Fast Retrieval of Large Entries With Incomplete Measurement Data

Kun Xie, *Member, IEEE*, Jiazheng Tian, Xin Wang, *Senior Member, IEEE*, Gaogang Xie, *Senior Member, IEEE*, Jiannong Cao, *Fellow, IEEE*, Hongbo Jiang, *Senior Member, IEEE*, and Jigang Wen

***Abstract*—In network-wide monitoring, finding the large monitoring data entries is a fundamental network management function. However, the retrieval of large entries is extremely difficult and challenging as a result of incompleteness of network measurement data. Enlightened by tensor model's strong capability of information representation and extraction, we model the network-wide monitoring data as a 3-way tensor. With tensor completion, the retrieval can be performed after recovering all missing entries. However, this not only incurs an extremely high cost when the tensor is large, but is also unnecessary. Instead, to quickly retrieve large entries at low cost, we transform the large entry retrieving problem to a cosine similarity searching problem, and propose two algorithms: 1) Quickly reordering the factor vectors based on Locality Sensitive Hashing (LSH) hash table so that vectors with small cosine distances are placed in the same hash bucket; 2) Quickly finding the similar vector of a queried one that the two together determine a large entry without incurring the high cost of recovering all entries through the dot products. In the process of LSH table building and similarity query, several novel techniques are proposed, including LSH table representation with the LSH forest, good hash table building to support the flexible search of cosine similarity, and bit-shifting-based quick similarity query. Our experimental studies on 4 real world datasets indicate that our technique is at least up to 60 times faster than the approach based on direct tensor completion.**

***Index Terms*—Large entry inference, tensor completion.**

I. Introduction

A. Background

NETWORK-WIDE monitoring is important for many network functions. Among which, finding the large monitoring data entries is a fundamental network management function. Depending on the type of KPI (Key Performance Indicator) recorded by the monitoring data, the large monitoring data entries may correspond to elephant flows, the large network latency paths, and the large packet loss paths in the network. Many network applications can benefit from the efficient identification of the large entries, including congestion control [1], network capacity planning [2], anomaly detection [3], network SLA (Service Level Agreement) tracking [4].

If we have complete network-wide monitoring data, such a task is trivial and can be solved by simply sorting the measurement data. Different types of KPI data between an origin and destination pair can be measured through different tools. For example, the traffic volume data can be measured by Netflow, and the end-to-end latency can be measured by ping probes. However, in practice, no matter what type the KPI data belong to, the network-wide monitoring data are incomplete due to the following reasons. 1) Network-wide monitoring among all the origin and destination pairs introduce high measurement overhead. Network systems usually adopt sampling-based measurement to reduce the measurement overhead; 2) The unavoidable data transmission losses under severe communication and system conditions, including network congestion, node misbehavior, monitor failure, a transmission of measurement information through an unreliable transport protocol. **The retrieval of large entries is extremely difficult and challenging as a result of the incompleteness of network measurement data**.

With incomplete Measurement Data, one possible approach to achieve the goal is to first recover the missing data, then return the large entries after sorting the recovered data. Various studies have been made to handle and recover the missing traffic data. Designed based on purely spatial or purely temporal information, the data recovery performance of most known approaches [5]–[7] is low. Recently matrix-completion-based algorithms are proposed to recover the missing traffic data by exploiting both spatial and temporal information [8]–[10]. Although the performance is good when the data missing ratio is low, the performance suffers when the missing ratio is large.

For more accurate missing data inference, a few recent studies [11]–[14] try to model the traffic data as a 3-way tensor, and then fill in the missing data of traffic matrices through tensor completion. Tensors, as the higher-order generalization of vectors and matrices, can take full advantage of the multilinear structures to provide better data understanding

Manuscript received 25 April 2020; revised 12 October 2021; accepted 14 February 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor N. Hegde. Date of publication 6 April 2022; date of current version 17 October 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61972144 and Grant 62025201; in part by NSF Electrical, Communications and Cyber Systems (ECCS) under Grant 2030063; and in part by NSF Communication and Information Foundations (CIF) under Grant 2007313. *(Corresponding author: Jiazheng Tian.)*

Kun Xie, Jiazheng Tian, Hongbo Jiang, and Jigang Wen are with the College of Computer Science and Electronics Engineering, Hunan University, Changsha 410082, China (e-mail: xiekun@hnu.edu.cn).

