IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 29, 2020, accepted March 8, 2020, date of current version March 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2980006

Integration of IoT Streaming Data With Efficient Indexing and Storage Optimization

QUANG-TU DOAN, A. S. M. KAYES, WENNY RAHAYU, AND KINH NGUYEN
Department of Computer Science and Information Technology, La Trobe University, Melbourne, VIC 3086, Australia
Corresponding author: A. S. M. Kayes (a.kayes@latrobe.edu.au)

ABSTRACT In the era of IoT, the world of connected experiences is created by the convergence of multiple technologies including real-time analytics, machine learning, and commodity sensors and embedded systems. However, with the proliferation of these IoT technologies and devices, there are challenges in integrating, indexing and managing time-series data from multiple sources to optimize the storage of those data and/or retrieve the information from them in real-time. Many researchers have addressed the data integration issue through developing time-series data compression techniques; however, they focused mainly on the application of integer value compression to IoT data. Moreover, existing work does not focus on the issues of data and information retrieval without decompression. In this paper, we solve these issues by constructing an indexing framework within a lossless compression 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. We conduct several sets of experiments and quantify the performance of our proposed approach. The experimental results, performed on 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.

INDEX TERMS Data integration, indexing, time-series data compression, floating point compression, decompression, IoT streaming data, window-based compression and integration.

I. INTRODUCTION

Due to the rapid advancement of big data platforms, the need to integrate data and then improve data access from multiple time-series data sources, such as in data analysis and decision support systems, has grown significantly over the last few years. However, with the unprecedented expansion of IoT streaming data [1], efficient access to the 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 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, we develop a framework to integrate time-series data from multiple IoT sources by using compression and indexing techniques for streaming data.

Data compression is a reduction in the numbers of bits that represent the data, and it can save storage capacity, speed up file transfer, and decrease costs for 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, Blalock *et al.* with Sprintz [1], Wegener with signal data compression [2] and Diffenderfer *et al.* with ZFP [3]. However, these techniques are merely compression approaches, which only focus on

The associate editor coordinating the review of this manuscript and approving it for publication was Noor Zaman.

saving storage capacity. In our work, we introduce an improvement to compression, not only for storage saving but for the ability to indexing from compressed time-series data.

In the recent years, there has been much research on similarity searches and the subsequent data indexing [4]–[6]. 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. In different aspects, in our research, we propose an indexing framework that features easy-to-find results based on timing requirements but not based on the similarity patterns. In particular, we observe 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*' that we refer in this paper is not likely ''summarizations'' that proposed in [6], which are representations of time-series data segments. Our term means summarized outcomes extracted from a segment of data by relevant user-defined functions. For example, with the ability to keep tracking time-series data based on time-stamps, our framework can summarize information such as the average, maximum or minimum of temperatures in a certain period of time.

A. THE CONTRIBUTIONS

Based on the above-mentioned investigations, the main contributions of this work are as follows.

1) We introduce a mechanism of time-series data integration with efficient indexing and storage optimization. Thus, at first, we discuss the research motivation of IoT data integration including compression techniques and indexing issues compared with existing work.
2) We propose an indexing model including a lossless compression technique for IoT data along with the benefits of bit-padding, bit-blocking and Huffman coding. We adopt an existing bit-padding technique and improve it by reducing the number of bits during the compression process. For instance, traditional bit-padding algorithms align bits in fixed 8-bit streams, whereas our proposed technique does not need fixed 8-bit streams.
3) For indexing, in this work, we do not focus on constructing a new data structure but introduce an index based on time-stamps which supports to access to compressed data without full decompression. This index utilizes the time-stamp attached during the compression process. The relevant algorithms are also discussed.
4) Through different experimental setups, we demonstrate that our solution can reduce space during the compression process. In addition, we show that our framework has the capability of integrating IoT time-series data in real-time from multiple data sources, and accessing those data to facilitate the information retrieval, for example, performing queries effectively based on time-stamps.
5) We highlight possible directions for future work that have not been well covered in current state-of-the-art time-series data integration and data compression research.

B. THE ORGANIZATION OF THE PAPER

The rest of this paper is organized as follows. The related work is discussed in Section 2. In Section 3, we propose a new indexing framework to integrate and manage IoT streaming data from multiple sources in real-time. We introduce our compression mechanism and data indexing in Section 3. In Section 4, we conduct several sets of experiments and demonstrate the benefits of our proposed framework, including the relevant algorithms and an overall discussion. Finally, we conclude the paper in Section 5, along with several future research directions.

II. RELATED WORK

As previously mentioned, our research work focuses on constructing a model of time series data integration, and we find a mechanism to store and access IoT efficiently. Hence, in this section, we discuss some existing work related to data compression, time-series data indexing and some streaming data integration techniques.

