IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 31, 2018, accepted January 16, 2019, date of publication February 4, 2019, date of current version March 1, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2897154

Near-Optimal Data Structure for Approximate Range Emptiness Problem in Information-Centric Internet of Things

XIUJUN WANG[1], ZHI LIU[2], (Member, IEEE), YAN GAO[1], XIAO ZHENG[1], XIANFU CHEN[3], (Member, IEEE), AND CELIMUGE WU[4], (Senior Member, IEEE)

[1]School of Computer Science and Technology, Anhui University of Technology, Ma'anshan 243032, China
[2]Department of Mathematical and Systems Engineering, Shizuoka University, Hamamatsu 432-8561, Japan
[3]VTT Technical Research Centre of Finland Ltd., 90571 Oulu, Finland
[4]Department of Computer and Network Engineering, Graduate School of Informatics and Engineering, The University of Electro-Communications, Tokyo 182-8585, Japan

Corresponding author: Zhi Liu (liu@ieee.org)

This work was supported in part by the Natural Science Foundation of China under Grant 61402008, in part by the Natural Science Foundation of Anhui Province under Grant 1408085QF128, and in part by the Major Program of the Natural Science Foundation of the Anhui Higher Education Institutions of China under Grant KJ2014ZD05.

ABSTRACT The approximate range emptiness problem requires a memory-efficient data structure $\mathscr{D}$ to approximately represent a set S of n distinct elements chosen from a large universe $U = \{0, 1, \cdots, N-1\}$ and answer an emptiness query of the form "$S \cap [a; b] = \emptyset$?" for an interval $[a; b]$ of length L ($a, b \in U$), with a false positive rate ε. The designed $\mathscr{D}$ for this problem can be kept in high-speed memory and quickly determine approximately whether a query interval is empty or not. Thus, it is crucial for facilitating online query processing in the information-centric Internet of Things applications, where the IoT data are continuously generated from a large number of resource-constrained sensors or readers and then are processed in networks. However, the existing works on the approximate range emptiness problem only consider the simple case when the set S is static, rendering them unsuitable for the continuously generated IoT data. In this paper, we study the approximate range emptiness problem over sliding windows in the IoT Data streams, denoted by *ε-ARESD-problem*, where both insertion and deletion are allowed. We first prove that, given a sliding window size n and an interval length L, the lower bound of memory bits needed in any data structure for *ε-ARESD-problem* is $n \log_2(nL/\varepsilon) + \Theta(n)$. Then, a data structure is proposed and proved to be within a factor of 1.33 of the lower bound. The extensive simulation results demonstrate the advantage of the efficiency of our data structure over the baseline approach.

INDEX TERMS Approximate range emptiness, data structure, information-centric network, Internet of Things, space lower bound.

I. INTRODUCTION

Approximate membership is an essential and well-studied data structure problem [1]–[3]. Given a set S of n elements chosen from a large universe U, the elements in S are called members, and the other elements are nonmembers. The task of this problem is to represent S by a memory-efficient data structure $\mathscr{D}$, and then, for a query element $q \in U$,

(1) $\mathscr{D}$ outputs *member* with a probability 1, if q truly belongs to S;

(2) $\mathscr{D}$ outputs *nonmember* with a probability no less than $1 - \varepsilon$, if q truly does not belong to S.

Note that when $q \notin S$, but $\mathscr{D}$ wrongly outputs *member*, we call this as a false positive error. Then, the false positive rate is the faction of the elements from $U - S$, for which $\mathscr{D}$ wrongly outputs *member*.

Goswami *et al.* [4] generalize the ε-approximate membership problem from asking whether a query element q ($q \in U$) belongs to S or not to asking whether a query interval $I = [a, b]$ of length L ($[a, b]$ contains L integers) intersects with S or not, i.e., whether $[a, b] \cap S$ is $\emptyset$ or not. This generalized problem is called the approximate range emptiness problem,

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

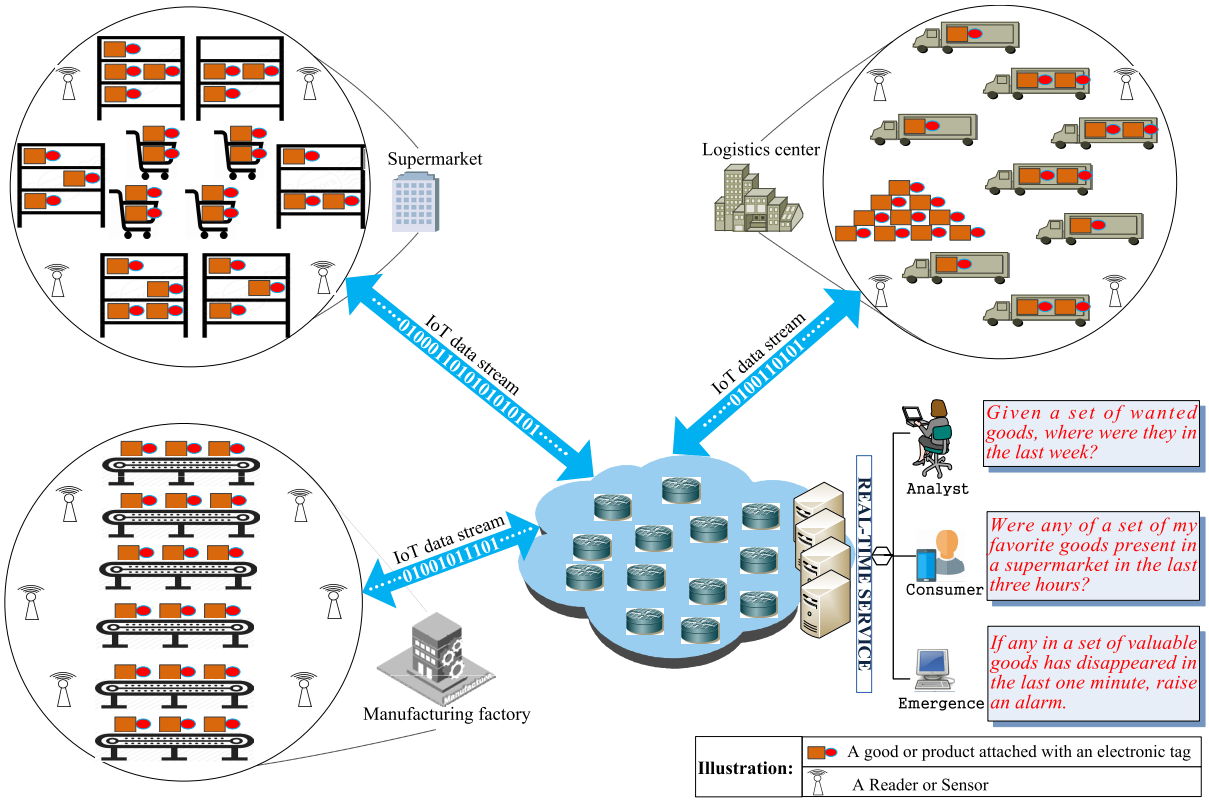

FIGURE 1. Example of the range emptiness problem in IoT data streams in Information-Centric Internet of Things environment.

as studied in [4], and is defined as follows: the task of this problem is to represent a set of S of n elements selected from U by a memory-efficient data structure $\mathscr{D}$, and then, for a query interval I of length L,

(1) $\mathscr{D}$ outputs *non-empty* with a probability 1, if $I \cap S \neq \emptyset$;

(2) $\mathscr{D}$ outputs *empty* with a probability at least $1-\varepsilon$, if $I \cap S$ truly equals to $\emptyset$.

A. PRACTICAL SIGNIFICANCE FOR IC-IOT

In the Information-Centric Internet of Things (IC-IoT), IoT data, such as radio frequency identification (RFID) tag data [6], or sensor data [7] are continuously generated from a large number of resource-constrained devices, and then are delivered and cached throughout a network for the convenience of close data copy retrieval, see [6], [8]–[13]. *However, as the potential amount of IoT data is huge, (for example, an RFID tag-ID can be* 128*-bit long [6]— there are* 2^{128} *possible tag-IDs), and these IoT data are continuously generated at a fast speed [8], [14]–[17], [19], quickly processing a query interval over IoT data streams poses a challenge for facilitating real-time services and monitoring in these systems.* In Fig. 1, we show that, in three different places, namely, factories, logistic centers, and supermarkets, goods with attached electronic tags can be connected to information-centric networks, and the tag-ID of a good is used as its resource identifier. The IoT data of a good in this scenario can be its tag-ID as well as other types of information, such as manufacturer name, date of production, location, and transportation destination. Many ordinary goods arrive and leave continuously and quickly to and from each place (for example, a factory or a supermarket) and there are also many such places. Therefore, a number of IoT data streams are continuously collected by the RFID readers or sensors deployed in these places for monitoring goods, and then cached in the networking nodes. Now suppose that we want to provide real-time monitoring of a specific group of goods, say a set of highly valuable goods with tag-IDs belonging to an interval I; naturally, we need to solve the approximate range emptiness problem. To be more specific, in this scenario, whenever a sequence of an IoT data stream, denoted by S, which contains the information of a set of newly arrived goods in one place, is generated and cached in a nearby networking node. We can construct a memory-efficient data structure $\mathscr{D}$ that approximately represents S, such that $\mathscr{D}$ can be stored and maintained at every networking node, say a switcher or router, without consuming too many memory resources. After that, the users at different geographical locations can quickly determine whether any good in I is present in S or not by querying the cached data structure $\mathscr{D}$ in a nearby node. This is significantly faster than the enumerative method whereby the user tries to retrieve each good $e \in I$ by sending out a request (an INTEREST packet [8], [9]) in which the resource identifier (data name) of e is set as the handle, and then waits for the reply, which is obviously a time-consuming process when none of the goods in I are actually present in S.

Another important issue in Fig. 1 is how to facilitate fine-grained access control to the IoT data streams. To be more specific, we want distributed users at different geographical locations to be quickly informed of the updates of the IoT data that are related to his or her own goods. For example, the goods belonging to a dealer can be represented by a number of query intervals of resource identifiers (such as the query intervals of tag-IDs), and every dealer needs to be quickly notified of the real-time logistical information of his or her own goods. For such a scenario, suppose that S is a sequence of IoT data streams that are newly collected and updated to a nearby networking node; after a memory-efficient data structure $\mathscr{D}$ that approximately represents S is constructed and cached throughout the networks, we can quickly identify those distributed users for which the query intervals overlap with S, and then inform them timely.