Xin Wang is with the Department of Electrical and Computer Engineering, The State University of New York at Stony Brook, Stony Brook, NY 11794 USA.

Gaogang Xie is with the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China, and also with the School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100190, China.

Jiannong Cao is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong.

Digital Object Identifier 10.1109/TNET.2022.3160233

and information precision. Compared with matrix-based data recovery, the tensor-based approach can better handle the missing monitoring data and will be used in this paper.

B. Problems

Although tensor completion is a promising technique for accurate recovery of missing data, using it to retrieve large entries with incomplete measurement data may suffer from high computation cost. With incomplete measurement data, one possible approach of finding the large entries is to first infer data entries not measured/missing, then return the large ones. Under tensor completion, two operations need to execute: 1)Training Phase: training the factor matrices through tensor factorization using the partial measurement samples, 2) Retrieving Phase: recovering un-measurement/missing data using the trained factor matrices, and retrieving all large entries whose values are above a predefined threshold ε. Although this is a promising way, given an $I \times J \times K$ network monitoring tensor with its rank equal to R, the time complexity of data recovering in the Retrieving Phase is $O(IJKR)$. The need of recovering all un-measurement entries before retrieving the large ones is a big bottleneck that prevents the efficient finding of large entries, especially when the size of the monitoring tensor is large.

Existing efforts [15]–[20] on tensor completion generally focus on the techniques of training phase. Only the recent work [21] study the fast retrieving of top-k entries in tensors with incomplete measurements. It proposes the use of discrete tensor completion, where real-valued factor matrices in the traditional tensor completion are represented by binary codes. As a result, large entry retrieving problem is transformed into the calculation of Hamming distance through lightweight XOR bit operations. Although the transformation helps quickly find the top-k entry locations, it compromises the accuracy of recovering the un-measurement data values. However, it is very important to know the data values in many applications. For example, knowing large traffic volume values of a network is the basis for designing a proper congestion control algorithm.

The goal of this work is to design an efficient algorithm that can quickly retrieve large entries whose values are larger than a threshold ε. To the best of our knowledge, we are not aware that any prior studies on tensor completion have been performed to retrieve both large entry locations and values. Addressing the challenge may open a new avenue for the tensor-based advanced data processing.

C. Our Contributions

Based on the analysis of relationship between tensor's CANDECOMP/ PARAFAC (CP) [22] decomposition and tensor's frontal slices, we transform the representation of tensor entry with the dot product of three factor vectors to the dot product of two vectors. With the observation that both the length and the direction of two vectors influence the values of the dot production, we transform the large entry retrieving problem to a cosine similarity searching problem. Thus retrieving a large entry is equivalent to the finding of a vector pair whose cosine distance is larger than a threshold. We propose an Fast Large Tensor Entry Retrieving framework based on the Locality Hash Function (FLTER-LSH), which includes the following two main processes.

- **Hash table building** We propose an algorithm based on LSH hash table to quickly store and reorder the vector set so that the vectors with small cosine distances are placed in the same hash bucket.
- **Similarity querying** Facilitated by the LSH hash tables, we propose a query algorithm to quickly find the similar vector of a queried one that the two together can determine a large entry without incurring the high cost of recovering all entries through the dot products.

The challenge and our technique contributions in above two main processes are listed as follows.

(1) Hash table building: The difficulties in building LSH hash tables to facilitate the cosine similarity searching lies in two points: (a) Different vector pairs with different vector lengths have different angular level requirements thus different cosine similarity requirements. (b) Current cosine locality-sensitive hashing (Cosine LSH) function can only output single bit hash index with two values (0 or 1), which cannot support different angular levels of requirement. To conquer the challenges, we propose two techniques:

- **LSH forest to represent LSH table.** To support different angular level requirements, instead of using a single hash function, we use multiple hash functions in Cosine LSH to construct hash tables. More specifically, we build a novel LSH forest to represent hash tables, where each leaf in the forest corresponds to one bucket in a hash table.
- **Good hash table building for flexible search of cosine similarity.** To achieve accurate similarity query for a given vector, we need a good LSH hash tables (LSH forest) that meet two conditions: 1) All vectors held in the same hash bucket as the one queried have a high cosine similarity with it, and 2) No vectors with high cosine similarity are missing from the bucket. From the theoretical analysis, we find the relationship between the angular requirement and the parameter setting in LSH forest to build good hash tables. To reduce the computation cost, we partition the vectors into multiple groups and build multiple groups of hash tables with different sizes of forest to store different vector groups. Each group has similar vector lengths thus similar angular requirements for vector query.