A. DATA COMPRESSION TECHNIQUES

Time series data has a special structure, which the gaps between the values of the two adjacent time-stamps are are taken into account. For example, in financial time series data, the price of 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 [2] 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 creating a function of quantization before encoding the data. Using blocks in another way, [7] 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, [3] improved ZFP by establishing a bound which is a well-know limitation of ZFP.

In addition, compression techniques rely on the small difference of consecutive values and make predictions for the next values. For example, the approach in [8] takes advantage of the correlation of 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 [9] compresses data in sequences by predicting the next value in the sequence and using hash tables as predictors. Lindstrom and Lindstrom and Isenburg [10] 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 [2], Sprintz [1] 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.

Our method 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.

B. DATA INDEXING TECHNIQUES

In terms of data indexing, SmallClient [11] improves query execution and search performance for big datasets and minimises the overhead of indexing. The framework is implementable on any distributed file system. Basically, the main part of the 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.* [12] 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.* [13] apply indexing methods and MapReduce into 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.

For indexing work, we also investigate other indexing surveys. According to Mamta [14], 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 summarises 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 [15] 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. 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 KRR to achieve a high detection rate for the prediction of missing user preferences.

C. DATA INTEGRATION TECHNIQUES

Data integration, which addresses the issue of data duplication and data fusion, has been the focus of a lot of research. For instance, using record linkage, the authors in [16], [17] combine and consolidate data from different sources. They create blocks to merge identical objects in a big data-set. Another technique to integrate data from multiple sources is schema mapping and matching [18]. Schema matching implements a ''match'' operator which is often the first step to determine schema mappings for data integration. In [19], the authors create a global schema and show that it is needed to specify a mediated schema and supply the descriptions of data 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 with respect to 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 match problems [20]–[22].

However, these traditional blocking techniques are not adequate to deal with the challenges and issues of big data in general and streaming data integration in particular.

Regarding ontology-based research, some work focused on streaming data integration [23]–[25]. From this perspective, the authors 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 works as a global schema for all data sources. In a different way, Pareek *et al.* [26] introduced a streaming analytics platform (simply, the Striim engine) for real-time data integration with respect to structured data from multiple sources. The Striim engine extracts data from sources to transform into SQL-based data so that it can integrate time-series data with different structured data [27]. Nevertheless, both ontology-based approaches and the Striim engine do not focus on dealing with the timing conflict for time-series data, which is one of the critical issues while integrating data from multiple sources. In this paper, we extend our previous work [28] to integrate IoT streaming data from multiple sources in real-time and deal with the time alignment. In addition, to the best of our knowledge, there is no existing streaming data integration work which considers the issues of information retrieval and data storage, which are our contributions in this paper. We propose a novel technique for time-series data compression to optimize storage, and we introduce a timestamp-based index for compressed data to facilitate the later work of information retrieval, and querying compressed data without full decompression.

III. PROPOSED DYNAMIC INDEXING FRAMEWORK

In this section, we introduce our dynamic indexing framework for streaming data from multiple IoT sources, which comprises two main contributions, time-series data compression and time-stamp indexing. In the framework, we also attach the process of time alignment which was introduced in our previous work [28].

Figure 1 illustrates our 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 an index of the compressed data in the storage. Our model comprises the following steps:

- The model can extract data from multiple sources continuously through our 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 [28].
- As most IoT data are floating points, we compress them using our floating-point compression technique, which is an improvement on the integer compression technique (Sprintz). The trace of data is stored in our timestamp index. Our data structure is <key,value> pairs, whereas keys are meta-data and store all attributes of each records. This step is implemented in the algorithm 1 in subsection 3.3.
- The floating-point compressed data can be compressed again by applying a lossless compression (e.g. Huffman compression [29], run-length encoding [30]), which improves the compression ratio to enhance our storage capability. The results of this step are presented in our experiment (indicate subsection).
- The timestamps index are refined so that it is easy to search the index based on the timestamps and users' queries.

A. THE COMPRESSION MECHANISM FOR FLOATING-POINT DATA

The compression process consists of six steps. In this section, we describe these steps 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 3.

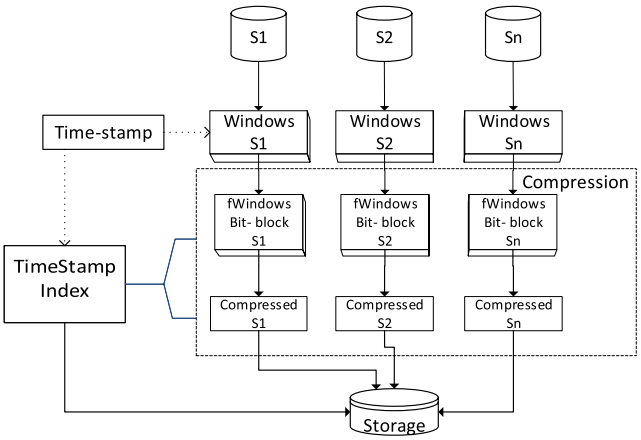

FIGURE 1. Indexing model for IoT Data.

