is stored in files for disk I/O efficiency. Records in the same section are more homogeneous than records in different sections, and to evaluate a structural predicate, the section index looking for candidate sections that contain attributes of interest needs to be scanned through. For each of the candidate sections, it is scanned through the bitmap index and the bitmap is used to evaluate the validity of the structural predicate. However, the work assume some pattern usage which is highly application specific. Our approach does not rely on such specific assumptions.

Some other existing work [90–92] on streaming data indexing makes use of sliding windows to index streams for information retrieval. In [91], the authors also investigate a number of data streaming indexing approaches with some extension versions, namely B-trees, burst tries and the advanced Judy implementation of compact tries. However, similar to other indexing techniques, although there is a scalable range search, insertion and deletion of data streams, these indices merely work in some specific scenarios of individual element retrieval and they are not attached and applied in any optimised framework which the proposed approach offers.

In [92], the authors provide a deep insight into the window indexing methods by providing a list of attribute values and their counts to answer set-valued queries, and those which provide direct access to tuples to answer attribute-valued queries. However, this work focuses on traditional sliding windows, without taking into account the different rates of incoming data from multiple sources and indexing on every tuple.

This section outlines the different types of indices developed by earlier work in the area. As described above, each of these indices targets a particular application domain or a particular performance measure (e.g., scalability of a pattern-based search, or indexing based on partitions to reduce communication overheads).

2.4 Summary

In this chapter, the existing work and approaches related to streaming data integration have been reviewed in relation to three topics, namely data integration, storage

optimisation and data access.

Firstly, in the review on streaming data integration, the integration approaches are categorized into three types, namely structured-based, semantic-based and time-based integration. The structure-based integration approaches are related to schema mapping, record linkage and data fusion while the semantic-based approaches rely mostly on ontology for sources sharing a unified schema. Time-based approaches are techniques based on time-based windowing techniques to integrate batches of data from time-series data. On the same topic, the chapter introduced three data integration features, namely time-based de-duplication and window-based integration and gave a comparative assessment of the existing work on these features. Secondly, the related works on storage optimisation with compression and pattern optimisation are discussed. These existing approaches focus on how to reduce the volume of data based on encoded compression or pattern based techniques. Finally, data access approaches using different indexing techniques are reviewed. These existing works are categorized into four types, namely key-value pair indexing, criteria-based indexing, AI-based indexing and partition-based indexing. While different types of indexing techniques are based on different criteria to index data attributes, they have the common purpose of supporting data access and search queries, which is also the focus of this thesis, that is, data access and query performance optimisation.

In the next chapter, the research motivation and the thesis problem statement are introduced. A solution framework for IoT streaming data integration is proposed and the main research contributions are discussed.

Chapter 3

Research Motivation, Problem Statement and Solution Framework

In this chapter, a motivating scenario in the domain of streaming data from IoT sources is presented and the general requirements for successful IoT streaming data integration are discussed. After this, the problem statements are detailed and a solution framework for streaming data integration is proposed to address the issues and ensure the general requirements are satisfied.

The chapter is organized as follows. In Section 3.1, the motivation of this research is presented. In Section 3.2, we detail the problem statements of the research and discuss the general requirements to ensure the success of IoT streaming data integration. In Section 3.3, we introduce a solution framework for integrating IoT streaming data from multiple sources.

3.1 Research Motivation

In today's competitive world, businesses derive meaningful insights quickly from datasets to make better business decisions and take faster action. This is more challenging if data presents insights about what is happening in real time. Streaming data is generated by a huge number of applications and IoT devices which have exponentially multiplied and continue to do so, for example, sensor data from machines in manufacturing firms, inventory backlogs at warehouses, activities on e-commerce websites, transactions at

point-of-sale systems in retail stores and many more. Consequently, it is challenging to synchronize data across IoT sources, replicate streaming data, maintain a single source of truth, and find meaningful information that can facilitate the success of the business. Streaming data integration (SDI) is useful for this purpose. However, it takes time and effort to write code and manually gather and integrate data from each system or application copy the data, reformat it, cleanse it, and then ultimately analyze it.

There are several challenging issues in SDI which need to be addressed, such as streaming data synchronization, migration and transformation. Streaming data synchronization is the process of synchronizing similar object instances and data structures across multiple IoT sources. This can be done by de-duplicating the gathered data to remove redundancy. In addition, because streaming data comes from different sources, it is necessary to look at the timing difference of the arrival flows to adjust and synchronize them to avoid timing conflict. This task is referred to as timing alignment in Chapter 2. In SDI, data migration is the process of selecting, preparing, extracting, and transforming data and permanently transferring it from sources to storage or other sources. Data migration reduces the cost of data storage, improves data access performance and improve data availability. Streaming data transformation includes but is not limited to changing data formats, combining data across multiple data sources, filtering or excluding certain data entries from the combined data set, summarizing values across data sets, and so on.

To elaborate on the identified issues, the following scenarios of SDI illustrate some of the issues which need to be addressed.

A scenario is illustrated in Figure 3.1. Each data sample or instance (e.g., $Data1$) has a list of attributes 'A'. The IoT streaming data can originate from multiple sources and can be of different time durations (e.g., duration 'D' can be seconds, minutes or hours). In such situations, the integrated data should be handled with a list of attributes, which is the intersection of all the attributes of different streaming data instances from multiple sources, as illustrated in Figure 3.1. The final duration 'D' represents for the integrated data, which is the lowest one of all 'Ds' (e.g., seconds, minutes or hours) to handle different timing orders. Also, there is a need to mark the starting time when the new streaming data is on the way for real-time integration. An example to take from the work using real streaming datasets from a global manufacturing company that is designed by many machines along with different IoT sensors to achieve safety goals

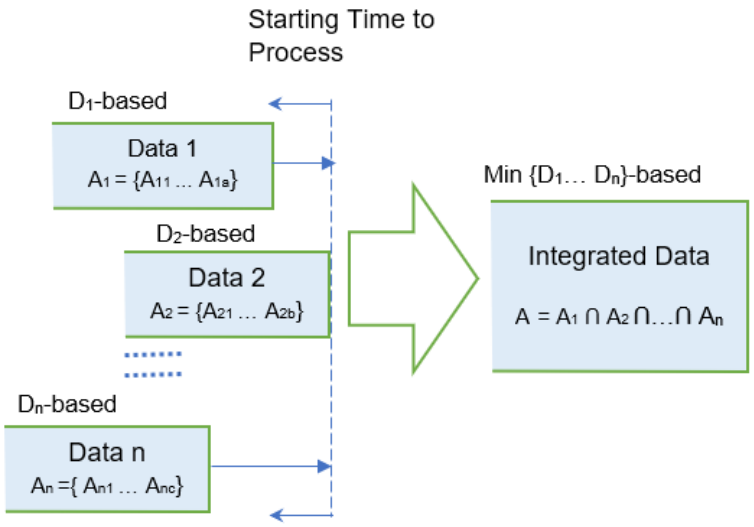


FIGURE 3.1: A Scenario of IoT Streaming Data Integration from Multiple Sources

and improve operational efficiency. These IoT sensors usually generate many different time-series data. The aim of this thesis is to integrate this data for predictive analytics in manufacturing (e.g., to determine the critical sensors that have a significant impact on the long-run maintenance costs). Integrating IoT data from these sensors is important to enable the business to react quickly to predict machine health.

As discussed previously, there is a need to develop a new model to integrate IoT data from multiple sources to address the identified issues. The model should meet the general requirements for successful IoT streaming data integration, which are listed as follows.

(Req.1) ***Process in order***: As IoT sensor data is obtained from multiple sources, they have different timing orders, so integrated data instances need to all have instances in the same time-stamp.

(Req.2) ***De-duplicate data instances***: The time-stamps (e.g., seconds, minutes or hours) of the different data instances may be different. This leads to some identical time-stamps while integrating them from multiple data sources along with different flows. Also, more than one data instance may have the same time-stamp. Overall, raw IoT data from many sensors are usually not ready for any kind of processing or integration. So, there is a need to deal with the data redundancy or duplication issue by merging or aggregating such data instances into one along with the associated attributes of such instances.

(Req.3) ***Integrate in real time***: Another important requirement is to integrate IoT data in real time, as streaming data may change its arrival rate. Also, such a process should be done without any delay in order to capture all the data instances with different flows. Thus, there is a need for a useful data integration technique with the goal of carrying out such processes in a real-time manner along with handling different timing orders and de-duplication.

(Req.4) ***Visualize on-the-fly***: The IoT streaming data with different time-stamps cannot be visualized as a whole. First, there is a need to summarize the expected time duration based on the requirements from the clients. Thus, there is a need for a mechanism to visualize on-the-fly according to such requirements. Also, this can be done either for a single source or integrated data from multiple sources.

(Req.5) ***Extract useful insights***: It is important to gather meaningful insights from data flows across applications. This reduces the waiting time associated with collecting and storing data to obtain useful information to make quick decisions or to conduct further data analysis.

(Req.6) ***Reduce the cost of storage***: In many cases, raw data from streams need to be stored because it can be processed as historical data for later access for the purpose of data analysis, for example, to develop a prediction model. Streaming data storage is an enormous challenge as the volume of data generated is infinite. Hence, it is necessary to reduce the cost of storage for integrated streaming data from various sources.

(Req.7) ***Improve data access performance***: One of the most important issues to address when manipulating data is how to access it efficiently. This requirement relates to the previous ones as data access depends on the data structure and storage. For example, if storage optimization involves a type of compression, it will be challenging to access data and gain insights efficiently and effectively.

In addition to these requirements, other general requirements of data integration are taken into account when collecting, gathering and aggregating data streams from various sources. These requirements include trust building in integrated data, data privacy maintenance by controlling access, an audit capability so that organizations can proactively comply with regulations, data sharing and collaboration between users so that

collectively they can make the most of the data, and so on. In the following section, the problem statements of this thesis are discussed in relation to the identified requirements.

3.2 Problem Statements

From the motivating scenario, it can be seen that the integration of IoT streaming data along with different timing flows/orders has become critically important in many real-life applications. Big data from industries and businesses are usually organized into data lakes or data warehouses for long-term storage and analytics. For example, in our application scenario, a manufacturing company runs many IoT sensors and is globally distributed across a wide variety of regions, hence data storage and real-time integration are vital to this manufacturing company, as it can support decision making for predictive analytics which can reduce subsequent maintenance efforts and costs. Using analytics for predictive maintenance has the potential to save a business millions of dollars by identifying machines which may be faulty before any damage occurs. However, before performing deeper analytics and applying data mining processes, one of the **most important tasks** is to integrate IoT streaming data in real time. Otherwise, data mining has no value as the streaming data continuously comes in various formats from different sources. Thus, a cutting edge approach to such data mining and analytics is one that starts from historical to dynamic streaming data integration.

Depending on the nature of the IoT sensors, these data may have similar or different attributes. For example, in our application scenario, there are many different IoT sensors associated with various types of manufacturing compressors. In addition, data can originate from different sensors in different time intervals (e.g., seconds, minutes). Thus, **the second important task** is to combine or integrate the sensor data in terms of aligning different time-stamps to provide integrated results for the users.

In addition to performing real-time processing and timing synchronization using an alignment solution, it is critical to react and respond to queries from clients accurately in a timely manner by accessing time-series data. Two factors should be considered in improving data access, namely how the data is accessed and how it is processed in real time from multiple sources. Therefore, to enable a quick response to queries from streaming data sources, there should be a mechanism for pre-processing streaming data which is

efficient in terms of storage and indexing. Hence, **one of the important tasks** for SDI is to devise a way to compress integrated streams into smaller versions in an efficient and productive manner. This requires **the following task** in SDI, which is to devise a way to access data effectively. For compressed data, it is necessary to have **a sub-sequence task** to create an index in advance to facilitate data access and query optimization. These tasks are necessary for an SDI model to assist an organisation's decision making and to enable data analysis to be conducted.

In the following, some terminologies with respect to the existing works are discussed and the IoT streaming data integration research is analyzed as a part of the problem statements.

Different Timing Arrival and Frequencies: The information (records, transactions) in streaming data along with different timing flows labelled by either identical or different time-stamps, which are called identities of data instances. These time-stamps have to be aligned and consistent when integrating data from multiple sources, which may come in different time intervals and frequencies. This idea of dealing with different timing conflicts can be found in [45]. Thus, there is a need for the integration to be processed simultaneously, which we call timing alignment.

Data Redundancy: Another issue of streaming data is the possibility of redundant data originating from different IoT sources with the same or different entities. In particular, the streaming data from different IoT sources may contain both identical and non-identical attributes, so the integration of this data should be processed by merging identical entities or by resolving entity resolution. The existing research on this issue [24, 93] does not address IoT streaming data. There is a need to remove redundancy in two ways: by considering the attributes of streaming data from multiple sources and by dealing with data fragments or windows in real time.

Window-Based Integration: The infinite volume of IoT streaming data is a problematic issue when processing and integrating data from multiple sources. Hence, a window-based approach is applied to process the fragments/instances of data as windows to ensure the real-time integration of streaming data. In the literature on processing streaming data [94, 95] several approaches have been proposed, including a windowing technique [41], however, these approaches are not adequate to integrate IoT data from multiple sources in real time. Other research works [38, 96] which are based on the data

integration perspective in the form of query-based integration, have been proposed in the literature. These existing works do not deal with the issues of timing conflict and data redundancy when IoT streaming data is obtained from multiple sources. Thus, there is a need to improve the window-based approach to deal with these issues.

IoT Streaming Data Storage Optimization: A significant problematic issue when dealing with big data is data storage due to the infinitely generated data streams. To overcome this, many researchers have investigated storage optimization by focusing on time-series data compression techniques, for example, Balck et al. with Sprintz [8], Wegener et al. with signal data compression [9] and Diffenderfer et al. with ZFP [10]. A data compression approach reduces the number of bits that represent the data, and it can save storage capacity, speed up file transfer, and decrease the costs of storage hardware and network bandwidth. However, the existing techniques are merely compression approaches, which only focus on saving storage capacity, and there are some limitations in storage ratios and the ability to access data due to compression. There is a need to improve both data storage and data access when data is compressed.

IoT Streaming Compressed Data Access One of the issues in SDI is how to access and query on the integrated data, especially compressed data. Some existing work [90–92] on streaming data indexing make use of sliding windows to index streams for information retrieval but their indices do not apply to compressed data. Thus, there is a need to optimize the access to compressed IoT data and retrieve data insights to answer the users' queries.

Overall, the aim of this thesis is to develop techniques to address the aforementioned issues. In particular, streaming data is generated from various sources and have different data structures, formats and time frequencies. The integrated data needs to be unified into one and it must be consistent, similar to a flow which is generated at regular time intervals from a single source. Therefore, the framework has to ensure not only a unified scheme but also must have the ability to synchronize flows from different sources with different time frequencies. In addition, although the streams are from different sources, they are in the same domain. This leads to the same information interest of applications from sources, and the different streams may have the same or identical attributes. When being integrated, the attributes of streams, both identical and non-identical ones, need

to be merged, aggregated and de-duplicated. The framework also has to ensure the de-duplication process is performed in real time. In the integration process, along with the integrated data, useful information, which is an insight or summarized data from a period of time, needs to extracted and stored so it can be accessed for further analysis. Hence, there is a need for IoT streaming data storage, and this storage should be optimized with a rational compression ratio to store as much data as possible. Finally, the purpose of IoT streaming data integration is to facilitate data access to enable fast responses to common queries so it is critical that the framework has this capability to optimize query performance. To address these issues a solution framework is introduced in the following section.

3.3 A Solution Framework for IoT Streaming Data Integration

In this section, a framework for IoT streaming data integration which meets the requirements outlined in the previous session is introduced. The design of this architecture is inspired by the basic layers in developing data integration systems and addressing the identified problems.

The framework has four layers, namely IoT sources, streaming data processing engine, streaming data optimized storage, and optimized query with indexing schemes, as shown in Figure 6.1. The layers and their components are described in the following.

The first layer of the framework is IoT sources. In practice, there are different types of industrial IoT data sources which facilitate smarter decision making and faster responses across organizations. IoT streaming data sources include web pages, applications, e-services, monitoring devices, sensors, wearables, mobile devices, location beacons, GIS systems and so on. These sources generate data streams in different formats (csv, html, etc.) and structures (structured or semi-structured). The frequency with which the data is generated is also different. For example, data streams are generated from a distributed manufacturing company which includes many machines along with IoT sensors. Some types of sensors generate data every second while other types generate streams every minute. Some IoT devices send data encoded in binary, some send data in a JSON or XML format and some send the data as text.

The second layer is the streaming data processing engine. The operation of this layer is the first main research contribution of the thesis. It plays a role in connecting and extracting data from the source layer and processes the data to deal with the issues of data integration, including, time alignment and de-duplication in real time. In particular, the processing engine manipulates data using two components, namely, *Data Managers* and *Integrator*. While the managers handle IoT streaming data in different formats (e.g., txt, csv, xml, html and so on), and process these data and map them to a unified format, the integrator component collates streaming data from multiple sources and handles data redundancy depending on whether they have identical or different time-stamps (i.e., timing controls). This work and the solution are discussed in Chapter 4.

The third layer is streaming data optimized storage. It comprises two main components, summarized windows and compression storage. The summarized windows are extracted from the second layer. The summarized windows information are insights based on the client's queries. The compression storage stores data with the ability of data searching based on the users' queries. In this layer, compressed data is transformed in window-wise records and index schemes. This layer is structured to facilitate data access for the next layer. The operation of the layer is illustrated in the Chapter 5.

The fourth layer is optimized query with index schemes. This layer is the front-end of a system which the user can access and interact with to receive a response to their queries. It contains index schemes, as a part of query optimization. The schemes are based on the search key from the queries, and to deal with multiple sources, the location of the sources is also the entry of the index. In the case of streaming processing, the index schemes are instantaneously created as long as data are coming, and can be accessed to respond to queries in nearly real time. The optimization is described in Chapter 6.

In conclusion, the components and layers of the framework are the sequence connections of the processes, namely data manipulation by the data manager, time alignment, de-duplication and integration processes by integrator, IoT streaming data compression by compressed storage, and data access by the index schemes. This research contributes to both academia and industry by dealing with the issues of IoT data integration and plays a vital role in today's interconnected environments. This solution framework can be widely applied to many applications and a variety of scenarios, and it can address a

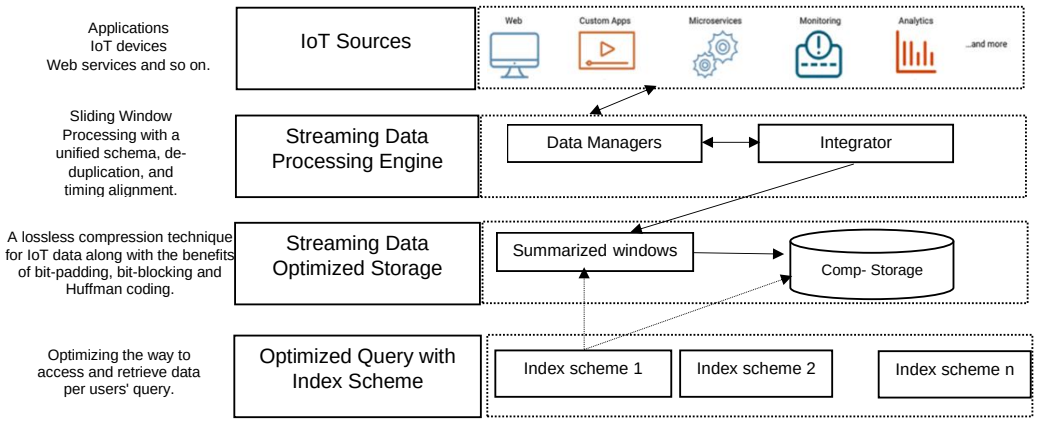


FIGURE 3.2: A Framework of IoT Streaming Data Integration from Multiple Sources

wide range of use cases that rely on key data delivery capabilities. The framework is also a pre-process and uses active metadata supported by machine learning.

3.4 Summary

This chapter presents a motivating scenario that requires the integration of streaming data from multiple sources and its associated requirements. The problem statement addressed in this thesis is discussed and a solution framework is proposed to address the issues.

Firstly, a generic scenario is given using the example of a global manufacturing company. Secondly, the general requirements needed for successful IoT streaming data integration are listed. Then, the problem statement is outlined with an explanation of some of the terminologies in the domain of streaming data processing. Finally, to address the identified issues, a solution framework is introduced.

In the following chapters, the work flows in the framework which are the main research contributions of the thesis are introduced. In particular, Chapter 4 details IoT streaming data processing using the windowing technique; Chapter 5 discusses IoT streaming data compression and storage; and Chapter 6 introduces IoT streaming data indexing and query optimization.