Last but not least, suppose the data servers (or routers) in IC-IoT systems cache a set of records S, where each record can be a long bit-vector, and this set can only be stored to disk completely of the data-servers in IC-IoT systems. Whenever a user issues a request to retrieve all the records within a time interval $[a, b]$, the information resource sharing and management in IC-IoT needs to search this set by loading each subpart of the set into the memory of a data server in order to find all the records within the time interval $[a, b]$. For such a scenario, we can use a space-efficient data structure, denoted by $\mathscr{D}$, which is capable of being maintained in the high-speed memory, to approximately represent S in each data-server. Then, when we want to list all the records within $[a, b]$, we first query $\mathscr{D}$ to avoid the slow disk accesses, if there does not exist a record in S within $[a, b]$. Please refer to [20] for other applications.

B. MOTIVATIONS

Goswami *et al.* [4] study the approximate range emptiness problem by focusing on the simple case in which S (the set to be represented) is static, rendering the derived lower bound and proposed data structure [4] unsuitable for the more general scenarios of IoT data streams, which are commonly seen in IoT environments [8], [14]–[17], [19]–[22].

In this work, we study the ε-approximate range emptiness problem in the sliding window model [1] for a data stream δ in IC-IoT systems, because of the following:

(1) IoT data usually occur in the form of data streams in IC-IoT systems, see [7], [14], [20], [21] for example.

(2) Usually, we are more interested in new data (for example, the latest n elements in an IoT data stream) rather than old data in IoT data streams, and the value or the hit ratio of cached IoT data in IC-IoT systems decreases with time [8], [14], [15], [19], [20], [23].

[1] The sliding window model in data streams is first raised by [23]; there are many studies on the sliding window model for various types of queries or estimations in data streams, such as [24].

(3) The designed data structure $\mathscr{D}$ can obviously provide an efficient solution for a wide range of applications in IC-IoT systems.

For example, in warehouse management in a smart supermarket, the readers continuously collect the tag-IDs (96-bit or 128-bit arrays) of newly arrived goods, see Fig. 1. In such scenarios, we care more about the arrival of the goods that have very short shelf-life or expiration time. A VIP customer may want to query whether a type of shipping good with their tag-ID belonging to a given interval $[a, b]$ has arrived or not (this VIP customer may also ask whether any of a desired set of valuable goods with their tag-ID belonging to a given interval $[a, b]$ is present in the n newly arrived shipping goods), so that he or she can obtain this good as soon as it arrives. Under this situation, a slow SQL statement in the database of servers cannot provide real-time service for this type of query when there are many VIP customers. Another example is in intelligent transportation, where each vehicle carries an electronic label, and a police officer wants to monitor a set of suspicious vehicles among the vehicles occurring on one main road of a large city in the last hour. For such scenarios, the designed data structure $\mathscr{D}$ can be quickly loaded into the IC networks, and then the police officer can rapidly obtain the information of suspicious vehicles.

C. CONTRIBUTIONS

Our main contributions in this paper are summarized as follows:

(1) We obtain a space lower bound for any data structure that solves the ε-approximate range emptiness problem over sliding windows for data streams in IC-IoT.

(2) We design a data structure, denoted by $\mathbf{A}$, for this problem, and prove that: the number of bits in $\mathbf{A}$ is within in a factor of 1.33 *of the space lower bound.*

(3) We conduct extensive simulations to verify the performance of the designed data structure $\mathbf{A}$, and the results clearly demonstrate its superior performance.

The rest of the paper is organized as follows. Section II presents related works. Section III presents and proves the space lower bound for the ε-approximate range emptiness problem in the sliding window model for data streams. Section IV provides extensive experimental results with synthetic datasets. Section V concludes the paper.

II. RELATED WORKS

A. THE APPROXIMATE MEMBERSHIP QUERY

There are many studies concerning the approximate membership problem, both in a data set S and data streams under different assumptions [1]–[3], [5], [25]–[30]. For example, a Bloom filter [3] is the first data structure for the approximate membership problem in a data set S. Given a false positive rate ε, and a set S of n elements from U, the number of memory bits needed in a Bloom filter is $\log_2(e)n\log_2(1/\varepsilon)$, and the lookup time for a query element $q \in U$ is $\log_2(1/\varepsilon)$. When the size of the set S is unknown, Pagh *et al.* [25] show

TABLE 1. Symbol definitions.

Symbol	Meaning
U	$U = \{0, 1, \cdots, u-1\}$ is a large universe that contains u elements.
δ	$\delta = e_1, e_2, \cdots, e_t, \cdots$ is a data stream, where each element e_t arrives at each time point $t > 0$ and $e_t \in U$.
W^t	W^t is the sliding window at each time point $t > 0$ and $W^t = e_{max\{1,t-n+1\}}, \cdots, e_{t-1}, e_t$ (W^t contains the latest n elements in δ at time point t).
n	The size of a sliding window W^t.
L	The length of a query interval.
$[a, b]$	A query interval of length L, i.e., $[a, b] = \{a, a+1, .., a+L-1\}$.
$\mathscr{A}$	A data structure that solves ε-**ARESD-problem**.
$\mathscr{A}(t, [a, b])$	The output result of $\mathscr{A}$ on a query interval $[a, b]$ at time point $t > 0$, where $\mathscr{A}(t, [a, b]) = 0$ means that $\mathscr{A}$ determines $W^t \cap [a, b] = \emptyset$ and $\mathscr{A}(t, [a, b]) = 1$ means that $\mathscr{A}$ determines $W^t \cap [a, b] \neq \emptyset$.
ε	ε is a given false positive rate and $\varepsilon \in (0, 1)$.
$\mathbf{B}$ and $\mathbf{D}$	Two arrays that are designed to prove the space lower bound of ε-**ARESD-problem** in Theorem .3.2.

that the space lower bound of the approximate membership problem is $(1 - o(1))n \log_2(1/\varepsilon) + \Omega(n \log_2(\log_2(n)))$. The time-decaying Bloom filter (T-DBF) [29], [30] is proposed to solve the approximate membership problem over sliding windows in data streams. The key idea is to substitute the m-bit array used in a Bloom filter with an array of m counters, such that the time information of each element in a data stream can be recorded into these counters. Given a false positive rate ε, and a sliding window size n, the number of bits used in T-DBF is $O(n \log_2(n) \log_2(1/\varepsilon))$.

B. THE APPROXIMATE RANGE EMPTINESS

Goswami *et al.* [4] first define and study the approximate range emptiness problem, which generalizes the approximate membership query problem from querying a single point "$q \in S$?" to an interval "$[a, b] \cap S = \emptyset$?" ($q, a, b \in U$). As far as we know, it is currently the only work that is relevant to the approximate range emptiness problem. Both the space lower bound and an optimal data structure are proposed in [4].

C. STREAMING DATA PROCESSING IN IC-IoT SYSTEMS

There are some studies [8], [14]–[17], [19], [30] on efficient streaming processing in IC-IoT systems. For example, in [15], Ahlgren *et al.* propose a method for finding the newest element in an IoT data stream. Li *et al.* [16] focus on the data authorization problem of IoT data streams, and propose a distributed secure data sharing structure, which contains a self-update mechanism for this problem.

III. SPACE LOWER BOUNDS

In this section, we prove a space lower bound for any data structure that solves the ε-approximate range emptiness problem in the sliding window model for data streams.[2]

A. PROBLEM DEFINITION

First, we list all the parameters that are needed for the definition of the ε-approximate range emptiness problem in the sliding window model for a data stream, which is abbreviated as ε-**ARESD-problem**.

(1) Let $\delta = e_1, e_2, \cdots, e_t, \cdots$ be a data stream, where each element e_t arrives at each time point $t > 0$ and is from a large universe $U = \{0, 1, \cdots, u-1\}$ of u elements.

(2) Let n be the size of a sliding window ($n \ll u$); then, at every time point $t > 0$, the current sliding window is defined as $W^t = e_{max\{1,t-n+1\}}, \cdots, e_t$, where e_t is the newest element in W^t, e_{t-1} is the second newest, and so on. Obviously, when $t > n-1$, the sliding window contains the n latest observed elements in δ, and when $t \leq n-1$, all the t elements $e_1, e_2, \cdots, e_t$ are in the current window.

(3) Let $[a, b]$ denote a query interval of length L, where $a \in U$ and $b = a + L - 1$ and $\varepsilon \in [0, 1]$ be a given false positive rate.

(4) Let $\mathscr{A}$ denote a data structure for the ε-**ARESD-problem**, and $\mathscr{A}(t, [a, b]) \in \{0, 1\}$ denote the result of $\mathscr{A}$ on a query interval $[a, b]$ at time point $t > 0$. When $\mathscr{A}(t, [a, b]) = 0$, $\mathscr{A}$ determines that $W^t \cap [a, b] = \emptyset$; otherwise, $\mathscr{A}$ determines that $W^t \cap [a, b] \neq \emptyset$.

Definition 1: The task of the ε-**ARESD-problem** is to approximately represent the sliding window W^t at each time point $t > 0$ in a memory-efficient data structure $\mathscr{A}$, by only a one-pass scan of the data stream δ. There are two processing steps for the data structure $\mathscr{A}$ at each $t > 0$:

Updating step: Before we give the details of the **Updating step** in $\mathscr{A}$, we point out two basic facts: $\mathscr{A}$ is given a newly arrived element e_t from δ, at each time point $t > 0$ (owing to the one-pass scan requirement, $\mathscr{A}$ does not have access to the previous element before time point t, unless this element is stored explicitly); $\mathscr{A}$ needs to evict the information of the expired element e_{t-n} from itself when $t > n$, and then store the information of e_t into itself. These two facts implicitly lead to $\mathscr{A}$ supporting an updating operation that changes its memory status from representing W^{t-1} at a time point $t-1$ to representing W^t at the new time point t by only taking e_t as the input. Furthermore, this updating step needs to guarantee ($Q1$) and ($Q2$) shown below in the **Querying step**, at each time point $t > 0$.

Querying step: $\mathscr{A}$ needs to answer a query interval $[a, b]$ such that for any $t > 0$, the following two statements are true:

($Q1$) $Pr[\mathscr{A}(t, [a, b]) = 1] = 1$, if $[a, b] \cap W^t \neq \emptyset$;

($Q2$) $Pr[\mathscr{A}(t, [a, b]) = 1] \leq \varepsilon$, if $[a, b] \cap W^t = \emptyset$.

Note that the definition of the ε-**ARESD-problem** is quite different from the problem studied in [4], where the authors only consider the simple case that the set S to be stored is static, i.e., no insertion or deletion is allowed. In this study, we consider a more general case that the set to be stored is a sliding window W^t, which changes with time.