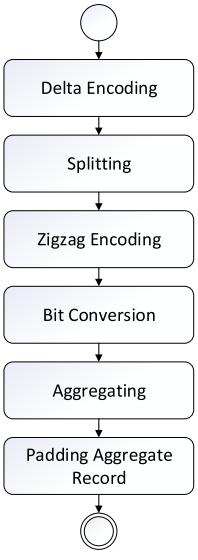

FIGURE 2. Real number bitblocking technique overview.

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

Table A

Delta

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

Table B

Split

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

Table C

FIGURE 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

Table D

Zigzag

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

Table E

Count significant bits

date	Air Compr Lub Oil 1	
	Part1	Part2
	00000000	0000000000000000
	00000000	0000010110001100
	00000000	0000101011101111
	00000000	0000011111101010
	00000000	0000001011110110
	00000000	0000010011011100
	00000000	0000000010100011
	0 bits	12 bits

Table F

FIGURE 4. Real number bitblocking - Phase 2.

STEP 1: DELTA ENCODING

In this step, for each record in the window, we compute the difference between the reference record's attributes. The result is shown in Table B - Figure 3.

It is obvious that the delta operation is reversible. That is, given the full data in Table B, we can retrieve Table A in Figure 3.

STEP 2: SPLITTING

In this step, instead of working with real numbers, we do the following.

1) We split each entry value into two parts: the whole number part and the fractional part.
2) If an entry is negative, we make the fractional part negative.
3) We store both parts as integers. And because, we store the second part as an integer, we also maintain a significance factor 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 the significance factor 4 for its column. The three 'components' allows us to retrieve the original value as $-(1 + 82 \times 10^{-4})$.

Applying this operation to Table B, we get the result shown in Table C - Figure 3. As shown above, this steps is reversible.

STEP 3: ZIGZAG ENCODING

In Table D - Figure 4, some of the entries are positive and some negative. It would be convenient to work with positive numbers only. The zigzag operation allows us to do that. The calculation is as follows.

1) If an entry is positive, we double it.
2) If an entry is negative, we double 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 4. It is obvious that this Zigzag step is reversible as well.

STEP 4: BIT CONVERSION

Now, we convert each integer value in Table E into a 16-bit binary representation. And we count the maximum number of significant bits 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, we take the data in Table F and put them in one record, made up of a series of bits. This aggregate record contains the data of 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 follows.

1) Number of significant bits for <attribute X, part W> (which is 0 bits or 000 in the table in Figure 5)

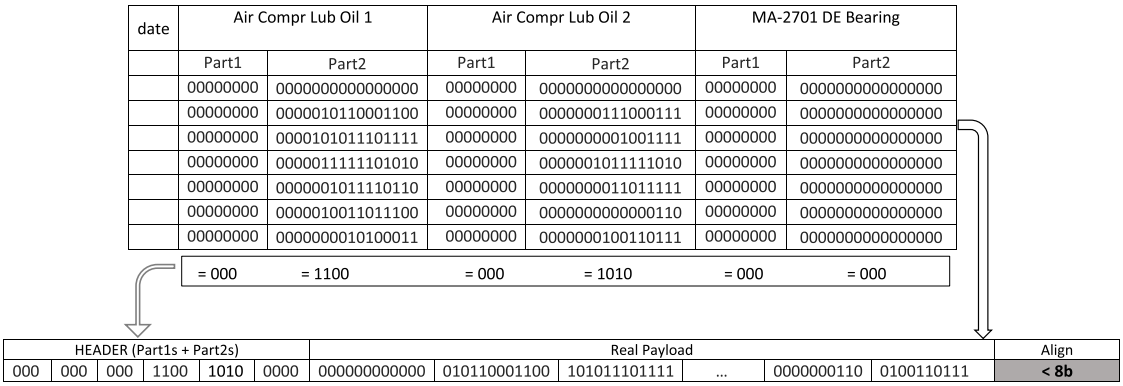

FIGURE 5. Real Number BitBlocking - Phase 3.

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 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 7. Time-stamp attachment.

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

As for the data segment (Real Payload), the contents consists 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 we can tell 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. Note that this aggregation step is reversible. From the aggregate record, we can retrieve data in the table in Figure 5.