Chapter 4

IoT Streaming Data Processing with Windowing Technique

In the previous chapter, the research motivation and the problem statement of this thesis were introduced and several requirements of streaming data and the issues affecting streaming data integration were discussed. After this, a solution framework of streaming data integration is proposed to deal with these issues. This chapter addresses the issues of different data arrival rates, the frequency of IoT streaming data and different data structures to remove data redundancy and integrate data into a single schema. Hence, this chapter describes the development of a new IoT Streaming Data Integration (ISDI) framework for time-series data. The chapter is organized as follows.

In Section 4.1, a generic Integrator model is introduced for the ISDI proposal with the goal of integrating IoT streaming data from multiple sources. Section 4.2 details the implementation algorithms for ISDI. In practice, IoT data obtained from multiple sources can arrive at different times. This requires an order for processing the streaming data. Furthermore, data from multiple sources can be generated in different frequencies so there is a need to align and sort out the time stamps of such data before integrating them. IoT streaming data integration also requires a unified schema for the integrated results, dealing with different data formats. Typically, streaming data from multiple sources are different in structure (e.g., semi-structured, relational), so before integrating them, there should be a unified schema to combine them into an integrated structure. Data redundancy occurs both in multiple sources or even in a single source itself, and

also presents the integration process. Several techniques are proposed in the literature to identify redundant data [24, 93]. However, in the context of IoT streaming data integration, data duplication is identified from single or multiple sources and resolves and integrates them to satisfy the streaming data flows in real time. In this perspective, several algorithms are introduced, namely a window-based algorithm to integrate IoT streaming data, a timing alignment algorithm to deal with timing conflict issues and a de-duplication algorithm to deal with data redundancy issue. Section 4.3 demonstrates the practicality of the proposed ISDI solution through an empirical evaluation with respect to several experiment setups. The real IoT manufacturing datasets described in Chapter 3 are used to evaluate our approach. Finally, the conclusion and some future research directions are presented in Section 4.4.

4.1 An Approach to IoT Streaming Data Integration

In this section, firstly the IoT Streaming Data Integration (ISDI) approach is formally presented, including preliminary definitions and formal notations to introduce the underlying concepts. Following the formal approach, a generic integrator model for ISDI is introduced.

4.1.1 Formal ISDI Model

In this research, IoT data integration in general is the process of integrating streaming data from multiple sources into an integrated time-series data. Timing alignment is the critical issue that is investigated and addressed in this chapter. There are a number of subsequent issues while integrating IoT streaming data from multiple sources, such as resolving timing conflicts with identical or different time-stamps and dealing with data redundancy and data integration in real time. The main tasks or techniques have been discussed in the existing literature, such as schema matching, entity resolution, duplication and the windowing technique, which can be applied to address these issues.

While schema matching is the process of attribute correspondence among multiple schemas [27], the task of entity resolution is responsible for recognizing the representation of different relevant entities that refer to the same entity. However, in the case of IoT streaming data, real-time integration is a critical issue when applying such a schema matching or

entity resolution task, as it is necessary that an infinite volume of data is processed continuously in real time. In addition, there is a need to align different time-stamps to the streaming data from multiple sources. As such, a unified schema is necessary to deal with such timing alignment. Last but not least, de-duplication also must be performed so that all data instances can be captured from multiple sources without redundant data.

Based on this analysis, the IoT data integration model is formalized as follows, including the key definitions and notations.

Definition 1. (Unified Schema). A unified schema (US) can be defined as the pairs of $< keys, values >$. These pairs can be created based on the local schemas from multiple data sources. The pairs of $< keys, values >$ are used for a unified data representation from multiple IoT sources, where a 'key' corresponds to an attribute that has a corresponding '$value$'.

$$US = \{keys, values\} \tag{4.1}$$

In the above relation,

- US represents a unified data schema,
- $\{keys\}$ represents a list of attributes in the integrated data, and
- $\{values\}$ represents the corresponding data according to the keys from different sources.

Definition 2. (Unique Keys). If $\{keys\}$ represents a set of unique keys according to the keys from multiple data sources and these keys correspond to the relevant values in such data sources, then the following relation can be represented.

$$\{keys\} = set(keys(k)) \tag{4.2}$$

In the above relation,

- $keys(k)$ represents a list of attributes according to the local data sources, and
- $\{keys\}$ represents a set of attributes in the integrated data.

Definition 3. (De-duplication). This refers to the process of mapping duplicate instances (i.e., data records associated with different fields/attributes) from multiple data sources into an integrated one. We identify the identical time stamps of all instances and aggregate them using a relevant mapping function, for example, a mapping function f could be a *avg()* function, which takes the average values of all duplicate/identical data instances. That is, in this research the de-duplication deals with the merging of all identical time-stamp records. For non-identical records, the values are taken from multiple data sources.

$$f = mapping() \tag{4.3}$$

In the above relation,

- $mapping()$ represents a user-defined function in terms of data records (a data record is formed based on the different fields/attributes) from different sources that are aggregated to an integrated format. For each data field in the unified schema, a user-defined mapping function (e.g., average function, $avg()$) can be used to collate data from multiple sources.

Definition 4. (Window-based Integration). This refers to the integration of IoT streaming data from multiple sources, taking individual windows from different sources and integrating them into integrated windows. These windows are defined as different clusters of a number of data records from multiple sources. A window is formed based on a specific duration of a time interval, such as days, weeks or months. According to the data windows from different sources, the timing alignment task is used while utilizing such a window-based integration.

$$Wi(t) = \bigcup_{k=1}^{n} W_k(t) \tag{4.4}$$

In the above relation,

- k is the number of data sources,
- $Wi(t)$ is an integrated window starting at time t with a specific duration, which is called the size of the window, and

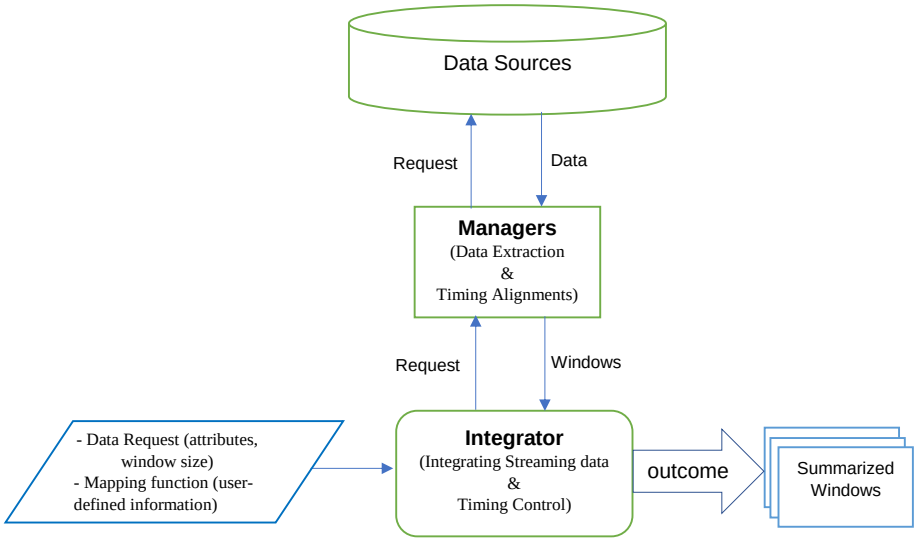


FIGURE 4.1: ISDI Integrator Model for IoT Time-Series Data from Multiple Sources

- $W_k(t)$ is the window of source k starting at time t with a duration of the window size. More details on the duration and window sizes are discussed in the later sections.

4.1.2 A Proposed Generic ISDI Integrator Model

A generic model is presented to integrate IoT streaming data (i.e., time-series data) from multiple sources, called the *ISDI Integrator*. Figure 4.1 illustrates the different components of the ISDI integrator model. The base model that is considered in the research is the basic windowing model to deal with IoT streaming data in real time from multiple sources. The ISDI integrator consists of two different layers: *(i) IoT Data sources and Managers* and *(ii) Integrator*.

4.1.2.1 Data Sources and Managers

The data source component in the first layer handles IoT streaming data in different formats (e.g., TXT, CSV, XML, HTML and so on) while the managers process these data and map them to a unified format. In particular, in this research, the managers are utilized to model a unified schema as the pairs of $< keys, values >$ format. In addition, the managers can pull data from data sources with the timing alignment mechanism. For example, because real-time incoming data from applications usually arrive at different

times, the managers are responsible for handling the differences. A timing alignment algorithm is proposed to work in this layer.

Each kind of IoT data format needs a manager corresponding to extract and manipulate streaming data. It aligns streaming data based on the identical or different time-stamps. Finally, it sends data to the integrator and marks a pointer according to the last processed data/record. This helps the integrator to process data in real time.

4.1.2.2 Generic Integrator

The integrator layer collates streaming data from multiple sources and consequently handles data redundancy according to identical or different time-stamps (i.e., timing controls). Two algorithms, named window-based integration and de-duplication, are proposed to work in this layer. This generic integrator is responsible for controlling the time of incoming multiple IoT data through different managers. It executes relevant queries from end-users through mapping functions and selecting relevant data based on their requirements from the queries as parameters.

For example, the inputs of this integrator are the attributes and the user-identified mapping function according to the data request, including the size of the window that is required to identify the fragments of data processed sequentially from multiple IoT sources. The result of the integrator is stored as summarized windows.

4.2 Implementation Algorithms for ISDI

This section discusses the implementation algorithms for ISDI. Firstly, a generic windowing algorithm is introduced to integrate IoT streaming data from multiple sources. In this research, semi-structured IoT data is considered in different formats (CSV and XML formats). Then, another algorithm is introduced to deal with timing alignment with respect to integrating IoT time-series data from multiple sources. In addition, a de-duplication algorithm is introduced to deal with the data redundancy issue.

As discussed previously, Algorithm 1, *Window-Based Integrator*, has the following inputs: *keys*, *windowSize*, *summarizedFuntion* and *sourceManagers*. The *keys* and the associated $< keys, values >$ pairs can be found based on the users' queries (i.e., requirements). The

Algorithm 1: *Window-Based Integrator* for ISDI

Input: keys, windowSize, summarizedFunction, sourceManagers
Output: integratedWindows

```
1  List iWindows;
2  while startingTime is before the current time do
3      List iRecords;
4      for each sM in sourceMangagers do
5          List recordsList = sM.getRecords(startingTime, windowSize);
6          for each r in recordsList do
7              Record newR = r.getRecord(keys);
8              iRecords.add(newR);
9          end
10     end
11     Window window = new Window(iRecords, startingTime, windowSize);
12     iWindows.add(window);
13     startingTime.plusMillis(windowSize);
14     listOfRecords;
15 end
16 return summarizedFunction.f(iWindows);
```

windowSize is defined by users as it is a time duration for what users want to extract information. The *summarizedFunction* is a mapping function and is defined based on the users' requirements. This function takes different data windows from multiple sources as inputs and returns integrated results as output, *integratedWindows*. Line 5 of Algorithm 1 shows the main contribution of the *sourceManagers*. Each source manager is controlled by a starting time (*startingTime*) and the time interval (*windowSize*). Without considering the timing alignment of source managers, data from different sources can be lost. Most importantly, if we do not align streaming data (using the loop of Lines 4 to 10) while collecting them from multiple sources through such managers, the processing time can be increased (see the experiments in Section 4.3). The data collection according to different source managers through the timing alignment is introduced in Algorithm 2. After collecting data from different sources, the integrator filters data to take only those that have relevant *keys* and adds them into an integrated storage *iRecords* (Lines 7 and 8). The filtering is done according to the de-duplication process introduced later in Algorithm 3. The de-duplication algorithm returns the list of records that are used in this algorithm (see Line 14 in Algorithm 1). The data windows are added into the integrated windows before resetting the starting time for the next integration process. During the process of integrating the windows, the output is based on the function *summarizedFuntion.f()*, as shown in Line 16.

Algorithm 2: *Timing alignment* while Extracting Data from Multiple Sources

Input: startingTime, windowSize
Output: listOfRecords

```
1  List recordsToProcess;
2  int fromIndex = 0; int toIndex = record.size();
3  if lastRecordProcessed != null then
4  |  fromIndex = records.indexOf(lastRecordProcessed);
5  end
6  endTime = startingTime.plusMills(windowSize);
7  for each r in records.subList(fromIndex, toIndex) do
8  |  if lastRecordProcessed between startingTime and endTime then
9  |  |  recordsToProcess.add(r);
10 |  |  lastRecordProcessed = r;
11 |  else
12 |  |  break;
13 |  end
14 end
15 return recordsToProcess;
```

Algorithm 2, *Timing alignment*, is another key contribution of this research. It works along with Algorithm 1, based on the last processed data/record (Line 10 in Algorithm 2). When the last processed data is marked, the source managers only need to scan the data behind it until the relevant windows finish. The *break* functionality in Line 12 helps to cut off all the data beyond the finish time (*endTime*) of the relevant windows. The timing alignment of the associated windows from different sources is controlled by the relevant input *startingTime*, which is usually worked with the integrator (Line 5 in Algorithm 1).

Algorithm 3, *De-duplication*, is used to process data duplication and it is also associated with the integrator model in Algorithm 1. Despite the different time-stamps (data frequencies, such as seconds, minutes, hours or days), the time-stamps also can be identical when collected from multiple sources. The algorithm uses two pointers (see Lines 1 and 3 in Algorithm 3) to count the number of records that are associated with identical time-stamps. The loop condition in Line 5 depends on the Lines from 8 to 12, in which the identical time-stamps are compared with a specified record and the next record. If the time-stamps are different, the loop will stop; otherwise, the pointer will scan the rest of the data to look for other duplicate records. It then executes their sum of values (named "*sov*" in Algorithm 3) to the non-identical attributes and aggregate the values of the identical attributes by an user-defined mapping function (named "$mfov$" in Algorithm 3), see line 16 and 18. Finally, all the duplicate records, which are between the

Algorithm 3: *De-duplication* for ISDI

input: listOfRecords, identicalKey, keyOfTime, keys
Result: remove *duplicate records* and return *list of records*

```
1  int recordsPointer = 0;
2  while recordsPointer < listOfRecords.size() - 1 do
3      int pointer = recordsPointer;
4      boolean continuingCondition;
5      while continuingCondition ∧ pointer < listOfRecords.size() do
6          d1 = listOfRecords.get(pointer).getKey(keyOfTime);
7          d2 = listOfRecords.get(pointer + 1).getKey(keyOfTime);
8          if d1.equals(d2) then
9              pointer += 1 ;
10         else
11             continuingCondition = false;
12         end
13         if recordsPointer < pointer then
14             for each k in keys do
15                 if !k.equals(identicalKey) then
16                     listOfRecords.get(pointer).getValue(sov);
17                 else
18                     listOfRecords.get(pointer).getValue(mfov);
19                 end
20             end
21         else
22         end
23         for  k from (recordsPointer + 1) to pointer do
24             listOfRecords.remove(k);
25         end
26         recordsPointer = pointer + 1;
27     end
28 end
29 return listOfRecords;
```

recordsPointer and the *pointer*, are removed (Line 24 in Algorithm 3).

4.3 Experiment and Evaluation

This section demonstrates the feasibility of the ISDI approach. Firstly, three sets of experiments are conducted in relation to integrating IoT streaming data from multiple sources in different formats (IoT data of CSV and XML formats). Then, the generic ISDI proposal is evaluated by measuring the performance with respect to non-window and window-based approaches.

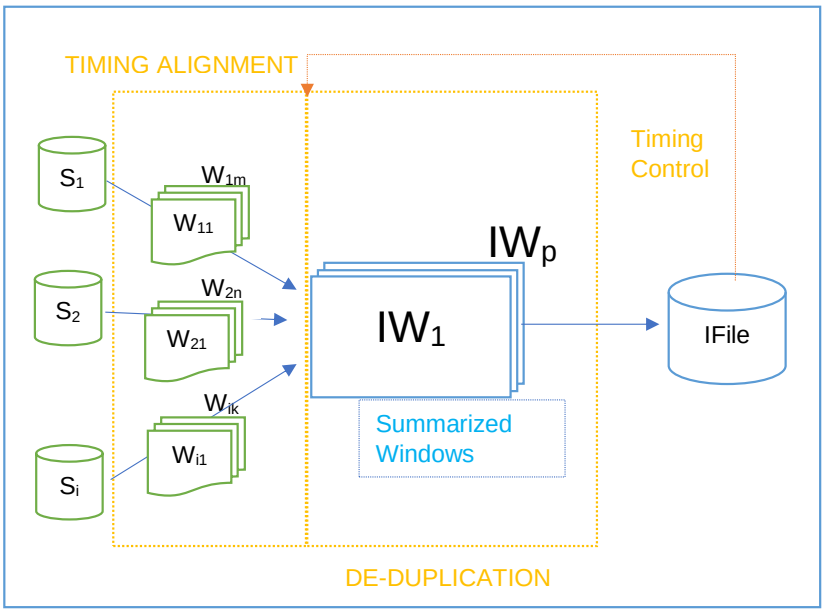


FIGURE 4.2: Different Components of the Generic ISDI Integrator

4.3.1 Experiment Setup #A

In the first set of experiments, a simple window-based approach is used to integrate streaming data from different sources. The left side of Figure 4.3 shows the different components of this simple integrator, including two IoT data sources (S1 and S2), source-based windows (W_i) and summarized windows. The simple integrator also comprises three main steps, namely timing alignment, de-duplication and integration.

Instead of processing the streaming data, based on the window-based approach, all the data files (CSV or XML files) can also be considered at once by following traditional data integration techniques (e.g., schema matching). In the second set of experiments, a non-windowing technique is used for experimental comparison.

The right side of Figure 4.3 shows the generic ISDI integrator, which is the main contribution in this chapter. The third set of experiments based on the proposed ISDI integrator is conducted. Similar to the simple integrator, all the components and steps are the same as in the ISDI integrator, the only difference being that there are different managers to extract streaming data from multiple sources and convert them into a unified schema (i.e., pairs of $< keys, values >$). These managers are also responsible for updating such pairs in real time when new streaming data originate from IoT sources. However, in the

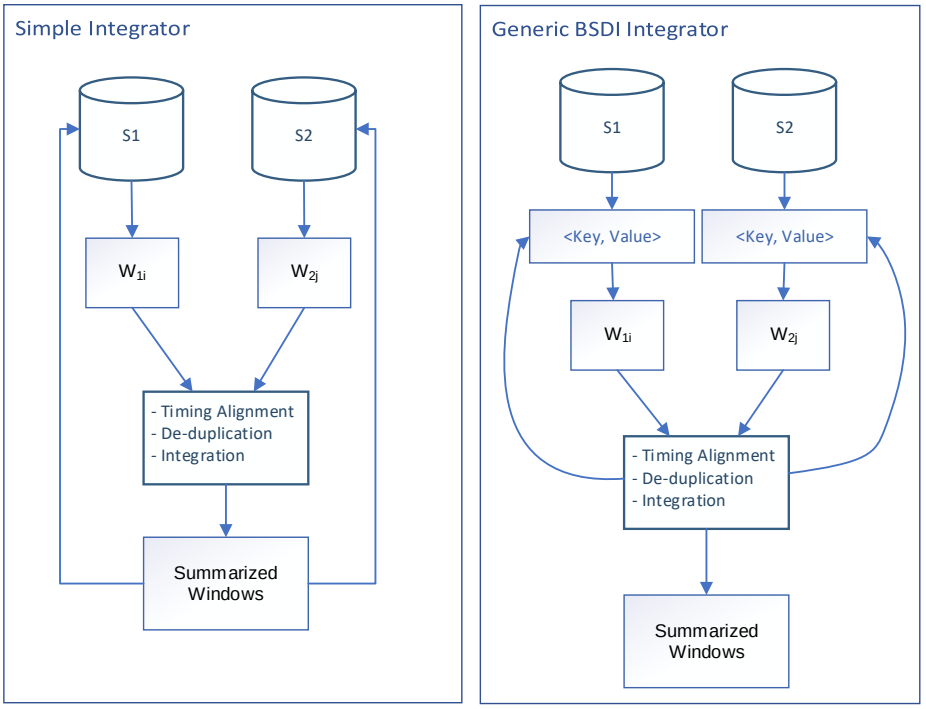


FIGURE 4.3: Simple Integrator (without Managers) vs Generic ISDI Integrator (with Managers)

simple integrator, there is always a check of the IoT sources for any new time-series data according to earlier time-stamps that are already processed.

In these window-based approaches, different fragments of data are taken from multiple sources as windows according to a specified size of the window, and they are then integrated into an integrated window (IW). Figure 4.2 shows the sequences of the necessary tasks for the ISDI integrator. In the simple integrator, windows are usually extracted directly from the data sources and the integrator always looks back to data sources for further windows, whereas in the generic ISDI integrator, windows are extracted and converted to pairs of $< keys, values >$, and the integrator works independently without looking back to data sources. The timing alignment is an important task that is implemented in the source managers, to obtain data windows. When a summarized window (IW) is created by integrating such data windows from multiple sources, it then generates a timing signal and controls the further windows through the source managers. The other operations such as de-duplication and window-based integration are the same in the simple and generic integrator approaches.