B. SPACE LOWER BOUND ANALYSIS

The space lower bound for any data structure that solves the ε-**ARESD-problem** is proved based on an encoding argument, which is commonly used in analyzing the lower bounds in the research areas of algorithms and complexity [4], [25], [26]. The high-level idea is as follows: first, we assume

[2]For ease of illustration, IoT data stream is abbreviated as data stream.

that there is a data structure $\mathscr{A}$ for the ε-**ARESD-problem**; second, we can use $\mathscr{A}$ and two additional arrays, **B** and **D**, to perfectly encode an ordered sequence **s** that contains n unique elements chosen from U; lastly, the lower bound of $\mathscr{A}$ is derived from the information gap between **s** and two arrays, **B** and **D** (the difference between the number of bits needed to perfectly represent a sequence **s** and the total number of bits in the arrays **B** and **D**). The main idea for designing the two additional arrays, **B** and **D**, is to correct the errors made by $\mathscr{A}$. Note that by Definition 1, $\mathscr{A}$ is allowed to have a false positive rate ε.

Theorem 2: Let c be equal to $u/(nL)$. Suppose that $c \geq 1$ and $\varepsilon \geq 1/c$. Let $|\mathscr{A}|$ represent the number of bits in any data structure $\mathscr{A}$ that solves the ε-**ARESD-problem**; then, we have the following

$$|\mathscr{A}| \geq n[\log_2(1/\varepsilon) + \log_2 L + \log_2 n] - O(n) = S_{LB}. \quad (1)$$

Proof: We choose the sequences of n unique elements following two steps:

STEP1: We divide the universe U into u/L disjoint subsets as $U_1 = \{0, 1, \cdots, L-1\}$, $U_2 = \{L, L+1, \cdots, 2*L-1\}$,$\cdots$,$\{(\frac{u}{L}-1)L, (\frac{u}{L}-1)L+1, \cdots, (\frac{u}{L}-1)L+L-1\}$;

STEP2: we choose n different subsets out of those u/L disjoint subsets, and then pick exactly one element from those chosen n subsets.

There are $(\frac{u}{L})^n L^n$ possible sequences, and the number of bits required to represent all these sequences exactly is $\log_2((\frac{u}{L})^n L^n) = \log_2((\frac{u}{L})^n) + n\log_2(L)$ bits. [3]

Now, let us say that we generate a sequence $\mathbf{s} = (s_1, s_2, \cdots, s_n)$ by STEP1 and STEP2 above. Assume that $\mathscr{A}$ is a data structure that is capable of solving the ε-**ARESD-problem**, and initially at time point 0, $\mathscr{A}$ does not store any elements. Then, the n elements in sequence **s** are given to $\mathscr{A}$ one by one. To be more specific, s_1 is given to $\mathscr{A}$ at time point 1 ($\mathscr{A}$ uses the **Updating step** in Definition 1 for storing s_1 at time point 1), s_2 is given at time point 2 ($\mathscr{A}$ uses the **Updating step** for storing s_2 at time point 2), and so on. Then, after time point n, we say that $\mathscr{A}$ is an approximate representation of sequence **s**, because the answers to the query interval of $\mathscr{A}$, which are $\mathscr{A}(n+1, [a, a+L-1])$, $a \in \{0, 1, \cdots, u-L\}$, can help us to recover the sequence **s** approximately (some answers can be wrong owing to the false positive rate of $\mathscr{A}$).

For the exact recovery of sequence **s**, we need to specify two pieces of information:

Info-part-a: The n subsets from $\{U_1, U_2, ..., U_{\frac{u}{L}}\}$ in which the n elements $s_1, s_2, ..., s_n$ are truly contained. For ease of illustration, let $U_{k_i} \in \{U_1, U_2, ..., U_{\frac{u}{L}}\}$ ($k_i \in \{1, 2, ..., \frac{u}{L}\}$) represent the interval that actually contains the element s_i of sequence **s**, for $i \in \{1, 2, ..., n\}$. Thus, for this part, we need to specify the values of $k_1, k_2, ..., k_n$.

Info-part-b: Which element contained in U_{k_i} is actually the element s_i, for all $i \in \{1, 2, ..., n\}$.

[3] We consider only these $(\frac{u}{L})^n L^n$ possible sequences, because the number of this type of sequences is sufficiently large for us to obtain a reasonable lower bound, when $u \geq nL$.

Now, we are going to show how to partially recover sequence **s** by issuing interval queries to $\mathscr{A}$. U_i is actually an interval of length L, because U_i represents L consecutive integers. We focus on these u/L intervals, and query each U_i, $i \in \{0, 1, \cdots, \frac{u}{l}\}$ at each time point $n+j, j \in \{1, 2, \cdots, n\}$. Let q_i^j to be the result related to query U_i at time point $(n+j)$ to $\mathscr{A}$. The querying process for $U_1, U_2, .., U_{\frac{u}{L}}$ is shown in Procedure 1.

Procedure 1 Querying Process for All $\frac{u}{L}$ Intervals $U_1, U_2, \cdots, U_{\frac{u}{L}}$ at n Time Points

1:**For** $j = 1$**to** n ||**for time point* $j+n$*||
2: **For** $i = 1$**to** $\frac{u}{L}$ ||**for each interval* U_i*||
3: $q_i^j = \mathscr{A}(n+j, U_i)$; ||**store the query result regarding interval* U_i *at each time point***||**End For**
4: $\mathscr{A}$ uses the **Updating step** in definition 1 for evicting the information of the expired s_i, and then stores an element $u \notin U$, at time point $n+j$;

||* *because the universe* $U = \{0, 1, .., u-1\}$, u *is an element that is not contained in* U *||
End For

From the above process, we can get an n-bit vector $\mathbf{q}_i = (q_i^1, q_i^2, \cdots, q_i^n)$, where $q_i^j \in \{0, 1\}, j \in \{1, 2, \cdots, n\}$, for each interval $U_i, i \in \{1, 2, \cdots, \frac{u}{L}\}$. According to the values of the n-bit vectors, these $\frac{u}{L}$ intervals $U_1, U_2, .., U_{\frac{u}{L}}$ can be classified into $n+1$ categories as follows:

$$\begin{cases} C_0 = \{U_i, |\mathbf{q}_i = (0, *, *, ...*), & i \in \{1, 2, .., u/L\}\}, \\ C_1 = \{U_i, |\mathbf{q}_i = (1, 0, *, ...*), & i \in \{1, 2, .., u/L\}\}, \\ C_2 = \{U_i, |\mathbf{q}_i = (1, 1, 0, ...*), & i \in \{1, 2, .., u/L\}\}, \\ \vdots & \\ C_n = \{U_i, |\mathbf{q}_i = (1, 1, 1, ..., 1), & i \in \{1, 2, .., u/L\}\}, \\ \text{where } * \text{ stands for either 0 or 1.} & \end{cases} \quad (2)$$

By (Q1) and (Q2) in Definition 1, we have two facts:

(S1) all the n elements $\mathbf{s} = (s_1, s_2, \cdots, s_n)$ cannot be contained in an interval from C_0;

(S2) the j-th element in **s**, which is s_j, can only possibly be in an interval from $\bigcup_{k=j}^{n} C_k$, for any $j \in \{1, 2, ..., n\}$.

We present the explanation for (S1) as follows. If an interval U_i ($i \in \{1, 2, \cdots, \frac{u}{L}\}$) really contains one of the n elements in **s**, then at the time point $n+1$, q_i^{n+1} must be equal to 1, by (Q1) in Definition 1 (if $U_i \cap \mathbf{s} \neq \emptyset$, $\mathscr{A}(n+1, U_i)$ is equal to 1 with a probability of 1). In particular, at the time point $n+1$, before we insert the element u into $\mathscr{A}$, all the n elements in sequence **s** are not evicted from $\mathscr{A}$. Thus, the query result to any interval that contains one of the n elements in **s** must be 1.

Next, we explain (S2) for $j = 1$. When $j = 1$, (S2) means that s_1 can only possibly be contained in an interval from $\bigcup_{k=1}^{n} C_k$. We can obtain this statement because, by Procedure 1, we know that s_1 is not evicted when we obtain q_i^1 at the

time point $n+1$ for each $i \in \{1, 2, \cdots, \frac{u}{L}\}$. Thus, the interval U_{k_1} that truly contains s_1 must have $q^1_{k_1} = 1$, and then we know $s_1 \in \bigcup_{k=1}^{n} C_k$. The other cases for $j > 1$ in (S2) can be obtained similarly to the case for $j = 1$.

Let the number of intervals contained in C_i be represented by $c_i \geq 0$, for $i \in \{1, 2, \cdots, \frac{u}{L}\}$. By $(Q2)$ in Definition 1, we know that for any interval U_i that does not contain any element of sequence $\mathbf{s}$, the query result $\mathscr{A}(n+1, U_i)$ is equal to 1 with a probability ε. Because there are $u/L - n$ intervals that do not contain any element of sequence $\mathbf{s}$, we have

$$c_0 = (u/L - n)(1 - \varepsilon). \tag{3}$$

Considering that there are u/L query intervals in total, $U_1, .., U_{\frac{u}{L}}$, we obtain

$$\sum\nolimits_{k=1}^{n} c_i = n + (u/L - n)\varepsilon. \tag{4}$$

Assume that s_1 truly belongs to U_{k_1}; then, we have $q^1_{k_1} = 1$ at the time point $n+1$. At the time point $n+2$, there are two possible cases: $q^2_{k_1} = 1$ with a probability ε and $q^2_{k_1} = 0$ with a probability $1 - \varepsilon$. If $q^2_{k_1} = 0$, according to the definition of n categories shown in (2), U_{k_1} is contained in C_1; otherwise ($q^2_{k_1} = 0$), U_{k_1} can be only contained in $\bigcup_{k=2}^{n} C_k$. In summary, U_{k_1} can be contained in C_1 with a probability of $1-\varepsilon$, and U_{k_1} is in $\bigcup_{k=2}^{n} C_k$ with a probability of ε. The other cases for $i \in \{2, 3, .., n\}$ can be analyzed in a similar way.

Generally, assume that s_j belongs to U_{k_j} for each $j \in \{1, 2, ..., n\}$; we use $Z_j = 0$ (for each interval U_{k_j}) to denote the event that $U_{k_j} \in C_j$, and $Z_j = 1$ otherwise ($U_{k_j} \in \bigcup_{k=j+1}^{n} C_k$). By the above analysis, we have the following:

$$\begin{cases} Pr(Z_j = 0) = 1 - \varepsilon, & j \in \{1, 2, .., n-1\}, \\ Pr(Z_n = 0) = 1. \end{cases} \tag{5}$$

Consider that there are n intervals $\hat{\mathbf{U}} = \{U_{k_1}, U_{k_2}, .., U_{k_n}\}$ in total for the n elements $s_1, s_2, ..., s_n$; thus we know, on average, there are $(n-1)\varepsilon$ intervals in $\hat{\mathbf{U}}$ that have $Z_j = 1$, and all the other $(n-1)(1-\varepsilon)+1$ intervals have $Z_j = 0$.

