Streaming Algorithms for Estimating High Set Similarities in LogLog Space

Yiyan Qi, Pinghui Wang, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, Guangjian Tian, John C.S. Lui, and Xiaohong Guan

Abstract—Estimating set similarity and detecting highly similar sets are fundamental problems in areas such as databases and machine learning. MinHash is a well-known technique for approximating Jaccard similarity of sets and has been successfully used for many applications. Its two compressed versions, b-bit MinHash and Odd Sketch, can significantly reduce the memory usage of the MinHash, especially for estimating high similarities (i.e., similarities around 1). Although MinHash can be applied to static sets as well as streaming sets, of which elements are given in a streaming fashion, unfortunately, b-bit MinHash and Odd Sketch fail to deal with streaming data. To solve this problem, we previously designed a memory-efficient sketch method, *MaxLogHash*, to accurately estimate Jaccard similarities in streaming sets. Compared with MinHash, our method uses smaller sized registers (each register consists of less than 7 bits) to build a compact sketch for each set. In this paper, we further develop a faster method, *MaxLogOPH++*. Compared with MaxLogHash, MaxLogOPH++ reduces the time complexity for updating each coming element from $O(k)$ to $O(1)$ with a small additional memory. We conduct experiments on a variety of datasets, and experimental results demonstrate the efficiency and effectiveness of our methods.

Index Terms—Streaming algorithms, sketch, jaccard similarity

1 Introduction

Data streams are ubiquitous in nature. Examples range from financial transactions to Internet of things (IoT) data, network traffic, call logs, trajectory logs, etc. Due to the nature of these applications which involve massive volume of data, it is prohibitive to collect the entire data streams, especially when computational and storage resources are limited [1]. Therefore, it is important to develop memory-efficient methods such as sampling and sketching techniques for mining large streaming data.

- *Y. Qi, Y. Zhang, and Q. Zhai are with MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, P.O. Box 1088, No. 28, Xianning West Road, Xi'an, Shaanxi 710049, China. E-mail: {qiyiyan, zhangyuanming}@stu.xjtu.edu.cn, qzzhai@mail.xjtu.edu.cn.*
- *P. Wang is with the MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, P.O. Box 1088, No. 28, Xianning West Road, Xi'an, Shaanxi 710049, China, and also with Shenzhen Research Institute, Xi'an Jiaotong University, Shenzhen, China. E-mail: phwang@mail.xjtu.edu.cn.*
- *X. Guan is with the MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, P.O. Box 1088, No. 28, Xianning West Road, Xi'an, Shaanxi 710049, China, the Shenzhen Research Institute, Xi'an Jiaotong University, Shenzhen, China, and also with the Center for Intelligent and Networked Systems, Tsinghua National Lab for Information Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: xhguan@mail.xjtu.edu.cn.*
- *C. Wang is with the School of Software Engineering, Xi'an Jiaotong University, Xi'an, Shaanxi 710049, China. E-mail: cxwang@mail.xjtu.edu.cn.*
- *G. Tian is with Huawei Noah's Ark Lab, Hong Kong. E-mail: Tian.Guangjian@huawei.com.*
- *J.C.S. Lui is with the Chinese University of Hong Kong, Hong Kong. E-mail: cslui@cse.cuhk.edu.hk.*

Manuscript received 12 Aug. 2019; revised 11 Dec. 2019; accepted 11 Jan. 2020. Date of publication 24 Jan. 2020; date of current version 10 Sept. 2021. (Corresponding author: Pinghui Wang.) Recommended for acceptance by C. Li. Digital Object Identifier no. 10.1109/TKDE.2020.2969423

Many datasets can be viewed as collections of sets and computing set similarities is fundamental for a variety of applications in areas such as databases [2], [3], machine learning [4], and information retrieval [5], [6]. For example, one can view each mobile device's trajectory as a set and each element in the set corresponds to a tuple of time t and the physical location of the device at time t. Then, mining devices with similar trajectories is useful for identifying friends or devices belonging to the same person. Other examples are datasets encountered in computer networks, mobile phone networks, and online social networks (OSNs), where learning user similarities in the sets of users' visited websites on the Internet, connected phone numbers, and friends on OSNs is fundamental for applications such as link prediction and friendship recommendation.

One of the most popular set similarity measures is the Jaccard similarity coefficient, which is defined as $\frac{|A\cap B|}{|A\cup B|}$ for two sets A and B. To handle large sets, MinHash (or, minwise hashing) [7] is a powerful set similarity estimation technique, which uses an array of k registers to build a sketch for each set. Its accuracy only depends on the value of k and the Jaccard similarity of two sets of interest, and it is independent of the size of two sets. MinHash has been successfully used for a variety of applications, such as similarity search [5], compressing social networks [2], advertising diversification [8], large scale learning [4], and webspam detection [9]. Many of these applications focus on estimating similarity values close to 1. Take similar document search in a sufficiently large corpus as an example. For a corpus, there may be thousands of documents which are similar to the query document, therefore our goal is not just to find similar documents, but also to provide a short list (e.g., top-10) and ranking of the most similar documents. For such an application, we need effective methods that are very accurate and memory-efficient for estimating high similarities. To achieve this goal, there are two compressed MinHash methods, b-bit MinHash [10] and Odd Sketch [11], which were proposed in the past few years to further reduce the memory usage of the original MinHash by dozens of times, while to provide comparable estimation accuracy especially for large similarity values. However, we observe that these two methods fail to handle data streams (the details will be given in Section 3).

To solve the above challenge, recently, Yu and Weber [12] develop a method, HyperMinHash. HyperMinHash consists of k registers, whereas each register has two parts, an FM (Flajolet-Martin) sketch [13] and a b-bit string. The b-bit string is computed based on the fingerprints (i.e., hash values) of set elements that are mapped to the register. Based on HyperMinHash sketches of two sets A and B, HyperMinhash first estimates $|A \cup B|$ and then infers the Jaccard similarity of A and B from the number of collisions of b-bit strings given $|A \cup B|$. Later in our experiments, we demonstrate that HyperMinHash not only exhibits a large bias and is also computationally expensive for estimating similarities, which results in a large estimation error and a big delay in querying highly similar sets. More importantly, it is difficult to analytically analyze the estimation bias and variance of HyperMinHash [12], which are of great value in practice– the bias and variance can be used to bound estimation errors and determine the smallest necessary sampling budget (i.e., k) for a desired accuracy.

We previously developed a memory-efficient method, *MaxLogHash*, to estimate Jaccard similarities in streaming sets [14]. Similar to MinHash, MaxLogHash uses a list of k registers to build a compact sketch for each set. Unlike MinHash which uses a 64-bit (resp. 32-bit) register for storing the minimum hash value of 64-bit (resp. 32-bit) set elements, our method MaxLogHash uses only 7-bit register (resp. 6-bit register) to approximately record the logarithm value of the minimum hash value, and this results in 9 times (resp. 5 times) reduction in memory usage. Another attractive property is that our MaxLogHash sketch can be computed incrementally, therefore, MaxLogHash is able to handle streaming-sets. Given any two sets' MaxLogHash sketches, we provide a simple yet accurate estimator for their Jaccard similarities and derive exact formulas for bounding the estimation error. We also developed a faster method, *MaxLogOPH* [14]. Compared to MaxLogHash, MaxLogOPH reduces the time complexity for updating each coming element from $O(k)$ to $O(1)$, but gives a large estimation bias for small sets.

To solve this problem, in this paper, we design a novel estimator, *MaxLogOPH++*, which significantly reduces the estimation error of MaxLogOPH. Specifically, we first build a likelihood function of generated sketches and then infer the Maximum Likelihood Estimation (MLE) [15] of Jaccard similarity using the Newton-Raphson method [16]. Compared with our method MaxLogHash [14], MaxLogOPH++ reduces the time complexity from $O(k)$ to $O(1)$ and gives comparable accuracy. MaxLogOPH++ also significantly reduces the estimation bias of MaxLogOPH [14]. We conduct experiments on a variety of synthetic and real-world datasets. Our experimental results show that MaxLogOPH++ is two orders of magnitude faster than MaxLogHash at the cost of 4 percent more memory usage, and is 2 times more accurate than MaxLogOPH for small sets.

The rest of this paper is organized as follows. The problem formulation is presented in Section 2. Section 3 introduces preliminaries used in this paper. Sections 4 and 5 present our method MaxLogHash and MaxLogOPH++ respectively. The performance evaluation and testing results are presented in Section 6. Section 7 summarizes related work. Concluding remarks then follow.

2 Problem Formulation

For ease of reading and comprehension, we say that each set belongs to a user, elements in the set are items (e.g., products) that the user connects to. Let U denote the set of users and I denote the set of all items. Let $\Pi = e^{(1)}e^{(2)}\cdots e^{(t)}\cdots$ denote the user-item stream of interest, where $e^{(t)} = (u^{(t)}, v^{(t)})$ is the element of Π occurred at discrete time $t > 0$, $u^{(t)} \in U$ and $v^{(t)} \in I$ are the element's user and item, which represents a connection from user $u^{(t)}$ to item $v^{(t)}$. We assume that Π has no duplicate user-item pairs,[1] that is, $e^{(i)} \neq e^{(j)}$ when $i \neq j$. Let $I_u^{(t)} \subset I$ be the item set of user $u \in U$, which consists of items that user u connects to before and including time t. Let $\cup^{(t)}(u_1, u_2)$ denote the union of two sets $I_{u_1}^{(t)}$ and $I_{u_2}^{(t)}$, that is, $\cup^{(t)}(u_1, u_2) = I_{u_1}^{(t)} \cup I_{u_2}^{(t)}$. Similarly, we define the intersection of two sets $I_{u_1}^{(t)}$ and $I_{u_1}^{(t)}$ as $\cap^{(t)}(u_1, u_2) = I_{u_1}^{(t)} \cap I_{u_2}^{(t)}$. Then, the Jaccard similarity of sets $I_{u_1}^{(t)}$ and $I_{u_2}^{(t)}$ is defined as

$$J_{u_1,u_2}^{(t)} = \frac{|\cap^{(t)}(u_1, u_2)|}{|\cup^{(t)}(u_1, u_2)|},$$

which reflects the similarity between users u_1 and u_2. We formulate our problem as:

Problem Definition. Given an insertion-only user-item stream $\Pi = e^{(1)}e^{(2)}\cdots e^{(t)}\cdots$ and current time $t > 0$, we aim to develop a memory-efficient method to estimate $J_{u_1,u_2}^{(t)}$ for any two users u_1 and u_2 based on their corresponding item sets appearing before and including time t.

3 Preliminaries

In this section, we first introduce MinHash [7]. Then, we elaborate two state-of-the-art memory-efficient methods *b-bit MinHash* [10] and *Odd Sketch* [11] that can decrease the memory usage of the original MinHash method. At last, we demonstrate that both b-bit MinHash and Odd Sketch fail to handle streaming sets.

3.1 MinHash

Given a random permutation (or hash function)[2] π from elements in I to elements in I, i.e., a hash function maps integers in I to *distinct* integers in I at random. Broder *et al.* [7] observed that the Jaccard similarity of two sets $A, B \subseteq I$ equals

$$J_{A,B} = \frac{|\cap(A, B)|}{|\cup(A, B)|} = P(\min(\pi(A)) = \min(\pi(B))), \tag{1}$$

1. Duplicated user-item pairs can be easily checked and filtered using fast and memory-efficient techniques such as Bloom filter [17].

2. MinHash assumes no hash collisions.

where $\pi(A) = \{\pi(w) : w \in A\}$. Therefore, MinHash uses a sequence of k independent permutations $\pi_1, \ldots, \pi_k$ and estimates $J_{A,B}$ as

$$\hat{J}_{A,B} = \frac{\sum_{i=1}^{k} \mathbf{1}(\min(\pi_1(A)) = \min(\pi_1(B))}{k}, \tag{2}$$

where $\mathbf{1}(\mathbb{P})$ is an indicator function that equals 1 when the predicate $\mathbb{P}$ is true and 0 otherwise. Note that $\hat{J}_{A,B}$ is an unbiased estimator for $J_{A,B}$, i.e., $\mathbb{E}(\hat{J}_{A,B}) = J_{A,B}$, and its variance is

$$\mathrm{Var}(\hat{J}_{A,B}) = \frac{J_{A,B}(1 - J_{A,B})}{k}. \tag{3}$$

Therefore, instead of storing a set A in memory, one can compute and store its MinHash sketch S_A, i.e.,

$$S_A = (\min(\pi_1(A)), \ldots, \min(\pi_k(A))),$$

which reduces the memory usage when $|A| > k$. The Jaccard similarity of any two sets can be accurately and efficiently estimated based on their MinHash sketches.