The sets of experiments are carried out on two different machines. The development environment of these sets of experiments is Java on NetBeans IDE 8.0.2 and we use a

TABLE 4.1: First Set of Streaming Data (Case 1)

Details	IoT Source 1	IoT Source 2
Duration	256 days	3 days
Number of Records	368,199	259,200
Frequency	record/1-minute	record/1-second
Size	94 MB	62.8 MB

time-based library, named Joda-Time to process the different time series of the windows from multiple data sources.

1. A Windows PC of 3.4 GHz CPU, 4 Core(s) and 8 Logical Processors with 16 GB of physical memory (namely, machine M1).

2. A Windows Laptop of 2.4 GHz Intel Core processor, 2 Core(s) and 4 Logical Processors with 16 GB of physical memory (namely, machine M2).

The above two machine configurations are selected for the following two reasons: (i) to represent the two categories of most widely used machine configurations in this type of applications, and (ii) to evaluate the scalability of the proposed algorithm in different machine configurations

4.3.2 IoT Data Sets

In this chapter, real streaming data sets have been used from a distributed manufacturing company in Australia, which are collected from IoT sensors installed on many pieces of industry machinery. In this section, time-series data of different sizes generated from these sensors are used.

Tables 4.1 and 4.2 depict two sets of IoT data. Table 5.1 contains streaming data from two IoT sources, including second and minute-based time-series data. In particular, IoT Source 2 has 259,200 records of 62.8 MB in size. Table 4.2 also contains streaming data from two IoT sources, including one new IoT Source 3, which has 7,257,600 records of 3.05 GB in size.

TABLE 4.2: Second Set of Streaming Data (Case 2)

Details	IoT Source 1	IoT Source 3
Duration	256 days	84 days
Number of Records	368,199	7,257,600
Frequency	record/1-minute	record/1-second
Size	94 MB	3.05 GB

4.3.3 Experiments #1 and #2

The first two sets of experiments based on the IoT data presented in the earlier section are described in this section. These experiments are carried out using both machines M1 and M2, and applying non-windowing and simple window-based integrator approaches.

4.3.3.1 Comparative Analysis w.r.t. Non-Window and Window-Based Approaches

The experiment results in terms of processing time are shown in Tables 4.3 and 4.4. Using the datasets presented in Table 4.1 (Case 1) and with machine M1, the simple integrator can take 24 seconds to integrate both streaming data from two sources, whereas the non-windowing approach takes 32 seconds to do the same task. The performance can be improved by using more powerful machines, which is also shown in Table 4.3.

It can be observed in Table 4.4 that the non-windowing approach is not able to perform the integration because we consider more than 7 million records which are 3.05 GB in size in this variation (Case 2 in Table 4.2). The main issue is that it cannot read the whole file at a time. For such a huge dataset, the simple integrator reads the data window-by-window and it takes more than 2 minutes to integrate streaming data from two sources. Table 4.4 also demonstrates the performance variation in terms of two machines with different processing powers.

4.3.3.2 Demonstration of Streaming Data

As streaming data can be an infinite time series, both non-window and window-based approaches are compared in terms of the IoT data demonstration. Figures 4.4 and 4.5 illustrate the streaming data from a source file and from the integrated file applying the

TABLE 4.3: 368,199 Records (a record per minute) of 94 MB Size (Case 1)

Machine	**Non-Windowing (NW)**	**Simple Integrator (W1)**
CPU 2.4 GHz (M1)	32 seconds	24 seconds
CPU 3.4 GHz (M2)	18 seconds	14 seconds

TABLE 4.4: 7,257,600 Records (a record per second) of 3.05 GB Size (Case 2)

Machine	**Non-Windowing (NW)**	**Simple Integrator (W1)**
CPU 2.4 GHz (M1)	not-identified	4 minutes 6 seconds
CPU 3.4 GHz (M2)	not-identified	2 minutes 15 seconds

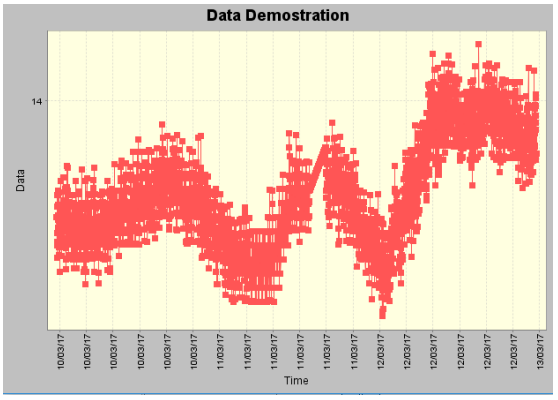


FIGURE 4.4: Minute-based Data from Case 2 with Simple Integrator (W1)

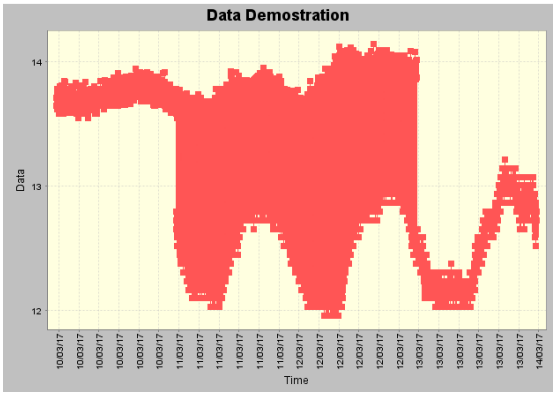


FIGURE 4.5: Integrated Data of Case 2 with Simple Integrator (W1)

simple integrator approach (in the first set of experiments). From the data demonstration, we observe that the data values change significantly in a window.

Figures 4.6 and 4.7 demonstrate another source file and the integrated file using the non-windowing approach. To demonstrate the data clearly, a snapshot is taken of the data in a short time period. For data analytics perspective, it can be observed that the window-based approach is more powerful than the non-window approach. This can also lead to the performance variation in terms of different window sizes which is discussed in the next section (see Experiment #3 in Section 4.3.4.3).

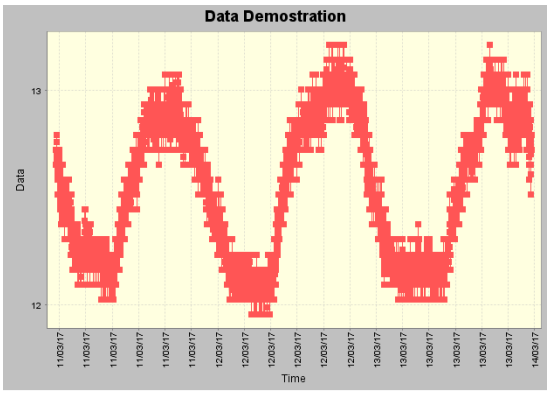


FIGURE 4.6: Second-based Data from Case 1 with Non-Windowing Approach (NW)

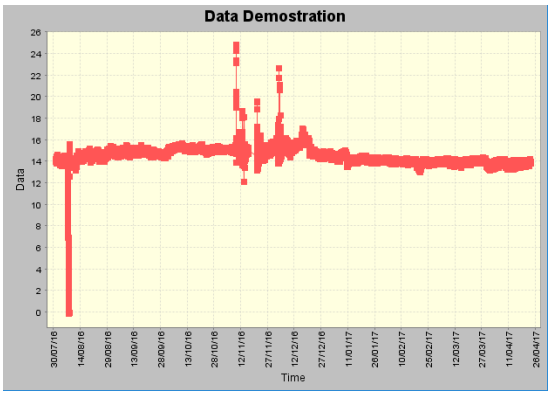


FIGURE 4.7: Integrated Data of Case 1 with Non-Windowing Approach (NW)

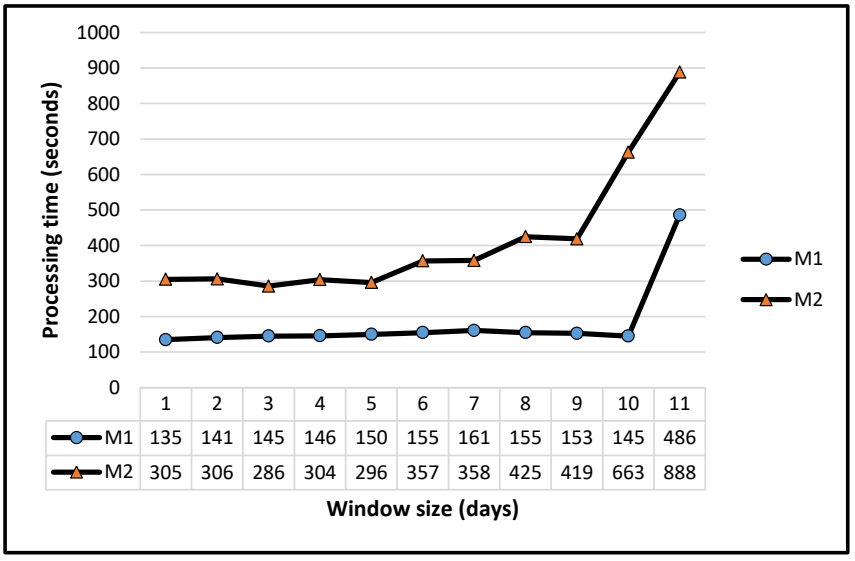
4.3.4 Performance Evaluation

In this section, first the performance variation of the simple integrator approach with respect to different machines is discussed. Then, the performance of the generic ISDI approach is compared to the simple integrator approach. In addition, the performance of the ISDI approach is discussed, including semi-structured streaming data in different formats.

4.3.4.1 Performance w.r.t. Different Machines

In the earlier section, a 1-day window size is considered to measure the processing time, using the simple window-based approach (called the simple integrator). As shown in Table 4.4, it takes 2 minutes and 15 seconds (i.e., 135 seconds) to integrate Case 2 data sources. Figure 4.8 illustrates the processing time of the simple integrator using windows of different sizes.

It can be observed that the variation in processing time is due to different factors: the size of the window and the processing power of the machine. In terms of programming efficiency, these factors can be the complexity of functions and approaches, and/or number of loops. Hence, it is important to revise the window size as a variation in size affects the number of loops in the algorithms. This set of experiments is run on two machines M1 and M2. Figure 4.8 shows that the processing time varies linearly and is almost stable for the window sizes from 1-day to 10-day, using the powerful machine M2. It takes approximately 140 seconds for all such cases. However, the integration time rises



	1	2	3	4	5	6	7	8	9	10	11
M1	135	141	145	146	150	155	161	155	153	145	486
M2	305	306	286	304	296	357	358	425	419	663	888

FIGURE 4.8: Performance w.r.t. Different Window Sizes

dramatically for the window size on 11-day. If the window size is too large, the process is nearly the same as the non-windows process on a huge volume of data. Hence, the performance worsens and the processing time cannot be identified as shown in Table 4.4.

4.3.4.2 Performance w.r.t. Different Window-Based Approaches

The performance of the window-based approach is measured using machine M2 with Case 2 datasets. In particular, based on the task sequences of the simple and generic ISDI integrator illustrated in Figure 4.3, the performance of the window-based approaches is empirically demonstrated. The variation of window sizes from 1-day to 10-day are considered.

Figure 4.9 compares the performance. It can be observed that the proposed ISDI integrator achieves better performance than the other traditional non-window and window-based approaches. The source managers and a unified data schema ($< keys, values >$ pairs) are the main reasons for this variation of performance. Overall, the source managers usually help the ISDI integrator to collate streaming data from multiple sources in real time and achieve better performance.

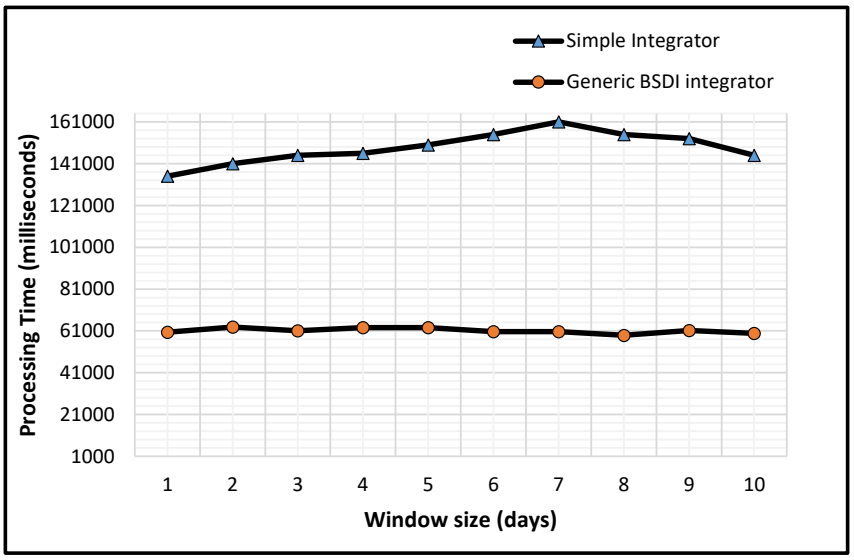


FIGURE 4.9: Simple Integrator vs Generic ISDI Integrator

TABLE 4.5: Third Set of Streaming Data (Case 3)

Details	IoT Source 1	IoT Source 4
Duration	256 days	180 days
Number of Records	368,199	56,570
Frequency	record/1-minute	record/5-minute
Size	94 MB	39.2 MB

4.3.4.3 Experiment #3 and Performance w.r.t. Different Data Formats

The third set of experiments is conducted using another real dataset. Table 4.5 depicts the new streaming dataset in XML format (IoT Source 4). In this case (Case 3), the same datasets (IoT Source 1) in CSV format are used, which are detailed in Tables 4.1 and 4.2. Table 4.5 contains 1-minute and 5-minute based streaming data in CSV and XML formats. In particular, IoT Source 4 has 56.570 records which are 39.2 MB in size (XML format).

In this set of experiments, the performance of the generic ISDI integrator is measured when processing windows. The average processing time per window is examined by varying the sizes of the window and using semi-structured data in different formats. Figure 4.10 shows the performance variation in terms of 1-day to 10-day window sizes and using streaming data in both CSV and XML formats. It can be observed that the performance of the proposed ISDI integrator is almost stable in both cases.

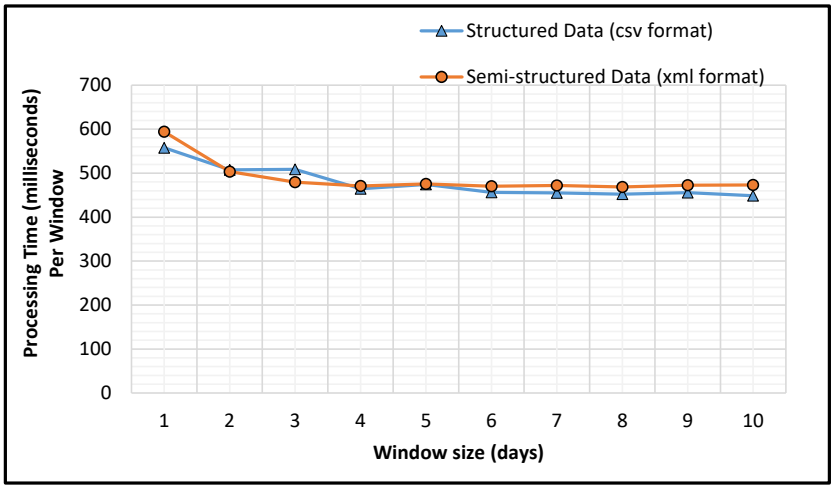


FIGURE 4.10: Performance w.r.t. Semi-structured Data in Different Formats

4.3.5 Experiment Setup #B

Using a new experiment in the Apache Spark environment, the ISDI algorithms are evaluated and their performance is measured by setting different Spark partition sizes.

4.3.5.1 Development Environment on Spark

In the Spark experiment, the 'solo' Java API implementation of the integrator component is replaced by Apache Spark, as shown in Figure 4.11. In this architecture, the IoT data sources and the implemented source-managers from the previous setup are used. The Spark Driver is implemented on the Maven-based build 3.5.4 and Java 8. Although the experiment is performed on a single computer (single node), this 4-core system's work resembles a distributed system more than a traditional single core machine. Note that, the Maven's memory usage needs a special setup (increasing the heap size) to avoid the error "*GC overhead limit exceeded*". Thus, VM arguments are set as $-Xmx2048M$. In the spark implementation, data requested from the Managers is transferred into resilient distributed datasets (RDDs), which are divided into logical partitions and then operated in parallel. The results of the experiments show that a large number of partitions is not an ideal solution. However, it is beyond the scope of this research to investigate the optimal number of partitions to address the issue, but different partition sizes will be applied to measure and compare their performance with the earlier experimental setup. During the task of partitioning, those RDDs are integrated by using the "$reduceByKey$" function and passing an associative function for the following tasks: integration, time

alignment and de-duplication. The experiment outcome is the Summarized Windows (presented in Figure 4.11).

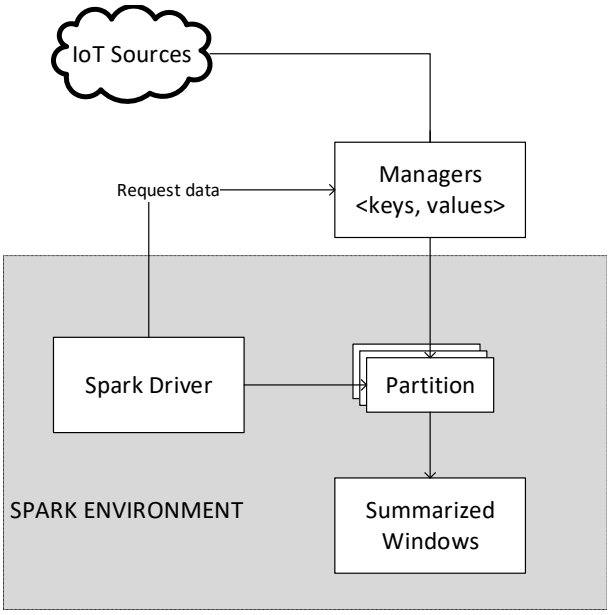


FIGURE 4.11: An Architecture for Integrating IoT Streaming Data from Multiple Sources Using Apache Spark

4.3.5.2 Performance w.r.t. Experimental Setups #A and #B

Different Spark partitions (Setup #B) are selected to measure their performance and compare them with the earlier experiment (Setup #A). The entire processing time with respect to different formats of streaming data is shown in Figure 4.12. The average processing time per window is shown in Figure 4.13. The performance of Setup #B with the partition sizes of 10 and 50 on processing windows and integrating them together is lower than the performance of the earlier setup #A (see Figure 4.12). The results show that when the partition size is 100 and the window sizes is more than 2, the performance of setup #B in the Apache Spark environment is better. However, it is observed when a partition size is between 10 and 50, the outcome gives a warning that *some stages contain a task of very large size, and the maximum recommended task size is smaller than the real ones*. This may reduce the performance of the entire processing. In contrast, the performance of setup #B with a partition size of 100 has all successful integration tasks without any warning of over-size. This produced much better performance. In addition, Figure 4.13 shows that while the average processing time per window increases linearly

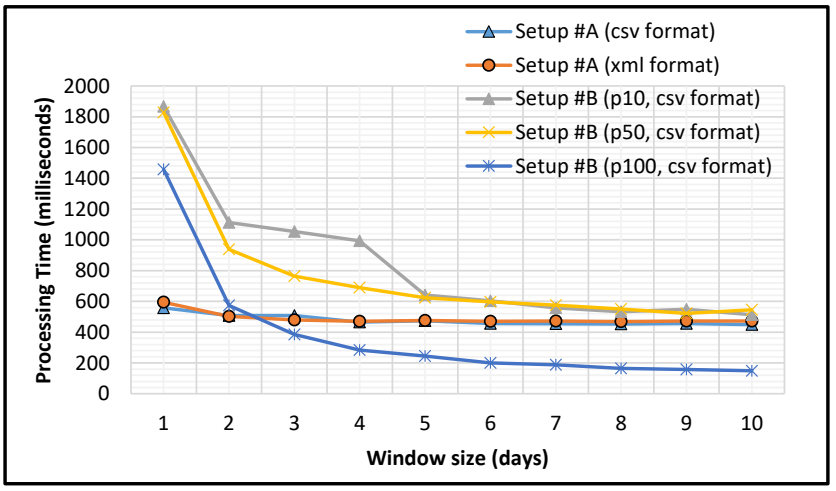


FIGURE 4.12: Setups #A vs #B w.r.t. Processing Time

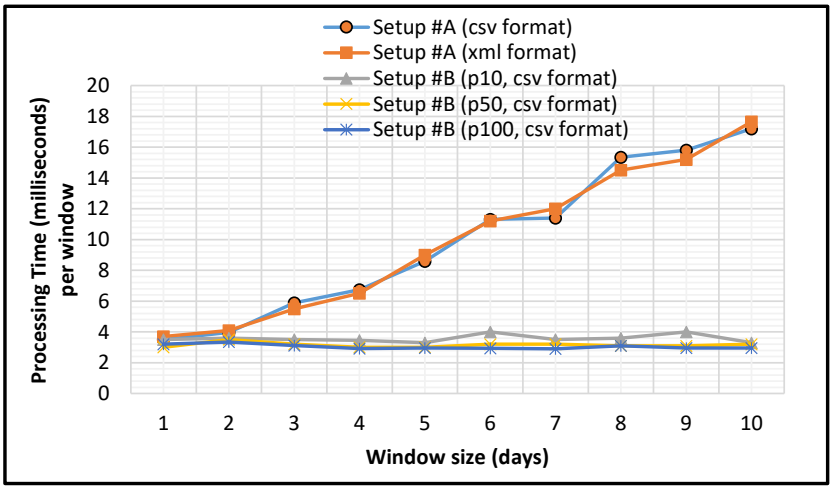


FIGURE 4.13: Setups #A vs #B w.r.t. Processing Time per Window

from 3.7ms to almost 18ms (in the previous setup #A), this current setup #B achieves better performance in the Spark environment (at around 3ms).