Therefore, we can see that by the output of $\mathscr{A}$, we cannot accurately recover sequence $\mathbf{s}$. In the following, we show the construction of two arrays $\mathbf{B}$ and $\mathbf{D}$ that correct the errors made by $\mathscr{A}$, and help us to recover sequence $\mathbf{s}$ perfectly.

Without loss of generality, we assume that $(n-1)\varepsilon = l$ is an integer, $Z_j = 1$ for $j \in \{1, 2, .., l\}$, and $Z_j = 0$ for $j \in \{l+1, l+2, .., n\}$. The first additional array $\mathbf{B}$ contains three parts as follows:

(B_1) For each $Z_j, j \in \{1, 2, .., l\}$, first, we need to specify the index $\mathcal{I}_j$ of the category that truly contains U_{k_j}, and then which interval from $C_{\mathcal{I}_j}$ ($\mathcal{I}_j \in \{j+1, j+2, .., n\}$) is U_{k_i}; this requires $\log_2(n) + \log_2(c_{\mathcal{I}_j})$ bits.

(B_2) For each $Z_j, j \in \{l+1, l+2, .., n\}$, we only need to specify which interval from C_j is U_{k_j}; this costs $\log_2 c_j$ bits.

(B_3) We need an n-bit vector to represent the values of each $Z_j, j \in \{1, 2, .., n\}$.

Thus, the number of bits in $\mathbf{B}$ is

$$|\mathbf{B}| = n + \sum_{j=1}^{l}(\log_2\ n + \log_2\ c_{\mathcal{I}_j}) + \sum_{j=l+1}^{n} \log_2\ c_j, \tag{6}$$

$$\text{where } \mathcal{I}_j \in \{j+1, j+2, .., n\} \quad \text{for } j \in \{1, 2, .., l\}, \tag{7}$$

$$c_j \geq 0 \quad \text{for } j \in \{1, 2, .., n\}, \tag{8}$$

$$\text{and } \sum\nolimits_{j=l}^{n} c_j = n + (u/L - n)\varepsilon. \tag{9}$$

We can see in (6) that each $c_{\mathcal{I}_j} \in \{j+1, j+2, .., n\}$ for $j \in \{1, 2, .., l\}$, and each c_i occurs exactly once in $\sum_{i=l+1}^{n} \log_2(c_i)$ of (6), for $i \in \{l+1, l+2, .., n\}$. Let us say that the number of times that $c_j, j \in \{1, 2, .., n\}$ occurs in $\sum_{j=1}^{l}(\log_2\ n + \log_2\ c_{\mathcal{I}_j})$ of (6) is $\alpha_i \geq 0$ times; then, the number of times that $c_i, i \in \{l+1, l+2, .., n\}$ occurs in (6) is $1 + \alpha_i \geq 1$ times ($\alpha_i \geq 0$), and we obtain the following:

$$\sum\nolimits_{i=1}^{n} \alpha_i = l \tag{10}$$

(see $\sum_{j=1}^{l} \log_2(c_{\mathcal{I}_j})$ shown in (6)).

We consider the maximization problem as follows:

$$z = \max_{c_1, c_2, .., c_n} \sum_{i=1}^{l} \alpha_i \log_2(c_i) + \sum_{i=l+1}^{n} (\alpha_i + 1)\log_2(c_i) \tag{11}$$

$$s.t. \begin{cases} \sum_{i=1}^{n} c_i = n + (u/L - n)\varepsilon, \\ c_i \geq 0 \quad (i = 1, 2, \cdots, n). \end{cases} \tag{12}$$

The formula shown in (11) is a concave function, which can be maximized by using the Lagrangian multiplier method. The details are shown in Appendix. The maximum of (11) is equal to the following:

$$n \log_2(1 + (c-1)\varepsilon) + \sum_{i=1}^{l} \alpha_i \log_2 \alpha_i + \sum_{i=l+1}^{n} (1 + \alpha_i) \log_2(1 + \alpha_i), \tag{13}$$

where $c = \frac{u}{nL}$. Next, we consider the maximization problem as follows:

$$v = \max_{\alpha_1, \alpha_2, .., \alpha_n} \sum_{i=1}^{l} \alpha_i \log_2 \alpha_i + \sum_{i=l+1}^{n} (1 + \alpha_i) \log_2(1 + \alpha_i) \tag{14}$$

$$s.t. \begin{cases} \sum_{i=1}^{n} \alpha_i = l, \\ \alpha_i \geq 0 \quad (i = 1, 2, \cdots, n). \end{cases} \tag{15}$$

Because both $\alpha_i \log_2(\alpha_i)$ and $(1+\alpha_i)\log_2(1+\alpha_i)$ are convex functions of the variable α_i, the linear combination of these functions shown in (14) is also a convex function. Thus, the maximal value of v can only occur at the boundary points. We choose the boundary point, where $\alpha_i = 0$ for $i \in \{1, 2, .., n-1\}$ and $\alpha_n = l$ (the values of this function over other boundary points are less than or equal to this point). Thus, the maximal value of v shown in (14) is equal to the following:

$$(l+1)\log_2(l+1) = ((n-1)\varepsilon + 1)\log_2((n-1)\varepsilon + 1). \tag{16}$$

Combining (6), (13), and (16), we can obtain the following:

$$|\mathbf{B}| \leq n\log_2(1+(c-1)\varepsilon) + ((n-1)\varepsilon+1)\log_2((n-1)\varepsilon+1), \quad (17)$$

where $c = u/(nL)$. So far, we have shown an additional array **B** that helps us to recover the n intervals in which the n elements of sequence **s** are truly contained. More specifically, by combining Procedure 1 and array **B**, we can complete the job needed by **Info-part-a**.

From now on, we know exactly which interval from $\{U_1, U_2, .., U_{\frac{u}{L}}\}$ is U_{k_j} for each element $e_j, j \in \{1, 2, .., n\}$. Next, we show the recovery process for **Info-part-b**, i.e., which element in U_{k_j} is e_j. To be more concrete, this is to recover the tailing $\log_2(L)$ bits for each element in sequence **s**.

Procedure 2 Querying Process for Recovering the Tailing $\log_2(L)$-Bit of Each Element e_j in Sequence **s**

0:**For** $j = 1$**to** n
||**assume the interval that truly contains* e_i *is* $I = [a, a+l-1]$*||
1:**For** $k = 1$**to** $\log_2(L)$ ||**for the j-th tailing bit of* e_i*||
2: let $I_1 = [a-\frac{l}{2}+1, a+\frac{l}{2}]$ and $I_2 = [a+\frac{l}{2}+1, a+l-1]$;
3: $X_j^k = \mathscr{A}(t, I_1)$ and $Y_j^k = \mathscr{A}(t, I_2)$;
||**t is the time point when we find which interval is I by Procedure 1 and the additional array* **B**.*||
||**I_1 and I_2 represent the left and right half of I, respectively.*||
4: **If** $X_j^k = 1$, $Y_j^k = 1$, check the additional array **D** for the true values of X_j^k and Y_j^k;
5: **If** $Y_j^k = 0$, set $I = I_1$ and the k-th bit of e_j is 0;
6: **If** $X_j^k = 0$, set $I = I_2$ and the k-th bit of e_j is 1;
7:**End For End For**

In Procedure 2, we show how to find the tailing $\log_2(L)$ bits for each $e_j, j \in \{1, 2, .., n\}$. Now, we show the number of bits needed for the additional array **D**. We know from Line 4 that when both X_j^k and Y_j^k are equal to 1, we need to specify where e_j is located. This costs us exactly one bit. Next, we calculate the number of times that $X_j^k = Y_j^k = 1$, for $k = 1, 2, .., \log_2(L)$. Considering that only one of I_1 and I_2 truly contains e_i, we know that the query answer to the other interval is 1 because of the false positive rate of $\mathscr{A}$ or the existence of some other element $e_i \in \mathbf{s}$ $(i \neq j)$ in interval I_1 or I_2. Without loss of generality, we assume that e_i is truly contained in I_1. Then, $Y_i^j = 1$ occurs when one of the following two events is true:

E_1: there is no element in **s** contained in I_2, and $Y_i^j = \mathscr{A}(t, I_2) = 1$;

E_2: there is at least one element $e_i \in \mathbf{s}$ $(i \neq j)$ contained in I_2 and $Y_i^j = \mathscr{A}(t, I_2) = 1$.

The probability that one element is located in an interval of length L is equal to L/u, and the probability that all the $n-1$ elements are not contained in I_2 is $(1-L/u)^{n-1}$. We can therefore obtain the following:

$$\begin{aligned} &Pr(E_1) + Pr(E_2) \\ &= (1-L/u)^{n-1}\varepsilon + (1-(1-L/u)^{n-1}) \\ &\approx (1-\frac{(n-1)L}{u})\varepsilon + \frac{(n-1)L}{u} < \varepsilon + (1-\varepsilon)\frac{1}{c}, \end{aligned} \quad (18)$$

where $c = u/(nL)$. The approximation in (18) comes from the fact that $u \gg n$ in the scenario of the approximate range emptiness problem, see [4]. Considering that we have $\log_2(L)$ queries for each e_i, and there are n elements, we thus have at most $\log_2(L)n(\varepsilon+(1-\varepsilon)\frac{1}{c})$ query errors, and each bit in array **D** can correct exactly one error. Therefore, the size of array **D**, denoted by $|\mathbf{D}|$, must satisfy the following:

$$|\mathbf{D}| \leq (\varepsilon + (1-\varepsilon)/c)n\log_2(L). \quad (19)$$

By Procedures 1 and 2, and the construction of array **B** and **D**, we know that we can recover sequence **s** perfectly. Thus, we have the following ($c = u/(nL)$):

$$\begin{aligned} |\mathscr{A}| + |\mathbf{B}| + |\mathbf{D}| &\geq \log_2(\frac{u}{L})^{\underline{n}} + n\log_2 L \\ &= \log_2(cn)^{\underline{n}} + n\log_2 L \\ |\mathscr{A}| &\geq \log_2((cn)^{\underline{n}}) + n\log_2 L - |\mathbf{B}| - |\mathbf{D}| \end{aligned} \quad (20)$$

By using inequalities (17) and (19), we obtain:

$$\begin{aligned} |\mathscr{A}| \geq\ & \log_2((cn)^{\underline{n}}) + n\log_2 Ln\log_2(1+(c-1)\varepsilon) \\ & - ((n-1)\varepsilon+1)\log_2((n-1)\varepsilon+1) \\ & - (\varepsilon + (1-\varepsilon)/c)n\log_2 L. \end{aligned} \quad (21)$$

By Stirling's formula (see [31, Lemma 7.3]), we have:

$$(cn)^{\underline{n}} = \frac{(cn)!}{((c-1)n)!} \geq e^{-n}(cn)^n(\frac{c}{c-1})^{(c-1)n}\frac{1}{2}. \quad (22)$$