3.2 b-bit MinHash

Li and König [10] proposed a method, b-bit MinHash, to further reduce the memory usage. b-bit MinHash reduces the memory required for storing a MinHash sketch S_A from $32k$ or $64k$ bits[3] to bk bits. The basic idea behind b-bit MinHash is that the same hash values give the same lowest b bits while two different hash values give the same lowest b bits with a small probability $1/2^b$. Formally, let $\min^{(b)}(\pi(A))$ denote the lowest b bits of the value of $\min(\pi(A))$ for a permutation π. Define the b-bit MinHash sketch of set A as

$$S_A^{(b)} = (\min^{(b)}(\pi_1(A)), \ldots, \min^{(b)}(\pi_k(A))).$$

To mine set similarities, Li and König first compute S_A for each set A, and then store its b-bit MinHash sketch $S_A^{(b)}$. At last, the Jaccard similarity $J_{A,B}$ is estimated as

$$\hat{J}_{A,B}^{(b)} = \frac{\sum_{i=1}^{k} \mathbf{1}(\min^{(b)}(\pi_i(A)) = \min^{(b)}(\pi_i(B))) - \frac{k}{2^b}}{k(1 - \frac{1}{2^b})}. \tag{4}$$

$\hat{J}_{A,B}^{(b)}$ is also an unbiased estimator for $J_{A,B}$, and its variance is

$$\mathrm{Var}(\hat{J}_{A,B}^{(b)}) = \frac{1 - J_{A,B}}{k}\left(J_{A,B} + \frac{1}{2^b - 1}\right). \tag{5}$$

3.3 Odd Sketch

Mitzenmacher *et al.* [11] developed a method Odd Sketch, which is more memory efficient than b-bit MinHash for mining highly similar sets. Odd Sketch uses a hash function h that maps each tuple $(i, \min(\pi_i(A)))$, $i = 1, \ldots, k$, to an integer in $\{1, \ldots, z\}$ at random. For a set A, its odd sketch $S_A^{(\mathrm{Odd})}$ consists of z bits. Function h maps tuples $(1, \min(\pi_1(A))), \ldots, (k, \min(\pi_k(A)))$ into z bits of $S_A^{(\mathrm{Odd})}$ at random. $S_A^{(\mathrm{Odd})}[j]$, $1 \le j \le z$, is the parity of the number of tuples that are mapped to the jth bit of $S_A^{(\mathrm{Odd})}$. Formally, $S_A^{(\mathrm{Odd})}[j]$ is computed as

$$S_A^{(\mathrm{Odd})}[j] = \oplus_{i=1,\ldots,k} \mathbf{1}(h(i, \min(\pi_i(A))) = j), \quad 1 \le j \le z.$$

The Jaccard similarity $J_{A,B}$ is then estimated as

$$\hat{J}_{A,B}^{(\mathrm{Odd})} = 1 + \frac{z}{4k} \ln\left(1 - \frac{2\sum_{i=1}^{z} S_A^{(\mathrm{Odd})}[j] \oplus S_B^{(\mathrm{Odd})}[j]}{z}\right). \tag{6}$$

Mitzenmacher *et al.* demonstrate that $\hat{J}_{A,B}^{(\mathrm{Odd})}$ is more accurate than $\hat{J}_{A,B}^{(\mathrm{b})}$ under the same memory usage (refer to [11] for details of the error analysis of $\hat{J}_{A,B}^{(\mathrm{Odd})}$).

3.4 Discussion

MinHash can be Directly Applied to Stream Data. We can easily find that MinHash sketch can be computed incrementally. That is, one can compute the MinHash sketch of set $A \cup \{v\}$ from the MinHash sketch of set A as

$$\min(\pi(A \cup \{v\})) = \min(\min(\pi(A)), \pi(v)).$$

Variants b-bit MinHash and Odd Sketch cannot be Used to Handle Streaming Sets. Let $\pi^{(b)}(v)$ denote the lowest b bits of $\pi(v)$. Then, one can easily show that

$$\min^{(b)}(\pi(A \cup \{v\})) \ne \min(\min^{(b)}(\pi(A)), \pi^{(b)}(v)).$$

It shows that computing $\min^{(b)}(\pi(A \cup \{v\}))$ requires the hash value $\pi(w)$ of each $w \in A \cup \{v\}$. In addition, we observe that $\min^{(b)}(\pi(A))$ cannot be approximated as $\min_{w\in A}\pi^{(b)}(w)$, which can be computed incrementally, because $\min_{w\in A}\pi^{(b)}(w)$ equals 0 with a high probability when $|A| \gg 2^b$. Later in the experiment section (Section 6.4), we show that the above approximation gives large estimate errors. Similarly, we cannot compute the odd sketch of a set incrementally. Therefore, both b-bit MinHash and Odd Sketch fail to deal with streaming sets.

4 MaxLogHash

4.1 Basic Idea

Let h be a function that maps any element v in I to a random number in range (0,1). i.e., $h(v) \sim Uniform(0, 1)$. Define the log-rank of v with respect to hash function h as $r(v) \leftarrow \lfloor -\log_2 h(v) \rfloor$. We compute and store

$$\mathrm{MaxLog}(h(A)) = \max_{v\in A} r(v) = \max_{v\in A} \lfloor -\log_2 h(v) \rfloor. \tag{7}$$

Let us now develop a simple yet accurate method to estimate Jaccard similarity of streaming sets based on the following properties of function $\mathrm{MaxLog}(h(A))$.

Observation 1. $\mathrm{MaxLog}(h(A))$ can be represented by an integer of no more than $\lceil \log_2 \log_2 |I| \rceil$ bits with a high probability. For each $v \in I$, we have $h(v) \sim Uniform(0, 1)$, and thus

$$r(v) \sim Geometric(1/2),$$

supported on set $\{0, 1, 2, \ldots\}$, that is,

$$P(r(v) < j) = 1 - \frac{1}{2^j}, \quad j \in \{0, 1, 2, \ldots\}.$$

3. A 32- or 64-bit register is used to store each $\min(\pi_i(A))$, $i = 1, \ldots, k$.

Then, one can easily find that

$$P(\mathrm{MaxLog}(h(A)) \le 2^{\lceil \log_2 \log_2 |I| \rceil} - 1) = \left(1 - \frac{1}{2^{2^{\lceil \log_2 \log_2 |I| \rceil}}}\right)^{|A|}.$$

For example, when $A \subseteq \{1, \dots, 2^{64}\}$ and $|A| \le 2^{54}$, we only require 6 bits to store $\mathrm{MaxLog}(h(A))$ with probability at least 0.999.

Observation 2. $\mathrm{MaxLog}(h(A))$ can be computed incrementally. This is because

$$\mathrm{MaxLog}(h(A \cup \{v\})) = \max(\mathrm{MaxLog}(h(A)), \lfloor -\log_2 h(v) \rfloor).$$

Observation 3. $J_{A,B}$ can be easily estimated from $\mathrm{MaxLog}(h(A))$ and $\mathrm{MaxLog}(h(B))$ with a little additional information. We find that

$$\begin{aligned}\gamma =& P(\mathrm{MaxLog}(h(A)) \neq \mathrm{MaxLog}(h(B))) \\ =& \sum_{j=1}^{+\infty} \frac{|A \setminus B|}{2^{j+1}} \left(1 - \frac{1}{2^{j+1}}\right)^{|A \setminus B| - 1} \left(1 - \frac{1}{2^j}\right)^{|B|} \\ &+ \sum_{j=1}^{+\infty} \frac{|B \setminus A|}{2^{j+1}} \left(1 - \frac{1}{2^{j+1}}\right)^{|B \setminus A| - 1} \left(1 - \frac{1}{2^j}\right)^{|A|}.\end{aligned}$$

Due to the limited space, we omit the details of how γ is derived. Similar to MinHash, we have $P(\max(h(A)) \neq \max(h(B))) = 1 - J_{A,B}$. Therefore, we have $\gamma < 1 - J_{A,B}$. Although γ can be estimated similar to MinHash using k hash functions $h_1, \dots, h_k$, that is,

$$\mathbb{E}(\gamma) = \frac{\sum_{i=1}^{k} \mathbf{1}(\mathrm{MaxLog}(h_i(A)) \neq \mathrm{MaxLog}(h_i(B)))}{k},$$

unfortunately, it is difficult to compute $J_{A,B}$ from γ. To solve this problem, we observe

$$\begin{aligned}&P(\mathrm{MaxLog}(h(A)) \neq \mathrm{MaxLog}(h(B)) \wedge \delta_{A,B} = 1) \\ &\approx 0.7213(1 - J_{A,B}),\end{aligned} \tag{8}$$

where $\delta_{A,B} = 1$ indicates that there exists one and only one element in $A \cup B$ of which log-rank equals $\mathrm{MaxLog}(h(A \cup B))$.

Based on the above three observations, we propose to incrementally and accurately estimate the value of $P(\mathrm{MaxLog}(h(A)) \neq \mathrm{MaxLog}(h(B)) \wedge \delta_{A,B} = 1)$ using k hash functions $h_1, \dots, h_k$. Then, we easily infer the value of $J_{A,B}$.

4.2 Data Structure

The MaxLogHash sketch of a user u, i.e., S_u, consists of k bit-strings, where each bit-string $S_u[i], 1 \le i \le k$, has two components, $s_u[i]$ and $m_u[i]$, i.e.,

$$S_u[i] = s_u[i] \parallel m_u[i].$$

At any time t, $m_u[i]$ records the maximum hash value of items in $I_u^{(t)}$ with respect to hash function $r_i(\cdot) = \lfloor -\log_2 h_i(\cdot) \rfloor$, i.e., $m_u[i] = \max_{w \in I_u^{(t)}} r_i(w)$, where $I_u^{(t)}$ refers to the set of items that user u connected to before and including time t; $s_u[i]$ consists of 1 bit and its value indicates whether there exists one and only one item $w \in I_u$ such that $r_i(w) = m_u[i]$. As we mentioned, we can use $\lceil \log_2 \log_2 |I| \rceil$ bits to record the value of $m_u[i]$ with a high probability (very close to 1). When $m_u[i] \ge 2^{\lceil \log_2 \log_2 |I| \rceil}$, we use a hash table to record tuples $(u, i, m_u[i])$ for all users.

Algorithm 1. The Pseudo-Code of MaxLogHash

```
Input: $\Pi, k, \alpha = 0.7213$.
1  $U \leftarrow \emptyset$;
2  foreach user-item pair $(u, v)$ arriving on stream $\Pi$ do
3      UpdateUIPair$(u, v)$;
4  Function UpdateUIPair$(u, v)$
5  if $u \notin U$ then
6      $U \leftarrow U \cup \{u\}$;
7      for $i \in \{1, \dots, k\}$ do
8          $s_u[i] \leftarrow 1$;
9          $m_u[i] \leftarrow \lfloor -\log_2 h_i(v) \rfloor$;
10 else
11     for $i \in \{1, \dots, k\}$ do
12         $r_i(v) \leftarrow \lfloor -\log_2 h_i(v) \rfloor$;
13         if $r_i(v) \ge m_u[i]$ then
14             if $r_i(v) == m_u[i]$ then
15                 $s_u[i] \leftarrow 0$;
16                 continue();
17             $s_u[i] \leftarrow 1$;
18             $m_u[i] \leftarrow r_i(v)$;
19 Function EstimateJaccard$(u_1, u_2)$
20 $\hat{k} \leftarrow 0$;
21 for $i \in \{1, \dots, k\}$ do
22     if $m_{u_1}[i] == m_{u_2}[i]$ then
23         continue();
24     if $m_{u_1}[i] > m_{u_2}[i]$ and $s_{u_1}[i] == 1$ then
25         $\hat{k} \leftarrow \hat{k} + 1$;
26         continue();
27     if $m_{u_1}[i] < m_{u_2}[i]$ and $s_{u_2}[i] == 1$ then
28         $\hat{k} \leftarrow \hat{k} + 1$;
29 $\hat{J}_{u_1,u_2} \leftarrow 1 - \hat{k}k^{-1}\alpha^{-1}$;
```

4.3 Update Procedure

For each user $u \in U$, when it first connects to an item w in stream Π, we initialize the MaxLogHash sketch of user u as

$$S_u[i] = 1 \parallel r_i(w), \quad i = 1, \dots, k,$$

where $r_i(w) = \lfloor -\log_2 h_i(w) \rfloor$. That is, we set indicator $s_u[i] = 1$ and register $m_u[i] = r_i(w)$. For any other item v that user u connects to after the first item w, i.e., an user-item pair (u, v) occurring on stream Π after the user-item pair (u, w), we update it as follows: We first compute the log-rank of item v, i.e., $r_i(v) = \lfloor -\log_2 h_i(v) \rfloor$, $i = 1, \dots, k$. When $r_i(v)$ is smaller than $m_u[i]$, we perform no further operations for updating the user-item (u, v). When $r_i(v) = m_u[i]$, it indicates that at least two items in I_u has a log-rank value $m_u[i]$. Therefore, we simply set $s_u[i] = 0$. When $r_i(v) > m_u[i]$, we set $S_u[i] = 1 \parallel r_i(v)$. Algorithm 1 shows the pseudo-code of MaxLogHash.