4.3.6 Discussion

This section summarizes the experience of implementing a new ISDI framework for IoT streaming data.The proposed ISDI framework is evaluated in response to traditional non-window and window-based approaches. The ISDI approach is tested on different local setups and machines with regard to different semi-structured IoT data in different formats (Setup #A). Also, Apache Spark (Setup #B) is used to improve the ISDI implementation. The performance is finally demonstrated and compared with different partition sizes inn the Apache Spark environment.

To evaluate the performance, in Setup #A, different window sizes are specified with respect to various window-based approaches. The test results in Figure 4.10 show that the average processing time varies from 450 ms to 500 ms approximately, as the number of window sizes changes from 1 day to 10 days. That is, the performance is affected by changing the window sizes. The processing overhead increases at a linear rate due to the increase in the window sizes with respect to the different formats of IoT streaming data. This performance is acceptable for reasonable window sizes with limited computing resources.

In Setup #B, the results shown in Figures 4.12 and 4.13 can be interpreted as follows. First, the larger the partition size, the better performance (i.e., processing time for integrating IoT data from multiple sources). This is because different partition sizes determine the degree of parallelism. For example, a window of 10 days is divided into10 partitions, which means 10 tasks are launched to process the data integration in parallel. In the case of the same window size and 100 partitions, the processing time for integrating IoT data from multiple sources is much lower, as each task executes smaller data. However, if there are too many partition sizes and the data chunks are small, then a small number of data tasks are scheduled. This affects the entire process because of the out of memory issues or excessive overheads in managing many small tasks. Secondly, if the window size is too small, the performance of the entire process using Apache Spark becomes slower (i.e., a longer time is needed to process the data). In such cases, the data is usually divided into too many small-sizes windows. For example, with a 1-day window size, the processing times for partition sizes of 10, 50 and 100 in Setup #B are 1867ms, 1827ms and 1457ms, respectively. However, when using Apache Spark with a partition size of 100 and a higher window size, the performance is better than when the partition size is smaller.

Overall, it can be said that the proposed ISDI framework has an acceptable response time in supporting IoT streaming data integration from multiple sources.

4.4 Summary

In the last few years, much attention has been devoted to developing solutions for integrating IoT streaming data from multiple sources. This creates a major challenge

in selecting the required fragments/windows of streaming data along with identical or different timing alignment from multiple data sources and the subsequent adoption of dynamic changes at regular or different intervals. A key factor in the success of these IoT data integration solutions is the need to cope with the changing nature of the streaming data sources while integrating them on-the-fly in real time. To date, some data integration approaches have been proposed to collate data from multiple sources. However, these existing approaches are not robust enough in today's dynamic and interconnected environments with the goal of integrating IoT time-series data due to the key issues of timing alignment and data duplication while integrating streaming data from multiple IoT sources in real time.

In this chapter, a new window-based approach has been proposed for IoT Streaming Data Integration (ISDI), extending the basic windowing technique. Firstly, a formal ISDI model is introduced, including the key concepts of timing alignment, de-duplication and window-based integration. The fundamental definitions and formal notations are included to introduce these underlying concepts. A generic integrator model for ISDI is then introduced, including different layers to utilize streaming data integration from multiple IoT sources in real time. It is generic for three reasons: i) it can work based on the user-defined mapping functions and consequently answers ad-hoc queries from clients; ii) it can answer ad-hoc queries from users based on useful information including the attributes required by them and the IoT data needed to be summarized; iii) it can deal with various types of streaming data from multiple sources, such as:

- integrating relational data with continuous data streams from environmental sensors, such as temperature, humidity and so on,
- integrating stock exchange data in real time from relevant sources, and
- integrating different medical and health data from multiple sources to create a hospital information system, including relevant time-series data.

The implementation algorithms were presented to realize the preliminary definitions and the generic ISDI model. The applicability of the ISDI approach was demonstrated by conducting different sets of experiments and presenting an empirical comparison of the proposed solution with respect to earlier simple data integration solutions. The results on an experimental setup in the real Spark streaming environment show that the ISDI

approach is effective in practice. Overall, the proposed ISDI approach can be applied to integrate different time-series data from multiple streaming data sources. In today's interconnected world, it is particularly important when the necessary streaming data are obtained from different sources to achieve integrated results for the end-users.

This chapter described the extractions, processing and integration of IoT streaming data from multiple sources. In the next chapter, a mechanism for optimized storage for integrated IoT data is introduced.

Chapter 5

IoT Streaming Data Compression and Storage

In the previous chapter, a new window-based approach is proposed for IoT streaming data integration which is extended from a basic windowing technique. The formal ISDI model is introduced to address the key concepts of timing alignment, de-duplication and window-based integration. The fundamental definitions and formal notations are included to introduce these underlying concepts. A generic integrator model for ISDI is then introduced, including different layers to utilize streaming data integration from multiple IoT sources in real time.

However, it is critical to be able to react and respond to queries from clients accurately in a timely manner by accessing the integrated time-series data. Factors that should be considered in improving data access are how the data are accessed and how the data are processed in real time from multiple sources. Therefore, to adopt the idea of quick response queries from streaming data sources, there should be a mechanism for pre-processing streaming data including storage efficiency and efficient access based on timestamps. A characteristic of streaming data is that it is potentially unbounded in size, so there is a need to improve data compression in relation to storage. In this thesis, time-series data is considered as the collection of data, which is ordered by time, from IoT sources. Also, it is necessary to access the compressed time-series data without decompression, which facilitates much better performance in queries. Hence, in this chapter a framework to integrate time-series data from multiple IoT sources is developed

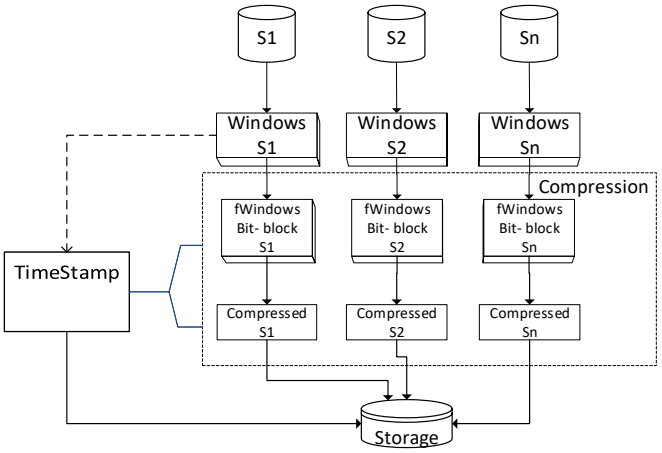


FIGURE 5.1: Compression Model for IoT Data

using compression and utilizing timestamps to access compressed data for streaming data.

The rest of this chapter is organized as follows. A a new compression framework is proposed to integrate and manage IoT streaming data from multiple sources in real-time ins Section 1. A compression mechanism and a data access algorithm based on timestamps are also introduced in this section. In Section 2, several sets of experiments are conducted and the benefits of the proposed framework are demonstrated, including the relevant algorithms and an overall discussion. Finally, the chapter is concluded and summarised in Section 3.

5.1 Proposed Compression Framework

In this section, a compression framework for streaming data from multiple IoT sources is introduced, which comprises two main contributions, time-series data compression and time-series data access based on timestamps. This framework, named ISDI-C, applies a newly designed compression technique to improve the storage space of the ISDI framework in Chapter 4.

Figure 5.1 illustrates the compression model to store data with the ability of data searching based on users' queries. In particular, the scope of queries, as mentioned previously, are timestamp-based requirements. Their responses can be searched from the timestamps of the compressed data in the storage. The model comprises the following steps:

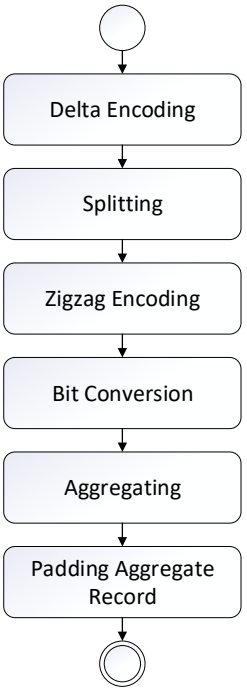


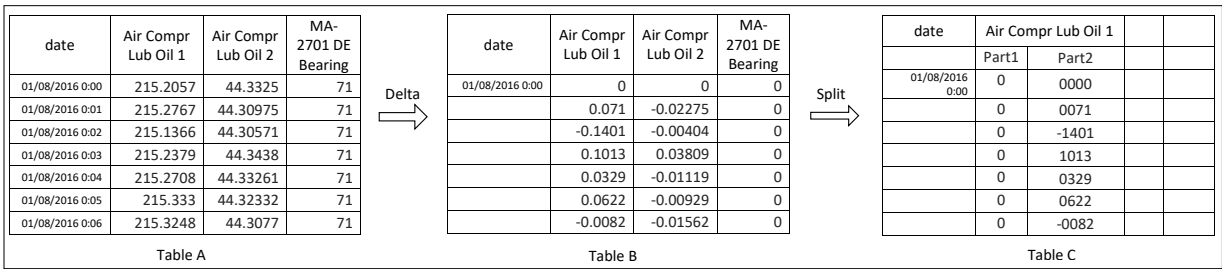
FIGURE 5.2: Real Number BitBlocking Technique Overview

- The model can extract data from multiple sources continuously through the windowing technique which defines every batch of processed data with a size which equals a window length (a period). This step utilizes our previous work on window extractions [97].

- As most IoT data are floating points, they are compressed using a floating-point compression technique, which is an improvement on the integer compression technique (Sprintz). Data traces are stored in the timestamp storage. The data structure is <key,value> pairs, whereas keys are meta-data and store all the attributes of each record. This step is implemented in Algorithm 4 in subsection 3.3.

- The floating-point compressed data can be compressed again by applying a lossless compression (e.g., Huffman compression [98], run-length encoding [99]), which improves the compression ratio to enhance storage capability. The results of this step are presented in our experiment (indicate subsection).

- The timestamps are refined so the can be used as an efficient access mechanism for users' queries.

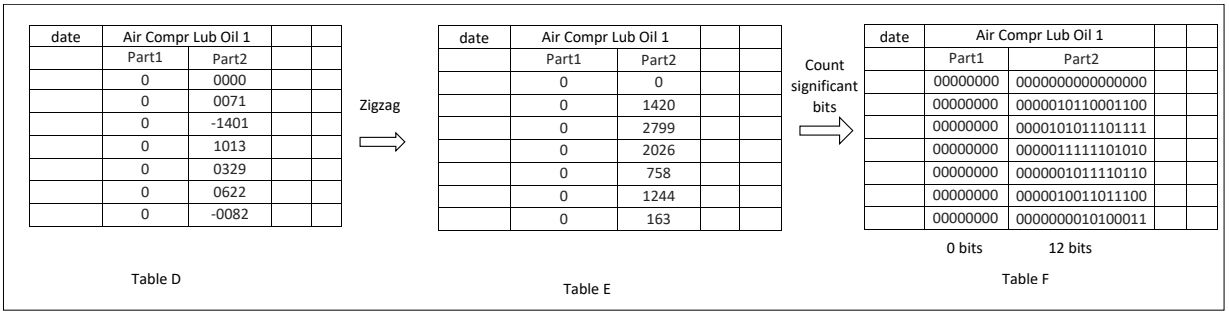


date	Air Compr Lub Oil 1	Air Compr Lub Oil 2	MA-2701 DE Bearing
01/08/2016 0:00	215.2057	44.3325	71
01/08/2016 0:01	215.2767	44.30975	71
01/08/2016 0:02	215.1366	44.30571	71
01/08/2016 0:03	215.2379	44.3438	71
01/08/2016 0:04	215.2708	44.33261	71
01/08/2016 0:05	215.333	44.32332	71
01/08/2016 0:06	215.3248	44.3077	71

date	Air Compr Lub Oil 1	Air Compr Lub Oil 2	MA-2701 DE Bearing
01/08/2016 0:00	0	0	0
	0.071	-0.02275	0
	-0.1401	-0.00404	0
	0.1013	0.03809	0
	0.0329	-0.01119	0
	0.0622	-0.00929	0
	-0.0082	-0.01562	0

date	Air Compr Lub Oil 1	
	Part1	Part2
01/08/2016 0:00	0	0000
	0	0071
	0	-1401
	0	1013
	0	0329
	0	0622
	0	-0082

FIGURE 5.3: Real Number BitBlocking - Phase 1

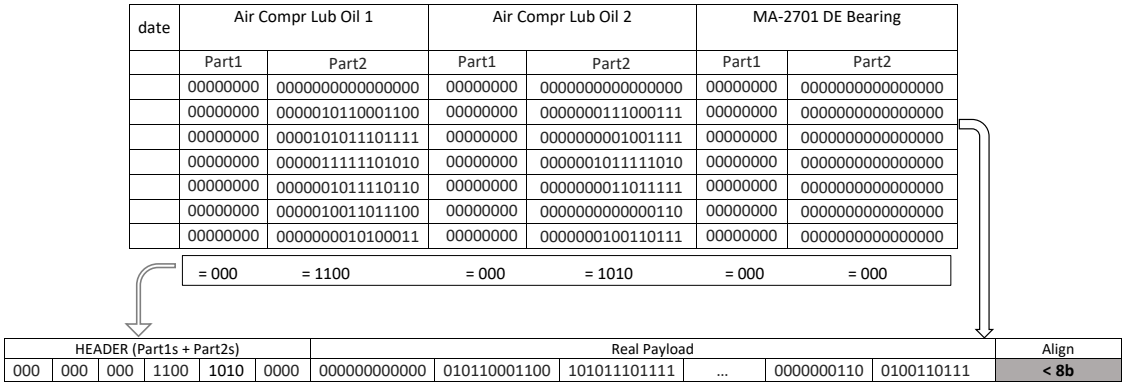


date	Air Compr Lub Oil 1	
	Part1	Part2
	0	0000
	0	0071
	0	-1401
	0	1013
	0	0329
	0	0622
	0	-0082

date	Air Compr Lub Oil 1	
	Part1	Part2
	0	0
	0	1420
	0	2799
	0	2026
	0	758
	0	1244
	0	163

date	Air Compr Lub Oil 1	
	Part1	Part2
	00000000	0000000000000000
	00000000	0000010110001100
	00000000	0000101011101111
	00000000	0000011111101010
	00000000	0000001011110110
	00000000	0000010011011100
	00000000	0000000010100011

FIGURE 5.4: Real Number BitBlocking - Phase 2



date	Air Compr Lub Oil 1		Air Compr Lub Oil 2		MA-2701 DE Bearing	
	Part1	Part2	Part1	Part2	Part1	Part2
	00000000	0000000000000000	00000000	0000000000000000	00000000	0000000000000000
	00000000	0000010110001100	00000000	0000000111000111	00000000	0000000000000000
	00000000	0000101011101111	00000000	0000000001001111	00000000	0000000000000000
	00000000	0000011111101010	00000000	0000001011111010	00000000	0000000000000000
	00000000	0000001011110110	00000000	0000000011011111	00000000	0000000000000000
	00000000	0000010011011100	00000000	0000000000000110	00000000	0000000000000000
	00000000	0000000010100011	00000000	0000000100110111	00000000	0000000000000000

= 000	= 1100	= 000	= 1010	= 000	= 000

HEADER (Part1s + Part2s)						Real Payload						Align
000	000	000	1100	1010	0000	000000000000	010110001100	101011101111	...	0000000110	0100110111	< 8b

FIGURE 5.5: Real Number BitBlocking - Phase 3

5.1.1 The Compression Mechanism for Floating-Point Data

The compression process comprises six steps. In this section, these steps are described through examples, in which each window has 7 records and each record has 3 attributes. When processing a window, we also maintain a reference record, which is the last record of the previous window. An example of a window's data and a reference record is shown in Table A - Figure 5.3.

HEADER (Part1s + Part2s)						Align	Real Payload (Byte-Algned)											
000	000	000	1100	1010	0000	3*8 – 21 = 3b	000000000000	4b	010110001100	4b	101011101111	4b	…	0000000110	4b	0100110111	6b	

FIGURE 5.6: An Example of Traditional Bit Padding

TimeStamp	HEADER (Part1s + Part2s)						Real Payload						Align
125D03A0AC40	000	000	000	1100	1010	0000	000000000000	010110001100	101011101111	…	0000000110	0100110111	<= 8b

FIGURE 5.7: Time-stamp Attachment

Step 1: Delta Encoding.

In this step, for each record in the window, the difference is computed between the reference record's attributes. The result is shown in Table B - Figure 5.3.

It is obvious that the delta operation is reversible. That is, given the full data in Table B, Table A is retrieved in Figure 5.3.

Step 2: Splitting

In this step, instead of working with real numbers, the following process is undertaken.

1. Each entry value is split into two parts: the whole number part and the fractional part.
2. If an entry is negative, the fractional part is made negative.
3. Both parts are stored as integers. Because the second part is stored as an integer, a significance factor is also maintained for its column.

For example, the value -1.0082 is split into 1 (the whole number part) and -0082 (the fractional part), and the fractional part is stored as 82 with a significance factor 4 for its column. The three 'components' allow us to retrieve the original value as $-(1+82\times10^{-4})$.

Applying this operation to Table B, we get the result shown in Table C - Figure 5.3. As shown above, this step is reversible.

Step 3: Zigzag Encoding

In Table D - Figure 5.4, some of the entries are positive and some are negative. It would be convenient to work with positive numbers only. The zigzag operation allows us to do this. The calculation is as follows.

1. If an entry is positive, we double it.
2. If an entry is negative, we double the absolute value and subtract 1 from the result.

Applying this operation to the data in Table D, we get the result shown in Table E - Figure 5.4. It is obvious that this Zigzag step is reversible as well.

Step 4: Bit Conversion

Now, each integer value in Table E is converted into a 16-bit binary representation, and the maximum number of significant bits is counted for each column (not to be confused with the column's significant factor in Step 2).

The result is shown in Table F. Note that this step is reversible.

Step 5: Aggregating

In this step, the data in Table F are taken and put in one record, made up of a series of bits. This aggregate record contains the data of the reference record and all the records in a window.

To describe the construction of this aggregate record, let us take the case where we have:

1. 7 records in the window.
2. Each record has three attributes X, Y and Z.
3. Each field (being a real number) is split into the whole number and the fraction part, denoted by W (part 1) and F (part 2).

The aggregate record has two parts: the header and the data.

The header has the following.

1. Number of significant bits for <attribute X, part W > (which is 0 bits or 000 in the table in Figure 5.5)
2. Number of significant bits for <attribute Y, part W> (which is 000 in the table in Figure 5.5)
3. Number of significant bits for <attribute Z, part W> (which is 000 in the table in Figure 5.5)
4. Number of significant bits for <attribute X, part F> (which is 12 bits or 1100 in the table in Figure 5.5)
5. Number of significant bits for <attribute Y, part F> (which is 1010 in the table in Figure 5.5)
6. Number of significant bits for <attribute Z, part F> (which is 0000 in the table in Figure 5.5)

As for the data segment (Real Payload), the contents consist of the data for record 1, record 2, ... record 7. For record 1, the data is arranged in the following order (logically).

1. <record 1, attribute X, part W>
2. <record 1, attribute Y, part W>
3. <record 1, attribute Z, part W>
4. <record 1, attribute X, part F>
5. <record 1, attribute Y, part F>
6. <record 1, attribute Z, part F>

Similarly, the above six logical sequences will be same for record 2 to record 7.

To save space, however, any part with 0 significant bits (as is evident from the header) can and will be *omitted* from the aggregate record, *without loss of information.* An

example of an aggregate record, to its structure, is presented in the record under the table in Figure 5.5. Note that this aggregation step is reversible. From the aggregate record, we can retrieve data in the table in Figure 5.5.