By substituting (22) into (21), we derive the following:

$$\begin{aligned} |\mathscr{A}| \geq\ & n\log_2(cn) - n\log_2 e + n\log_2((\frac{c}{c-1})^{c-1}) - 1 \\ & + n\log_2 L - n\log_2(1+(c-1)\varepsilon) \\ & - ((n-1)\varepsilon+1)\log_2((n-1)\varepsilon+1) \\ & - (\varepsilon+(1-\varepsilon)/c)n\log_2 L \\ \geq\ & n\log_2(\frac{c}{1+(c-1)\varepsilon}) + n(1-\varepsilon-\frac{1-\varepsilon}{c})\log_2 L \\ & + n\log_2 n - n\log_2 e + n(c-1)\log_2(c/(c-1)) \\ & - 1 - (\varepsilon n)\log_2(\varepsilon n) \\ \geq\ & n\Big[\log_2 c - \log_2(c+\frac{1}{\varepsilon}-1) \\ & + \log_2\frac{1}{\varepsilon} + (1-\varepsilon-\frac{1-\varepsilon}{c})\log_2 L\Big] \\ & + n\log_2 n - 1.44n - 1 - \varepsilon n\log_2 n + \varepsilon n\log_2(1/\varepsilon) \\ =\ & n\Big[\log_2 c - \log_2(c+\frac{1}{\varepsilon}-1) + (1+\varepsilon)\log_2\frac{1}{\varepsilon} \\ & + (1-\varepsilon-\frac{1-\varepsilon}{c})\log_2 L + (1-\varepsilon)\log_2 n\Big] - 1.44n - 1 \\ \geq\ & n[\log_2(1/\varepsilon) + \log_2 L + \log_2 n] - O(n). \end{aligned} \quad (23)$$

The inequality in (23) is derived from the fact that $\log_2 e \approx 1.44$ and $(c-1)\log_2(c/(c-1)) \geq 0$, for $c \geq 1$. Then because $c = \frac{u}{nL} \geq \frac{1}{\varepsilon}$, $\log_2 c - \log_2(c+\frac{1}{\varepsilon}-1)$ in (23) is a

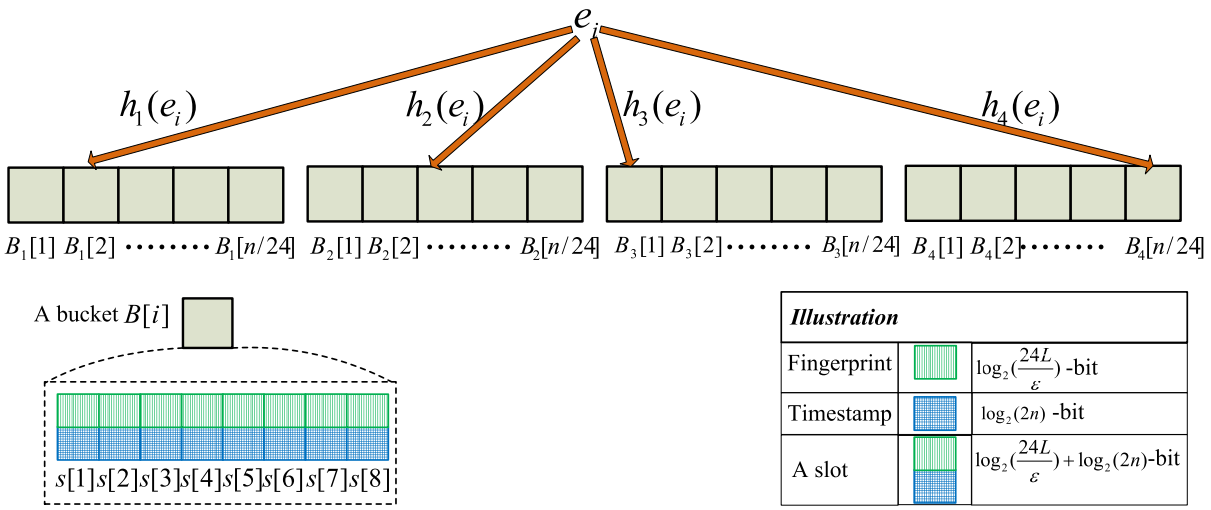

FIGURE 2. Designed data structure **A** for the ε-ARESD-problem.

monotonically increasing function with respect to c, we know $\log_2 c - \log_2(c+\frac{1}{\varepsilon}-1) \geq \log_2(\frac{1}{\varepsilon}) - \log_2(\frac{1}{\varepsilon}+\frac{1}{\varepsilon}-1) \geq -1$. We also see that $1-\varepsilon-\frac{1-\varepsilon}{c} \geq 1-2\varepsilon$ is true, because $c \geq \frac{1}{\varepsilon}$. Finally, by $\varepsilon n \ll n$, we obtain the last inequality in the above, and the conclusion. ■

DISCUSSIONS ON THE LOWER BOUNDS

In Theorem 2, we assume that $c = \frac{u}{nL} \geq 1$ and $\varepsilon \geq \frac{1}{c} = \frac{nL}{u}$. We want to show that this is not a strong restriction. The size of the universe U is very large in many applications. For example, in RFID-enabled networks [6] [14]–[17], [19], [30], a tag-ID is a 96-bit string, which leads to a universe of $2^{96} \approx 8 \times 10^{28}$ unique IDs. n (the size of a sliding window) and L (the size of a query interval) are typically far less than u. For example, suppose we want to identify if a tag that belongs to a specific interval $[a, a+L-1]$ appears in the most recently arrived n tags of this system or not. Usually, the length of the query interval $L \leq 10^7$ and $n \leq 10^7$. Thus, we have $c = u/(nL) \geq 8 \times 10^{14}$, and then the allowed false positive rate ε must be larger than 10^{-15}. However, we should point out that 10^{-14} is already a very small value of ε for many real applications.

When $c < 1/\varepsilon$, in (23), we can see that $\log_2 c - \log_2(c+\frac{1}{\varepsilon}-1)+(1+\varepsilon)\log_2\frac{1}{\varepsilon}$ is approximately equal to $\log_2 c$. This means that by introducing false positive errors, we cannot reduce the number of bits for any data structure $\mathscr{D}$ that solves the ε-**ARESD-problem**. This is another reason for using this assumption.

IV. DATA STRUCTURE FOR THE ε-ARESD-problem

In this section, we describe a data structure **A** for the ε-**ARESD-problem**. The number of bits in **A** is approximately 1.33 times of the space lower bound S_{LB} shown in (1), the time for the insertion operation is $O(1)$, the amortized cost of the updating operation is $O(1)$, [4] and the time of querying an interval of length L is $O(L)$.

A. DESIGN OF DATA STRUCTURE A

The data structure **A** uses four independent hash functions: h_1, h_2, h_3, h_4, with range $\{1, 2, .., n/24\}$, and four hash tables, denoted by B_j, $j \in \{1, 2, 3, 4\}$. For each element $e_t \in \delta$ $(t > 0)$, the hash function h_j, $j \in \{1, 2, 3, 4\}$, maps the element e_i to a bucket $B_j[h_j(e_t)]$ in the j-th hash table. **A** also contains another hash function f with range $\{0, 1, .., 24 \times 2^{1/\varepsilon} - 1\}$, which produces $f(e_t)$, called the fingerprint of the element e_t.

The explicit data structure of **A** is shown in Fig. 2. Each hash table B_j, $j \in \{1, 2, 3, 4\}$, contains $n/24$ buckets: $B_j[1], B_j[2], \cdots B_j[n/24]$, and each bucket $B_j[i]$, $i \in \{1, 2, \cdots, n/24\}$, contains eight slots: $s[1]$,$s[2]$, ..., $s[8]$. Each slot $s[i]$, $i \in \{1, 2, \ldots, 8\}$ includes two parts: $s[i]$.Fp and $s[i]$.Ts, which are used respectively for storing $f(e_t)$ (the fingerprint of e_t), and the timestamp of e_t ($\log_2(2n)$-bit long), denoted by T_{e_t}.

The reason for adopting four hash tables, each of which contains $n/24$ buckets (each bucket contains eight slots), is that this parameter setting can guarantee that for each newly arrived element $e_t \in \delta$, we can find a free slot among its hashed buckets $B_1[h_1(e_t)]$, $B_2[h_2(e_t)]$, $B_3[h_3(e_t)]$, and $B_4[h_4(e_t)]$ in **A** with a very high probability [32], [33], which shall be formally stated in Theorem 4.

Data structure **A** supports three operations: **Insertion**, **Update**, and **Query**, which are explained in detail in the following:

Insertion: When a new element e_t arrives at time point $t > 0$, we first calculate the current timestamp T_{e_t} stored in a wraparound counter $T = [(t-1) \bmod (n)]+n+1, t = 1, 2, ...$;

[4] We do need to perform the update operation, because in a data stream δ, the past element e_{i-n} expires whenever the new element e_i arrives at each time point $i > n$, as the window slides forward over δ.

second, we compute the four hashed buckets: $B_1[h_1(e_t)]$, $B_2[h_2(e_t)]$, $B_3[h_3(e_t)]$, and $B_4[h_4(e_t)]$, and the fingerprint of e_t: $f(e_t)$; third, we store the fingerprint and timestamp of e_t as follows:

(I1): If there exists a slot s with $s.\text{Fp} = f(e_t)$ in the four hashed buckets, we only store the information of e_t into this slot. Without loss of generality, we assume that the second slot $s[2]$ in $B_4[h_4(e_t)]$ contains the same fingerprint as that of e_t, and then we set $s[2].\text{Ts} = T_{e_t}$.

(I2): Otherwise, we calculate the number of free slots, of which the stored timestamp is less than or equal to $t \bmod n$, in each of the four hash tables. We place the $f(e_t)$ (fingerprint) and T_{e_t} (timestamp) of e_t into the least loaded bucket (the bucket that contains the largest number of free slots). In case of a tie, we choose the free slot s, from the leftmost hash table to store the two parts of the information of e_t.

Notice that, if a slot s has $s.\text{Ts} \leq t \bmod n$ at time point t, we call s a free slot, because the stored information (fingerprint and timestamp) in s belongs to an expired element in δ at time point t. We shall formally prove this statement in Theorem 3.

It is worth pausing at this point to consider the case in which we cannot find a free slot for e_t (when the four hashed buckets are full). In this case, we simply omit e_t completely, and we shall show that this is rarely the case. More formally, the expected number of the elements of the current sliding window that are omitted is proved to be less than 1, and thus, this case has a minor effect on the number of incorrect responses of **A**. Please see Theorem 4 and 5 in Section IV-B for formal statements and proofs.