(2) Similarity query: Although building multiple forests can reduce computation cost in hash table building, it may introduce a high cost to perform the similar query, as a query vector needs to be matched against multiple groups of hash tables with each group corresponding to a forest. To address the issue, we propose the following strategy.

- **Bit-shifting-based quick similarity query.** Taking advantage of the features of our proposed common hash group shared by multiple forests, we develop a light-weight query algorithm. When querying against M groups of hash tables, only the hash indexes for one group with the largest forest size need to be calculated, based on which the hash indexes of other groups can be deduced with simple bits sift operations.

We analyze the time complexity and space complexity of FLTER-LSH. With the help of our well designed LSH hash table, we can quickly retrieve large entries through low cost hash computation. In addition, LSH forests are virtually built rather than physically exist, thus the storage cost is low.

We conduct comprehensive experiments on 4 publicly real world datasets to comparatively evaluate and demonstrate the effectiveness of the proposed method. Compared with the state of art tensor completion algorithms, our algorithm can achieve high data retrieving accuracy using significantly smaller time.

The rest of the paper is organized as follows. We introduce the related work and the preliminaries of tensor in Sections II and III. We introduce the problem and its challenge in Section IV. We transform the large entry retrieving problem to a cosine similarity searching problem in Section V. In Section VI, we present our solution overview. In Section VII and VIII, we present in details our algorithms for LSH hash table building and similarity query, respectively. We analyze the time complexity and space complexity of our FLTER-LSH in Section IX. Finally, we evaluate the performance of the proposed algorithm through extensive experiments in Section X, and conclude the work in Section XI.

II. Related Work

During the data-acquisition and processing, there may exist data missing due to mechanical failures, human-induced factors, and measurement cost saving. Estimating the missing values based on a few observed entries in a tensor is usually referred to as tensor completion.

With the rapid progress of sparse representation techniques, following the compressive sensing [23], [24] and matrix completion [25]–[29] to process one-dimensional and two-dimensional data arrays, tensor completion has attracted lots of research interests recently. Compared with compressive sensing and matrix completion, tensor completion can more accurately recover the missing data, taking advantage of the multidimensional data structure.

Besides its traditional application scenarios in signal processing [30], recently, tensor completion has started to be used in network field for complete network data analysis [11]–[14]. Different from these initial applications [11]–[14] apply tensor completion to recover the complete monitoring data with a few measurement samples, this paper wants to apply tenor completion to quickly and accurately retrieve the large entries with a few measurement samples.

Generally, to retrieve large entries in tensor with partial samples, we should first train the factor matrices to recover the missing entries, and then return the large entries. Many variants [15]–[20] of tensor completion algorithms are proposed to train the factor matrices. Depending on the optimization algorithms adopted to train the factor matrices, these algorithms can be divided into three types: alternating least squares (ALS) [18], [19], stochastic gradient descent (SGD) [18], [20], and coordinate descent (CCD++) [19], [20]. This paper does not focus on factor matrix training, and we can apply these existing studies to train the factor matrices.

As introduced in Section 1, only one recent work [21] studies the problem of retrieving the top-k large entries of the tensor. It proposes the use of discrete tensor completion to represent the real-valued factor matrices in traditional tensor completion model with binary codes. Accordingly, it can calculate Hamming distance through lightweight XOR bit

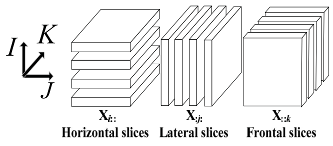


Fig. 1. Tensor slices.

operations to quickly retrieve the locations of top-k entries. Rather than only identifying the locations of the top-k entries, we aim to find both the locations and values of the entries whose values are larger than a threshold.

Different from existing studies, to quickly retrieve large entries of tensor with partial measurements, this paper transforms the large entry retrieving problem to a cosine similarity searching problem. To provide a simple solution, we propose an algorithm based on LSH hash table to quickly reorder the factor vectors so that the vectors with small cosine distances are placed into the same hash bucket. Facilitated by the LSH tables, we further propose a query algorithm to quickly find the similar vector of a queried one so they together determine a large entry without incurring the high cost of recovering all entries through the dot products.