Step 6: Padding Aggregate Record

There is a need to store the aggregate record as a sequence of bytes. But the last byte may only be partially filled, i.e., some bits are not part of the actual data. We refer to this byte as *the last data byte*.

Figure 5.6 shows an example of aligning bits in a traditional way. In this case, bytes are aligned by adding bits whenever a byte is created without adding more values. This leads to a lot of redundant bits and takes up storage space. In the improvement, as data is managed in the window, the number of bits is controlled and known in a window. Hence, a method for a byte-align mechanism is developed by adding bits only at the end of each window. To take this into account the "partially filled" possibility, one more byte is added to the aggregate record to indicate how many bits in the last data byte are part of the data. This additional record is referred to as *the padding byte*.

A value of 1 in the padding byte means that the first bit of the last data byte is part of the data, a value of 2 means the first two bits are part of the data, etc. A value of 0 means that there is no partially filled record, and all the bits of the last data byte are part of the data. An example of the aggregate record, with partially filled data, is shown in Figure 5.7. This is the actual compressed record that is being stored.

Note that this padding operation is clearly reversible in the sense that from a padded aggregate record, the data can be retrieved, and there is no need to reconstruct the record of Step 5: Aggregating.

As a critical overall feature, because each step is reversible, the whole compression process is reversible, i.e., final aggregate record of Step 6 can be decoded to retrieve the original data of the window in Figure 3.

Algorithm 4: Window-Bit-Block Compression

Input: window, setOfKeys, referencedRecord
Output: integratedWindows

```
1  Let firstPart be a two-dimension array;
2  Let secondPart be a two-dimension array;
3  for i = 1,...,window.getRecords().size() do
4  |  keyIndex = 0;
5  |  refRecord = referencedRecord;
6  |  if i>1 then
7  |  |  refRecord = window.getRecords().get(i-1);
8  |  end
9  |  for each key in setOfKeys do
10 |  |  valueF = r.getValue(key) - refRecord.getValue(key);
11 |  |  Let components be an array with the size 2;
12 |  |  components = BinaryComponents(ValueF);
13 |  |  // 8bits
14 |  |  firstPart[keyIndex][i] = BitsRepresentation(components[0], 8);
15 |  |  // 16bits
16 |  |  secondPart[keyIndex][i] = BitsRepresentation(components[1], 16);
17 |  |  keyIndex ++ ;
18 |  end
19 end
20 for i = 1,...,keyIndex do
21 |  significant1 = NumberOfSignificantBits(firstPart[i]);
22 |  significant2 = NumberOfSignificantBits(secondPart[i]);
23 |  //3bits + 4bits per a header of a value
24 |  hearder1 += BitsRepresentation(significant1,3);
25 |  hearder2 += BitsRepresentation(significant2,4);
26 |  for j = 1,...,w.getRecords().size() do
27 |  |  payLoad1 += BitsRepresentation(firstPart[i][j],significant1);
28 |  |  payLoad2 += BitsRepresentation(secondPart[i][j],significant2);
29 |  end
30 end
31 return BitPadding(header1 + hearder2 + payLoad1 + payLoad2)
```

5.1.2 A Time-series Data Access Technique

In addition to compression, the time-series data access based on timestamps is another main contribution of the model. Assuming a given index structure, there is a need to denote each entry of the index by a timestamp. Hence, in this subsection, an entry of the index is defined as a timestamp and a mechanism is found to attach a timestamp to the window-bit blocks. In the model, the key for a window is a pair of timestamps and window size. As a result, it is trivial to extract the timestamp for each window. In order to attach it to the block, the DateTime format must be transferred to the bit blocks. To

save storage, they are transferred to a hexadecimal and the number of first bytes is fixed to store these timestamps in each block. Figure 5.7 demonstrates an example timestamp attachment.

For example, the timestamps are normalized into the format 'YYYYMMDDhhmmss' which can be parsed into a long variable. They are then converted into a binary or a hexadecimal. Notice that, the attachment of the timestamp is only performed at the first record of each block which is the encoding of a window.

5.1.3 Compression Mechanism with Time-series Data Access Support

Algorithm 4 is used to encode a batch of data into a real number bit block. This algorithm is enhanced from Sprintz (time series compression for the IoT) which is mainly applied for compressing multivariate integer time series. The improvement can be used for real-industry data, and floating-point, and it can ignore the floating-point quantization process similar to other floating-point compression techniques. In particular, the algorithm takes inputs including a set of data (a window), the set of keys or attributes as users' requirements, and a referenced record which is the last record of the previous window. The referenced record supports the delta encoding of the first record of the data-set/window. First, the first parts and the second parts of the floating point values for each attribute are identified after delta encoding (see phase 1 - Figure 5.3). Therefore, there is a a need for two-dimension arrays to store these values. The first dimension is the index of the record, and the second is the index of the attributes or keys. This work is presented in the loop from line 2 to line 19. In this loop, it is first delta-encoded (line 10). Then, the results are split into two integer parts (in front of and after the dot). Each part is transferred into binary; and the sign is moved to the second part if the first part has a zero value. Again, delta encoding is applied for the integer parts and then zigzag encoding for all the values. An example of this implementation is presented in Figure 5.4. The BinaryComponents() in line 12 perform all of these operations; and it transfers the result to the *components* array with a size of two. The first element is converted into an 8-bits representation to become a value of the first part, and the second element is converted into a 16-bits representation which is the value of the second part. The loop from line 20 to line 30 is used to identify all components for the bit block of a window including the headers and the payloads (real values) of the two integer parts. Finally, all

Algorithm 5: Attaching Timestamp to Compressed Data

Input: dataSources, timeIndex, startingTime
Output: time-seriesIndex-base

```
1  Let compressedData be an array with the size equals number or sources
2  for i = 1,...,dataSources.size() do
3  |  compressedData[i] <- Window-Bit-Block();
4  end
5  // TimeAlignment:
6  granularity = getMaxSize(compressedData);
7  for i = 1,...,dataSources.size() do
8  |  dataEntry <- compressdata[i].getData(startingTime, granularity);
9  |  add dataEntry to timeIndex ;
10 |  startingTime += granularity;
11 end
```

the parts are blocked together using a function BitPadding() in line 31. An illustration of these steps is shown in Figure 5.5.

Algorithm 5 is our processing step (Attaching Timestamp to Compressed Data) after the compression step (Figure 5.1). First, data is extracted in compression version from each source. The data is stored in the array *compressedData* (line 1 to line 4). Then, a granularity is identified from all sources in line 6. Lastly, data is obtained with a period of granularity and they are added as entries to the index (Line 7 to line 11).

5.2 Experiment Results

The same real streaming dataset which was used in the experiment in the previous chapter is used for this experiment. This dataset is from a distributed manufacturing company which is designed with many machines along with IoT sensors. Table 5.1 contains streaming data from two IoT sources, namely a small dataset and big dataset within second and minute-based time series data. In particular, the small dataset contains IoT Source 1 with 368,199 records of 94 MB in size and IoT Source 2 with 2592,200 records of 62.8 MB; whereas, the big dataset contains IoT Source 3 and IoT Source 4 with 1,472,800 records and 6,480,000 records which are 376 MB and 1.5 GB in size, respectively. The experiment is performed on both the small and big datasets.

TABLE 5.1: Set of Streaming Data

Details	**IoT Source 1**	**IoT Source 2**	**IoT Source 3**	**IoT Source 4**
Duration	256 days	3 days	1024 days	75 days
No of Records	368,199	259,200	1,472,800	6,480,000
Frequency	record/ min	record/ sec	record/ min	record/ sec
Size	94 MB	62.8 MB	376 MB	1.5 GB

5.2.1 Storage Space Reduction

Table 5.2 illustrates the compression ratios and storage saving abilities of our compression techniques within the timestamp. The formulas for the compression ratio and storage saving are defined as follows.

$$compressionRatio = \frac{sizeUsingCompression}{sizeWithoutCompression} (1)$$

$$storageSaving = 1 - \frac{sizeUsingCompression}{sizeWithoutCompression} (2)$$

In formulas (1) and (2), *sizeUsingCompression* is the size of the data storage needed when the compression technique is applied; *sizeWithoutCompression* is the size of the data storage needed when we implement the ISDI (IoT Streaming Data Integration) model from the previous chapter with the same data. In particular, in ISDI, data is extracted in windows (blocks) and then data is integrated from sources using a user-defined function attached in the integrator (in Figure 5.8), for example, calculating the average temperatures in each window. The storage in this case includes semantic information. In this experiment, data is tested on one source (the data in IoT source 1 in Table 5.1) with different compression levels to investigate the *compressionRatio* and the *storagSaving*.

Table 5.2 provides details on the four compression techniques. The real number *bit blocks* technique, which is the contribution in this chapter, is illustrated in Figure 5.2; *byte transfer* is the technique that transfers all bits into bytes, and *Huffman* coding is an algorithm for performing data compression [100]. These techniques are different levels of this compression. In order to apply *Huffman* coding, bits presentation is transferred into

TABLE 5.2: Compression Ratio Using Different Techniques

Technique	**Compression Ratio** (IoT sources 1 & 2)	**Storage Saving** (IoT sources 1 & 2)	**Compression Ratio** (IoT sources 3 & 4)	**Storage Saving** (IoT sources 3 & 4)
SprintZ [8]	3.68%	96.32%	3.72%	96.28%
Real number bit-blocks	27.24%	72.76%	27.5%	72.5%
Real number bit-blocks & Byte transfer	4.94%	95.06%	5.05%	04.95%
Real number bit-blocks & Huffman	2.12%	97.88%	2.15%	97.85%

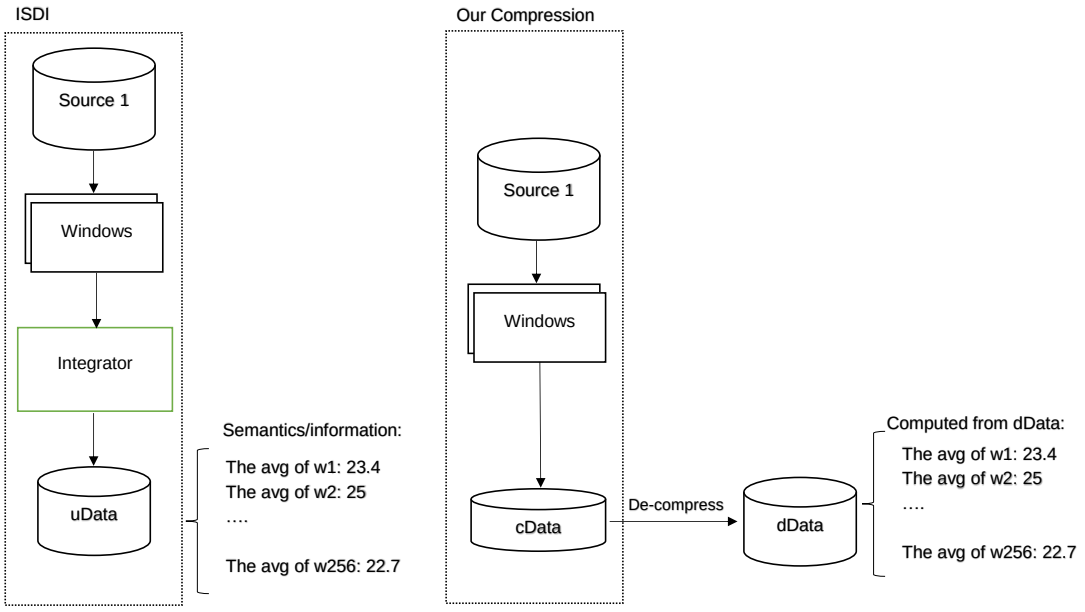


FIGURE 5.8: ISDI vs ISDI-C

symbols (or characters), so *Byte transfer* is always performed before applying *Huffman* coding. Lastly, they are compared with an existing time series compression technique, SprintZ [8]. With small dataset, in the first level of our technique which applies bit blocks

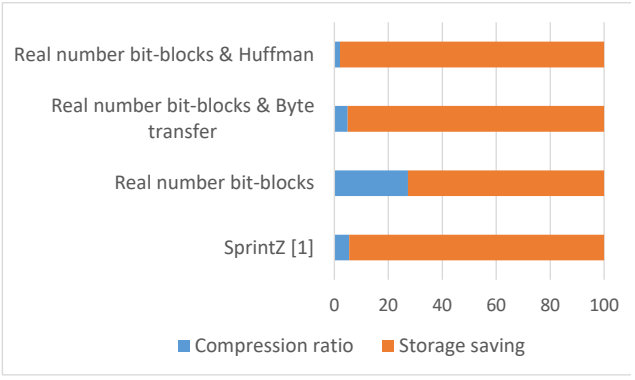


FIGURE 5.9: Compression Ratio Using Different Techniques with a Big Dataset

only, the compression ratio is 27.24% and the storage saving is 72.76%. However, the compression ratio is much better when all the bits of blocks are transferred into bytes, this being only 4.94%. This compression ratio again reduces when Huffman coding is applied, being half the previous level at only 2.12%, and the storage saving is 97.88% which is the best result. In comparison with SprintZ [8], this comprehensive technique gives a better storage saving, 96.32% vs 97.88%. Notice that, with the big dataset, the results are similar to those of the small dataset, for example, 2.15% vs 2.12% of the compression ratio. This shows that the system is scalable. This is because data are processed in partitions and the compression technique is applied for each window into those partitions. The outcome will be the same with a data partition or multiple partitions. For a different observation, the same results are presented in a bar chart in Figure 5.9.

Figure 5.8 illustrates the process of extracting summarized information from the ISDI and this current work, ISDI-C, to obtain data with *sizeWithoutCompression* (uncompressed data - uData) and *sizeUsingCompression* (compressed data - cData), respectively. As previously discussed, uData contains semantic information which is set as the average of the temperatures for each window in the experiment. For the compression version, data is compressed on each window and they are combined into cData. The cData is the decompressed and the average temperature is calculated for each window. The two results are exactly the same, which means semantic information will not be lost when applying the compression. Hence, it can be concluded that the compression technique is a lossless compression and offers a very good compression ratio.

5.2.2 Time-series Data Processing Capability

In this subsection, the capability of on-the-fly processing time-series data is measured using the model (Figure 5.10). The model includes SourceManagers, Compression and Indexing based on timestamps. While SourceManagers or source controllers convert IoT data to different structures (including semi-structured and non-structured data) into key-value pairs (<k,v>) and send these data to the distributed streaming platform Apache Kafka, the compression model and supporting access processes receive data to facilitate a quick response to the clients' queries as previously discussed. In particular, a streaming data processing pipeline is set, which transfers data from IoT sources to the model. A comparison is made of the time to transfer data (each window) from a source to our model (T_w) versus the processing time of our model including compression and building time-series access (T_p); whereas, T_w is measured by the time it takes to convert the data into <key, value> pairs and transfer it through the distributed platform Apache Kafka, and T_p is the elapsed time for the processes of compressing and building the data access. If T_p is less than T_w, it is confirmed that the model satisfies the condition of processing time-series data on the fly.

Figure 5.11 shows processing time T_w and T_p with different volumes of data. Typically, the time to process both steps increases linearly if the volume of data grows as well. In addition, the figure shows that T_p is less than T_w if the data volume is less than 25200 records, which means that the model can process data completely before other data arrives from sources through Kafka. In contrast, if the data volume is too big (> 25200 records), T_p is greater than T_w, so the arriving data must wait for the framework to process its job. However, in processing streaming data, the volume of on-the-fly processed data is normally small enough to run through a streaming pipeline. To conclude, in good conditions, when streaming data are processed as usual, the framework can definitely be deployed in streaming pipeline processing.

In practice, T_w is determined by the slowest rate of incoming data (for example, 1 record per hour), and the critical volume (from 3600 to under 25200 in Figure 5.11) for a window is determined by the fastest rate (for example, 1 record per second) and the slowest rate.

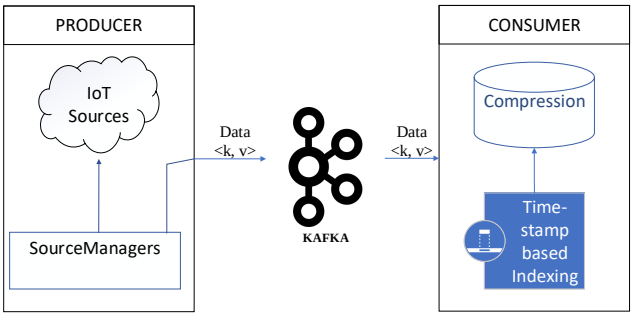
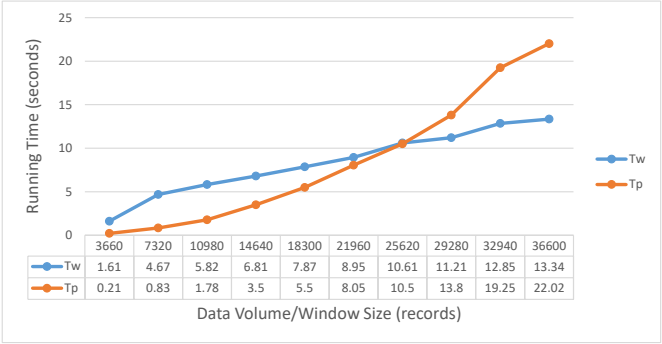


FIGURE 5.10: On-the-fly Processing for Time-series Access Mechanism



	3660	7320	10980	14640	18300	21960	25620	29280	32940	36600
Tw	1.61	4.67	5.82	6.81	7.87	8.95	10.61	11.21	12.85	13.34
Tp	0.21	0.83	1.78	3.5	5.5	8.05	10.5	13.8	19.25	22.02

FIGURE 5.11: On-the-fly Processing Time based on the Volume of Data From Single source

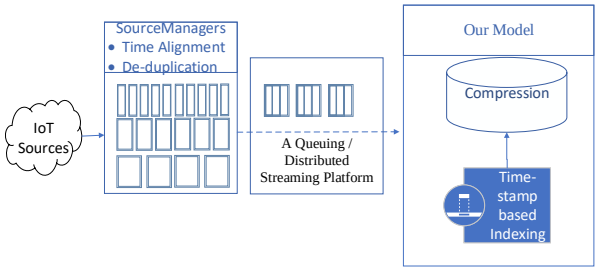
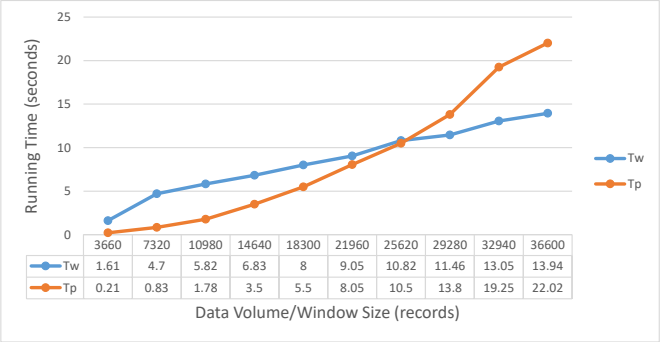


FIGURE 5.12: Time-series Compression and Access Mechanism from Multiple Sources

5.2.3 Time-series Data Integration through Timing Alignments and De-duplication

In this subsection, the experiment which was described in the previous subsection is performed on two sources in the proposed algorithms, time alignment and de-duplication.

Figure 5.12 illustrates our experiment for multiple sources. As discussed in the previous section, windows are extracted from each source. The volume of data which is processed on-the-fly is the size of the window (from one source). The results are shown in Figure 5.11. It is observed that their timing performance greatly depends on the size of the window or the number of records in a window. In addition, there are differences in the



	3660	7320	10980	14640	18300	21960	25620	29280	32940	36600
Tw	1.61	4.7	5.82	6.83	8	9.05	10.82	11.46	13.05	13.94
Tp	0.21	0.83	1.78	3.5	5.5	8.05	10.5	13.8	19.25	22.02

FIGURE 5.13: On-the-fly Processing Time Based on the Volume of Data from 2 Sources (S_1) and S_2

data volume of each window from the sources, for example, there are 60 records in a window with a 1-hour size from source 1, but there are 3600 records from source 2 with the same size (refer to frequency in Table 5.1). Thus, when applying the timing alignment (Algorithm 5), the volume granularity of the on-the-fly processed data is defined as the maximum number of records from different windows of multiple sources. In other words, this granularity depends on the source with the minimum base (to calculate the number of records) and the source with the maximum base (to decide the window size). For example, with source 1 (second-based where data are generated every second) and with source 2 (minute-based) in the framework, the granularity is 60 records corresponding to 1 minute (window size). Hence, in this experiment, different scenarios are discussed which effect volume granularity. They are referred to as an 'extreme case' and a 'realistic case'.

In the 'extreme case', source E_1 is millisecond-based and source E_2 is 24-hour-based (day-based). This means the granularity for on-the-fly processed data is 1000*60*60*24 = 86,400,000 records, which violates the framework's performance as analysed in the previous subsection. A result cannot be generated in this case.