4.4 Jaccard Similarity Estimation

Define variables

$$\chi_{u_1 \cup u_2}[i] = \mathbf{1}(m_{u_1}[i] \neq m_{u_2}[i]), \quad i = 1, \dots, k, \tag{9}$$

$$\psi_{u_1 \cup u_2}[i] = \begin{cases} s_{u_1}[i], & m_{u_1}[i] > m_{u_2}[i] \\ s_{u_2}[i], & m_{u_1}[i] < m_{u_2}[i] \\ -1, & m_{u_1}[i] = m_{u_2}[i] \end{cases} \quad (10)$$

$$\delta_{u_1 \cup u_2}[i] = \mathbf{1}(\chi_{u_1 \cup u_2}[i] = 1)\mathbf{1}(\psi_{u_1 \cup u_2}[i] = 1). \quad (11)$$

Note that $\delta_{u_1 \cup u_2}[i] = 1$ indicates that there exists one and only one element in set $\cup(u_1, u_2)$ of which log-rank equals $\max_{w \in \cup(u_1,u_2)} r_i(w)$ with respect to function r_i. Let $d_{u_1 \cup u_2} = |\cup(u_1, u_2)|$, we have the following theorem to compute $P(\delta_{u_1 \cup u_2}[i] = 1)$.

Theorem 1. *For non-empty sets I_{u_1} and I_{u_2}, we have $P(\delta_{u_1 \cup u_2}[i] = 1) = 0$, $i = 1, \ldots, k$, when $d_{u_1 \cup u_2} = 1$. Otherwise, we have*

$$P(\delta_{u_1 \cup u_2}[i] = 1) = \alpha_{d_{u_1 \cup u_2}}(1 - J_{u_1,u_2}), \quad i = 1, \ldots, k,$$

where $\alpha_n = n \sum_{j=1}^{+\infty} \frac{1}{2^{j+1}} \left(1 - \frac{1}{2^j}\right)^{n-1}$, $n \geq 2$.

Proof. Let r^* be the maximum log-rank of all items in $\cup(u_1, u_2)$. When two items w and v in I_{u_1} or I_{u_2} has the log-rank value r^*, we easily find that $\psi_{u_1 \cup u_2}[i] = 0$. When only one item w in I_{u_1} and only one item v in I_{u_2} have the log-rank value r^*, we easily find that $\chi_{u_1 \cup u_2}[i] = 0$. Let

$$\Delta(u_1, u_2) = (I_{u_1} \setminus I_{u_2}) \cup (I_{u_2} \setminus I_{u_1}) = \cup(u_1, u_2) \setminus \cap(u_1, u_2),$$

and $d_{u_1 \Delta u_2} = |\Delta(u_1, u_2)|$. Then, we find that event $\chi_{u_1 \cup u_2}[i] = 1 \wedge \psi_{u_1 \cup u_2}[i] = 1$ happens (i.e., $\delta_{u_1 \cup u_2}[i] = 1$) only when one item w in $\Delta(u_1, u_2)$ has a log-rank value larger than all items in $\cup(u_1, u_2) \setminus \{w\}$. For any item $v \in I$, we have $h_i(v) \sim Uniform(0, 1)$ and so $r_i(v) \sim Geometric(1/2)$, supported on the set $\{0, 1, 2, \ldots\}$. Based on the above observations, when $d_{u_1 \cup u_2} \geq 2$, we have

$$\begin{aligned} &P(\delta_{u_1 \cup u_2}[i] = 1 \wedge r^* = j) \\ &= \sum_{w \in \Delta(u_1,u_2)} P(r_i(w) = j) \prod_{v \in \cup(u_1,u_2) \setminus \{w\}} P(r_i(v) < j) \\ &= \frac{d_{u_1 \Delta u_2}}{2^{j+1}} \left(1 - \frac{1}{2^j}\right)^{d_{u_1 \cup u_2} - 1}. \end{aligned} \quad (12)$$

Therefore, we have

$$\begin{aligned} P(\delta_{u_1 \cup u_2}[i] = 1) &= \sum_{j=0}^{+\infty} P(\delta_{u_1 \cup u_2}[i] = 1 \wedge r^* = j) \\ &= \sum_{w \in \Delta(u_1,u_2)} P(r_w = j) \prod_{v \in \cup(u_1,u_2) \setminus \{w\}} P(r_v < j) \\ &= \sum_{j=1}^{+\infty} \frac{d_{u_1 \Delta u_2}}{2^{j+1}} \left(1 - \frac{1}{2^j}\right)^{d_{u_1 \cup u_2} - 1} \end{aligned} \quad (13)$$

where the last equation holds for $|\Delta(u_1, u_2)| = |\cup(u_1, u_2)| - |\cap(u_1, u_2)|$. □

Define variable $\hat{k} = \sum_{i=1}^{k} \mathbf{1}(\delta_{u_1 \cup u_2}[i] = 1)$. From Theorem 1, the expectation of $\hat{k}$ is computed as

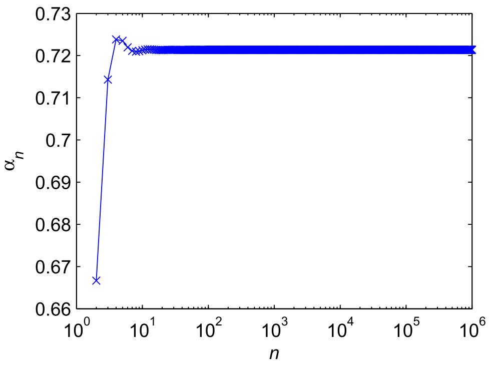


Fig. 1. Value of α_n, $n = 2, \ldots, 10^6$ where $|\alpha_2 - 0.7213| = 0.0546$, $|\alpha_n - 0.7213| \leq 0.007$ when $n \geq 3$.

$$\begin{aligned} \mathbb{E}(\hat{k}) &= \mathbb{E}\left(\sum_{i=1}^{k} \mathbf{1}(\delta_{u_1 \cup u_2}[i] = 1)\right) \\ &= \sum_{i=1}^{k} \mathbb{E}(\mathbf{1}(\delta_{u_1 \cup u_2}[i] = 1)) \\ &= k\alpha_{d_{u_1 \cup u_2}}(1 - J_{u_1 \cup u_2}). \end{aligned} \quad (14)$$

Therefore, we have

$$J_{u_1,u_2} = 1 - \frac{\mathbb{E}(\hat{k})}{k\alpha_{d_{u_1 \cup u_2}}}. \quad (15)$$

Note that the cardinality of set $\cup(u_1, u_2)$ (i.e., $d_{u_1 \cup u_2}$) is unknown. To solve this challenge, we find that

$$\begin{aligned} \alpha_n &= \frac{n}{2} \sum_{j=1}^{+\infty} \frac{1}{2^j} \left(1 - \frac{1}{2^j}\right)^{n-1} \\ &= \frac{n}{2} \sum_{j=1}^{+\infty} \frac{1}{2^j} \sum_{l=0}^{n-1} \binom{n-1}{l} \left(-\frac{1}{2^j}\right)^{n-l-1} \\ &= \frac{n}{2} \sum_{l=0}^{n-1} (-1)^{n-l-1} \binom{n-1}{l} \sum_{j=1}^{+\infty} \frac{1}{2^{j(n-l)}} \\ &= \frac{n}{2} \sum_{l=0}^{n-1} (-1)^{n-l-1} \binom{n-1}{l} \frac{1}{2^{n-l} - 1}. \end{aligned} \quad (16)$$

Fig. 1 shows that the value of α_n, $n = 2, 3, \ldots$. We find that $\alpha_n \approx \alpha = 0.7213$ when $n \geq 2$. Therefore, we estimate J_{u_1,u_2} as

$$\hat{J}_{u_1,u_2} = 1 - \frac{\hat{k}}{k\alpha}. \quad (17)$$

4.5 Error Analysis

Theorem 2. *For any users $u_1, u_2 \in U$, we have*

$$\mathbb{E}(\hat{J}_{u_1,u_2}) - J_{u_1,u_2} = (1 - \beta_{d_{u_1 \cup u_2}})(1 - J_{u_1,u_2}),$$

where $\beta_n = \frac{\alpha_n}{\alpha}$. The variance of $\hat{J}_{u_1,u_2}$ is computed as

$$\mathrm{Var}(\hat{J}_{u_1,u_2}) = \frac{\beta_{d_{u_1 \cup u_2}}(1 - J_{u_1,u_2})(\alpha^{-1} - \beta_{d_{u_1 \cup u_2}}(1 - J_{u_1,u_2}))}{k}.$$

When $d_{u_1 \cup u_2} \geq 3$, we have $|\beta_{d_{u_1 \cup u_2}} - 1| \leq 0.01$, and so $\mathbb{E}(\hat{J}_{u_1,u_2}) \approx J_{u_1,u_2}$ and $\mathrm{Var}(\hat{J}_{u_1,u_2}) \approx \frac{(1 - J_{u_1,u_2})(J_{u_1,u_2} + 0.3864)}{k}$.

Proof. From Equation (14), we easily have

$$\begin{aligned}\mathbb{E}(\hat{J}_{u_1,u_2}) &= \mathbb{E}\left(1-\frac{\hat{k}}{k\alpha}\right) = 1-\frac{k\alpha_{d_{u_1\cup u_2}}(1-J_{u_1,u_2})}{k\alpha}\\ &= \beta_{d_{u_1\cup u_2}}J_{u_1,u_2}+1-\beta_{d_{u_1\cup u_2}}.\end{aligned} \tag{18}$$

To derive $\mathrm{Var}(\hat{J}_{u_1,u_2})$, we first compute

$$\begin{aligned}\mathbb{E}(\hat{k}^2) &= \mathbb{E}\left(\left(\sum_{i=1}^{k}\mathbf{1}(\delta_{u_1\cup u_2}[i]=1)\right)^2\right)\\ &= \sum_{i=1}^{k}\mathbb{E}\Big((\mathbf{1}(\delta_{u_1\cup u_2}[i]=1))^2\Big)\\ &\quad+\sum_{i\neq j,1\le i,j\le k}\mathbb{E}\big(\mathbf{1}(\delta_{u_1\cup u_2}[i]=1)\mathbf{1}(\delta_{u_1\cup u_2}[j]=1)\big)\\ &= k\alpha_{d_{u_1\cup u_2}}(1-J_{u_1,u_2})+k(k-1)\alpha^2_{d_{u_1\cup u_2}}(1-J_{u_1,u_2})^2.\end{aligned}$$

Then, we have

$$\begin{aligned}&\mathrm{Var}(\hat{k}) = \mathbb{E}(\hat{k}^2)-(\mathbb{E}(\hat{k}))^2\\ &= k\alpha_{d_{u_1\cup u_2}}(1-J_{u_1,u_2})(1-\alpha_{d_{u_1\cup u_2}}(1-J_{u_1,u_2})).\end{aligned} \tag{19}$$

From the definition of $\hat{J}_{u_1,u_2}$, we have

$$\mathrm{Var}(\hat{J}_{u_1,u_2}) = \mathrm{Var}\left(1-\frac{\hat{k}}{k\alpha}\right) = \frac{\mathrm{Var}(\hat{k})}{k^2\alpha^2}. \tag{20}$$

Then, we easily obtain a closed-form formulas of $\mathrm{Var}(\hat{J}_{u_1,u_2})$ from Equation (19). □

5 MaxLogOPH++

5.1 Basic Idea

Although our method MaxLogHash is memory-efficient, it suffers with the same efficiency problem as MinHash, i.e., hashing k times for each coming element. Inspired by one permutation hashing (OPH) [18], which significantly reduces the time complexity of MinHash processing each element, we develop a faster method, MaxLogOPH++, to further reduce the time complexity of MaxLogHash from $O(k)$ to $O(1)$. In particular, we use a hash function which splits items in I_u into k subsets without overlapping at random, and each register $S_u[i]$, $1\le i\le k$ records the maximum rank value[4] $m_u[i]$ as well as the value of indicator $s_u[i]$ for elements in the ith subset, which is similar to the MaxLogHash method in Section 4. Different from MaxLogHash, MaxLogOPH++ introduces a counter d_u for each user u which records the number of elements in I_u, i.e., $d_u = |I_u|$, over time.