STEP 6: PADDING AGGREGATE RECORD

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

Figure 6 is 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 our improvement, as we manage data in window, we control and know the number of bits in a window. Hence, we develop a method for a byte-align mechanism by adding bits only at the end of each window. To take this into account the "partially filled" possibility, we add one more byte to the aggregate record to indicate how many bits in the last data byte are part of the data. We will refer to this additional record as *the padding byte*.

A value 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 aggregate record, with partially filled data, is shown in Figure 7. This is the actual compressed record that we are going to store.

Note that this padding operation is clearly reversible in the sense that from a padded aggregate record, we can retrieve the data, and we do not 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., we can

decode final aggregate record of Step 6 to retrieve the original data of the window in Figure 3.

B. AN INDEXING TECHNIQUE BASED ON TIME-STAMPS

In addition to compression, the time-stamp index is another main contribution of our model. Assuming a given index structure, we need to denote each entry of the index by a time-stamp. Hence, in this subsection, we define an entry of the index as a time-stamp and find a mechanism to attach a time-stamp to the window-bit blocks. In our model, the key for a window is a pair of time-stamps and window size. As a result, it is trivial to extract the time-stamp for each window. In order to attach it to the block, we also have to transfer the DateTime format to the bit blocks. To save storage, we transfer them to a hexadecimal and fix the number of first bytes to store these time-stamps in each block. Figure 7 demonstrates an example time stamp attachment.

For example, we normalize the time-stamps into the format 'YYYYMMDDhhmmss' which can be parsed into a long variable. We then convert them into a binary or a hexadecimal. Notice that, the attachment of time-stamp is only performed at the first record of each block which is the encoding of a window.

C. COMPRESSION AND INDEXING ALGORITHMS

Algorithm 1 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. Our 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, we identify the first parts and the second parts of the floating point values for each attribute after delta encoding (see phase 1 - Figure 3). Therefore, we need 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, we first delta-encode (line 10). We then split the results 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, we apply delta encoding for the integer parts and then zigzag encoding for all the values. An example for this implementation is presented in Figure 4. The BinaryComponents() in line 12 performs 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

Algorithm 1 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 += BitsRepresenta-
           tion(secondPart[i][j],significant2);