In the 'realistic case', source C_1 is second-based and source C_2 is minute-based (day-based) (the same dataset as in Table 5.1). This means the granularity for on-the-fly processed data is 60 records. The number is too small, so the window size can be defined as bigger (1 hour or 3600 records). In this case, the volume of the on-the-fly processed data is 3600 (from source 1) + 60 (from source 2) = 3660 records. This is a very good condition when deploying the model in a streaming pipeline. The performance in this case is shown in Figure 5.13.

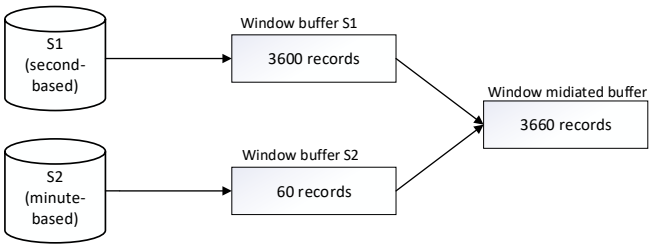


FIGURE 5.14: The Implementation of Integrating Windows from 2 Sources (second-based and minute-based)

5.2.4 Discussion

During the implementation of the proposed compression technique, timestamps are attached so that when constructing an supporting time-series data access, these timestamps can be utilized to obtain data without decompression. Hence, with this framework, the storage of time-series data can be reduced and the data is retrieved in real time. The experiment results in Table 2 demonstrate that the proposed compression technique outperforms the other two techniques by saving 97.88% of storage space compared with the other two techniques, which save 72.76% and 95.06% storage space, respectively.

The result for the 'realistic case' (Figure 5.13) shows that the model can be scalable when integrating data from different sources. This can also be inferred from the scenario and what we have analyzed in subsection 4.2. The performance and the standard threshold (the maximum volume of records that the model can process on-the-fly) are determined by the slowest rate and the fastest rate of incoming data. Hence, in this case, the fastest rate is 3,600 records per hour (second-based), so if the slowest rate is hour-based, it will be a 'realistic case'; and if the slowest rate is 24-hour-based, it will be an 'extreme case'. In addition, in the implementation to integrate data from the sources (Figure 5.14), temporary buffers are used to store the data from different sources and then they are merged and combined into a mediated buffer before transferring them into the model. In this way, in the 'realistic case', because there are not many differences between the volume of records (3,600) in the case of the single source with the fastest rate and the volume of records (3,660) in the case of the integrated ones, the performance of the single-source case with the fastest rate and the performance of the multiple-source case is quite similar (Figure 5.11 vs Figure 5.13).

5.3 Summary

In this chapter, a new compression framework (ISDI-C) for IoT streaming data was proposed. A time-series data compression technique was introduced, in which an access support mechanism is formed according to the timestamps on the compressed data. The proposed compression technique is a lossless compression technique for floating point time-series data, which has the advantage of binary-bit representation, bit-padding and bit-block. The existing technique of bit-padding was improved, and it was optimized by adding less bits to get multiples of 8-bit for bit-block creation.

Several sets of experiments were conducted with a single IoT data source and the capability of our storage reduction was demonstrated. Using the proposed compression technique (based on the real number of bit blocks and Huffman coding), the optimization is 97.88% of the storage space, whereas earlier techniques can only save 95.06% of the storage space at best.

A streaming pipeline was built to demonstrate the applicability of the framework with multiple IoT sources in real time. The results of the experimental setup using the Apache Kafka streaming environment showed that the framework can be effectively used in practice. Overall, the new compression framework can be applied to integrate different time-series data from streaming data sources.

In the next chapter, the time-based access mechanism will be further optimised by constructing indexing structures to efficiently and effectively respond to users' queries. The aim of this optimisation is to delay the decompression of the data as far as possible by getting the smallest sub-set of compressed data to decompress for the search.

Chapter 6

IoT Streaming Data Indexing and Query Optimisation

In the previous chapters, the underlying techniques are discussed to deal with some of the identified issues in streaming data processing (chapter 3), including IoT data alignment from multiple sources with different timing conflicts, different streaming data mapping from multiple sources to a single unified schema and data redundancy and subsequent de-duplication when retrieving data from sources. To address these issues, a framework ISDI is developed in Chapter 4 with a new window-based approach.

A model is then introduced to facilitate data access efficiently with streaming data compression and indexing (Chapter 5). In detail, an existing bit-padding technique is adopted and it is improved as a lossless compression for IoT data. Also, an indexing mechanism based on timestamps is built, which supports access to compressed data without full decompression.

In this chapter, a variety of queries is firstly identified from a scenario, and then a framework, namely ISDI-CI, is developed to optimise the way to access and retrieve these data per the users' queries. This framework reduces the search space by finding the smallest sub-dataset that needs to be searched on to delay the decompression of data as far as possible. Finally, the optimisation is evaluated by conducting experiments on the response for each query using different indexing schemes.

6.1 Proposed Indexing Approach for Query optimisation

As introduced in the previous chapter, this thesis develops a framework to collect and integrate IoT streaming data from multiple sources and store these data under a compressed version. This framework adapts the abilities of both streaming data integration and storage space. However, it is not easy to access this form of data for ad-hoc queries or a variety of queries with different search keys. Therefore, it is necessary to have a mechanism to organize and structure data in an optimal way so that it is effortless to access data and respond to users' queries. In this section, we re-describe the scenario of how data is processed and its pre-processing flow before index schemes are constructed to access data. We also give examples of queries with responses taken from the records in this form. After this, the optimised model with an extended data structure is explained, and the performance of the query responses is improved.

6.1.1 Scenario and Data Representation

In this scenario, we extract IoT streaming data sources and treat this information as useful numerical values or a preview of the potential power of IoT devices. For example, one of the advantages of these values is that healthcare professionals can track patient health in real time and provide on-demand care; manufacturing is able to understand the details of production lines and predict issues before they happen; the automotive industry is able to leverage sensors not only for self-driving but also to provide deeper and real-time insights. Furthermore, it is critical to access historical information from these data to analyse the prediction models. Hence, the previous chapter introduced a compression and indexing model to both store and facilitate data access in nearly real-time. It is noted that streaming data from sources is processed in windows. Because all the information of each window is compressed, a window record contains details of the compressed data, a timestamp and the location of sources, as shown in Table 6.1.

Table 6.1: Old Data Representation

Timestamp - starting time of a window (TS)	Location of the data source (SL)	Detailed data of the window - Compressed (CD)

In the following discussion, examples of the expected queries from the measurement source/sensors are given, and then these are generalized into possible scenarios.

- Example 1: Obtain all the useful/summarized information of a streaming data source, for example, the average temperature for a period of time, say last Sunday. This query is interested in the summarized information but not the entire data set, and it looks for the timestamp of streaming data to get the results. Hence, the expected response is day-windows of data.
- Example 2: Investigate when the average temperature of a machine is equal to or greater than 100 degrees Celsius in a period of time (last year) at several locations. In this case, the summarized information or the user-defined function (UDF) is the average temperature (AvgTemp). To obtain the result, we need to search on the timestamps (all the timestamps for the year 2019), the summarized information (the average temperature greater than 100 Celsius) and the location of sources. The expected response is timestamps.
- Example 3: Investigate how a heartbeat rate changed hourly last month where the maximum heartbeat rate is $>$ 120 bpm. The UDF is to get the maximum heartbeat rate. This is the same as in example 2 with the search keys being the summarized information and timestamps but the expected result is the whole data at that period of time. To do this, we need to decompress the compressed data to get this information.

Using the aforementioned examples, we identify the following two scenarios as below:

1. Scenario 1: Search for all the information of IoT streaming data information from the sources in a period of time.
2. Scenario 2: Search for one or more features of the data in a period of time, for example, properties/features including timestamp, temperature, humidity and other attributes.

These two scenarios generally cover search query situations from applications. Simple queries with a timing search fall into scenario 1. More complicated ones, including ad-hoc

queries, belong to scenario 2. To respond to all the queries from the aforementioned examples or to obtain the results from the two scenarios, we can only search on timestamps, even search keys are non-timestamp attribute in the scenario 2, and then decompress all the data, that is, as many records as we have in that period of time to obtain the information. It is expensive to fully decompress the entire data set for the certain period of time. To improve the query performance, we need to reduce the number of records to be decompressed for many of the common queries. To do this, some features need to be considered as directed access keys in queries, not just timestamps; however these features are compressed and become invisible. To address this issue, useful/summarized information or extra properties of the data can be extracted to support the search. We illustrate our optimisation using the definition of the summarized information and a window record along with the following examples:

Definition 6.1. (Summarized Information). Summarized information (SI) comprises the new attributes/properties extracted from the detailed data of a window. It is functioned by users to extract from the range of data and obtain useful information. For example, a function could be the average or minimum or maximum temperature of the window.

$$\text{SI} = < f1(dt),\ f2(dt),\ \ldots,\ fn(dt) >$$

In the above relation,

- SI represents the summarized/useful information of a window,
- dt represents the detailed data of a window which is all the records in the window, and
- f() represents a user-defined function from the data in the window.

Definition 6.2. (Window Record). A window record (Re) is a record obtained after the compression of a sliding window. Its properties include timestamp, source location, compressed data of the window and attributes from the compressed data.

$$\text{Re} = < \text{TS, SL, CD, SI} >$$

In the above relation,

- TS represents the timestamp which is the starting time of a window,
- SL represents the source location of the window record,
- CD represents the compression of the detailed data of a window, and
- SI = < *f1(CD), f2(CD), ..., fn(CD)* > represents the summarized information of the compressed data in the window.

The following is an example of a window record with some functions defined from specific attributes.

TS	SL	CD	D1	D2	D3	E1	E2	E3	...

where

1. TS is the timestamp of the window record,
2. SL is the data source location,
3. CD is compressed data for the whole window,
4. D1 is related to a property of the data, for example the average temperature of the window,
5. D2 is the minimum temperature of the window,
6. D3 is the maximum temperature of the window,
7. E1 is the average humidity of the window,
8. E2 is the minimum humidity of the window,
9. E3 is the maximum humidity of the window, and
10. etc.

In this example, to facilitate the search queries, each field, with the exception of CD, will be indexed according to a certain scheme, e.g., B-plus tree. Notice that, the index on each field depends on the query. To elaborate on the optimisation, we take an example of a record where the summarized information is minimum temperature and maximum temperature, and a query is "Retrieve records whose time is between t1 and t2 and

whose temperature is between T1 and T2." Without optimisation, we need to search on the timestamp first, and then decompress all the found records before matching their temperatures with the temperature in the query. In contrast, with optimisation, we can filter out the information by taking the records with minimum temperature less than T1 and maximum temperature greater than T2, so the number of records which need to be decompressed are reduced and the search space on the timestamp is smaller as well.

6.1.2 Framework

In the previous subsection, we introduced a new data structure to improve query performance. For greater practicality, we propose a framework to optimise on data access to return results.

The components shown in Figure 6.1 are described as follows:

- Multiple IoT sources from S1, S2 to Sn, for example, sensors, industrial wearables, monitoring system, business applications, and so on. In our research, we focus on numerical values of the generated sources.

- Features are properties extracted from widows including attributes of objects and summarized information. For example, they are timestamps, source locations, average temperature, minimum temperature, maximum temperature (summarized from each window) and so on.

- An index scheme based on the search key from queries, for example, B-Plus tree on timestamps and/or maximum temperature. To deal with multiple sources, the location of sources is also the entry of the index.

- Storage contains compressed data which has been transformed in window-wise records and index schemes. In the case of streaming processing, the index schemes are instantaneously created as long as data are coming, and it can be accessed to respond to queries in nearly real time.

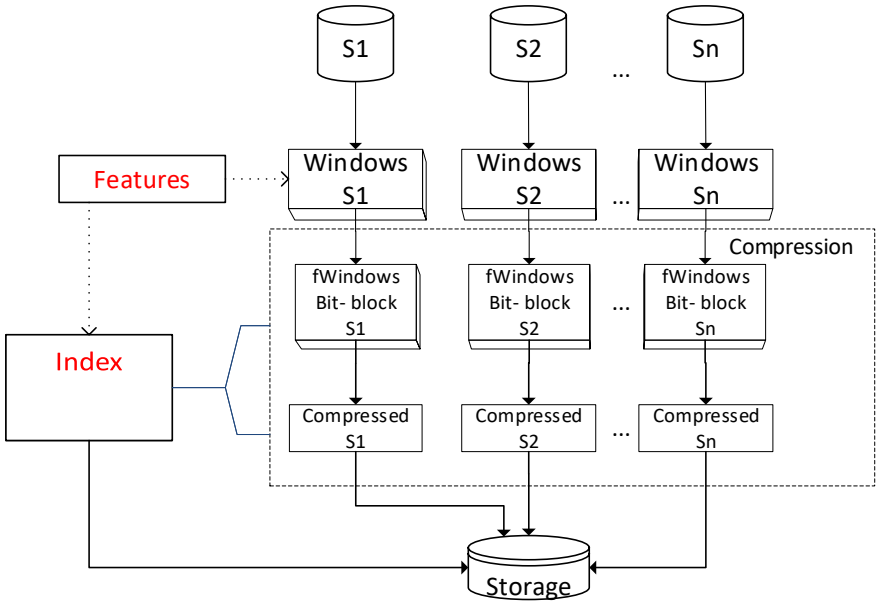


FIGURE 6.1: A Framework of Streaming Data Indexing from Multiple Sources with Optimisation

6.2 Illustrative Queries

In this section, we give some illustrative queries and indicate how these queries can support our claims about query optimisation.

For the purpose of optimisation, the framework needs to evaluate the performance of many common queries or satisfy as many query predicates as it can. Common queries are those which cover as many search keys and different types of results as possible. Therefore, we focus on the types of results and search keys to categorize queries to explore the performance when obtaining a response. Also, the processes that affect query performance need to be taken into account. In our scenario, we need to consider if the process of decompression is necessary for the search or not. In particular, because data is in window-wise records, queries could find either specific single windows or multiple windows either with or without the need to decompress the data. In addition, the query search keys are based on the attributes of records which are timestamps or non-timestamp attributes. Following is a discussion of the features that contribute to our common queries.

Table 6.2 shows the three features which are chosen to make common queries, namely search keys, types of results from the queries, and the requirement of decompression during the search, which is a factor impacting query performance. Search keys are

Table 6.2: Identified Features for Common Queries

Search Keys	**Types of Results**	**Requirement of Decompression**
timestamp, location (e.g. url), non-timestamp attributes (e.g. Temp., Humidity and so on)	single windows, multiple windows	yes, no

the attributes of the queries, so in our scenarios, they are timestamps, locations and non-time attributes. While it is obvious that a timestamp is a typical attribute or key, many locations represent multiple sources of streaming data. The common query search keys also contain other non-timestamp attributes such as temperature, humidity, machine vibrations and so on. In terms of the query results, their types actually depend on the search keys and their ranges, which also contribute to the differences in query performance. For example, a result of a single window from a specific timestamp in the query is retrieved faster than retrieving a result from multiple windows. It is noted that our pre-processed data contains mostly compressed information, with very specific search keys or non-timestamp attributes in queries, so it is possible that the data needs to be decompressed before the real search is performed. The process of decompression is very costly in every search.

As discussed, we categorize the common queries based on the identified features into four groups as follows:

Group 1: retrieve a single window with a specific timestamp.

Group 2: retrieve multiple windows in a period of time.

Group 3: retrieve single windows based on timestamps and non-timestamp attributes during a search which does not require decompression.

Group 4: retrieve single windows based on timestamps and non-timestamp attributes during a search which requires the decompression.

All these types of queries support our claims of query optimisation. By addressing the issues related to searching on compressed data with common queries and comparing them with an existing model, the optimisation of the framework is proved. We describe this task in the following section.

6.3 Experiment

This section demonstrates the ability to search on timestamp and non-timestamp attributes which are the original properties of data or are summarized from streams as useful information per the users' requirement. Some test cases are performed to measure the response time of the identified queries with different time ranges. The test cases are evaluated against a set of general searching queries to determine the effectiveness of the optimized framework in comparison to the earlier non-optimized version. The results of the evaluation show that the optimized framework is able to: i) facilitate quicker data access to the compressed data sources, and ii) reduce search space by removing record candidates which subsequently improve searching performance time.

The experiment data is the outcome of the previous chapter (our previous work [101]). The original data is a set of IoT streaming data, and each record is generated every second. The attributes of the data set are the timestamp and the temperatures of the compressors in the construction industry. We process the data set of 10.8 billion second-based records (each record is generated in one second) into 3 million windows. This means a window of an one-hour size contains the information of 3600 records, and its attributes are timestamp, compressed data and summarized information including average temperature, maximum temperature and minimum temperature of each window (Table 6.3). In the experiment, we specify the identified queries to measure the response time as follows:

Query 1 (*single window*): Retrieve all records in **one specific hour**.

Query 2 (*multiple windows)*: Retrieve all records in **a day or a few hours**.

Query 3 (*single windows, temperature (non-timestamp attributes), timestamp, no-decompression*): Retrieve records whose time is between t1 and t2 and whose **average temperature** is between T1 and T2

Query 4 (*multiple windows, temperature (non-timestamp), timestamp, decompression*): Retrieve records whose time is between t1 and t2 and whose **temperature** is between T1 and T2.

Query 5 (*single windows, temperature (non-timestamp attributes), timestamp, location, no-decompression*): Retrieve records whose timestamp is between t1 and t2 and **maximum temperature** is between T1 and T2 **from a particular set of sources {s1, s2 and s3}**

TABLE 6.3: Set of Streaming Data

original data	**processed data**
10.8 billion records	3 million window records
second-based	one-hour size
timestamps, temperatures and other attributes	timestamps, source location (url), compressed data, summarized information (avgTemp, minTemp, maxTemp)

6.3.1 Response for Query 1 (Selection on specific timestamp)

In this experiment, the performance of different scenarios is compared with the processing time. First, we investigate the performance of a non-optimisation approach, the ISDI-C framework with compression [101], responding to query 1. In this case, we pre-process the streaming data in the windows and extract and summarize the information from the window as its new attributes. Note that, because we search data on ordered timestamps, this is a linear search. The second scenario is applied with optimisation. A B-plus tree is taken as a representation of the index on the timestamp. The tree is constructed with keys being timestamps and values being the rest of each window record. In contrast to the complexity of the previous scenario, the complexity of this search depends on the height of the tree, which is O(log n), where n is number of keys or the number of window records.

Algorithm 6: optimisation Response for Query 1

Input: BPlusTreeOnTimeStamp, timeStamp

Output: records

1 $w \leftarrow BPlusTreeOnTimeStamp.retrieve(timeStamp)$;

2 return decompression($windows$);

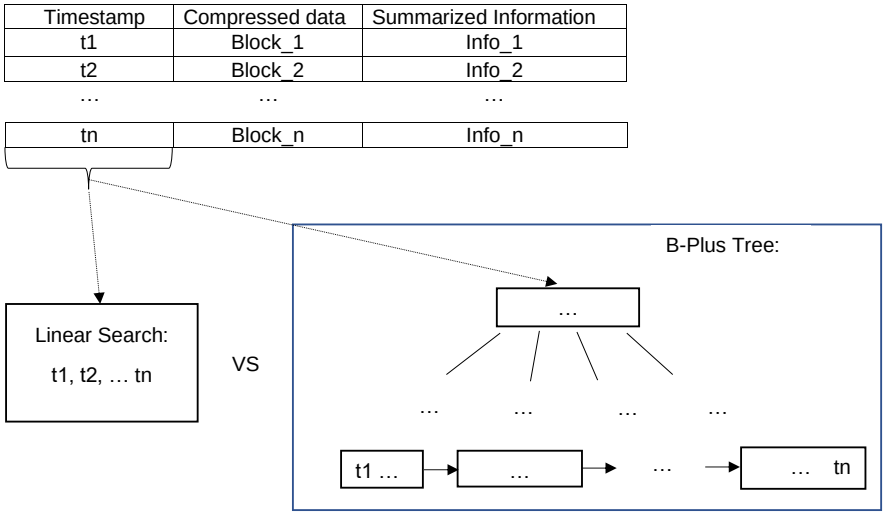


FIGURE 6.2: Scenario 1 vs Scenario 2 for Query 1

Figure 6.2 demonstrates the implementation scenarios to respond to query 1. Scenario 1 is the linear search on timestamps, and scenario 2 is the optimisation search on the B-Plus Tree which is created to index the timestamps of the data. The optimisation algorithm is shown in Algorithm 1. The results are retrieved when searching on the B-plus tree based on a specific timestamp.

The results are shown in Table 6.4. For the first scenario, the average case on the linear search is performed, taking 1,132.973 milliseconds to obtain the response. However, it retrieves data very quick in 1000 testing cases in the scenario 2, with an average of 0.01 milliseconds.