III. Preliminaries

In this paper, scalars are denoted by lowercase letters $(a,\ b,\ \cdots\ )$, vectors are written in boldface lowercase $(\mathbf{a}, \mathbf{b}, \cdots)$, and matrices are represented with boldface capitals $(\ \mathbf{A}, \mathbf{B}, \cdots\ )$. Higher-order tensors are written as calligraphic letters $(\mathcal{X},\ \mathcal{Y},\ \cdots)$. The elements of a tensor are denoted by its symbolic name with indexes in subscript. For example, the i-th entry of a vector $\mathbf{a}$ is denoted by a_i, the element (i, j) of a matrix $\mathbf{A}$ is denoted by a_{ij}, and the element (i, j, k) of a third-order tensor $\mathcal{X}$ is denoted by x_{ijk}. The row (column) vectors of a matrix are denoted by the symbolic name of the matrix with indexes in subscript. For example, the i-th row (column) vector of a matrix $\mathbf{A}$ is denoted by $\mathbf{a}_i$ $(\mathbf{a}_{(i)})$. And $||\mathbf{a}|| = \sqrt{\sum_i a_i^2}$ denotes the length (Euclidean norm) of vector $\mathbf{a}$, $||\mathbf{A}||^2 = \sqrt{\sum_{i,j} a_{ij}^2}$ denotes Frobenius norm of a matrix $\mathbf{A}$. Some preliminaries in this paper can be found in [31]–[33].

Definition 1: A tensor *is a multidimensional array, and is a higher-order generalization of a vector (first-order tensor) and a matrix (second-order tensor). An N-way or Nth-order tensor (denoted as $\mathcal{A} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$) is an element of the tensor product of N vector spaces, where N is the order of $\mathcal{A}$, also called way or mode.*

The element of $\mathcal{A}$ is denoted by $a_{i_1,i_2,\cdots,i_N}$, $i_n \in \{1, 2, \cdots, I_n\}$ with $1 \le n \le N$. In this paper, we take an 3-way tensor as an example to illustrate our algorihtm.

Definition 2: Slices of a 3-way tensor are two-dimensional sub-arrays, defined by fixing all indexes but two.

In Fig.1, a 3-way tensor $\mathcal{X}$ has horizontal, lateral and frontal slices, which are denoted by $\mathbf{X}_{i::}$, $\mathbf{X}_{:j:}$ and $\mathbf{X}_{::k}$, respectively. In this paper, we denote the frontal slice $\mathbf{X}_{::k}$ as $\mathbf{X}_k$.

Definition 3: The outer product of two column vectors $\mathbf{a} \circ \mathbf{b}$ is the matrix defined by: $(\mathbf{a} \circ \mathbf{b})_{ij} = a_i b_j$.

Definition 4: The outer product of three column vectors $\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$ is a rank one 3-way tensor defined by: $(\mathbf{a} \circ \mathbf{b} \circ \mathbf{c})_{ijk} = a_i b_j c_k$.

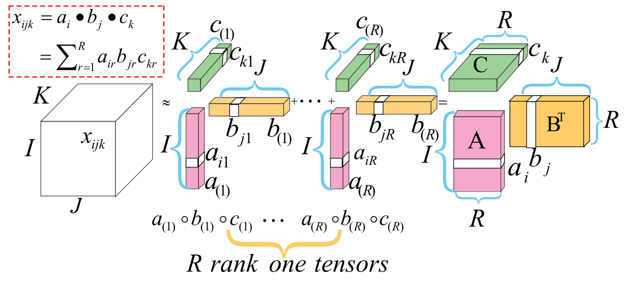

$x_{ijk} = a_i \bullet b_j \bullet c_k = \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr}$
$a_{(1)} \circ b_{(1)} \circ c_{(1)} \cdots a_{(R)} \circ b_{(R)} \circ c_{(R)}$
R rank one tensors

Fig. 2. CP decomposition of three-way tensor.

Definition 5: The rank of a 3-way tensor is the minimal number of rank one tensors, that generate the tensor as their sum, i.e. the smallest R, such that $\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_{(r)} \circ \mathbf{b}_{(r)} \circ \mathbf{c}_{(r)}$ where $\mathbf{a}_{(r)}$, $\mathbf{b}_{(r)}$, and $\mathbf{c}_{(r)}$ are column vectors.