To estimate similarities via MaxLogOPH++, we first build a probabilistic model based on generated sketches. However, we find that the exact probabilistic model is hard to solve. Inspired by [19], we instead use the Poisson approximation model to approximate the exact model and then infer the Maximum Likelihood Estimation [15] of Jaccard similarity using the Newton-Raphson method [16].

4. For MaxLogOPH++, the rank value of an empty bucket is always smaller than that of any element.

Algorithm 2. The Pseudo-Code of MaxLogOPH++

```
Input: $\Pi, k, \alpha = 0.7213$.
1  $U \leftarrow \emptyset$;
2  foreach user-item pair $(u, v)$ arriving on stream $\Pi$ do
3      UpdateUIPair($u, v$);
4  Function UpdateUIPair($u, v$)
5  if $u \notin U$ then
6      $U \leftarrow U \cup \{u\}$, $d_u \leftarrow 1$;
7      for $i \in \{1, \ldots, k\}$ do
8          $s_u[i] \leftarrow 0$, $m_u[i] \leftarrow 0$;
9      $i_v \leftarrow g(v)$;
10     $m_u[i_v] \leftarrow \lfloor -\log_2 h(v) \rfloor$;
11 else
12     $d_u \leftarrow d_u + 1$;
13     $i_v \leftarrow g(v)$;
14     $r(v) \leftarrow \lfloor -\log_2 h(v) \rfloor$;
15     if $r(v) \ge m_u[i_v]$ then
16         if $r(v) == m_u[i_v]$ then
17             $s_u[i_v] \leftarrow 0$;
18             continue();
19         $s_u[i_v] \leftarrow 1$, $m_u[i_v] \leftarrow r(v)$;
20 Function EstimateJaccard($u_1, u_2$)
21 $\hat{k} \leftarrow 0$;
22 for $i \in \{1, \ldots, k\}$ do
23     if $m_{u_1}[i] == m_{u_2}[i]$ then
24         continue();
25     if $m_{u_1}[i] > m_{u_2}[i]$ and $s_{u_1}[i] == 1$ then
26         $\hat{k} \leftarrow \hat{k} + 1$;
27         continue();
28     if $m_{u_1}[i] < m_{u_2}[i]$ and $s_{u_2}[i] == 1$ then
29         $\hat{k} \leftarrow \hat{k} + 1$;
30 $J^{(0)} \leftarrow 1 - \hat{k}k^{-1}\alpha^{-1}$;
   /* $l_T$: the maximum number of iterations. */
31 for $l = 1, 2, \ldots, l_T$ do
32     $J^{(l)} \leftarrow J^{(l-1)} - \frac{f(J^{(l-1)})}{f'(J^{(l-1)})}$;
33 return $J^{(l_T)}$;
```

5.2 Jaccard Similarity Estimation

Exact Probabilistic Model. Let g be a hash function that maps any element $v \in I$ to a random integer in range $[1, k]$. For any users $u_1, u_2 \in U$, let $\cup_i(u_1, u_2)$ denote the subset of $\cup(u_1, u_2)$ in which each element's hash value equals i with respect to function g, that is,

$$\cup_i(u_1,u_2) = \{v : g(v) = i, v \in \cup(u_1,u_2)\}, \quad 1 \le i \le k.$$

Our algorithm randomly splits items from $\cup(u_1, u_2)$ into k non-overlapped subsets $\cup_1(u_1, u_2)$, …, $\cup_k(u_1, u_2)$. Define $d_{u_1\cup u_2} = |\cup(u_1, u_2)|$ and $d_{u_1\cup u_2,i} = |\cup_i(u_1, u_2)|$. Using the classical balls-in-urns model [20] (randomly throwing balls into k bins), we derive the probability distribution of vector $(d_{u_1\cup u_2,1}, \ldots, d_{u_1\cup u_2,k})$ as

$$P(d_{u_1\cup u_2,1},\ldots,d_{u_1\cup u_2,k}|d_{u_1\cup u_2}) = \frac{\binom{d_{u_1\cup u_2}}{d_{u_1\cup u_2,1},\ldots,d_{u_1\cup u_2,k}}}{k^{d_{u_1\cup u_2}}}. \tag{21}$$

Similar to MaxLogHash, we let $\delta_{u_1\cup u_2}[i]$ indicate whether there exists one and only one element in set $\cup_i(u_1, u_2)$ of which log-rank equals $\max_{w\in\cup_i(u_1,u_2)} r(w)$ with respect to function r. We give the following theorem to compute $P(\delta_{u_1\cup u_2}[i]|d_{u_1\cup u_2,i})$.

Theorem 3. *For non-empty sets I_{u_1} and I_{u_2}, we have*

$$P(\delta_{u_1\cup u_2}[i]=1|d_{u_1\cup u_2,i})=\alpha_{d_{u_1\cup u_2,i}}(1-J_{u_1,u_2}),$$

where $\alpha_1=1$ and $\alpha_2,\alpha_3,\ldots$ have the same definition as those in Theorem 1.

We omit the proof which is similar to Theorem 1. Let $\boldsymbol{\delta}_{u_1\cup u_2}=(\delta_{u_1\cup u_2}[1],\ldots,\delta_{u_1\cup u_2}[k])$. Let $P(\boldsymbol{\delta}_{u_1\cup u_2}|d_{u_1\cup u_2})$ denote the probability density function (PDF) of $\boldsymbol{\delta}_{u_1\cup u_2}$ given $d_{u_1\cup u_2}$. In what follows we drop the subscript $u_1\cup u_2$ for brevity. From Equation (21), we compute $P(\boldsymbol{\delta}|d)$ as

$$\begin{aligned}P(\boldsymbol{\delta}|d)&=\sum_{d_1+\ldots+d_k=d_u}P(d_1,\ldots,d_k|d)P(\boldsymbol{\delta}|d_1,\ldots,d_k)\\&=\sum_{d_1+\ldots+d_k=d_u}\frac{\binom{d}{d_1,\ldots,d_k}}{k^d}\prod_{i=1}^{k}P(\delta[i]|d_i),\end{aligned}\tag{22}$$

where $P(\boldsymbol{\delta}|d_1,\ldots,d_k)=\prod_{i=1}^{k}P(\delta[i]|d_i)$ because the value of $\delta[i]$ can only be changed by the elements thrown into subset $\cup_i(u_1,u_2)$.

Poisson Approximation Model. The above probabilistic model is hard to compute because $d_1,\ldots,d_k$ are not independent. Therefore, inspired by [19], we use the Poisson approximation technique to remove the dependence of $d_1,\ldots,d_k$. Specifically, we assume that the value of d is distributed according to a Poisson distribution with parameter λ, i.e., $d\sim\mathrm{Poisson}(\lambda)$. Then, the PDF of $\boldsymbol{\delta}$ given λ is

$$\begin{aligned}P(\boldsymbol{\delta}|\lambda)&=\sum_{d=0}^{+\infty}P(\boldsymbol{\delta}|d)\frac{e^{-\lambda}\lambda^d}{d!}\\&=\sum_{d=0}^{+\infty}\sum_{d_1+\ldots+d_k=d}\frac{\binom{d}{d_1,\ldots,d_k}}{k^d}\prod_{i=1}^{k}P(\delta[i]|d_i)\frac{e^{-\lambda}\lambda^d}{d!}\\&=\sum_{d_1=0}^{+\infty}\cdots\sum_{d_k=0}^{+\infty}\prod_{i=1}^{k}P(\delta[i]|d_i)\frac{e^{-\frac{\lambda}{k}}\lambda^{d_i}}{d_i!k^{d_i}}\\&=\prod_{i=1}^{k}\sum_{d_i=0}^{+\infty}P(\delta[i]|d_i)\frac{e^{-\frac{\lambda}{k}}\lambda^{d_i}}{d_i!k^{d_i}}\end{aligned}\tag{23}$$

where the third equation holds because $\binom{d}{d_1,\ldots,d_k}=\frac{d!}{d_1!\cdots d_k!}$. Given $d\sim\mathrm{Poisson}(\lambda)$, the above equation indicates that the values of $\delta[1],\ldots,\delta[k]$ are independent and identically distributed. In addition, we note that the values of $d_1,\ldots,d_k$ are independent and identically distributed according to a Poisson distribution $\mathrm{Poisson}(\frac{\lambda}{k})$.

Then, we briefly discuss why the Poisson approximation model (i.e., $P(\boldsymbol{\delta}|d)$) works. Let $\phi(\boldsymbol{\delta})$ be an estimator of J, we denote the expectation and the variance of $\phi(\boldsymbol{\delta})$ under $P(\boldsymbol{\delta}|d)$ as $\mathbb{E}_d(\phi(\boldsymbol{\delta}))$ and $\mathrm{Var}_d(\phi(\boldsymbol{\delta}))$ respectively. We let $\mathbb{E}_{\mathrm{Poisson}(\lambda)}(\phi(\boldsymbol{\delta}))$ and $\mathrm{Var}_{\mathrm{Poisson}(\lambda)}(\phi(\boldsymbol{\delta}))$ denote the expectation and the variance of $\phi(\boldsymbol{\delta})$ under $P(\boldsymbol{\delta}|d)$. When setting $\lambda=d$, [19], [21], [22] prove that statistical properties of $\mathbb{E}_d(\phi(\boldsymbol{\delta}))$ and $\mathrm{Var}_d(\phi(\boldsymbol{\delta}))$ are well approximated by $\mathbb{E}_{\mathrm{Poisson}(\lambda)}(\phi(\boldsymbol{\delta}))$ and $\mathrm{Var}_{\mathrm{Poisson}(\lambda)}(\phi(\boldsymbol{\delta}))$, which is the *depoissonization step*.

Estimator of J. Next, we elaborate our method to estimate the Jaccard similarity J of any given user pair under the Poisson approximation model. Let $p_J=P(\delta[i]=1|\lambda)$ and $\hat{k}=\sum_{i=1}^{k}\mathbf{1}(\delta[i]=1)$. Then, given k independent observed variables $\delta[1],\ldots,\delta[k]$, the likelihood function is formalized as

$$\begin{aligned}\log\mathcal{L}(\boldsymbol{\delta}|\lambda)&=\log P(\boldsymbol{\delta}|\lambda)\\&=\log\prod_{i=1}^{k}P(\delta[i]|\lambda)\\&=\log p_J^{\hat{k}}(1-p_J)^{k-\hat{k}}\\&=\hat{k}\log p_J+(k-\hat{k})\log(1-p_J).\end{aligned}$$

By solving $\frac{\partial\log\mathcal{L}(\boldsymbol{\delta}|\lambda)}{\partial p_J}=0$, we have

$$\hat{p}_J=\frac{\hat{k}}{k}.\tag{24}$$

Then, we compute p_J as

$$\begin{aligned}p_J&=P(\delta[i]=1|\lambda)\\&=\sum_{d_i=1}^{\infty}P(\delta[i]=1|d_i)P(d_i|\lambda)\\&=(1-J)P(d_i=1|\lambda)+\alpha_2(1-J)P(d_i=2|\lambda)\\&\quad+\alpha_3(1-J)P(d_i=3|\lambda)+\cdots\\&\approx(1-J)P(d_i=1|\lambda)+\alpha_2(1-J)P(d_i=2|\lambda)\\&\quad+\alpha(1-J)\left(1-\sum_{d_i=0}^{2}P(d_i|\lambda)\right),\end{aligned}\tag{25}$$

where the last approximate equality holds because $\alpha_n\approx\alpha=0.7213$ when $n\geq 3$. From Equations (24) and (25), we have

$$\begin{aligned}&(1-J)P(d_i=1|\lambda)+\alpha_2(1-J)P(d_i=2|\lambda)\\&+\alpha(1-J)\left(1-\sum_{d_i=0}^{2}P(d_i|\lambda)\right)=\frac{\hat{k}}{k}.\end{aligned}\tag{26}$$

Recall that $\lambda=d_{u_1\cup u_2}=\frac{d_{u_1}+d_{u_2}}{1+J}$. We use the Newton-Raphson method [16] to obtain J from the above equation. Let

$$\begin{aligned}f(J)&=(1-J)P(d_i=1|\lambda)+\alpha_2(1-J)P(d_i=2|\lambda)\\&\quad+\alpha(1-J)\left(1-\sum_{d_i=0}^{2}P(d_i|\lambda)\right)-\frac{\hat{k}}{k}.\end{aligned}\tag{27}$$

The Newton-Raphson method starts from an initial estimation $J^{(0)}$, and then repeats the following procedure

$$J^{(l+1)}\leftarrow J^{(l)}-\frac{f(J^{(l)})}{f'(J^{(l)})},\quad l\geq 0,$$

until $f(J)$ converges, where $f'(J^{(l)})$ is the derivative of $f(J)$ at $J=J^{(l)}$. To set the initial estimation, we initialize

$$J^{(0)}=1-\hat{k}k^{-1}\alpha^{-1},\tag{28}$$

which is the estimate given by MaxLogOPH [14].