TABLE 6.4: Results for Query 1 (Selection on specific timestamp)

Scenario 1: ISDI-C Framework with Compression [101]	
Average Case	1132.973
Scenario 2: Optimisation with Indexing	
Average time (1000 tests)	0.01

6.3.2 Response for Query 2 (Selection on timestamp interval)

In this experiment, we measure the performance of the same scenarios in the previous sub-section. As the results of the search are larger in volume, we investigate the search with different ranges of time periods in the queries. However, as shown in the result of Query 1 above, there is a big difference between the performance of scenario 1 with average case vs scenario 2, so in this subsection, we take the best case of linear search on the scenario 1 to compare with the average time of our optimised framework.

The optimisation implementation is shown in Algorithm 2. The inputs are the B-Plus tree on the timestamp and a period of time is defined by a pair of timestamps (t1, t2). The first window record can be retrieved by a specific timestamp t1 (line 3), which is the same in Algorithm 1. It is noted that all the leaf nodes of the B-Plus tree are linked-list and those nodes contain the index of the timestamps which are in order, so the other records are traced from the node containing t1 to the node containing t2 (lines 6-10).

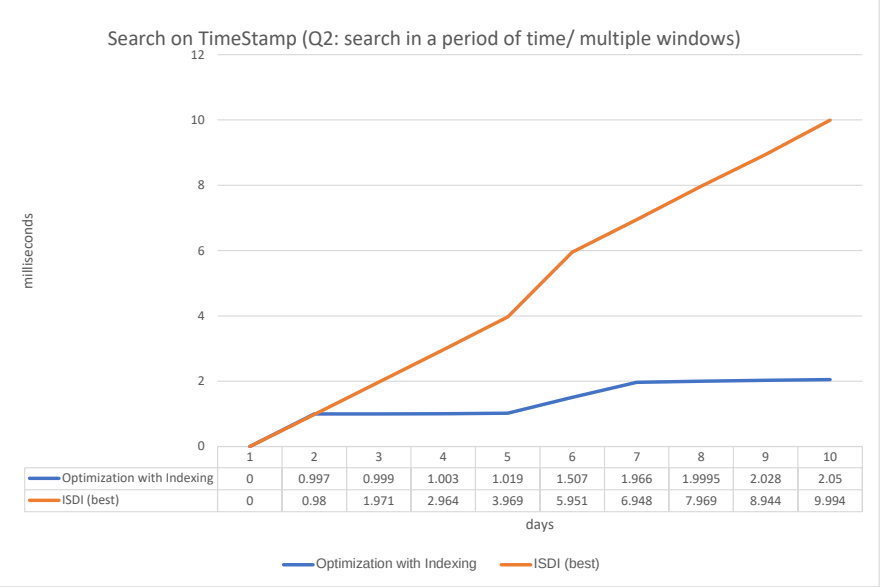
The results are shown in Figure 6.3 and Table 6.5. For both scenarios, we perform test cases with a different range of window sizes, from one to ten days. The ideal case performance in scenario 1 is worse than the performance of the framework with optimisation. In particular there are small differences between the two performances with small window sizes of one and two days, which are 0.001 and 0.997 milliseconds vs 0.002 and 0.98 milliseconds. However, the differences are incrementally larger with larger window sizes. While the processing time is 0.999 ms with windows size of 3 days in scenario 1, it doubles to 1.971 ms with windows of the same size in scenario 2. The difference in the performance is much bigger even much bigger when the window size is 10 days, this being 2.05 ms in scenario 1 vs 9.994 ms in scenario 2 in Table 6.5. Also, in Figure 6.3, the processing time of the ISDI-C framework in scenario 2 is gradually increasing with the increase in the windows size while there is a slow increase in the performance of our optimisation framework with different window sizes.

Algorithm 7: Optimisation Response for Query 2

Input: BPlusTreeOnTimeStamp, t1,t2

Output: records

```
Declare a collection of windows windows;
Declare a timestamp timeStamp;
nodeW ← BPlusTreeOnTimeStamp.retrieve(t1);
windows.add(nodeW.data);
Declare and initialize a temporary node newNode ← nodeW;
while timeStamp < t2 do
    newNode ← newNode.nextNode;
    windows.add(newNode.data);
    timeStamp ← newNode.timeStamp;
end
return decompression(windows);
```



	1	2	3	4	5	6	7	8	9	10
Optimization with Indexing	0	0.997	0.999	1.003	1.019	1.507	1.966	1.9995	2.028	2.05
ISDI (best)	0	0.98	1.971	2.964	3.969	5.951	6.948	7.969	8.944	9.994

FIGURE 6.3: Results for Query 2

6.3.3 Response for Query 3 (Selection on timestamp and aggregated non-timestamp attributes)

In this experiment, we demonstrate the effectiveness of the response to query 3. Again, we compare the performance between the ISDI-C framework (Scenario 1) and the optimisation framework, ISDI-CI, (Scenario 2) with the following different index schemes.

- optimisation index scheme 1: B-plus tree on average temperature

- optimisation index scheme 2: Hash table on average temperature

TABLE 6.5: Results for Query 2 (Selection on timestamp interval)

Scenario 1: ISDI-C Framework with compression [101]										
Days	1	2	3	4	5	6	7	8	9	10
ms	0.002	0.98	1.971	2.964	3.969	5.951	6.948	7.969	8.944	9.994
Scenario 2: optimisation with Indexing										
Days	1	2	3	4	5	6	7	8	9	10
ms	0.001	0.997	0.999	1.003	1.019	1.507	1.966	1.999	2.028	2.05

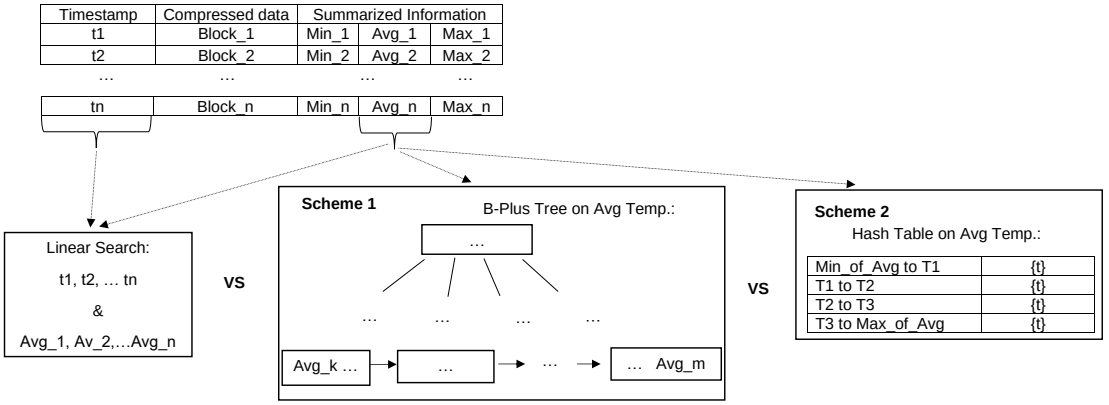


FIGURE 6.4: Index Scheme for Query 3

Figure 6.4 demonstrates the different searching strategies on the timestamps and the average temperatures of a window slide. From the left side at the bottom of the figure, the basic search (scenario 1) is a linear search on the timestamp to match with the other search key, average temperature. The optimisation framework with scheme 1 and 2 are illustrated.

In contrast to searching on the timestamp only, which is a linear search on a sorted collection, the search in scenario 1 on temperature requires the whole data set to be searched. Therefore, when we consider temperature as another search key, we need to optimise the search by constructing an index for this key as mentioned in the previous section. Algorithm 3 demonstrates the optimisation response for query 3. In this case, the implementation is same as in Algorithm 2. However, instead of indexing based on the timestamp, the index schemes in this experiment are based on average temperature.

Table 6.6 shows the performance results of the different strategies. Without optimisation, the search time to obtain result is 466.755 ms while the performances improve significantly by using indexing schemes 1 and 2 , at 74.766 ms and 96.778 ms, respectively.

Algorithm 8: optimisation Response for Query 3

Input: IndexSchemeOnAvgTemp, T1, T2

Output: records

```
1  Declare a collection of windows windows;
2  Declare a variable of average temperature avgTemp;
3  nodeW ← IndexSchemeOnAvgTemp.retrieve(T1);
4  windows.add(nodeW.data);
5  Declare and initialize a temporary node newNode ← nodeW;
6  while avgTemp < T2 do
7  |   newNode ← newNode.nextNode;
8  |   avgTemp ← newNode.avgTemp;
9  |   windows.add(nodeW.data);
10 end
11 return decompression(windows);
```

TABLE 6.6: Results for Query 3 (Selection on timestamp and aggregated non-timestamp attributes)

ISDI	466.755 ms
Idx Scheme 1	74.766 ms
Idx Scheme 2	96.778 ms

6.3.4 Response for Query 4 (Selection on timestamp and actual non-timestamp attributes)

Query 4 is much more complicated than the other queries and proves the ability of our optimised approach. Note that, the search is on the value of the temperature, which is compressed in pre-processing. Hence, without optimisation, we need to decompress all the data in the period of time in the query and then find the temperatures that match the ones from the query. It is costly to decompress; however, with the optimisation schemes, scheme 1 and scheme 2, we filter out some of the data that are out of the query interest based on the summarised information then decompress the rest, hence reducing the cost of decompression significantly. The search strategies are shown in Figure 6.5 and described as follows:

- ISDI: performs a linear search on the timestamp in the query, then decompresses each record to find the temperatures that match T (between T1 and T2) from the query.

- Scheme 1: a B-plus tree on the maximum Temp to retrieve a subset of records Sub1, a B-plus tree on minimum Temp to retrieve a subset of records Sub2. Take the intersection of S1 and S2, and then filter them further on the timestamp.

- Scheme 2: a hash table for the maximum Temp to retrieve the subset of records Sub1, a B-plus tree on the minimum Temp to retrieve a subset of records Sub2. Take the intersection of S1 and S2, and then filter them further on the timestamp.

Algorithm 4 demonstrates the implementation of the search on index schemes. The results windowsMin and windowsMax are retrieved by the index scheme on minimum temperature and maximum temperature, respectively. In this case, Algorithm 3 can be implemented with the two arguments being a pair of lower bound timestamp (minT or T2) and upper bound timestamp (T1 or maxT) in line 1 and 2. After filtering all the records which need to be compressed by taking the intersection of windowsMin and windowsMax (line 4), the results are retrieved by filtering further through T1 and T2 (line 5 to 7).

Algorithm 9: optimisation Response for Query 4

Input: IndexSchemeOnMax, IndexSchemeOnMin, T1,T2
Output: records

```
1 windowsMin ← Algorithm3(IndexSchemeOnMin, minT, T1);
2 windowsMax ← Algorithm3(IndexSchemeOnMax, T2, maxT);
3 windows ← windowsMin ∩ windowsMax;
4 decompressedRecords ← decompression(windows);
5 if decompressedRecords.temperature between (T1 and T2) then
6 |   records ← decompressedRecords;
7 end
8 return records;
```

Table 6.7 shows the results of the query using different strategies. With the same data used in previous queries (3 million windows), there is a memory error for the linear search because of being overloading in decompression processing. However, the outcome of schemes 1 and 2 are delivered, and it is seen that the performance of a tree based index is better than the performance of hashing techniques, these being 452573.961 ms and 817199.855 ms, respectively. When the data set is reduced to 1 million windows, the results are much better, the performance of non-optimisation, optimisation with scheme 1 and optimisation with scheme 2 being respectively 1,044,397 ms, 147,495 ms and 300,410 ms.

The results of this query illustrate the benefit of our optimisation. While a traditional search requires full decompression of the entire data set to find the exact required temperature, the optimisation approach scans and filters the candidate records or a subset of

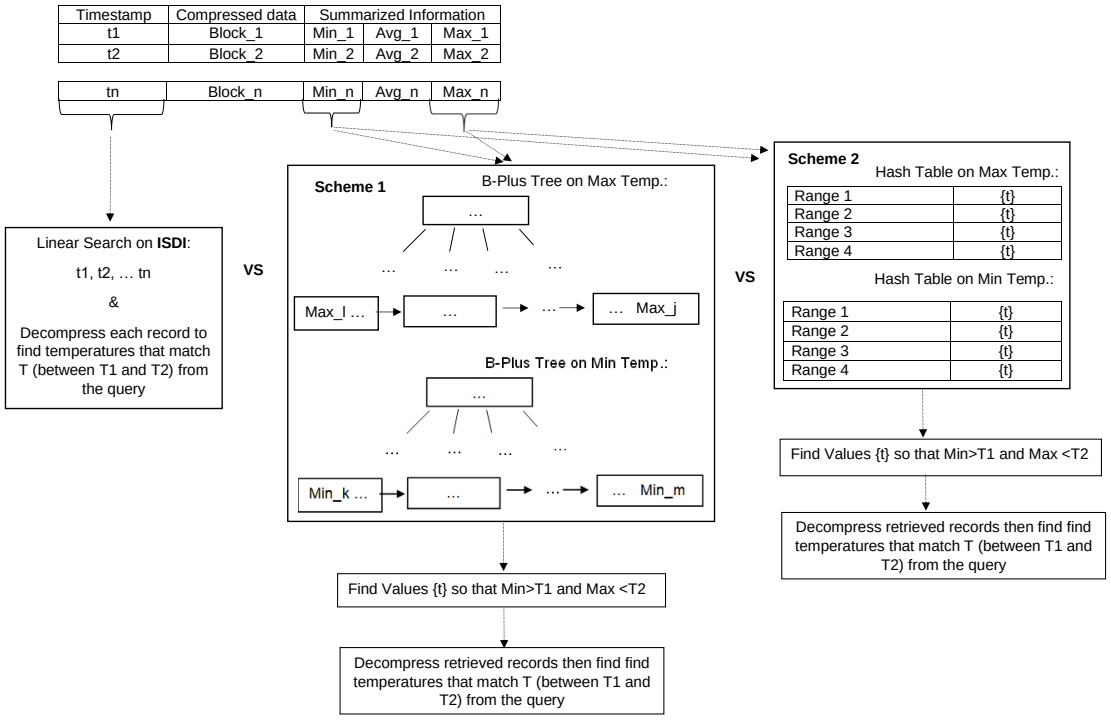


FIGURE 6.5: Index Scheme for Query 4

the data that mostly satisfies the predicate of the query. With the optimised approach, not only are fewer decompression processes performed, the search space is also reduced.

TABLE 6.7: Results for Query 4 (Selection on timestamp and actual non-timestamp attributes)

	3 million windows	1 million windows
Linear Search on **ISDI**	Memory Error	1044397 ms
Idx Scheme 1	452573.961 ms	147495 ms
Idx Scheme 2	817199.855 ms	300410 ms

6.3.5 Response for Query 5 (Selection from multiple sources)

Query 5 is performed to illustrate its ability to search data on multiple sources so the locations of the sources are taken into account. The demonstration of the implementation for the query is shown in Figure 6.6 and described as follows:

- ISDI: Linear search on location and timestamp to retrieve the maximum temperatures that match the search keys in the query.

- Scheme 1: B-plus tree on the maximum Temp to retrieve the initial subset of records, and then filters them further on timestamp and location.

- Scheme 2: A hash table for maximum Temp to retrieve the initial subset of the records and then filters them further on timestamp and location.

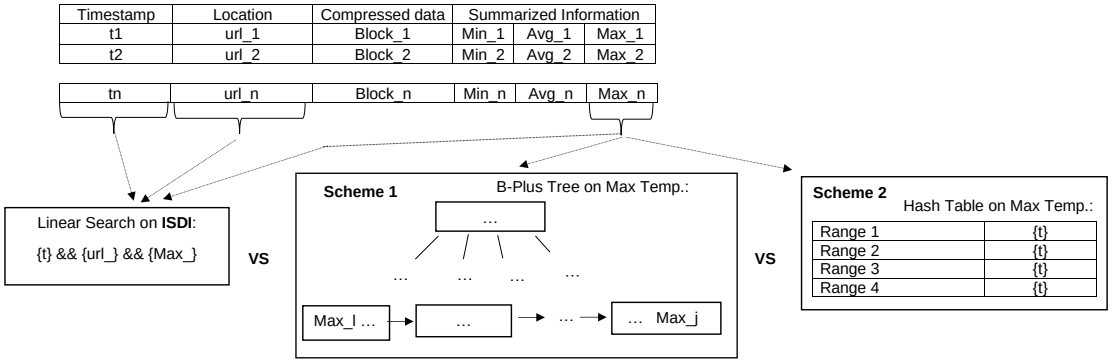


FIGURE 6.6: Index Scheme for Query 5

TABLE 6.8: Results for Query 5 (Selection from multiple sources)

ISDI	480.560 ms
Idx Scheme 1	80.235 ms
Idx Scheme 2	102.560 ms

Table 6.8 shows the running time to respond to query 5. Without optimisation, it takes 480.560 milliseconds to perform the query but the results are delivered in 80.235 milliseconds and 102.560 milliseconds with the optimisation schemes.

6.4 Summary

In this chapter, we describe the improvement of the ISDI-C framework for streaming data integration into ISDI-CI with indexing strategies. We first analyse the scenario of streaming data, which is pre-processed in a window-wise compression version and useful information is attached to facilitate later queries. Then, an optimisation framework with indexing schemes from multiple sources is introduced, and some common and illustrative queries are discussed to prove our claim of optimisation. Finally, we perform experiments on five queries to illustrate the ability of optimisation framework on a wide range of different query scenarios. The results prove that the optimisation (ISDI-CI) is much better than the existing framework, ISDI.

Chapter 7

Conclusion and Future Research

The aim of this thesis is to address the challenges associated with streaming data integration by developing a new framework for integrating IoT streaming data from multiple sources and facilitating quick data access for future analysis purposes. The thesis first reviews the existing works related to the issues of streaming data processing and IoT streaming data integration. The thesis then discusses the research motivation and problem statement. The limitations of the existing related research are discussed, followed by details on the general requirements for successful IoT streaming data integration and the problem statement, after which a novel framework of streaming data integration is introduced. The framework incorporates the data integration process, data storage optimization and data access optimization to provide an entire picture of information flow from multiple sources to respond to clients' queries.

In this chapter, firstly the major contributions are summarized, then some interesting research directions are discussed, which may be explored further in the future.

7.1 Contributions

In this thesis, a new streaming data integration framework has been introduced for data analysis. It includes a general and extensible model to represent and reason about IoT streaming data integration from multiple sources, a compression approach to facilitate storage optimization, and a data access model with index schemes. This thesis makes a number of research contributions regarding the design and implementation of this

novel streaming data integration framework for both industry and academic areas. In the following, the research contributions in relation to IoT streaming data integration processes are summarized.

- **IoT streaming data integration model (ISDI integrator model)**

  For the streaming data integration framework, the ISDI integrator model adopts and extends basic windowing processing on streaming data. It is a generic integrator model which has two components, namely 'managers' and 'integrator'. The former component processes data from sources and maps them into a unified format, and the latter component collates the streaming data, handles data redundancy and is responsible for controlling the time of the incoming multiple IoT data through different managers. In this work, the key concepts of timing alignment, de-duplication and window-based integration were introduced along with their implementation algorithms. Several sets of experiments were conducted and an empirical comparison of the model with the existing work on data integration were presented to demonstrate the applicability of the ISDI approach. The results obtained in an experiment setup in the Apache Spark streaming environment showed that this approach can be effectively used in practice. Overall, the proposed approach can be applied to integrate different time-series data from multiple streaming data sources.

- **Integration of IoT streaming data with storage optimization**

  In this work, a new version framework for IoT streaming data was proposed. A time-series data compression technique was introduced, in which an index is formed according to the time-stamps on the compressed data. The proposed compression technique was a lossless compression technique for floating point time-series data, which has the advantage of binary-bit representation, bit-padding and bit-block. An existing technique of bit-padding was extended and optimized the storage ratio by adding less bits to get multiples of 8-bit for bit-block creation. Several sets of experiments were conducted with a single IoT data source and demonstrated the storage reduction capability. Using the proposed approach (based on the real number of bit blocks and Huffman coding), the framework optimised 97.88% of the storage space, whereas earlier techniques can only save 95.06% storage space at best. A streaming pipeline was also built to demonstrate the applicability of the

framework with multiple IoT sources in real time. The results of the experiment setup using the Apache Kafka streaming environment showed that the framework can be effectively used in practice. Overall, the novel framework can be applied to integrate different time-series data from streaming data sources.

- **IoT Streaming Data Indexing and Query Optimization**

  This thesis improves streaming data integration using indexing strategies. Firstly, the scenario of streaming data was analyzed, which is pre-processed in a window-wise compression version and useful information is attached to facilitate user queries. Then, an optimization framework with indexing schemes from multiple sources was introduced, and some common and illustrative queries were presented for better optimization. A set of experiments on five queries was performed to illustrate the ability of the optimization framework on a wide range of scenarios. The results proved that the optimization is much better than the existing ISDI framework.