29     end
30 end
31 return BitPadding(header1 + hearder2 + payLoad1 +
   payLoad2)
```

all components for the bit block of a window including the headers and the payloads (real values) of the two integer parts. Finally, we block all the parts together by using a function BitPadding() in line 31. An illustration for these steps is shown in Figure 5.

Algorithm 2 is our processing step (Index Generation) after the compression step (Figure 1). First, we extract data 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, we obtain data

Algorithm 2 Time-Series Indexing Generation

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**

with a period of granularity and add them as entries to the index (Line 7 to line 11).

IV. EXPERIMENT RESULTS

In our experiment, we use the same real streaming dataset as the experiment in our previous work [28]. This dataset is from a distributed manufacturing company, which is designed with many machines along with IoT sensors. Table 1 contains streaming data from two IoT sources, namely small dataset and big dataset within second and minute-based time-series data. In particular, the small dataset contains IoT Source 1 of having 368,199 records of 94 MB in size and IoT Source 2 of having 2592,200 records of 62.8 MB; whereas, the big dataset contains IoT Source 3 and IoT Source 4 of having 1,472,800 records and 6,480,000 records with the sizes of 376 MB and 1.5 GB, respectively. We perform our experiment on both the small and big datasets.

TABLE 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

A. STORAGE SPACE REDUCTION

Table 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.

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

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

TABLE 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 [1]	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%

In formulas (1) and (2), *sizeUsingCompression* is the size of the data storage needed when we apply our compression technique; *sizeWithoutCompression* is the size of the data storage needed when we implement the model of ISDI (IoT Streaming Data Integration) from our previous work [28] with the same data. In particular, in ISDI, we extract data in windows (blocks) and then integrate data from sources by using a user-defined function attached in the integrator (in Figure 8), for example, calculating the average of temperatures in each window. The storage in this case includes semantic information. In this experiment, we test data on one source (the data in IoT source 1 in Table 1) with different compression levels to investigate the *compressionRatio* and the *storagSaving*.

Table 2 provides details on four compression techniques. Real number *bit blocks* technique, which is our contribution in this paper, is illustrated in the figure 2; *byte transfer* is the technique that transfers all bits into bytes, and *Huffman* coding is an algorithm for performing data compression [31]. These techniques are different levels of our compression. In order to apply *Huffman* coding, we transfer bits presentation into symbols (or characters), so *Byte transfer* is always performed before applying *Huffman* coding. Last but not least, we compare them with an existing time series compression technique, SprintZ [1]. With small dataset, in the first level of our technique which applies bit blocks only, the compression ratio is 27.24% and the storage saving is 72.76%. However, the compression ratio is much better when we transfer all the bits of blocks 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 [1], our comprehensive technique gives a better storage saving, 96.32% vs 97.88%. Notice that, with the big dataset, our results are similar with those with small dataset, for example, 2.15% vs 2.12% of compression ratio. This shown that our system is scalable. It is because we process data in partitions and the compression technique is

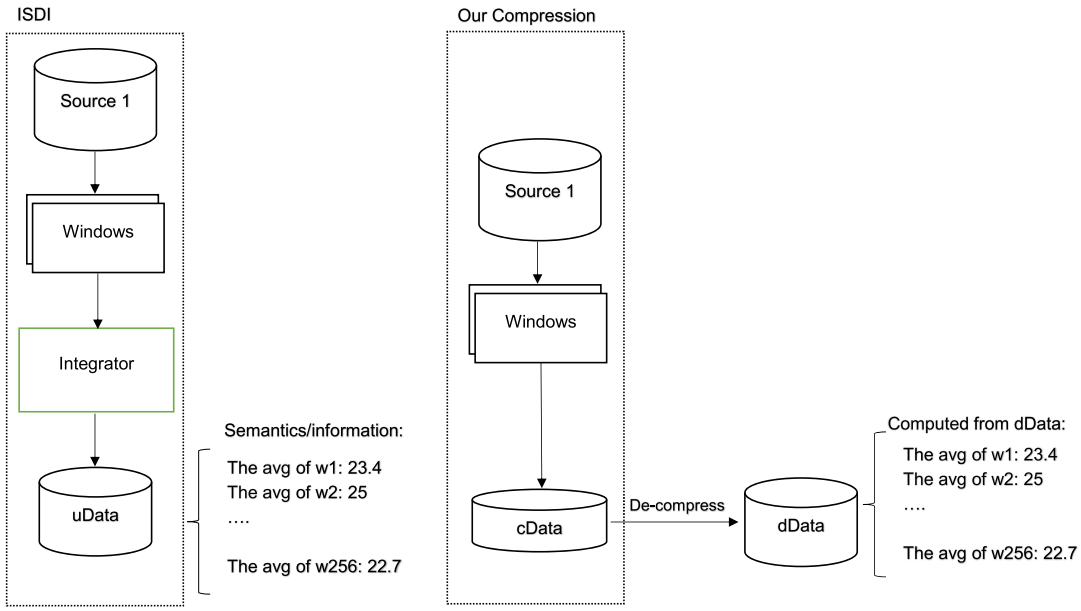

FIGURE 8. Our ISDI [28] vs our compression.

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 9.

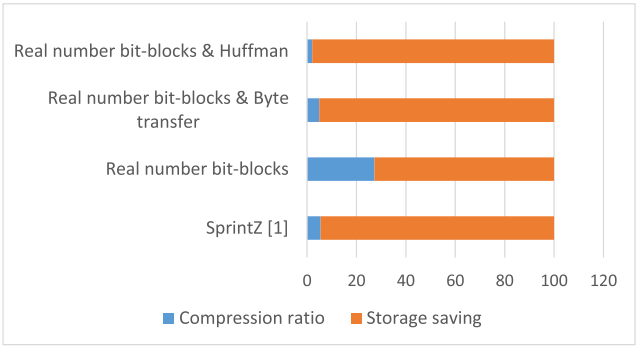

FIGURE 9. Compression ratio using different techniques with big dataset.