Algorithm 2 shows the pseudo-code of MaxLogOPH++, which includes the update and estimation modules. In update module (Lines 4 - 19), for each coming user-item pair (u, v), we first assign it an index $1 \leq i \leq k$. Then, we update registers $s_u[i]$ and $m_u[i]$ according to the log-rank values of item v, which is similar to MaxLogHash. In estimation module (Lines 20 - 33), we first estimate $J^{(0)}$ according to Equation (28). Then, we iterate on $J^{(l)}$ using Newton-Raphson method and return $J^{(l_T)}$, where l_T is the maximum number of iterations.

5.3 Discussion

Compared with MaxLogOPH [14]. When the cardinalities of sets is large, i.e., $d \gg k$, we can approximate Equation (26) as

$$\alpha(1-J) \approx \frac{\hat{k}}{k},$$

because $\sum_{d_i=0}^{2} P(d_i|\lambda) \approx 0$. For example, when $d = 10,000$ and $k = 100$, we have $\sum_{d_i=0}^{2} P(d_i|\lambda) = 1.86 \times 10^{-40}$. We find that the above equation is exactly the estimator of MaxLogOPH, which explains why MaxLogOPH only gives large errors when the cardinalities are small. However, our new method MaxLogOPH++ uses the Newton-Raphson method to solve Equation (26), which significantly reduces the estimate error for small sets. Later in the experiments, we will show that MaxLogOPH++ is several times more accurate than MaxLogOPH.

Compared With OPH [18]. OPH randomly splits the stream into k buckets and each bucket uses a register to keep tracking of the element with the minimum value among the bucket. Similar to MinHash, OPH uses a 64-bit (resp. 32-bit) register for storing the minimum of 64-bit (resp. 32-bit) set elements. Therefore, OPH and MinHash require the same amount of memory and our method MaxLogOPH++ reduces their memory usage by $5 \sim 9$ times. In addition, OPH may also have many empty buckets and exhibit large estimation errors. Although several densification techniques [23], [24], [25], [26] have been proposed to resign the registers of empty buckets by other non-empty ones to improve the estimation accuracy, these algorithms may consume large time (i.e., $O(k^2)$ for [23], [24], [25] and $O(k \log k)$ for [26]). In contrast, our method builds a likelihood function of generated sketches and then directly infers the Jaccard similarity using the MLE and Newton-Raphson method. In our experiments, we find that the Newton-Raphson method usually coverages in $5 \sim 10$ iterations, which takes small and constant time (i.e., 1×10^{-5} seconds as shown in Fig. 6).

Compared With HyperLogLog [19]. Similar to HyperLogLog, MaxLogOPH++ randomly splits the stream into k registers and stores the maximal log-rank values. Different from HyperLogLog, MaxLogOPH++ introduces a flag value $\delta_{(u_1,u_2)}[i]$ for each register to indicate whether there exists one and only one element in set $\cup_i(u_1, u_2)$ of which log-rank equals $\max_{w \in \cup_i(u_1,u_2)} r(w)$ with respect to function r. In addition, MaxLogOPH++ builds a likelihood function of generated sketches and then directly infers the Jaccard similarity with MLE and Newton-Raphson method. As a result, MaxLogOPH++ is more accurate than HyperLogLog with similar update and estimation time.

TABLE 1
Real-World Datasets Used in Our Experiments, Where "size" Refers to the Number of Elements in the Stream

	$\lvert U\rvert$	$\lvert I\rvert$	size
MUSHROOM	119	8,124	186,852
CONNECT	127	67,557	2,904,951
EPINIONS	876,252	120,492	13,668,320
TREC	1,729,302	556,077	83,629,405

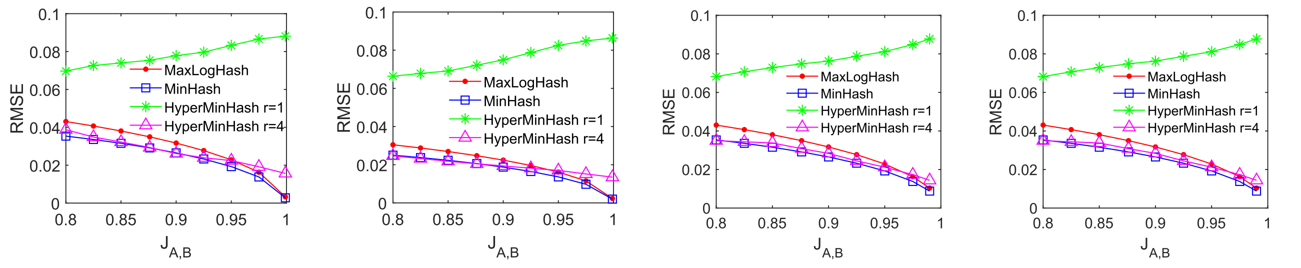
6 Evaluation

In this section, we evaluate our methods with the state-of-the-arts on both synthetic and real-world datasets. All algorithms run on a computer with a Quad-Core Intel(R) Xeon (R) CPU E3-1226 v3 CPU 3.30 GHz processor.

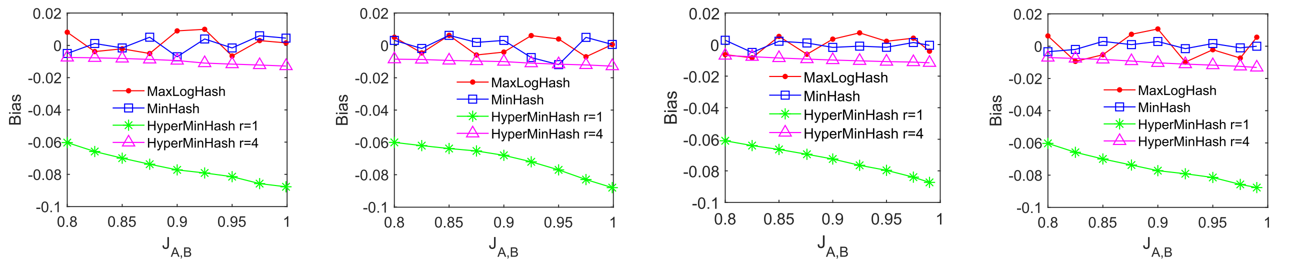
6.1 Datasets

For simplicity, we assume that elements in sets are 32-bit numbers, i.e., $I = \{0, 1, \ldots, 2^{32} - 1\}$. We evaluate the performance of our method MaxLogHash and MaxLogOPH++ on a variety of datasets.

1) *Synthetic datasets.* Our synthetic datasets consist of set-pairs A and B with various cardinalities and Jaccard similarities. We conduct our experiments on the following two different settings:
   - *Balanced set-pairs* (i.e., $|A| = |B|$). We set $|A| = |B| = n$ and vary $J_{A,B}$ in $\{0.80, 0.81, \ldots, 1.00\}$. Specially, we generate set A by randomly selecting n different numbers from I and generate set B by randomly selecting $|A \cap B| = \frac{J_{A,B}|A|}{1+J_{A,B}}$ different numbers from set A and $n - |A \cap B|$ different numbers from set $I \setminus A$. In our experiments, we set $n = 10,000$ by default.
   - *Unbalanced set-pairs* (i.e., $|A| \neq |B|$). We set $|A| = n$ and $|B| = J_{A,B}n$, where we vary $J_{A,B} \in \{0.80, 0.81, \ldots, 0.99\}$. Specially, we generate set A by randomly selecting n different numbers from I and generate set B by selecting $J_{A,B}n$ different elements from A.
2) *Real-world datasets.* Similar to [11], we evaluate the performance of our method on the detection of item-pairs (e.g., pairs of products) that always appear together in the same records (e.g., transactions). For association rule learning (also known as the market-basket problem), discovering item-pairs such as "beer and diapers" and "Beluga vaviar and Ketel vodka" that are commonly-purchased or nearly always bought together is critical for learning pairwise associations. Cohen *et al.* [27] applied MinHash to solve this problem. We conduct experiments on four real-world datasets from different areas: MUSHROOM [11], CONNECT [11], TREC [28], and EPINIONS [29]. MUSHROOM and CONNECT are also used in [11]. We generate a stream of item-record pairs for each dataset, where a record can be viewed as a transaction and items in the same record can be viewed as products bought together. For each record x in the dataset of interest and every item w in x, we append an element (w, x) to the stream of item-record pairs. We summarize all datasets in Table 1.



(a) (balanced) RMSE, $k = 128$ (b) (balanced) RMSE, $k = 256$ (c) (unbalanced) RMSE, $k = 128$ (d) (unbalanced) RMSE, $k = 256$



(e) (balanced) Bias, $k = 128$ (f) (balanced) Bias, $k = 256$ (g) (unbalanced) Bias, $k = 128$ (h) (unbalanced) Bias, $k = 256$

Fig. 2. Estimation error of our method MaxLogHash in comparison with MinHash and HyperMinHash on both balanced and unbalanced set-pairs with $k = 128$.

6.2 Baselines

Our methods use k 6-bit registers to build a sketch for each set. We compare our methods with the following state-of-the-art methods:

- *MinHash [7]*. MinHash builds a sketch for each set. A MinHash sketch consists of k 32-bit registers.
- *b-bit MinHash [10]*. b-bit MinHash compresses each 32-bit register used by MinHash into a b-bit register. In our experiment, we update b-bit MinHash in an incremental manner and set b to different values.
- *HyperLogLog [19]*. A HyperLogLog sketch consists of k 5-bit registers, and is originally designed for estimating a set's cardinality. One can easily obtain a HyperLogLog sketch of $A \cup B$ by merging the HyperLogLog sketches of sets A and B and then use the sketch to estimate $|A \cup B|$. Therefore, HyperLogLog can also be used to estimate $J_{A,B}$ by approximating $\frac{|A|+|B|-|A\cup B|}{|A\cup B|}$.
- *HyperMinHash [12]*. A HyperMinHash sketch consists of k q-bit registers and k r-bit registers. The first k q-bit registers can be viewed as a HyperLogLog sketch. To guarantee the performance for large sets (including up to 2^{32} elements), we set $q = 5$.

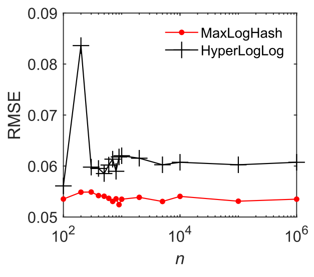
6.3 Metrics

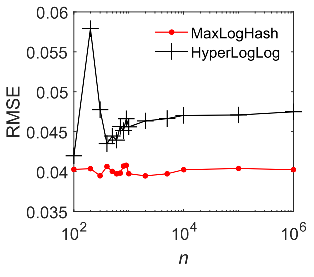
We evaluate both efficiency and effectiveness of our methods in comparison with the above baseline methods. For efficiency, we evaluate the running time of all methods. Specially, we study the time for updating each set element and estimating set similarities, respectively. The update time determines the maximum throughput that a method can handle, and the estimation time determines the delay in querying the similarity of set-pairs. For effectiveness, we evaluate the error of estimation $\hat{J}$ with respect to its true value J using metrics: bias and root mean square error (RMSE), i.e., $\text{Bias}(\hat{J}) = \mathbb{E}(\hat{J}) - J$ and $\text{RMSE}(\hat{J}) = \sqrt{\mathbb{E}((\hat{J} - J)^2)}$. Our experimental results are empirically computed from 1,000 independent runs by default. We further evaluate our methods on the detection of association rules, and use *precision* and *recall* to evaluate the performance.