Update: After the insertion of each newly arrived element e_t at time point t, if $[t \bmod n] \neq 0$, no update is conducted; otherwise, we check every slot s in each bucket for the four hash tables, and then update this slot s as follows:

$$\begin{cases} \text{if } s.\text{Ts} \in \{0, 1, .., n\}, & \text{then } s.\text{Ts} = 0 \\ \text{if } s.\text{Ts} \in \{n+1, n+2, , .., 2n\}, & \text{then } s.\text{Ts} = s.\text{Ts} - n. \end{cases} \tag{24}$$

Query: Given a query interval $[a, b]$ at time point $t \geq 0$ we check all the L integers contained in this interval one after another. The query process for an integer $q \in [a, b]$ is as follows. We first calculate the four hashed buckets of q: $B_1[h_1(q)]$, $B_2[h_2(q)]$, $B_3[h_3(q)]$, and $B_4[h_4(q)]$, and its fingerprint $f(q)$. Then, the output of **A** for I is as follows:

(1) If there exists a slot s among the 32 slots in the four hashed buckets with $s.\text{Fp} = f(q)$, and $s.\text{Ts} > (t \bmod n)$, q is taken to be an integer that belongs to the current sliding window W^t, and **A** outputs **1** for this case;

(2) Otherwise, **A** outputs **0**.

B. THEORETICAL ANALYSIS OF A

We shall analyze the performance of the designed structure **A** in this section.

Theorem 3: At any time point $t > 0$, after the **Insertion** and **Update** operations are finished, the stored timestamp in data structure **A** for an element belonging to the current sliding window W^t ($e_i \in W^t$) takes values in set $T^t = \{(t \bmod n) + 1, .., n-1, n, n+1, .., n + (t \bmod n)\}$.

Proof: We first consider the case that $t < n$. Then, the sliding window at the time point t is $W^t = \{e_1, e_2, .., e_t\}$. By the **Insertion** operation, we know the stored timestamp of the element $e_i \in W^t$ ($i \in \{1, 2, .., t\}$) is $((i-1) \bmod n) + n + 1 = n + i$, which belongs to set T^t.

Next, we consider the case that $t \geq n$. Without loss of generality, we assume that the current time point $t = kn + r$, where $k \geq 1$ is an integer, and $r \in \{0, 1, .., n-1\}$. Then, the sliding window at the time point t is $W^t = \{e_{(k-1)n+r+1}, e_{(k-1)n+r+2}, .., e_{kn}, e_{kn+1}, .., e_{kn+r}\}$, which can be divided into two disjoint sets W_1^t and W_2^t as follows:

$$W_1^t = \{e_{(k-1)n+r+1}, e_{(k-1)n+r+2}, .., e_{kn}\}$$
$$W_2^t = \{e_{kn+1}, e_{kn+2}, .., e_{kn+r}\}$$

By the **Insertion** operation, we know that the stored timestamp of the element $e_{(k-1)n+r+i} \in W_1^t$ is $n + r + i$ ($i \in \{1, 2, .., n-r\}$). After the insertion of e_{kn}, data structure **A** needs to execute the **Update** operation, because ($kn \bmod n$) $= 0$. Then, by the **Update** operation, we know that the stored timestamp $n + r + i$ associated with the element $e_{(k-1)n+r+i} \in W_1^t$ is reduced to $r + i$ (see the (24)). Thus, we can construct a set of stored timestamps for the elements in W_1^t, which is $T_1^t = \{t \bmod n + 1, (t \bmod n) + 2, .., n\}$. After the time point kn, by the **Insertion** operation, we obtain that the stored timestamp of the element e_{kn+j} is $n + j$ ($j \in \{1, 2, .., (t \bmod n)\}$). Thus, we can build a set of stored timestamps for the elements in W_2^t, which is $T_2^t = \{n+1, n+2, .., n+(t \bmod n)\}$. The conclusion follows from the fact that $T^t = T_1^t \cup T_2^t$. ■

By the **Insertion** operation, it is clear that we adopt the randomized allocation scheme called the Always-Go-Left Algorithm in [32]; in the following, we shall analyze the number of elements belonging to the current sliding window W^t that are not stored in **A** (omitted in the **Insertion** operation), based on [32, Th. 1].

Theorem 4: At any time point $t > 0$, the number of the elements, belonging to the current sliding window W^t, of which the information is not stored in cell array **A**, is less than 1.

Proof: Based on [32, Th. 1] (this theorem shows the maximum load of the n buckets, where at most $h \times n$ balls exist at any time point), we can obtain that, at any time point $t > 0$, with a probability $1 - 1/n^\alpha$ ($\alpha > 1$ is a constant), the maximum load of the slots in **A** is $6 + \ln\ln(n)/4$, which is strictly smaller than 8 even when we consider a window size of $2^{100} \approx 10^{30}$. Therefore, we know that at any time point $t > 0$, with a probability less than $1/n^\alpha$, a bucket B in A is full.

Next, considering a distinct element $e_i \in W^t$, we shall analyze the probability that e_i is omitted in the **Insertion** operation. First, we know the probability that e_i chooses a bucket B, which is full, is less than $1/n^\alpha$, and there are at most n distinct elements contained in W^t; therefore, the expected

number of omitted elements in W^t is less than $n \times 1/n^\alpha < 1$ ($\alpha > 1$). ■

Theorem 5: Given, at time point $j > 0$, if a query interval $I = [a, b]$ intersects with the current sliding window W^t, then **A** shall output **1** with a probability $1 - 1/n^\alpha \approx 1$ ($\alpha > 1$ is a constant); if I does not intersect with W^t, then **A** shall output **1** with a probability ε.

Proof: First, we shall analyze the case in which the given interval I does intersect with the sliding window W^t at time point t. Then, there must exist an element $e_k \in W^t$ that also belongs to I. By the **Insertion** operation of **A**, and Theorem 4, we know the following two cases:

Case1: With a probability at least $1 - 1/n^\alpha$, the fingerprint $f(e_k)$ and timestamp T_{e_k} are saved in one slot s among the four hashed buckets $B_i[h_i(e_k)]$, $i \in \{1, 2, 3, 4\}$, at the time point k;

Case2: With a probability at most $1/n^\alpha$, e_t is omitted.

For **Case1**, the analysis is as follows: Because $e_k \in W^t$, based on Theorem 3, we know that the stored timestamp of e_k takes a value in T^t, and then we have $s.\text{Ts} \geq t \bmod n$. Note that, if slot s is used by another element that comes after e_k, the stored timestamp can only be increased, because the element that comes after e_k must have a larger timestamp, as compared with e_k. By the **Query** process, we know the query result for I is **1** (see the first case (1) in **Query** operation).

From **Case2**, we see that the probability that the query result for I is **1** is greater than or equal to 0.

Combining **Case1** and **Case2**, we can see that **A** outputs **1** with a probability $1 - 1/n^\alpha \approx 1$, because the window size is usually large.

Next, we analyze the situation in which the query interval I does not intersect with W^t. We first consider the case for an integer $q \notin W^t$, and we analyze the event defined as follows:

the element $e_i \in W^t$ has $f(e_i) = f(q)$ (e_i has the same fingerprint as that of q), and e_i is placed into one of the 32 slots of the four hashed buckets $B_1[h_1(q)]$, $B_2[h_2(q)]$, $B_3[h_3(q)]$, and $B_4[h_4(q)]$.

We denote the above event by $C(e_i, q)$, which means that hash collisions happen between q and e_i. Without loss of generality, assume that n_j elements of W^t are placed into the j-th hash table of data structure **A**, $j \in \{1, 2, 3, 4\}$. Considering that there are at most n elements contained in the sliding window W^t, we know $\sum_{i=1}^{4} n_i = n$. Now, we can obtain the probability that $C(e_i, q)$ occurs as follows:

$$Pr[C(q, e_i)] = \sum_{j=1}^{4} \frac{n_j}{n} \frac{24}{n} \frac{\varepsilon}{24L}, \tag{25}$$

where $\frac{n_j}{n}$ is the probability that the information of e_i (fingerprint and timestamp of e_i) is stored into the j-th hash table, $\frac{24}{n}$ is the probability that e_i is placed into the bucket $B_j[h_j(q)]$, and $\frac{\varepsilon}{24L}$ is the probability that $f(e_i) = f(q)$. By the **Query** operation of **A**, it is not hard to see that if there exists an integer $q \in I$ such that the event $C(e_i, q)$ occurs, data structure **A** shall output **1** for I, even though the query interval I does not intersect with W^t. Therefore, we have the following:

$$\begin{aligned} &Pr[\mathbf{A} \text{ outputs } \mathbf{1} | I \cap W = \emptyset] \\ &= Pr[\bigcup\nolimits_{q \in I} \bigcup\nolimits_{j=1}^{n} C(e_j, q)] \\ &\leq \sum\nolimits_{q \in I} \sum\nolimits_{j=1}^{n} Pr[C(q, e_j)] = \varepsilon, \end{aligned} \tag{26}$$

where the last equation is based on (25) and $|I| = L$. ■

Theorem 6: Given, at time point $j > 0$, the total number of bits in data structure **A** is as follows:

$$(4/3)n \left[\log_2(1/\varepsilon) + \log_2 L + \log_2 n + \log_2(24) + 1\right], \tag{27}$$

which is approximately 4/3 times the lower bound shown in (1). Furthermore, the querying time for an interval of length L is $O(L)$, amortized updating time is $O(1)$, and insertion time is $O(1)$.

Proof: As shown in the first paragraph of Section IV-A, in a slot s in **A**, we need $\log_2(2n)$-bit for s.Ts and $\log_2(\frac{24L}{\varepsilon})$-bit for s.Fp. Furthermore, there is a total of $4 \times \frac{n}{24} \times 8$ slots in **A**. Therefore, we can see that the total number of bits contained in **A** is as shown in (27).

By the **Query** operation in Section IV-A, the querying time of **A** is $O(L)$, because we need to try L integers contained in a query interval I in the worst case. By the **Update** operation in Section IV-A, we only need to scan all of the $4 \times \frac{n}{24} \times 8$ slots in **A** at specific time points: n, $2n$, $3n$, $\cdots$, and thus the amortized updating time is $(4/3)n/n = 4/3 = O(1)$. By the **Insertion** operation in Section IV-A, we can easily obtainthat the insertion time is $O(1)$. ■

V. EXPERIMENTS

We compare the designed data structure **A** with the classic Time-decaying Bloom filter [29], [30], denoted by T-DBF, which supports point-wise membership querying over sliding windows in data streams. T-DBF can always represent the current sliding window W^t for each time point t, and then for a querying integer $q \in U$, T-DBF can answer the question of the form "$q \in W^t$?", with a predetermined false positive rate. When we apply T-DBF to the **ε-ARESD-problem**, for each query interval I, we simple check every integer e contained in I by probing T-DBF, and then we decide that $I \cap W^t$ is not empty whenever T-DBF outputs *member* for an integer $e \in I$; otherwise, we take $I \cap W^t$ to be empty.