Figure 8 illustrates the process of extracting summarized information from the ISDI and this current work to obtain data with *sizeWithoutCompression* (uncompressed data - uData) and *sizeUsingCompression* (compressed data - cData), respectively. As previously discussed, uData contains semantic information which we set as the average of the temperatures for each window in the experiment. For the compression version, we compress data on each window and combine them into cData. We then uncompress the cData and calculate the average of temperature for each window. The two results are exactly the same, which means the semantic information will not be lost when applying our compression. Hence, we can conclude that our compression technique is a lossless compression and offers a very good compression ratio.

B. TIME-SERIES DATA PROCESSING CAPABILITY

In this subsection, we measure the capability of on-the-fly processing time-series data through using our model (Figure 10). Our model includes SourceManagers, Compression and Indexing. 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, our model of compression and indexing processes receives data to facilitate a quick response to clients' queries as previously discussed. In particular, we set a streaming data processing pipeline which transfers data from IoT sources to the model. We compare 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 the index (T_p); whereas, T_w is measured by the time

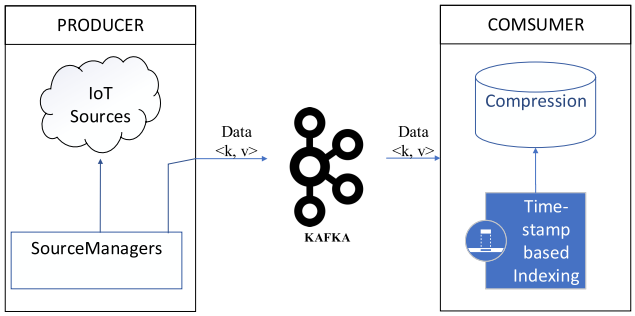

FIGURE 10. On-the-fly processing for time-series indexing.

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 index. If T_p is less than T_w, we confirm that our model satisfies the condition of processing time-series data on the fly.

Figure 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 up 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 our 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 streaming processing 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, our framework can definitely be deployed in a streaming pipeline processing.

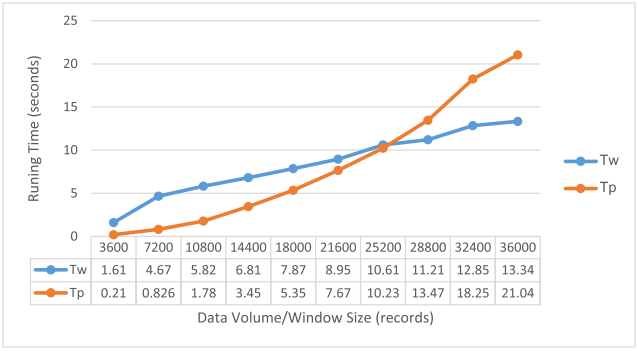

FIGURE 11. On-the-fly Processing Time based on the volume of data from one source.

In practical, T_w is determined by the slowest rate of in-coming data (for example, 1 record per hour), and the critical volume (from 3600 to under 25200 in the figure 11) for a window is determined by the fastest rate (for example, 1 record per second) and the slowest rate.

C. TIME-SERIES DATA INTEGRATION THROUGH TIMING ALIGNMENTS AND DE-DUPLICATION

In this subsection, we conduct the same experiment as described in the previous subsection but we perform it on two sources in our recommended algorithms, time alignment and de-duplication.

Figure 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 11. We observe that their timing performance depends much on the size of the window or the number of records in a window. In addition, there are differences in the data volume of each window from the sources, for example, there are 60 records in a window with a 1-hour

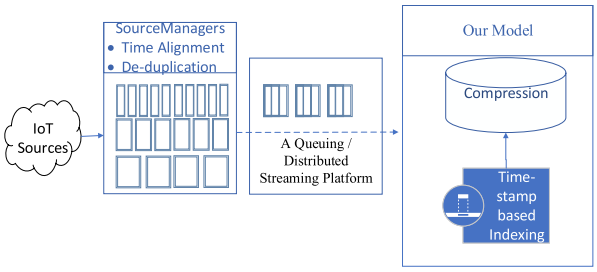

FIGURE 12. Time-series compression and index from multiple sources.

size from source 1, but there are 3600 records from source 2 with the same size (refer to frequency in Table 1). Thus, when applying timing alignment (Algorithm 2), we define the volume granularity of the on-the-fly processed data as the maximum number of records from different windows of multiple sources. In other words, this granularity depends on the source having the minimum base (to calculate the number of records) and the source having the maximum base (to decide the window size). For example, with source 1 (second-based, data is 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, we discuss different scenarios, which effect the volume granularity. We refer to them as an 'extreme case' and a 'realistic case'.

In the 'extreme case', source E_1 is milisecond-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. We cannot generate a result 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 in Table 1). This means the granularity for on-the-fly processed data is 60 records. The number is too small, so we can define the window size 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 our model in a streaming pipeline. The performance in this case is shown in Figure 13.

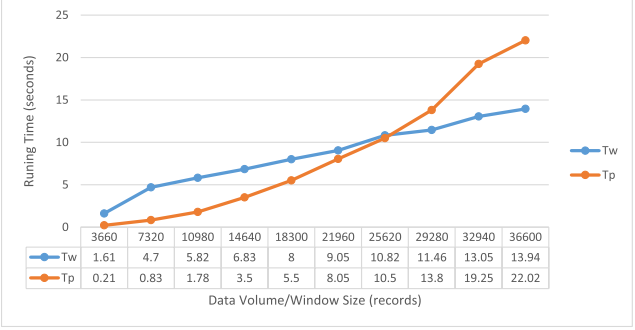

FIGURE 13. On-the-fly processing time based on the volume of data from two sources (S_1) and S_2.

D. OVERALL DISCUSSION