In summary, the proposed framework has addressed the key requirements of streaming data processing and integration, i.e., addressing the timing conflicts of data streams by time alignment; reducing the redundancy and merging information into data instances by de-duplication; integrating streaming data and performing those operations using window-based processing; optimizing data storage by lossless compression; and facilitating data access by structuring data compression and attaching index schemes efficiently. These research contributions improved data access from multiple sources in terms of integrating IoT streaming data, storing the data and indexing them efficiently and effectively. In the following section, suggestions for future work are discussed with the purpose of improving and enhancing the IoT streaming data integration framework more comprehensively.

7.2 Future Research

Much work can be done to further enhance the integration of IoT streaming data from multiple sources. The following several interesting research directions might be explored in the future.

- **Query Optimization with a Focus on Query Patterns**

  With the convergence of physical-digital systems, an increasing number of data fields have been generated and are available for industry and research use. It is a significant challenge to effectively exploit many data fields and manage the performance. In this context, the strategy of IoT data indexing is introduced, which is based on various query patterns which can subsequently result in better optimization, such as artificial intelligence (AI) indexing approaches [75]. In particular, AI indexing approaches observe patterns and categorize objects (i.e., data fields) with similar traits. These approaches also support the identification of patterns between terms in an unstructured data set. Hence, in future work, these kinds of indexing approaches can be investigated to improve the mass volume of data fields from multiple IoT sources.

- **Privacy Issue in Integrating IoT Streaming Data from Multiple Sources**

  A significant research problem is how to preserve privacy and how much is privacy worth when data integration is performed by utilizing multiple IoT streaming sources. Future research can explore different privacy control algorithms and policies to address this issue, such as differential privacy and k-anonymity [102], and subsequently investigate the boundary between privacy and utility in IoT data integration from multiple sources.

- **Capturing Semantic Relationships between Multiple Data Sources**

  In this thesis, at the early stage of extracting streaming data from multiple sources, data was transformed into 'key-value pairs' without considering the semantic relationships between the data stored in these sources. Therefore, a semantic-based approach can be utilised to work in the front-end of the framework for data modelling and information retrieval. Different semantic approaches, such as ontology and schema matching/mapping, can be adapted to represent a shared, agreed and detailed data model for better integration [103]. Thus, in future work, semantic approaches can be adapted to process and integrate data comprehensively from multiple IoT sources.

- **Dynamic Windowing and Optimal Window Size**

  In this research, several experiments were conducted with predefined window sizes. The performance of streaming data processing was affected by the different window

sizes, with growing window sizes (Chapter 4) and data volume in a window affecting the data transferred through a streaming pipeline (Chapter 5). Different factors can influence performance, such as the speed and time interval of data streams and the number of records in a window. Hence, future research could explore dynamic windowing with an optimal window size in the streaming data integration framework.

- **Future Prototype Framework for Mobile Platforms**

  Users nowadays are becoming increasingly dependent on applications and various mobile devices. The integration of IoT streaming data in mobile devices such as cellphones, PDAs or laptops has been receiving increasing attention in pervasive and mobile computing environments [104]. As future work, the existing prototype framework can be extended and applied to support mobile platforms. Although the experimental evaluation in this thesis demonstrates the feasibility of the framework in the desktop platform, it may not sufficient for mobile platforms. Leveraging the scalability of the framework in mobile platforms will be another future direction.

Bibliography

[1] S Barker and M Rothmuller. The internet of things: Consumer, industrial & public services 2020–2024. *Juniper Research*, 2020.

[2] Michael Grieves. Digital twin: manufacturing excellence through virtual factory replication. *White paper*, 1:1–7, 2014.

[3] Zheng Liu, Norbert Meyendorf, and Nezih Mrad. The role of data fusion in predictive maintenance using digital twin. In *AIP Conference Proceedings*, volume 1949, pages 1–6. AIP Publishing LLC, 2018.

[4] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[5] Jinchuan Chen, Yueguo Chen, Xiaoyong Du, Cuiping Li, Jiaheng Lu, Suyun Zhao, and Xuan Zhou. Big data challenge: a data management perspective. *Frontiers of Computer Science*, 7(2):157–164, 2013.

[6] Gordon J Harris, Stephen A Rago, and Timothy H Williams. Distributed storage resource management in a storage area network, November 30 2004. US Patent 6,826,580.

[7] S Kalyan Chakravarthy, N Sudhakar, E Srinivasa Reddy, D Venkata Subramanian, and P Shankar. Dimension reduction and storage optimization techniques for distributed and big data cluster environment. In *Soft Computing and Medical Bioinformatics*, pages 47–54. Springer, 2018.

[8] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):93, 2018.

[9] Albert W Wegener. Block floating point compression of signal data, October 30 2012. US Patent 8,301,803.

[10] James Diffenderfer, Alyson L Fox, Jeffrey A Hittinger, Geoffrey Sanders, and Peter G Lindstrom. Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing*, 41(3):A1867–A1898, 2019.

[11] Hailin Li. Piecewise aggregate representations and lower-bound distance functions for multivariate time series. *Physica A: Statistical Mechanics and its Applications*, 427:10–25, 2015.

[12] ASM Kayes, Jun Han, Alan Colman, and Md Saiful Islam. Relboss: a relationship-aware access control framework for software services. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 258–276. Springer, 2014.

[13] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut: Sortable summarizations for scalable indexes over static and streaming data series. *The VLDB Journal*, 28(6):847–869, 2019.

[14] Edith Cohen and Martin Strauss. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–233, 2003.

[15] Linfeng Zhang and Yong Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 77–84. IEEE, 2008.

[16] Xin Luna Dong and Divesh Srivastava. Big data integration. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1245–1248. IEEE, 2013.

[17] Christian Bizer, Peter Boncz, Michael L Brodie, and Orri Erling. The meaningful use of big data: four perspectives–four challenges. *Acm Sigmod Record*, 40(4): 56–60, 2012.

[18] Patrick Ziegler and Klaus R Dittrich. Data integration—problems, approaches, and perspectives. In *Conceptual modelling in information systems engineering*, pages 39–58. Springer, 2007.

[19] Vladimir Gligorijević and Nataša Pržulj. Methods for biological data integration: perspectives and challenges. *Journal of the Royal Society Interface*, 12(112): 20150571, 2015.

[20] Venkat N Gudivada, Ricardo A Baeza-Yates, and Vijay V Raghavan. Big data: Promises and problems. *IEEE Computer*, 48(3):20–23, 2015.

[21] Veda C Storey and Il-Yeol Song. Big data technologies and management: What conceptual modeling can do. *Data & Knowledge Engineering*, 108:50–67, 2017.

[22] Matthew Herland, Taghi M Khoshgoftaar, and Richard A Bauder. Big data fraud detection using multiple medicare data sources. *Journal of Big Data*, 5(1):29, 2018.

[23] Thomas N Herzog, Fritz J Scheuren, and William E Winkler. *Data quality and record linkage techniques*. Springer Science & Business Media, 2007.

[24] N McNeill, Hakan Kardes, and Andrew Borthwick. Dynamic record blocking: efficient linking of massive databases in mapreduce. In *Proceedings of the 10th International Workshop on Quality in Databases (QDB)*, pages 1–7, 2012.

[25] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38(6):946–969, 2013.

[26] Tomer Sagi, Avigdor Gal, Omer Barkol, Ruth Bergman, and Alexander Avram. Multi-source uncertain entity resolution: Transforming holocaust victim reports into people. *Information Systems*, 65:124–136, 2017.

[27] Zohra Bellahsène, Angela Bonifati, and Erhard Rahm. *Schema matching and mapping*. Springer, 2011.

[28] AnHai Doan, Pedro M Domingos, and Alon Y Levy. Learning source description for data integration. In *WebDB (Informal Proceedings)*, pages 81–86, 2000.

[29] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. Enabling ontology-based access to streaming data sources. In *International Semantic Web Conference*, pages 96–111. Springer, 2010.

[30] Agustina Buccella, Alejandra Cechich, and Nieves Rodríguez Brisaboa. An ontology approach to data integration. *Journal of Computer Science & Technology*, 3, 2003.

[31] Isabel F Cruz, Huiyong Xiao, et al. The role of ontologies in data integration. *Engineering intelligent systems for electrical engineering and communications*, 13 (4):245, 2005.

[32] Cinzia Daraio, Maurizio Lenzerini, Claudio Leporelli, Henk F Moed, Paolo Naggar, Andrea Bonaccorsi, and Alessandro Bartolucci. Data integration for research and innovation policy: an ontology-based data management approach. *Scientometrics*, 106(2):857–871, 2016.

[33] Cinzia Daraio, Maurizio Lenzerini, Claudio Leporelli, Paolo Naggar, Andrea Bonaccorsi, and Alessandro Bartolucci. The advantages of an ontology-based data management approach: openness, interoperability and data quality. *Scientometrics*, 108(1):441–455, 2016.

[34] Gaihua Fu. Fca based ontology development for data integration. *Information processing & management*, 52(5):765–782, 2016.

[35] Erhard Rahm. Towards large-scale schema and ontology matching. In *Schema matching and mapping*, pages 3–27. Springer, 2011.

[36] Sean M Falconer and Natalya F Noy. Interactive techniques to support ontology matching. In *Schema Matching and Mapping*, pages 29–51. Springer, 2011.

[37] Avigdor Gal. Enhancing the capabilities of attribute correspondences. In *Schema Matching and Mapping*, pages 53–73. Springer, 2011.

[38] Johannes Fürnkranz. Integrative windowing. *Journal of Artificial Intelligence Research*, 8:129–164, 1998.

[39] B Boutsinas, DK Tasoulis, and MN Vrahatis. Estimating the number of clusters using a windowing technique. *Pattern Recognition and Image Analysis*, 16(2):143–154, 2006.

[40] Paul Trachian. Machine learning and windowed subsecond event detection on pmu data via hadoop and the openpdc. In *IEEE PES General Meeting*, pages 1–5. IEEE, 2010.

[41] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007.

[42] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, Mar 2013.

[43] Alok Pareek, Bhushan Khaladkar, Rajkumar Sen, Basar Onat, Vijay Nadimpalli, Manish Agarwal, and Nicholas Keene. Striim: A streaming analytics platform for real-time business decisions. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, pages 1–8. ACM, 2017.

[44] Alok Pareek, Bhushan Khaladkar, Rajkumar Sen, Basar Onat, Vijay Nadimpalli, and Mahadevan Lakshminarayanan. Real-time etl in striim. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, page 3. ACM, 2018.

[45] Paul Marinier, Bhaskar M Anepu, Ghyslain Pelletier, and Robert L Olesen. Maintaining time alignment with multiple uplink carriers, 2015. US Patent 8,934,459.

[46] Ravneet Kaur, Inderveer Chana, and Jhilik Bhattacharya. Data deduplication techniques for efficient cloud storage management: a systematic review. *The Journal of Supercomputing*, 74(5):2035–2085, 2018.

[47] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):1–30, 2014.

[48] Ruijin Zhou, Ming Liu, and Tao Li. Characterizing the efficiency of data deduplication for big data storage management. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 98–108. IEEE, 2013.

[49] Ohad Rodeh and Avi Teperman. zfs-a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pages 207–218. IEEE, 2003.

[50] Zhiguo Ding and Minrui Fei. An anomaly detection approach based on isolation forest algorithm for streaming data using sliding window. *IFAC Proceedings Volumes*, 46(20):12–17, 2013.

[51] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[52] Grigorios Chrysos, Odysseas Papapetrou, Dionisios Pnevmatikatos, Apostolos Dollas, and Minos Garofalakis. Data stream statistics over sliding windows: How to summarize 150 million updates per second on a single node. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 278–285. IEEE, 2019.

[53] Klemen Kenda, Blaž Kažič, Erik Novak, and Dunja Mladenić. Streaming data fusion for the internet of things. *Sensors*, 19(8):1955, 2019.

[54] Evgeny Kharlamov, Sebastian Brandt, Martin Giese, Ernesto Jiménez-Ruiz, Steffen Lamparter, Christian Neuenstadt, Özgür Lütfü Özçep, Christoph Pinkel, Ahmet Soylu, Dmitriy Zheleznyakov, et al. Semantic access to siemens streaming data: the optique way. In *International Semantic Web Conference (Posters & Demos)*, volume 1486. Citeseer, 2015.

[55] Mohd Abdul Ahad and Ranjit Biswas. Dynamic merging based small file storage (dm-sfs) architecture for efficiently storing small size files in hadoop. *Procedia Computer Science*, 132:1626–1635, 2018.

[56] ASM Kayes, Jun Han, Wenny Rahayu, Tharam Dillon, Saiful Islam, and Alan Colman. A policy model and framework for context-aware access control to information resources. *The Computer Journal*, 2018. doi: https://doi.org/10.1093/comjnl/bxy065.

[57] ASM Kayes, Wenny Rahayu, Tharam Dillon, and Elizabeth Chang. Accessing data from multiple sources through context-aware access control. In *TrustCom*, pages 551–559. IEEE, 2018.

[58] ASM Kayes, Jun Han, and Alan Colman. Icaf: A context-aware framework for access control. In *Australasian Conference on Information Security and Privacy*, pages 442–449. Springer, 2012.

[59] ASM Kayes, Wenny Rahayu, Tharam Dillon, Elizabeth Chang, and Jun Han. Context-aware access control with imprecise context characterization through a combined fuzzy logic and ontology-based approach. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 132–153. Springer, 2017.

[60] ASM Kayes, Wenny Rahayu, Tharam Dillon, Elizabeth Chang, and Jun Han. Context-aware access control with imprecise context characterization for cloud-based data resources. *Future Generation Computer Systems*, 93:237–255, 2019.

[61] Abdul Ghaffar Shoro and Tariq Rahim Soomro. Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology*, 2015.

[62] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[63] Apache Spark. Apache spark. *Retrieved January*, 17:2018, 2018.

[64] Anatoliy Batyuk and Volodymyr Voityshyn. Apache storm based on topology for real-time processing of streaming data from social networks. In *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)*, pages 345–349. IEEE, 2016.

[65] B. Yadranjiaghdam, S. Yasrobi, and N. Tabrizi. Developing a real-time data analytics framework for twitter streaming data. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 329–336, 2017. doi: 10.1109/BigDataCongress.2017.49.

[66] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[67] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006.

[68] Martin Burtscher and Paruj Ratanaworabhan. High throughput compression of double-precision floating-point data. In *2007 Data Compression Conference (DCC'07)*, pages 293–302. IEEE, 2007.

[69] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[70] Aisha Siddiqa, Ahmad Karim, and Victor Chang. Smallclient for big data: an indexing framework towards fast data retrieval. *Cluster Computing*, 20(2):1193–1208, 2017.

[71] Tamer Elsayed, Jimmy Lin, and Douglas W Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pages 265–268. Association for Computational Linguistics, 2008.

[72] Taerim Lee, Hyejoo Lee, Kyung-Hyune Rhee, and Uk Sang Shin. The efficient implementation of distributed indexing with hadoop for digital investigations on big data. *Computer Science and Information Systems*, 11(3):1037–1054, 2014.

[73] M Mittal. Indexing techniques and challenge in big data. *International Journal of Current Engineering and Technology*, 7(3):1225–1228, 2017.

[74] Abdullah Gani, Aisha Siddiqa, Shahaboddin Shamshirband, and Fariza Hanum. A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowledge and information systems*, 46(2):241–284, 2016.

[75] Fatimah Adamu, Adib M Monzer Habbal, Suhaidi Hassan, R Les Cottrell, Bebo White, and Ibrahim Abdullahi. A survey on big data indexing strategies. *UUM InterNetWorks Research Laboratory in collaboration with the Malaysian Communications and Mutimedia Commission (MCMC)*, 2015.

[76] Susan T Dumais et al. Latent semantic indexing (lsi): Trec-3 report. *Nist Special Publication SP*, pages 219–219, 1995.

[77] Avinash Atreya and Charles Elkan. Latent semantic indexing (lsi) fails for trec collections. *ACM SIGKDD Explorations Newsletter*, 12(2):5–10, 2011.

[78] Y Tang and L Liu. Multi-keyword privacy-preserving search in personal server networks. *Knowledge and Data Engineering, IEEE Transactions on, vol. PP*, PP (99):1–1, 2015.

[79] Guigang Zhang, Jian Wang, Weixing Huang, Chao Li, Yong Zhang, and Chunxiao Xing. A semantic++ mapreduce: A preliminary report. In *2014 IEEE International Conference on Semantic Computing*, pages 330–336. IEEE, 2014.

[80] Flora Amato, Aniello De Santo, Francesco Gargiulo, Vincenzo Moscato, Fabio Persia, Antonio Picariello, and Silvestro Roberto Poccia. Semtree: An index for supporting semantic retrieval of documents. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 62–67. IEEE, 2015.

[81] Maheshkumar H Kolekar and S Sengupta. Hidden markov model based video indexing with discrete cosine transform as a likelihood function. In *Proceedings of the IEEE INDICON 2004. First India Annual Conference, 2004.*, pages 157–159. IEEE, 2004.

[82] Ganeshchandra Mallya, Shivam Tripathi, Sergey Kirshner, and Rao S Govindaraju. Probabilistic assessment of drought characteristics using hidden markov model. *Journal of Hydrologic Engineering*, 18(7):834–845, 2013.

[83] Ayaka Matsui, Satoshi Nishimura, and Seiichiro Katsura. A classification method of motion database using hidden markov model. In *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pages 2232–2237. IEEE, 2014.

[84] Wahyu Catur Wibowo et al. Improving classification performance by extending documents terms. In *2014 International Conference on Data and Software Engineering (ICODSE)*, pages 1–5. IEEE, 2014.

[85] Widia Permata Sari, Moch Arif Bijaksana, and Arief Fatchul Huda. Indexing name in hadith translation using hidden markov model (hmm). In *2019 7th International Conference on Information and Communication Technology (ICoICT)*, pages 1–5. IEEE, 2019.

[86] Paolo Cappellari, Mark Roantree, and Soon Ae Chun. Optimizing data stream processing for large-scale applications. *Software: Practice and Experience*, 48(9): 1607–1641, 2018.

[87] Saima Gulzar Ahmad, Chee Sun Liew, M Mustafa Rafique, and Ehsan Ullah Munir. Optimization of data-intensive workflows in stream-based data processing models. *The Journal of Supercomputing*, 73(9):3901–3923, 2017.

[88] Zoltán Zvara, Péter G.N. Szabó, Barnabás Balázs, and András Benczúr. Optimizing distributed data stream processing by tracing. *Future Generation Computer Systems*, 90:578–591, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.06.047.

[89] Ken Q Pu and Ying Zhu. Efficient indexing of heterogeneous data streams with automatic performance configurations. In *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*, pages 34–34. IEEE, 2007.

[90] Sirish Chandrasekaran and Michael J Franklin. Streaming queries over streaming data. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 203–214. Elsevier, 2002.

[91] Sobhan Badiozamany and Tore Risch. Scalable ordered indexing of streaming data. In *ADMS@ VLDB*, pages 57–67, 2012.

[92] Lukasz Golab, Shaveen Garg, and M Tamer Özsu. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology*, pages 712–729. Springer, 2004.

[93] Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J Miller. Framework for evaluating clustering algorithms in duplicate detection. *Proceedings of the VLDB Endowment*, 2(1):1282–1293, 2009.

[94] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[95] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44 (3):15, 2012.

[96] Anish Das Sarma, Xin Luna Dong, and Alon Y Halevy. Uncertainty in data integration and dataspace support platforms. In *Schema Matching and Mapping*, pages 75–108. Springer, 2011.

[97] Doan Quang Tu, ASM Kayes, Wenny Rahayu, and Kinh Nguyen. Isdi: A new window-based framework for integrating iot streaming data from multiple sources. In *International Conference on Advanced Information Networking and Applications*, pages 498–511. Springer, 2019.

[98] Esra Satir and Hakan Isik. A huffman compression based text steganography method. *Multimedia tools and applications*, 70(3):2085–2110, 2014.

[99] Dong-Hui Xu, Arati S Kurani, Jacob D Furst, and Daniela S Raicu. Run-length encoding for volumetric texture. *Heart*, 27(25):452–458, 2004.

[100] Mamta Sharma. Compression using huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 10(5):133–141, 2010.

[101] Q. Doan, A. S. M. Kayes, W. Rahayu, and K. Nguyen. Integration of iot streaming data with efficient indexing and storage optimization. *IEEE Access*, 8:47456–47467, 2020. doi: 10.1109/ACCESS.2020.2980006.

[102] Jordi Soria-Comas, Josep Domingo-Ferrer, David Sánchez, and Sergio Martínez. Enhancing data utility in differential privacy via microaggregation-based k-anonymity. *The VLDB Journal*, 23(5):771–794, 2014.

[103] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[104] Dan Chalmers and Morris Sloman. A survey of quality of service in mobile computing environments. *IEEE Communications surveys*, 2(2):2–10, 1999.