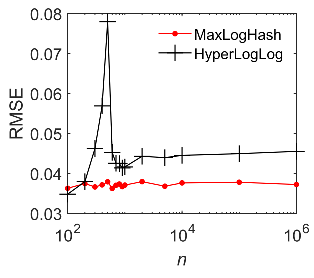
6.4 Accuracy of Similarity Estimation

MaxLogHash versus MinHash and HyperMinHash. From Figs. 2a, 2b, 2c, and 2d, we see that our method MaxLogHash gives comparable results to MinHash and HyperMinHash with $r = 4$. Specially, the RMSEs of these three methods differ within 0.006 and continually decrease as the similarity increases. The RMSE of HyperMinHash with $r = 1$ significantly increases as $J_{A,B}$ increases. We observe that the large estimation error occurs because HyperMinHash exhibits a large estimation bias. Figs. 2e, 2f, 2g, and 2h show the bias of our method MaxLogHash in comparison with MinHash and HyperMinHash. We see that the empirical biases of MaxLogHash and MinHash are both very small and no systematic biases can be observed. However, HyperMinHash with $r = 1$ shows a significant bias and its bias increases as the similarity value increases. To be more specific, its bias raises from -0.06 to -0.089 when the similarity increases from 0.80 to 0.99. One can increase r to reduce the bias of HyperMinHash. However, HyperMinHash with large r desires more memory space. For example, HyperMinHash with $r = 4$ has comparable accuracy but requires 1.5 times more memory space compared with our method MaxLogHash. Compared with MinHash, MaxLogHash gives a 5.4 times reduction in memory usage while achieves a similar estimation accuracy. Later in Section 6.5, we show that our method MaxLogHash has a computational cost similar to Minhash, but is several orders of magnitude faster than HyperMinHash when estimating set similarities.

MaxLogHash versus b-bit MinHash. Next, we compare our method MaxLogHash with b-bit MinHash in streaming setting. In this experiment, we update b-bit MinHash incrementally, i.e., approximating $\min^{(b)}(\pi(A))$ by using $\min_{w\in A}\pi^{(b)}(w)$. As shown in Fig. 4, we see that b-bit MinHash fails to give an accurate estimate regardless of $b = 4$ or $b = 8$. We also find


(a) $m = 2^9$, $J_{A,B} = 0.8$


(b) $m = 2^9$, $J_{A,B} = 0.9$


(c) $m = 2^{10}$, $J_{A,B} = 0.8$

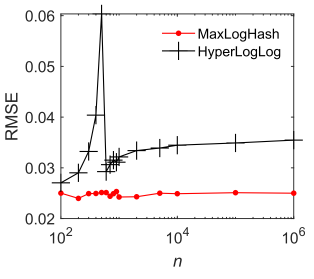

(d) $m = 2^{10}$, $J_{A,B} = 0.9$

Fig. 3. Estimation error of our method MaxLogHash in comparison with HyperLogLog on synthetic set-pairs A and B with the same memory space m bits, where $|A| = |B| = n$.

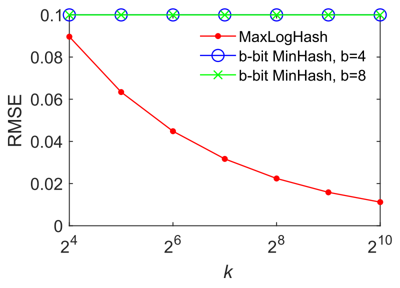
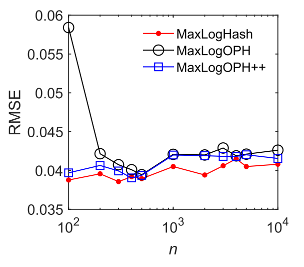


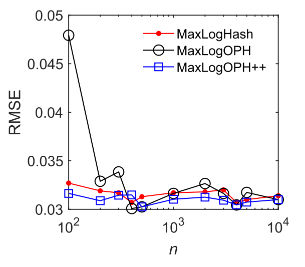
Fig. 4. Estimation error of our method MaxLogHash in comparison with b-bit MinHash on synthetic set-pairs $|A| = |B| = 10,000$ and $J_{A,B} = 0.9$ with the same number of registers.

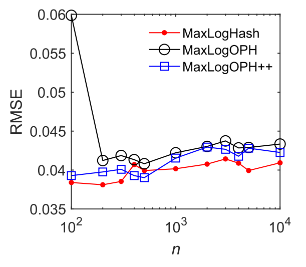
that b-bit MinHash always estimates the Jaccard similarity as 1 because the value in each register equals 0 with a high probability when $|A|, |B| \gg 2^b$. We omit the similar result on unbalanced set-pairs.

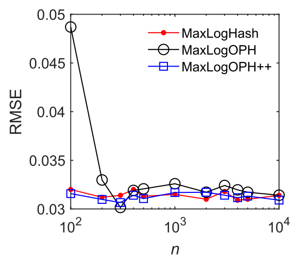
MaxLogHash versus HyperLogLog. To make a fair comparison, we allocate the same amount of memory space, m bits, to each of MaxLogHash and HyperLogLog. As discussed in Section 4, the attractive property of our method MaxLogHash is that its estimation error is almost independent with the cardinality of sets A and B, which does not hold for HyperLogLog. Fig. 3 shows the RMSEs of MaxLogHash and HyperLogLog on sets of different sizes. We see that the RMSE of our method MaxLogHash is almost a constant. Figs. 3a and 3b show the performance of HyperLogLog suddenly degrades when $m = 2^9$ and the cardinalities of A and B are around 200, because HyperLogLog uses two different estimators for cardinalities within two different ranges respectively [19]. As a result, our method MaxLogHash decreases the RMSE of HyperLogLog by up to 36 percent. As shown in Figs. 3c and 3d, similarly, the RMSE of our method MaxLogHash is about 2.5 times smaller than HyperLogLog when $m = 2^{10}$ and the cardinalities of A and B are around 500.

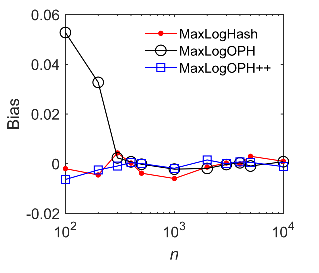
MaxLogHash versus MaxLogOPH++. We compare MaxLogOPH++ with MaxLogHash and MaxLogOPH [14] on sets with increasing cardinalities. In this experiment, we conduct experiments with the same number of buckets k. As we mentioned, compared with MaxLoHash and MaxLogOPH, MaxLogOPH++ uses an additional 32-bit register d_u to record the cardinality of set I_u, which results in a slightly increase of memory usage (e.g., 4 percent when setting $k = 128$). We set the maximal iteration $l_T = 5$ for MaxLogOPH++. As shown in Figs. 5a, 5b, 5c, and 5d, MaxLogOPH exhibits relatively large estimation errors for small cardinalities. When $k = 128$ and the cardinality increases to 200 (about $2k$), we see that MaxLogOPH achieves similar accuracy to MaxLogHash. MaxLogOPH++ overcomes the shortcomings of MaxLogOPH. We can see that MaxLogOPH++

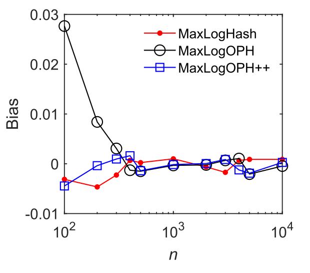

(a) (balanced) RMSE, $J_{A,B} = 0.8$

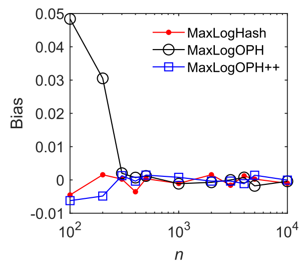

(b) (balanced) RMSE, $J_{A,B} = 0.9$


(c) (unbalanced) RMSE, $J_{A,B} = 0.8$


(d) (unbalanced) RMSE, $J_{A,B} = 0.9$


(e) (balanced) Bias, $J_{A,B} = 0.8$


(f) (balanced) Bias, $J_{A,B} = 0.9$


(g) (unbalanced) Bias, $J_{A,B} = 0.8$

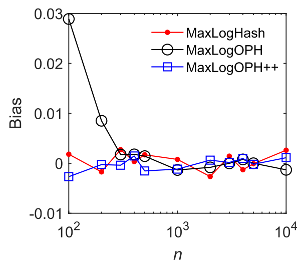
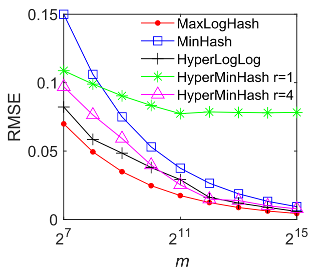
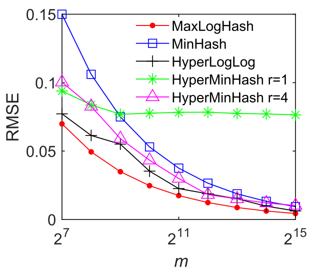

(h) (unbalanced) Bias, $J_{A,B} = 0.9$

Fig. 5. Estimation error of our methods MaxLogHash, MaxLogOPH, and MaxLogOPH++ on both balanced and unbalanced synthetic data pairs A and B with the same number of registers, $k = 128$.



(a) $n = 10^4$



(b) $n = 10^7$

Fig. 6. Estimation error of our method MaxLogHash in comparison with other baselines on synthetic set-pairs A and B with the same memory space m bits, where $|A| = |B| = n$ and $J_{A,B} = 0.9$.

achieves similar estimation accuracy as MaxLogHash even for n smaller than k. In Fig. 5e, 5f, 5g, and 5h, we also show the bias of MaxLogOPH++, MaxLogOPH, and MaxLogHash. We find that the estimation error of MaxLogOPH is mainly caused by the bias, and our new method MaxLogOPH++ significantly reduces the bias. Later in Section 6.5, MaxLogOPH++ significantly accelerates the speed of updating elements compared with MaxLogHash.

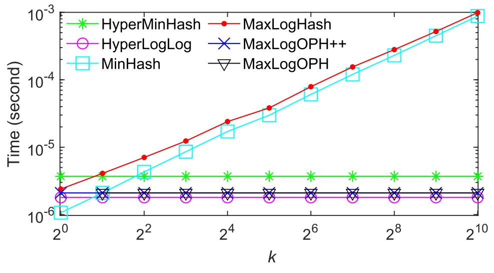
Accuracy versus Memory Usage. In addition, we compare all the competitors under different memory usage. Specially, we allocate m bits of memory for each algorithm and compute these algorithms' estimation errors. In Fig. 6, we set $J_{A,B} = 0.9$ and vary m from 2^7 to 2^{15} bits. We see that our method outperforms other competitors when using the same memory. In particular, MaxLogHash is up to 54, 42, and 56 percent more accurate than MinHash, HyperLogLog, and HyperMinHash respectively.

6.5 Efficiency

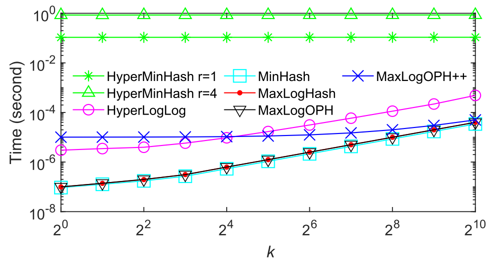
We further evaluate the efficiency of our method MaxLogHash, MaxLogOPH, and MaxLogOPH++ in comparison with MinHash and HyperLogLog. Specially, we present the time for updating each coming element and computing Jaccard similarity, respectively. We conduct experiments on synthetic balanced datasets. We omit similar results for real-world datasets and synthetic unbalanced datasets. Fig. 7a shows that the update time of MaxLogOPH, MaxLogOPH++, and HyperLogLog is almost a constant and our method outperforms other baselines. The update time of HyperMinHash is almost irrelevant to its parameter r and thus we only plot the curve for $r = 1$. Specially, MaxLogOPH and MaxLogOPH++ share the same update time and are about 2 and 420 times faster than HyperMinHash and MinHash. Fig. 7b shows that our methods MaxLogHash and MaxLogOPH++ have estimation time similar to MinHash, while they are about 6 times faster than HyperLogLog and 4 to 5 orders of magnitude faster than HyperMinHash. We also notice that the time required for each Newton-Raphson iteration of MaxLogOPH++ is almost a constant because it is irrelevant to the number of buckets. When k increases to 2^{10}, MaxLogOPH++ is slightly slower than MaxLogHash and MaxLogOPH.