We evaluate the two data structures: **A** and T-DBF with respect to the false positive rate, false negative rate, insertion time, and querying time. We implement both **A** and T-DBF in MATLABR2014b on a PC with an Intel(R) Core(TM) i7-7700 2.8GHz processor and 8.00 GB of RAM, running Windows 7.

Let $\mathcal{U}$ represent a universe of 10^8 integers ($\mathcal{U} = \{1, 2, .., 10^8\}$), and $\mathcal{L}$ a set of all the possible intervals of length L over $\mathcal{U}$. We generate two synthetic datasets: *UNIFORM* and *NON-UNIFORM*, for testing the performance of **A** and T-DBF. *UNIFORM* includes a data stream of 10^7 elements that are drawn from a uniform distribution over U, and a sequence of 10^7 query intervals of length L are chosen

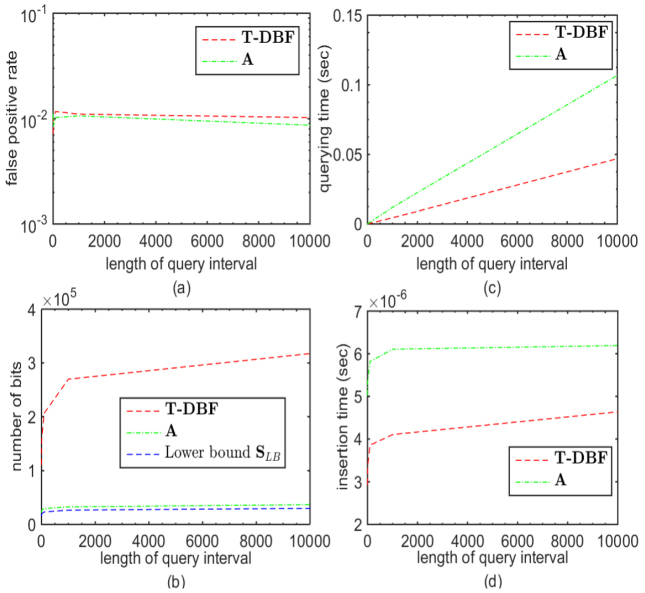

FIGURE 3. Performance of A and T-DBF on *UNIFORM* for various interval lengths.

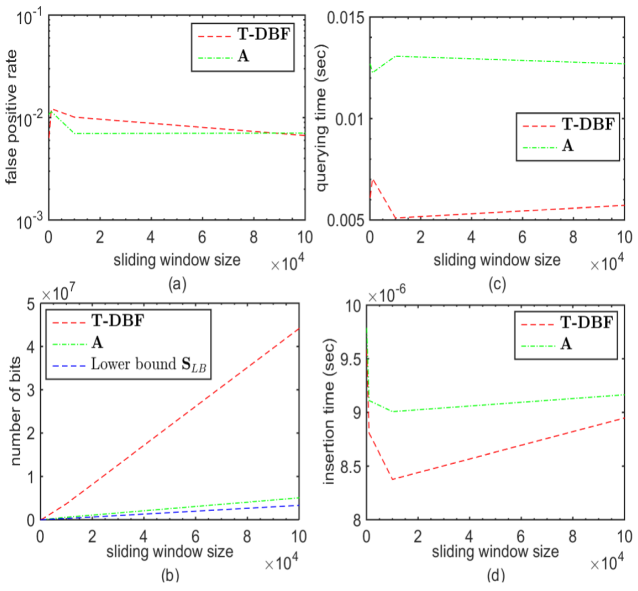

FIGURE 4. Performance of A and T-DBF on *UNIFORM* for various sliding window sizes.

TABLE 2. Summary of the performances of A and the classic time-decaying bloom filter.

	A	Time-Decaying Bloom Filter (T-DBF)
Memory usage	1.33 times the lower bound S_{LB} shown in (1)	At least $O(\log_2(n))$ times the lower bound S_{LB} shown in(1)
Querying time	$O(L)$	$O(L\log_2(L/\varepsilon))$
Insertion time	$O(1)$	$O(\log_2(L/\varepsilon))$
Updating time	$O(1)$	$O(1)$

Note that L is length of a query interval, n is the size of a sliding window, and ε is the false positive rate.

randomly and uniformly from $\mathcal{L}$. *NON-UNIFORM* includes a data stream of 10^7 elements that are drawn from a Zipfian distribution with $Z = 1$ (Z is the value of the exponent that controls the skewness of the distribution) over U, and a sequence of 10^7 query intervals of length L are chosen randomly and uniformly from $\mathcal{L}$.

We set the false positive rate $\varepsilon = 10^{-2}$ and sliding window size $n = 10^3$, while varying the query interval length l from 10^0 to 10^4, in order to test the performance of **A** and T-DBF. The performances of **A** and T-DBF over *UNIFORM* and *NON-UNIFORM* are reported in Fig. 3 and Fig. 5, respectively.

We can observe from Fig. 3 and Fig. 5 that **A** and T-DBF have almost the same false positive rate. Both **A** and T-DBF have similar querying times and insertion times. To be more specific, the querying time of **A** is at most 1.7 times that of T-DBF; the insertion of **A** is at most three times that of T-DBF. However, Fig. 3 and Fig. 5 clearly show that the number of memory bits is used by **A** is close to the space lower bound S_{LB}, and is significantly less that that of T-DBF. For example, in Fig. 3, when $l = 10^4$, the number of bits used by **A** is only 11.6% of that used by T-DBF.

To achieve the same false positive rate, **A** consumes much less memory space as compared with T-DBF, which

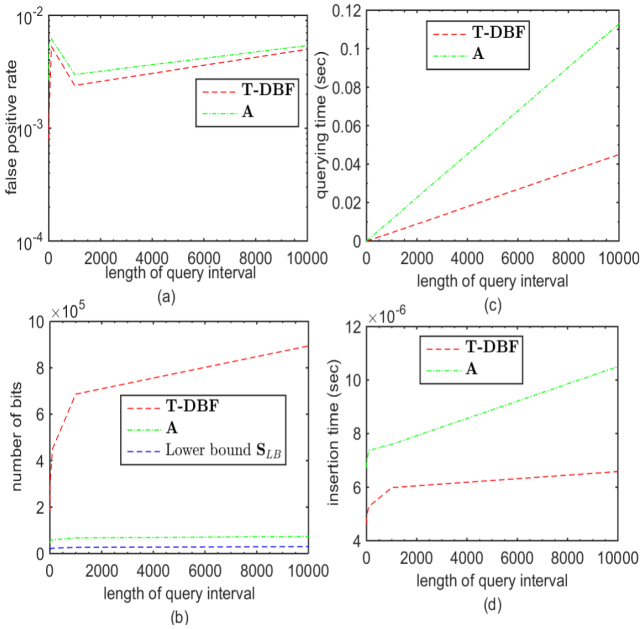

FIGURE 5. Performance of A and T-DBF on *NON-UNIFORM* for various interval lengths.

demonstrates what we have proved in Theorem 6. The insertion time of **A** is slightly slower than that of T-DBF, because **A** needs to find the least loaded free slot among the four hashed buckets, whereas T-DBF simply finds k hashed positions in a bit-array.

We set the false positive rate $\varepsilon = 10^{-2}$ and sliding window size $l = 10^3$, while varying the sliding window size from 10^2 to 10^5, in order to test the performance of **A** and T-DBF. The performances of **A** and T-DBF over *UNIFORM* and *NON-UNIFORM* are reported in Fig. 4 and Fig. 6, respectively.

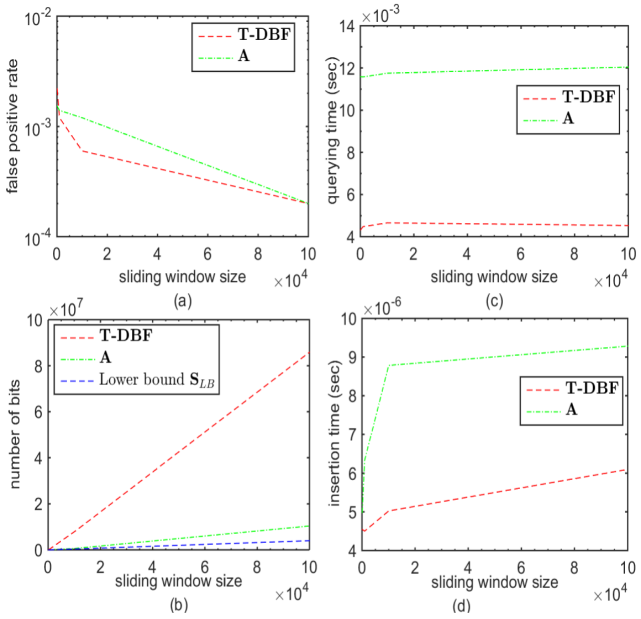

FIGURE 6. Performance of **A** and T-DBF on *NON-UNIFORM* for various sliding window sizes.

From these two figures, we also see clearly that the false positive rates of **A** and T-DBF are almost the same, the querying time of **A** is at most three times that of T-DBF, and the insertion time of **A** is 1.7 times that of T-DBF. However, the **A** uses much less memory as compared with T-DBF. For example, in Fig. 6, when $n = 10^4$, the number of bits used by**A** is only 7.6% of T-DBF.

Note that in all four experiments, neither **A** nor T-DBF produce any false negative errors.

VI. CONCLUSIONS

In this work, we have defined and studied the ε-approximate range emptiness problem over sliding windows in data streams in IC-IoT environments. The main objective is to design a space-efficient data structure that can always represent the sliding window W^t at every time point $t > 0$, and then approximately and quickly answers the range emptiness query of the form "$W \cap I = \emptyset$?", for any interval I of length L. To achieve this, we first proposed a space lower bound for any data structure that solves the ε-approximate range emptiness problem over sliding windows in data streams. Then, a data structure denoted by **A** was proposed and proved to be within a factor of 1.33 of the lower bound. Extensive simulation results demonstrated the efficiency of our data structures.

APPENDIX
DETAILS OF THE MAXIMIZATION PROBLEM SHOWN IN (11) AND (12)

It is easy to see that $\log_2(c_i)$ is a concave function of the variable c_i for $i \in \{1, 2, .., n\}$, and then the linear combination of these functions is also concave. Then, we can find the maximizer for z shown in (11) by setting up a function as follows:

$$F(c_1, .., c_n, \lambda) = \sum_{i=1}^{l} \alpha_i \log_2(c_i) + \sum_{i=l+1}^{n} (\alpha_i + 1)\log_2(c_i) + \lambda(\sum_{i=1}^{n} c_i - n + (u/L - n)\varepsilon),$$

where λ is a Lagrange multiplier.