During the implementation of the proposed compression technique, we attach time-stamps so that when constructing an index, we can utilize these time-stamps to obtain data without decompression. Hence, with this indexing framework, we can reduce the storage of time-series data and retrieve the data in real-time. The experiment results in Table 2 demonstrate that our proposed indexing 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 13) shows that our 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 model can process on-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 our implementation to integrate data from the sources (Figure 14), we use temporary buffers to store the data from different sources and then merge and combine them 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 11 vs Figure 13).

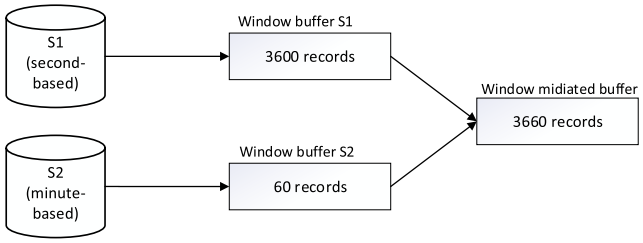

FIGURE 14. The implementation of integrating windows from 2 sources (second-based and minute-based).

V. CONCLUSION AND FUTURE RESEARCH

In this paper, we proposed a new indexing framework for IoT streaming data. We introduced a time-series data compression technique in which an index is formed according to the time-stamps on the compressed data. Our 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. We improved the existing technique of bit-padding, and optimized it by adding less bits to get multiples of 8-bit for bit-block creation.

We conducted several sets of experiment with single IoT data source and demonstrated the capability of our storage reduction. Using our proposed indexing technique (based on the real number of bit blocks and Huffman coding), we optimise 97.88% of the storage space, whereas earlier techniques can only save 95.06% storage space at best.

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

In our future work, we will optimize our indexing technique by experimenting with different sizes and speeds of IoT streaming data, as well as focusing on the query inputs and outputs. We can also improve our proposed framework by dealing with timing alignments and de-duplication issues while IoT streaming data come from multiple sources. Last but not least, we will explore our research in big data to adapt real industrial areas.

REFERENCES

[1] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time series compression for the Internet of Things," *Proc. ACM Interact., Mobile, Wearable Ubiquitous Technol.*, vol. 2, no. 3, pp. 1–23, 2018.

[2] A. W. Wegener, "Block floating point compression of signal data," U.S. Patent 8 301 803, Oct. 30, 2012.

[3] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom, "Error analysis of ZFP compression for floating-point data," *SIAM J. Sci. Comput.*, vol. 41, no. 3, pp. A1867–A1898, Jan. 2019.

[4] H. Li, "Piecewise aggregate representations and lower-bound distance functions for multivariate time series," *Phys. A, Stat. Mech. Appl.*, vol. 427, pp. 10–25, Jun. 2015.

[5] Y. Sakurai, Y. Matsubara, and C. Faloutsos, "Smart analytics for big time-series data," in *Proc. KDD*, 2017.

[6] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, "Coconut: Sortable summarizations for scalable indexes over static and streaming data series," *VLDB J.*, vol. 28, no. 6, pp. 847–869, Dec. 2019.

[7] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec. 2014.

[8] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in *Proc. Data Compress. Conf. (DCC)*, 2006, pp. 133–142.

[9] M. Burtscher and P. Ratanaworabhan, "High throughput compression of double-precision floating-point data," in *Proc. Data Compress. Conf. (DCC)*, 2007, pp. 293–302.

[10] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Trans. Vis. Comput. Graphics*, vol. 12, no. 5, pp. 1245–1250, Sep. 2006.

[11] A. Siddiqa, A. Karim, and V. Chang, "SmallClient for big data: An indexing framework towards fast data retrieval," *Cluster Comput.*, vol. 20, no. 2, pp. 1193–1208, Jun. 2017.

[12] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with MapReduce," in *Proc. 46th Annu. Meeting Assoc. Comput. Linguistics Hum. Lang. Technol. (HLT)*, 2008, pp. 265–268.

[13] T. Lee, H. Lee, K.-H. Rhee, and U. Shin, "The efficient implementation of distributed indexing with Hadoop for digital investigations on big data," *Comput. Sci. Inf. Syst.*, vol. 11, no. 3, pp. 1037–1054, 2014.

[14] M. Mittal, "Indexing techniques and challenge in big data," *Int. J. Current Eng. Technol.*, vol. 7, no. 3, pp. 1225–1228, 2017.

[15] A. Gani, A. Siddiqa, S. Shamshirband, and F. Hanum, "A survey on indexing techniques for big data: Taxonomy and performance evaluation," *Knowl. Inf. Syst.*, vol. 46, no. 2, pp. 241–284, Feb. 2016.

[16] N. McNeill, H. Kardes, and A. Borthwick, "Dynamic record blocking: Efficient linking of massive databases in mapreduce," in *Proc. 10th Int. Workshop Qual. Databases (QDB)*, 2012, pp. 1–7.