6.6 Accuracy of Association Rule Learning

In this experiment, we evaluate the performance of our method compared with other baselines on the detection of



(a) update time



(b) estimation time

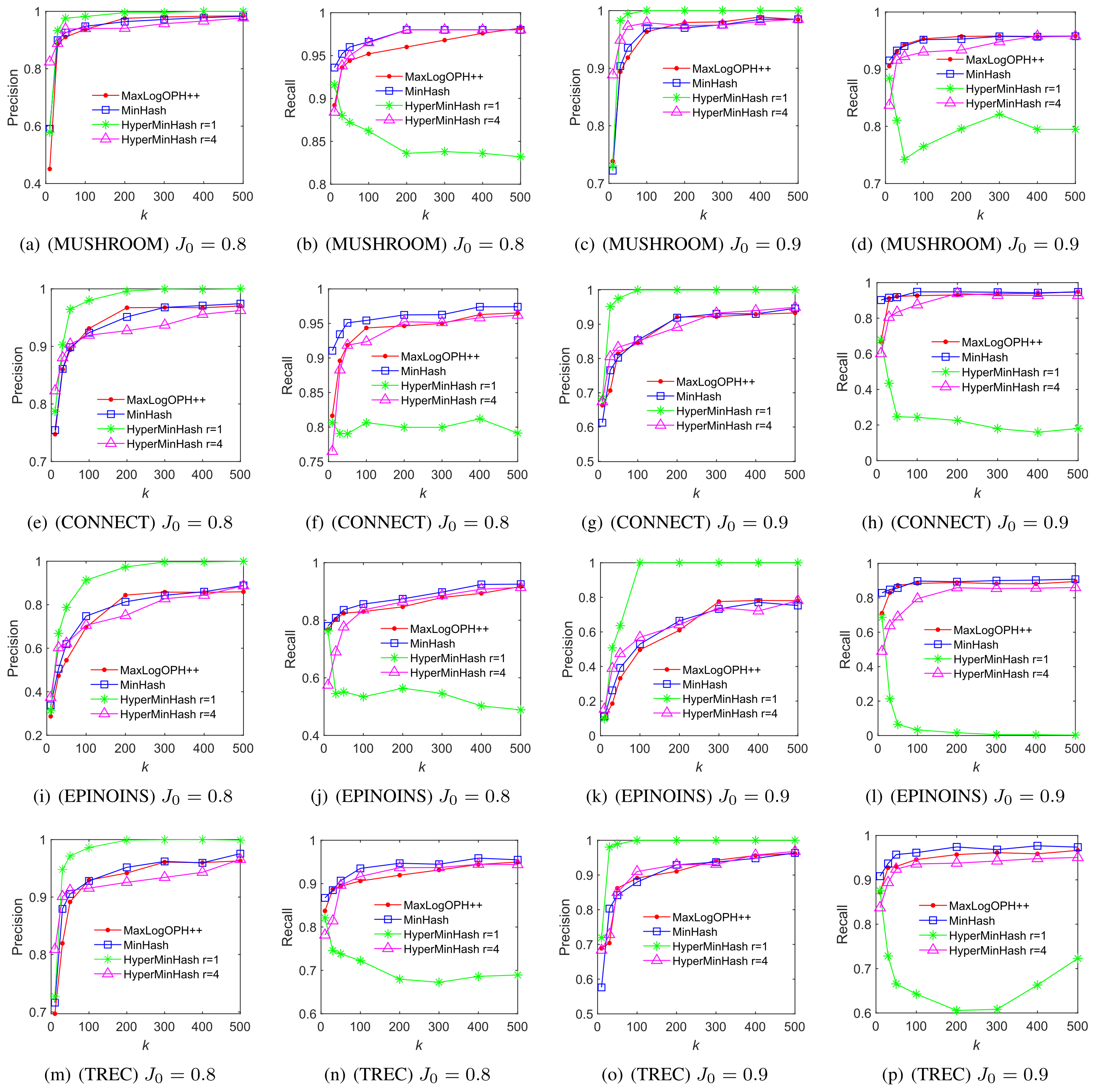
Fig. 7. Computational cost of our methods MaxLogHash, MaxLogOPH, and MaxLogOPH++ in comparison with MinHash, HyperLogLog, and HyperMinHash.

items (e.g., products) that almost always appear together in the same records (e.g., transactions). We conduct the experiments on four real-world datasets listed in Table 1. We first estimate all pairwise similarities among items' record-sets, and retrieve every pair of record-sets with similarity $J > J_0$. As discussed previously (results in Fig. 3), HyperLogLog is not robust, because it exhibits large estimation errors for sets of particular sizes. We also find that our method MaxLogOPH++ and MaxLogHash give similar results. Therefore, in what follows we compare our method MaxLogOPH++ only with MinHash and HyperMinHash. As shown in Fig. 8, MaxLogOPH++ gives comparable precision and recall to MinHash and HyperMinHash with $r = 4$. We see that MaxLogOPH++ gives similar performance in comparison with MinHash and HyperMinHash.

Table 2 further shows the amount of memory required for our methods MaxLogHash, MaxLogOPH, and MaxLogOPH++ in comparison with MinHash and HyperMinHash ($r = 4$) with the same precision and recall scores, where dataset TREC is used. Specially, for specific precision and recall scores, we report the memory usage for all the algorithms. Our methods MaxLogHash and MaxLogOPH use the same amount of memory usage due to their same sketch structure. MaxLogOPH++ requires a slightly more memory space than MaxLogHash and MaxLogOPH, because we additionally use a 32-bit register to record the number of presented items for each set. Compared with MinHash and HyperMinHash, our methods give up to 5.9 and 4.6 times reduction in memory usage respectively.

7 Related Work

Jaccard Similarity Estimation for Static Sets. Broder *et al.* [7] proposed the first sketch method MinHash to compute the Jaccard similarity of sets, which builds a sketch consisting



(a) (MUSHROOM) $J_0 = 0.8$ (b) (MUSHROOM) $J_0 = 0.8$ (c) (MUSHROOM) $J_0 = 0.9$ (d) (MUSHROOM) $J_0 = 0.9$

(e) (CONNECT) $J_0 = 0.8$ (f) (CONNECT) $J_0 = 0.8$ (g) (CONNECT) $J_0 = 0.9$ (h) (CONNECT) $J_0 = 0.9$

(i) (EPINOINS) $J_0 = 0.8$ (j) (EPINOINS) $J_0 = 0.8$ (k) (EPINOINS) $J_0 = 0.9$ (l) (EPINOINS) $J_0 = 0.9$

(m) (TREC) $J_0 = 0.8$ (n) (TREC) $J_0 = 0.8$ (o) (TREC) $J_0 = 0.9$ (p) (TREC) $J_0 = 0.9$

Fig. 8. Precision and recall of our method MaxLogOPH++ in comparison with MinHash and HyperMinHash on datasets listed in Table 1.

of k registers for each set. To reduce the amount of memory space required for MinHash, [10], [11] developed methods b-bit MinHash and Odd Sketch, which are dozens of times more memory efficient than the original MinHash. The basic idea behind b-bit MinHash and Odd Sketch is to use probabilistic methods such as sampling and bitmap sketching to build a compact digest for each set's MinHash sketch. Recently, several methods [18], [23], [24], [25] were proposed to reduce the time complexity of processing each element in a set from $O(k)$ to $O(1)$.

Weighted Similarity Estimation for Static Vectors. There are also some works for estimating similarity between real-value weighted vectors. SimHash (or, sign normal random projections) [30] was developed for approximating angle

TABLE 2
Memory Usage (in MB) of All Algortihms on Dataset TREC When Achieving the Same Precision and Recall Scores

		MinHash	HyperMinHash	MaxLogHash	MaxLogOPH	MaxLogOPH++
$J_0 = 0.8$	Precision=0.95	1383.2	836.4	324.1	324.1	337.2
	Recall=0.95	2213.1	778.0	518.7	518.7	539.4
$J_0 = 0.9$	Precision=0.95	2766.4	719.7	466.8	466.8	485.5
	Recall=0.95	290.5	778.0	168.6	168.6	175.3

similarity (i.e., cosine similarity) of weighted vectors. CWS [31], [32], ICWS [33], 0-bit CWS [34], CCWS [35], Weighted MinHash [36], PCWS [37], and BagMinHash [38] were developed for approximating generalized Jaccard similarity of weighted vectors,[5] and Datar *et al.* [39] developed an LSH method using p-stable distribution for estimating l_p distance for weighted vectors, where $0 < p \leq 2$. Campagna and Pagh [40] developed a biased sampling method for estimating a variety of set similarity measures beyond Jaccard similarity.

Similarity Estimation for Data Streams. The above similarity estimation methods fail to deal with streaming vectors, whereas elements in vectors come in a stream fashion. To solve this problem, Kutzkov *et al.* [41] extended AMS sketch [42] for the estimation of cosine similarity and Pearson correlation in streaming weighted vectors. Yang *et al.* [43] developed a streaming method HistoSketch for approximating Jaccard similarity with concept drift. Set intersection cardinality (i.e., the number of common elements in two sets) is also a popular metric for evaluating the similarity in sets. A variety of sketch methods such as LPC [44], FM [13], LogLog [45], HyperLogLog [19], HLL-TailCut+ [46], and MinCount [47] were proposed to estimate the stream cardinality (i.e., the number of distinct elements in the stream), and can be easily extended to estimate $|A \cup B|$ by merging the sketches of sets A and B. Then, one can approximate $|A \cap B|$ because $|A \cap B| = |A| + |B| - |A \cup B|$. To further improve the estimation accuracy, Cohen *et al.* [48] developed a method combining MinHash and HyperLogLog to estimate set intersection cardinalities. Our experiments reveal that these sketch methods have large errors when first estimating $|A \cap B|$ and $|A \cup B|$, and then approximating the Jaccard similarity $J_{A,B}$. As mentioned in Section 3, MinHash can be easily extended to handle streaming sets, but its two compressed versions, b-bit MinHash and Odd Sketch fail to handle data streams. To solve this problem, Peng [49] proposed to build a shared sketch based on Odd Sketches [11] to handle fully dynamic stream and gave a biased estimator. Yu and Weber [12] developed a method, HyperMinHash, which can be viewed as a joint of HyperLogLog and b-bit MinHash. HyperMinHash consists of k registers, whereas each register has two parts, an FM sketch and a b-bit string. The b-bit string is computed based on the fingerprints (i.e., hash values) of set elements that map to the register. HyperMinhash first estimates $|A \cup B|$ and then infers the Jaccard similarity of sets A and B from the number of collisions of b-bit strings given $|A \cup B|$. Our experiments demonstrate that HyperMinHash exhibits a large bias for high similarities and it is several orders of magnitude slower than our methods when estimating the similarity.

Fast Similarity Search. Locality Sensitive Hash (LSH) is an effective technique for fast similarity search in high dimensional spaces [30], [39], [50], [51]. The basic idea behind LSH is to construct a family of hash functions such that sets with high similarities are more likely to be hashed to the same bucket of hash tables. Clearly, many of the above sketching methods [7], [10], [23], [24], [25], [31], [32] belongs to the LSH family. In addition to these sketching methods, Satuluri *et al.* [52] proposed a Bayesian approach to fast prune set pairs of which similarity is smaller than the user-specific threshold with a high probability. Zhai *et al.* [53] developed a filtering-based algorithm for generating set pairs of which similarities are likely to be greater than the given threshold. Gao *et al.* [54] gave a learning-based method to improve the effectiveness of LSH. Recently, Zhang *et al.* [55] proposed to transform sets into representative vectors and then used R-Tree to index all these vectors to support fast similarity search.

5. The Jaccard similarity between two positive real value vectors $\vec{x} = (x_1, x_2, \ldots, x_p)$ and $\vec{y} = (y_1, y_2, \ldots, y_p)$ is defined as $J(\vec{x}, \vec{y}) = \frac{\sum_{1 \leq j \leq p} \min(x_j, y_j)}{\sum_{1 \leq j \leq p} \max(x_j, y_j)}$.