Taking the derivative with respect to c_i and setting it to zero, $\partial F(c_1, .., c_n, \lambda)/\partial c_i = 0$, we have $c_i = \frac{\alpha_i}{-\lambda \ln 2}$ for $i \in \{1, 2, .., l\}$, and $c_i = \frac{\alpha_i + 1}{-\lambda \ln 2}$ for $i \in \{l + 1, l + 2, .., n\}$. Then, using the equality constraints shown in (12) and (10), we obtain the following:

$$\sum_{i=1}^{l} c_i + \sum_{i=l+1}^{n} c_i = n + (u/L - n)\varepsilon$$
$$\sum_{i=1}^{l} \frac{\alpha_i}{-\lambda \ln 2} + \sum_{i=l+1}^{n} \frac{\alpha_i + 1}{-\lambda \ln 2} = n + (u/L - n)\varepsilon$$
$$\frac{1}{-\lambda \ln 2}(\sum_{i=1}^{n} (\alpha_i) + n - l) = n + (u/L - n)\varepsilon$$
$$\frac{1}{-\lambda \ln 2} = 1 + (\frac{u}{nL} - 1)\varepsilon.$$

Thus, we obtain that when $c_i = \alpha_i(1 + (\frac{u}{nL} - 1)\varepsilon)$, for $i \in \{1, 2, .., l\}$, and $c_i = (\alpha_i + 1)(1 + (\frac{u}{nL} - 1)\varepsilon)$, for $i \in \{1, 2, .., l\}$, the global maximal value of z in Eq. (11) is as shown in (13).

REFERENCES

[1] A. Pagh, R. Pagh, and S. S. Rao, "An optimal bloom filter replacement," in *Proc. 16th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA*, vol. 2, 2005, pp. 823–829.
[2] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *Proc. Eur. Symp. Algorithms*, 2006, pp. 456–467.
[3] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.
[4] M. Goswami, A. Gronlund, K. G. Larsen, and R. Pagh, "Approximate range emptiness in constant time and optimal space," in *Proc. Symp. Discrete Algorithms*, 2015, pp. 769–775.
[5] S. Dutta, A. Narang, and S. K. Bera, "Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams," *Very Large Data Bases*, vol. 6, no. 8, pp. 589–600, 2013.
[6] J. Yu, L. Chen, R. Zhang, K. Wang, "Finding needles in a haystack: Missing tag detection in large rfid systems," *IEEE Trans. Commun.*, vol. 65, no. 5, pp. 2036–2047, May 2017.
[7] R. K. Lomotey, J. Pry, and S. Sriramoju, "Wearable IoT data stream traceability in a distributed health information system," *Pervasive Mobile Comput.*, vol. 40, pp. 692–707, Sep. 2017.
[8] S. Arshad, M. A. Azam, M. H. Rehmani, and J. Loo, "Recent advances in information-centric networking based internet of things (ICN-IoT)," *IEEE Internet Things J.*, to be published.
[9] M. Amadeo *et al.*, "Information-centric networking for the Internet of Things: Challenges and opportunities," *IEEE Netw.*, vol. 30, no. 2, pp. 92–100, Mar./Apr. 2016.
[10] Z. Zhou, H. Yu, C. Xu, Z. Chang, S. Mumtaz, and J. Rodriguez, "BEGIN: Big data enabled energy-efficient vehicular edge computing," *IEEE Commun. Mag.*, vol. 56, no. 12, pp. 82–89, Dec. 2018.
[11] Z. Zhou, H. Yu, C. Xu, Y. Zhang, S. Mumtaz, and J. Rodriguez, "Dependable content distribution in D2D-based cooperative vehicular networks: A big data-integrated coalition game approach," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 3, pp. 953–964, Mar. 2018.
[12] P. Le Nguyen, Y. Ji, Z. Liu, H. Vu, and K.-V. Nguyen, "Distributed hole-bypassing protocol in wsns with constant stretch and load balancing," *Comput. Netw.*, vol. 129, pp. 232–250, Dec. 2017.
[13] Z. Liu, T. Tsuda, H. Watanabe, S. Ryuo, and N. Iwasawa, "Data driven cyber-physical system for landslide detection," *Mobile Netw. Appl.*, pp. 1–12, Mar. 2018. [Online]. Available: https://link.springer.com/article/10.1007/s11036-018-1031-1

[14] A. Singh, S. Garg, S. Batra, N. Kumar, and J. J. P. C. Rodrigues, "Bloom filter based optimization scheme for massive data handling in IoT environment," *Future Gener. Comput. Syst.*, vol. 82, pp. 440–449, May 2018.
[15] B. Ahlgren, A. Lindgren, and Y. Wu, "Experimental feasibility study of CCN-lite on Contiki motes for IoT data streams," in *Proc. ACM Conf. Inf.-Centric Netw.*, 2016, pp. 221–222.
[16] R. Li, H. Asaeda, and J. Li, "A distributed publisher-driven secure data sharing scheme for information-centric IoT," *IEEE Internet Things J.*, vol. 4, no. 3, pp. 791–803, Jun. 2017.
[17] R. Li, H. Asaeda, J. Li, and X. Fu, "A verifiable and flexible data sharing mechanism for information-centric IoT," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–7.
[18] R. Li, H. Harai, and H. Asaeda, "An aggregatable name-based routing for energy-efficient data sharing in big data era," *IEEE Access*, vol. 3, pp. 955–966, 2015.
[19] H. Xu, W. Shen, P. Li, D. Sgandurra, and R. Wang, "VSMURF: A novel sliding window cleaning algorithm for RFID networks," *J. Sensors*, vol. 2017, pp. 1–11, Jun. 2017. [Online]. Available: https://www.hindawi.com/journals/js/2017/3186482/
[20] Y. Qin, Q. Z. Sheng, N. J. G. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When things matter: A survey on data-centric Internet of Things," *J. Netw. Comput. Appl.*, vol. 64, pp. 137–153, Apr. 2016.
[21] O. Ascigil, S. Rene, G. Xylomenos, I. Psaras, and G. Pavlou, "A keyword-based ICN-IoT platform," in *Proc. ACM Conf. Inf.-Centric Netw.*, 2017, pp. 22–28.
[22] Z. Zhou *et al.*, "When mobile crowd sensing meets UAV: Energy-efficient task assignment and route planning," *IEEE Trans. Commun.*, vol. 66, no. 11, pp. 5526–5538, Nov. 2018.
[23] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, vol. 31, no. 6, pp. 1794–1813, 2002.
[24] Z. Pervaiz, A. Ghafoor, and W. G. Aref, "Precision-bounded access control using sliding-window query views for privacy-preserving data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1992–2004, Jul. 2015.
[25] R. Pagh, G. Segev, and U. Wieder, "How to approximate a set without knowing its size in advance," in *Proc. Annu. IEEE Symp. Found. Comput. Sci.*, Oct. 2013, vol. 19, no. 17, pp. 80–89.
[26] M. Naor and E. Yogev, "Tight bounds for sliding bloom filters," *Algorithmica*, vol. 73, no. 4, pp. 652–672, 2015.
[27] D. Belazzougui and R. Venturini, "Compressed static functions with applications," in *Proc. 24th ACM-SIAM Symp. Discrete Algorithms*, 2013, pp. 229–240.
[28] A. Metwally, D. Agrawal, and A. El Abbadi, "Duplicate detection in click streams," in *Proc. Int. Conf. World Wide Web*, 2005, pp. 12–21.
[29] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2008, pp. 77–84.
[30] G. Liao *et al.*, "Approximately filtering redundant data for uncertain RFID data streams," in *Proc. IEEE Int. Conf. Mobile Data Manage.*, Jun. 2017, pp. 56–61.
[31] M. Mitzenmacher and E. Upfal, *Probability and Computing*. Cambridge, U.K.: Cambridge Univ. Press, 2005.
[32] B. Vocking, "How asymmetry helps load balancing," in *Proc. Symp. Found. Comput. Sci.*, 1999, pp. 131–141.
[33] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. Conf. Eur. Symp.*, 2006, pp. 684–695.

XIUJUN WANG received the B.E. degree in computer science and technology from Anhui Normal University and the Ph.D. degree in computer software and theory from the University of Science and Technology of China. He is currently a Lecturer with the School of Computer Science and Technology, Anhui University of Technology. His research interests include data stream processing, randomized algorithm, and the Internet of Things.

ZHI LIU received the B.E. degree from the University of Science and Technology of China, China, and the Ph.D. degree in informatics from the National Institute of Informatics, Japan. He was a Junior Researcher (Assistant Professor) with Waseda University and a JSPS Research Fellow with the National Institute of Informatics. He is currently an Assistant Professor with Shizuoka University. His research interests include video network transmission, vehicular networks, and mobile edge computing. He is a member of the IEEE and the IEICE. He was a recipient of the IEEE StreamComm2011 Best Student Paper Award, the 2015 IEICE Young Researcher Award, and the ICOIN2018 Best Paper Award. He has been serving as the Chair for the number of international conferences and workshops. He is and has been a Guest Editor of journals, including the *Wireless Communications and Mobile Computing*, *Sensors*, and the *IEICE Transactions on Information and Systems*.

YAN GAO received the B.S. degree in physics from Anhui Normal University and the M.E. degree in physics from the University of Science and Technology of China. She is currently an Engineer with the School of Mathematics and Physics, Anhui University of Technology. Her research interests include randomized algorithms and stochastic differential equation theory.

XIAO ZHENG received the B.E. degree from Anhui University, China, and the Ph.D. degree in computer science and technology from Southeast University, China. He is currently a Professor with the School of Computer Science and Technology, Anhui University of Technology. His research interests include computer networking, service computing, mobile cloud computing, and machine learning.

XIANFU CHEN received the Ph.D. degree in signal and information processing from the Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China, in 2012. Since 2012, he has been with the VTT Technical Research Centre of Finland Ltd., Oulu, Finland, where he is currently a Senior Scientist. His research interest includes various aspects of wireless communications and networking, with an emphasis on human level and artificial intelligence for resource awareness in the next-generation communication networks. He is an IEEE Member.

CELIMUGE WU received the M.E. degree from the Beijing Institute of Technology, China, in 2006, and the Ph.D. degree from The University of Electro-Communications, Japan, in 2010, where he is currently an Associate Professor with the Graduate School of Informatics and Engineering. His current research interests include vehicular ad hoc networks, sensor networks, intelligent transport systems, the IoT, 5G, and mobile cloud computing. He is a Senior Member of the IEEE. He is/has been a TPC Co-Chair of Wireless Days 2019 and ICT-DM 2018 and a Track Co-Chair for many international conferences, including the ICCCN 2019 and the IEEE PIMRC 2016.