[17] T. Sagi, A. Gal, O. Barkol, R. Bergman, and A. Avram, "Multi-source uncertain entity resolution: Transforming holocaust victim reports into people," *Inf. Syst.*, vol. 65, pp. 124–136, Apr. 2017.
[18] Z. Bellahsène, A. Bonifati, and E. Rahm, *Schema Matching and Mapping*. Springer, 2011.
[19] A. Doan, P. M. Domingos, and A. Y. Levy, "Learning source description for data integration," in *Proc. WebDB, Informal*, 2000, pp. 81–86.
[20] E. Rahm, "Towards large-scale schema and ontology matching," in *Schema Matching and Mapping*. Springer, 2011, pp. 3–27.
[21] S. M. Falconer and N. F. Noy, "Interactive techniques to support ontology matching," in *Schema Matching and Mapping*. Springer, 2011, pp. 29–51.
[22] A. Gal, "Enhancing the capabilities of attribute correspondences," in *Schema Matching and Mapping*. Springer, 2011, pp. 53–73.
[23] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, "Enabling ontology-based access to streaming data sources," in *Proc. Int. Semantic Web Conf.* Springer, 2010, pp. 96–111.
[24] C. Daraio, M. Lenzerini, C. Leporelli, H. F. Moed, P. Naggar, A. Bonaccorsi, and A. Bartolucci, "Data integration for research and innovation policy: An ontology-based data management approach," *Scientometrics*, vol. 106, no. 2, pp. 857–871, Feb. 2016.
[25] C. Daraio, M. Lenzerini, C. Leporelli, P. Naggar, A. Bonaccorsi, and A. Bartolucci, "The advantages of an ontology-based data management approach: Openness, interoperability and data quality," *Scientometrics*, vol. 108, no. 1, pp. 441–455, Jul. 2016.
[26] A. Pareek, B. Khaladkar, R. Sen, B. Onat, V. Nadimpalli, M. Agarwal, and N. Keene, "Striim: A streaming analytics platform for real-time business decisions," in *Proc. Int. Workshop Real-Time Bus. Intell. Anal.*, 2017, pp. 1–8.
[27] A. Pareek, B. Khaladkar, R. Sen, B. Onat, V. Nadimpalli, and M. Lakshminarayanan, "Real-time ETL in Striim," in *Proc. Int. Workshop Real-Time Bus. Intell. Anal. (BIRTE)*, 2018, p. 3.
[28] D. Q. Tu, A. Kayes, W. Rahayu, and K. Nguyen, "ISDI: A new window-based framework for integrating IoT streaming data from multiple sources," in *Proc. Int. Conf. Adv. Inf. Netw. Appl.* Springer, 2019, pp. 498–511.
[29] E. Satir and H. Isik, "A Huffman compression based text steganography method," *Multimedia Tools Appl.*, vol. 70, no. 3, pp. 2085–2110, Jun. 2014.
[30] D.-H. Xu, A. S. Kurani, J. D. Furst, and D. S. Raicu, "Run-length encoding for volumetric texture," *Heart*, vol. 27, no. 25, pp. 452–458, 2004.
[31] M. Sharma, "Compression using Huffman coding," *Int. J. Comput. Sci. Netw. Secur.*, vol. 10, no. 5, pp. 133–141, 2010.

QUANG-TU DOAN is currently pursuing the Ph.D. degree with the Department of Computer Science and Computer Engineering, La Trobe University, Australia. He works on various advanced topics on data integration, data mining, software engineering, and information and knowledge management. He is also an active member with the Data Engineering and Knowledge Management Laboratory, La Trobe University.

A. S. M. KAYES received the Ph.D. degree from the Swinburne University of Technology, Australia, in 2014. He is currently a Lecturer of cyber security with the Department of Computer Science and Information Technology, La Trobe University, Australia. His research interests include information modeling, cyber security, access control, big data, the IoT streaming data integration and indexing, advanced data analytics, and privacy preservation.

WENNY RAHAYU is currently a Professor and the Head of the School of Engineering and Mathematical Sciences, La Trobe University, Australia. Prior to this appointment, she was the Head of the Department of Computer Science and Information Technology, from 2012 to 2014. In the last ten years, she has published two authored books, three edited books, and more than 150 research papers in international journals and conference proceedings. Her research interest includes the integration and consolidation of heterogeneous data and systems to support a collaborative environment within a highly data-rich environment.

KINH NGUYEN received the B.Sc. and M.Sc. degrees (Hons.) from Canterbury University, New Zealand, and the Ph.D. degree in computer science from La Trobe University, Australia. He is currently a Lecturer of computer science with the Department of Computer Science and Information Technology, La Trobe University. His current research interests include data-intensive systems development, web information systems, semantics web, formal specification and rigorous software development processes, and model-driven engineering.