8 Conclusions and Future Work

We previously developed a memory-efficient method, MaxLogHash, to estimate Jaccard similarities in streaming sets [14]. Compared with MinHash, MaxLogHash uses smaller sized registers to build a compact sketch for each set. In this paper, we further develop a memory-efficient yet fast method, MaxLogOPH++. Compared with MaxLogHash, MaxLogOPH++ further reduces the time complexity of updating each coming element from $O(k)$ to $O(1)$ with a small additional memory. We conduct experiments on both real-world and synthetic datasets. Experimental results demonstrate that our methods can reduce around $3 \sim 6$ times the amount of memory required for other baselines with the same desired accuracy and computational cost. In addition, our methods can be orders of magnitudes faster than other baselines on both update and estimation time. In the future, we plan to extend our methods to weighted streaming vectors and fully dynamic streaming sets that include both set element insertions and deletions.

Acknowledgments

The research presented in this paper was supported in part by National Key R&D Program of China (2018YFC0830500), Shenzhen Basic Research Grant (JCYJ20170816100819428), National Natural Science Foundation of China (61922067, U1736205, 61902305), MoE-CMCC "Artifical Intelligence" Project (MCM20190701), Natural Science Basic Research Plan in Shaanxi Province of China (2019JM-159), Natural Science Basic Research Plan in ZheJiang Province of China (LGG18F020016). The work of John C.S. Lui was supported in part by the GRF R4032-18.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, 2002, pp. 1–16.

[2] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 219–228.

[3] K. Yang, X. Ding, Y. Zhang, L. Chen, B. Zheng, and Y. Gao, "Distributed similarity queries in metric spaces," *Data Sci. Eng.*, vol. 4, no. 2, pp. 93–108, 2019.

[4] P. Li, A. Shrivastava, J. L. Moore, and A. C. König, "Hashing algorithms for large-scale learning," in *Proc. 24th Int. Conf. Neural Inf. Process. Syst.*, 2011, pp. 2672–2680.

[5] A. Broder, "On the resemblance and containment of documents," in *Proc. Compression Complexity Sequences*, 1997, pp. 21–29.

[6] Y. Yang, W. Zhang, Y. Zhang, X. Lin, and L. Wang, "Selectivity estimation on set containment search," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2019, pp. 330–349.

[7] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, no. 3, pp. 630–659, Jun. 2000.

[8] S. Gollapudi and A. Sharma, "An axiomatic approach for result diversification," in *Proc. 18th Int. Conf. World Wide Web*, 2009, pp. 381–390.

[9] T. Urvoy, E. Chauveau, P. Filoche, and T. Lavergne, "Tracking web spam with HTML style similarities," *ACM Trans. Web*, vol. 2, no. 1, Mar. 2008, Art. no. 3.

[10] P. Li and A. C. König, "b-bit minwise hashing," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 671–680.

[11] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 109–118.

[12] Y. W. Yu and G. Weber, "HyperMinHash: Jaccard index sketching in LogLog space," 2017, *arXiv:1710.08436*.

[13] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, Oct. 1985.

[14] P. Wang *et al.*, "A memory-efficient sketch method for estimating high similarities in streaming sets," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2019, pp. 25–33.

[15] J. S. Cramer, *Econometric Applications of Maximum Likelihood Methods*. Cambridge, U.K.: Cambridge Univ. Press, 1986.

[16] T. Dence, "Cubics, chaos and newton's method," *Math. Gazette*, vol. 81, no. 492, pp. 403–408, 1997.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[18] P. Li, A. B. Owen, and C. Zhang, "One permutation hashing," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 3122–3130.

[19] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Int. Conf. Anal. Algorithms*, 2007, pp. 127–146.

[20] A. Kheyfits, *A Primer in Combinatorics*. Berlin, Germany: Walter de Gruyter, 2010.

[21] P. Jacquet and W. Szpankowski, "Analytical depoissonization and its applications," *Theor. Comput. Sci.*, vol. 201, no. 1/2, pp. 1–62, 1998.

[22] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. New York, NY, USA: Cambridge Univ. Press, 2005.

[23] A. Shrivastava and P. Li, "Improved densification of one permutation hashing," in *Proc. 30th Conf. Uncertainty Artif. Intell.*, 2014, pp. 732–741.

[24] A. Shrivastava and P. Li, "Densifying one permutation hashing via rotation for fast near neighbor search," in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, pp. 557–565.

[25] A. Shrivastava, "Optimal densification for fast and accurate minwise hashing," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 3154–3163.

[26] T. Mai, A. Rao, M. Kapilevich, R. Rossi, Y. Abbasi-Yadkori, and R. Sinha, "On densification for minwise hashing," in *Proc. Conf. Uncertainty Artif. Intell.*, 2019, p. 302.

[27] E. Cohen *et al.*, "Finding interesting associations without support pruning," *IEEE Trans. Knowl. Data Eng.*, vol. 13, no. 1, pp. 64–78, Jan. 2001.

[28] National Institute of Standards and Technology, "Text REtrieval Conference (TREC) English documents," Aug. 2010. [Online]. Available: http://trec.nist.gov/data/docs_eng.html

[29] P. Massa and P. Avesani, "Controversial users demand local trust metrics: An experimental study on epinions.com community," in *Proc. 20th Nat. Conf. Artif. Intell.*, 2005, pp. 121–126.

[30] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proc. 34th Annu. ACM Symp. Theory Comput.*, 2002, pp. 380–388.

[31] M. Manasse, F. McSherry, and K. Talwar, "Consistent weighted sampling," Microsoft, Redmond, WA, Tech. Rep. MSR-TR-2010–73, Jun. 2010.

[32] B. Haeupler, M. Manasse, and K. Talwar, "Consistent weighted sampling made fast, small, and easy," 2014, *arXiv:1410.4266*.

[33] S. Ioffe, "Improved consistent sampling, weighted minhash and L1 sketching," in *Proc. IEEE Int. Conf. Data Mining*, 2010, pp. 246–255.

[34] P. Li, "0-bit consistent weighted sampling," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 665–674.

[35] W. Wu, B. Li, L. Chen, and C. Zhang, "Canonical consistent weighted sampling for real-value weighted min-hash," in *Proc. IEEE 16th Int. Conf. Data Mining*, 2016, pp. 1287–1292.

[36] A. Shrivastava, "Simple and efficient weighted minwise hashing," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 1498–1506.

[37] W. Wu, B. Li, L. Chen, and C. Zhang, "Consistent weighted sampling made more practical," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 1035–1043.

[38] O. Ertl, "BagMinHash - Minwise hashing algorithm for weighted sets," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2018, pp. 1368–1377.

[39] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. 20th Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.

[40] A. Campagna and R. Pagh, "Finding associations and computing similarity via biased pair sampling," *Knowl. Inf. Syst.*, vol. 31, no. 3, pp. 505–526, 2012.

[41] K. Kutzkov, M. Ahmed, and S. Nikitaki, "Weighted similarity estimation in data streams," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage.*, 2015, pp. 1051–1060.

[42] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proc. 28th Annu. ACM Symp. Theory Comput.*, 1996, pp. 20–29.

[43] D. Yang, B. Li, L. Rettig, and P. Cudr é-Mauroux, "HistoSketch: Fast similarity-preserving sketching of streaming histograms with concept drift," in *Proc. IEEE Int. Conf. Data Mining*, 2017, pp. 545–554.

[44] K. Whang, B. T. Vander-zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, Jun. 1990.

[45] M. Durand and P. Flajolet, *Loglog Counting of Large Cardinalities*. Berlin, Germany: Springer, 2003, pp. 605–617.

[46] Q. Xiao, Y. Zhou, and S. Chen, "Better with fewer bits: Improving the performance of cardinality estimation of large data streams," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[47] F. Giroire, "Order statistics and estimating cardinalities of massive data sets," *Discrete Appl. Math.*, vol. 157, no. 2, pp. 406–427, 2009.

[48] R. Cohen, L. Katzir, and A. Yehezkel, "A minimal variance estimator for the cardinality of big data set intersection," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2017, pp. 95–103.

[49] P. Jia, P. Wang, J. Tao, and X. Guan, "A fast sketch method for mining user similarities over fully dynamic graph streams," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 1682–1685.

[50] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. 30th Annu. ACM Symp. Theory Comput.*, 1998, pp. 604–613.

[51] A. Gionis *et al.*, "Similarity search in high dimensions via hashing," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, vol. 99, pp. 518–529.

[52] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *Proc. VLDB Endowment*, vol. 5, no. 5, pp. 430–441, 2012.

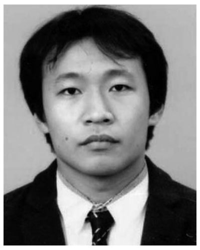
[53] J. Zhai, Y. Lou, and J. Gehrke, "ATLAS: A probabilistic algorithm for high dimensional similarity search," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 997–1008.

[54] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi, "DSH: Data sensitive hashing for high-dimensional k-nnsearch," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1127–1138.

[55] Y. Zhang, J. Wu, J. Wang, and C. Xing, "A transformation-based framework for KNN set similarity search," *IEEE Trans. Knowl. Data Eng.*, to be published, doi: 10.1109/TKDE.2018.2886189.

Yiyan Qi received the BS degree in automation engineering from Xi'an Jiaotong University, Xi'an, China, in 2014. He is currently working toward the graduate degree in NSKeyLab, Xi'an Jiaotong University, Xi'an, China. His research interests include Internet traffic measurement and modeling, abnormal detection, and online social network measurement.

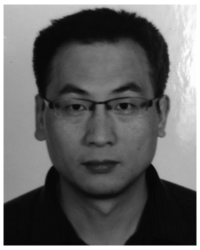
Pinghui Wang received the BS degree in information engineering and the PhD degree in automatic control from Xi'an Jiaotong University, Xi'an, China, in 2006 and 2012, respectively. He is currently a professor in MOE Key Laboratory for intelligent networks and network security, Xi'an Jiaotong University, Xi'an, China. His research interests include Internet traffic measurement and modeling, abnormal detection, and online social network measurement.



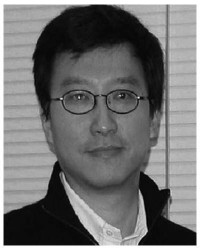
Yuanming Zhang received the BS degree in the automation from Chongqing University, Chongqing, China, in 2017. He is currently working toward the graduate degree in NSKeyLab, Xi'an Jiaotong University, Xi'an, China. His research interests include anomaly detection, encrypted traffic analysis, and Internet traffic measurement and modeling.

Qiaozhu Zhai received the BS and MS degrees in applied mathematics and the PhD degree in systems engineering from Xi'an Jiaotong University, Xi'an, China, in 1993, 1996, and 2005, respectively. He is currently a professor with the Systems Engineering Institute, Xi'an Jiaotong University, Xi'an, China. His research interests include optimization of large-scale systems and integrated resource bidding and scheduling in the deregulated electric power market.

Chenxu Wang received the BS degree in communication engineering and the PhD degree in control science and engineering from Xi'an Jiaotong University, Xi'an, China, in 2009 and 2015, respectively. He was a postdoctoral research fellow with The Hong Kong Polytechnic University, Hong Kong. He is currently an assistant professor with the School of Software Engineering, Xi'an Jiaotong University, Xi'an, China. His current research interests include data mining, network security, online social network analysis, and information diffusion.



Guangjian Tian received the PhD degree in computer science and technology from Northwestern Polytechnical University, Xi'an, China, in 2006. He is currently a principle researcher in Huawei Noah's Ark Lab. Before that, he was a postdoctoral research fellow with the Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, Hong Kong. His research interests include temporal data analysis, deep learning, and data mining with the specific focus on different industry applications.



John C.S. Lui received the PhD degree in computer science from University of California, Los Angeles, Los Angeles, California. He is currently a professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong. His current research interests include communication networks, network system security, network economics, network sciences, cloud computing, large-scale distributed systems, and performance evaluation theory.

Xiaohong Guan received the BS and MS degrees in automatic control from Tsinghua University, Beijing, China, in 1982 and 1985, respectively, and the PhD degree in electrical engineering from the University of Connecticut, Storrs, Connecticut, in 1993. He is currently a professor with the Systems Engineering Institute, Xi'an Jiaotong University, Xi'an, China. His research interests include allocation and scheduling of complex networked resources, network security, and sensor networks.