Definition 6: The idea of CANDECOMP/PARAFAC (CP) decomposition is to express a tensor as the sum of a finite number of rank one tensors. A 3-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ can be expressed as

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_{(r)} \circ \mathbf{b}_{(r)} \circ \mathbf{c}_{(r)}, \tag{1}$$

with an entry calculated as

$$x_{ijk} = \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr} \tag{2}$$

where $R > 0$, a_{ir}, b_{jr}, c_{kr} are the i-th, j-th, and k-th entry of column vectors $\mathbf{a}_{(r)} \in \mathbb{R}^I$, $\mathbf{b}_{(r)} \in \mathbb{R}^J$, and $\mathbf{c}_{(r)} \in \mathbb{R}^K$, respectively.

By collecting the vectors in the rank one components, we have tensor $\mathcal{X}$'s factor matrices $\mathbf{A} = \left[\mathbf{a}_{(1)}, \cdots, \mathbf{a}_{(R)}\right] \in \mathbb{R}^{I \times R}$, $\mathbf{B} = \left[\mathbf{b}_{(1)}, \cdots, \mathbf{b}_{(R)}\right] \in \mathbb{R}^{J \times R}$, and $\mathbf{C} = \left[\mathbf{c}_{(1)}, \cdots, \mathbf{c}_{(R)}\right] \in \mathbb{R}^{K \times R}$. Using the factor matrices, we can rewrite the CP decomposition as follows.

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_{(r)} \circ \mathbf{b}_{(r)} \circ \mathbf{c}_{(r)} = [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \tag{3}$$

In Fig.2, a three-way tensor can be represented through CP decomposition as the sum of R outer products (rank one tensors). That is, $\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_{(r)} \circ \mathbf{b}_{(r)} \circ \mathbf{c}_{(r)}$, with column vectors $\mathbf{a}_{(r)} \in \mathbb{R}^I$, $\mathbf{b}_{(r)} \in \mathbb{R}^J$, and $\mathbf{c}_{(r)} \in \mathbb{R}^K$. In addition, an entry x_{ijk} can be calculated as the sum of the dot product of row vectors $\mathbf{a}_i$, $\mathbf{b}_j$, and $\mathbf{c}_k$. That is, $x_{ijk} = \mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k$, where $\mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k = \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr}$ is the dot product of the three row vectors $\mathbf{a}_i$, $\mathbf{b}_j$, and $\mathbf{c}_k$ with $\mathbf{a}_i \in \mathbb{R}^{1 \times R}$, $\mathbf{b}_j \in \mathbb{R}^{1 \times R}$, and $\mathbf{c}_k \in \mathbb{R}^{1 \times R}$.

Definition 7: Given a 3-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with observed entries denoted by $\Omega = \{(i,j,k)\,|x_{ijk}\ is\ known\}$, the tensor completion problem is to train the factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ through solving following problem

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \sum_{i,j,k \in \Omega} (x_{ijk} - \mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k)^2 + \alpha||\mathbf{A}||^2 + \beta||\mathbf{B}||^2 + \gamma||\mathbf{C}||^2 \tag{4}$$

where $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ are factor matrices with $\mathbf{a}_i$, $\mathbf{b}_j$, and $\mathbf{c}_k$ being the i-th row vector of $\mathbf{A}$, the j-th row vector of $\mathbf{B}$, and the k-th row vector of $\mathbf{C}$.

In Eq.(4), x_{ijk} is the observed entry at (i,j,k) while $\mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k$ is the recovered entry. Eq.(4) aims to train factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ to minimize the recovery error at the observed entries. α, β, and γ are the regularization coefficients. $\alpha||\mathbf{A}||^{\mathbf{2}} + \beta||\mathbf{B}||^{\mathbf{2}} + \gamma||\mathbf{C}||^{\mathbf{2}}$ is added in the formulation to prevent over-fitting problem. After factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are obtained, the un-observed entries can be recovered with $\hat{x}_{ijk} = \mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k$.

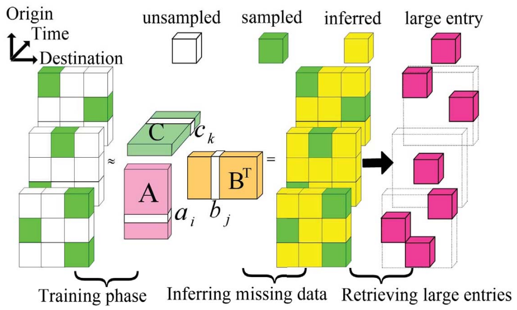


Fig. 3. Tensor completion based approach.

IV. Problem and Its Challenge

We model the network monitoring data as a 3-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, where I, J, and K correspond to the number of origin nodes, the number of destination nodes, and the number of time intervals monitored, respectively. Each entry m_{ijk} indicates the end-to-end monitoring data from the origin node i to the destination node j in the time interval k. If the tensor records traffic volume data of a network, each entry m_{ijk} indicates the end-to-end traffic volume data from the origin node i to the destination node j in the time slot k. If the tensor records the latency of a network, an entry m_{ijk} indicates the latency from the origin node i to the destination node j in the time slot k.

Given some observed entries (i.e., measurement samples) in a monitoring tensor, the aim of the paper is to retrieve all large entries whose values are above a predefined threshold ε. As shown in Fig.3, based on tensor completion, a straightforward way of retrieving large entries includes two phases:

Training Phase: Given a 3-way monitoring tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with observed entries in the set $\Omega = \{(i,j,k)\,|x_{ijk}\ is\ known\}$, training the factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ through tensor factorization by solving the problem in Eq.(4). This paper does not focus on factor matrix training, and we can apply existing studies [15]–[20] to train the factor matrices.

Inferring and Retrieving Phase: After three factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are trained, the large entry retrieving problem can be expressed as

$$\{(i,j,k) \in [I] \times [J] \times [K]\,|\mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k > \varepsilon\} \tag{5}$$

where ε is the threshold and the dot product $\mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k$ corresponds to the recovered entry $\hat{x}_{ijk}$.

Directly solving the problem in Eq.(5) requires recovering all entries through the dot products with the order of computation cost at $O(IJKR)$. This will become the bottleneck when the size of the tensor is large. Large entries are often resulted from anomaly, and often have lower number in reality. To reduce the computation cost, we transform the large entry inferring problem in Eq.(5) to a cosine similarity searching problem in Section V.

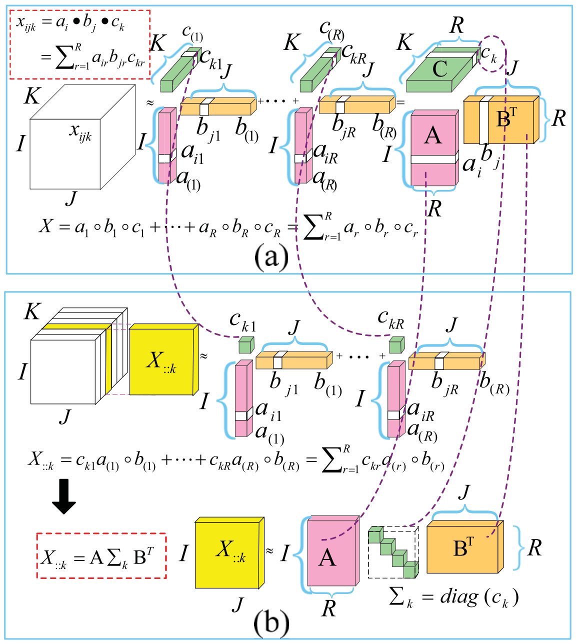

$x_{ijk} = a_i \bullet b_j \bullet c_k = \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr}$
$X = a_1 \circ b_1 \circ c_1 + \cdots + a_R \circ b_R \circ c_R = \sum_{r=1}^{R} a_r \circ b_r \circ c_r$
(a)
$X_{::k} = c_{k1} a_{(1)} \circ b_{(1)} + \cdots + c_{kR} a_{(R)} \circ b_{(R)} = \sum_{r=1}^{R} c_{kr} a_{(r)} \circ b_{(r)}$
$X_{::k} = A \Sigma_k B^T$
$\Sigma_k = diag(c_k)$
(b)

Fig. 4. The relationship between tensor CP decomposition and frontal slice matrix decomposition: (a) CP decomposition, (b) decomposition of frontal slice matrix.

V. Problem Transformation

To facilitate the problem transformation, in this section, we first investigate the relationship between tensor CP decomposition and the decomposition of a frontal slice of the tensor, based on which, the dot product with three vectors is transformed to the dot product based on two vectors. According to Eq.(3), the CP decomposition of a 3-way tensor $\mathcal{X}$ can be written as follows.

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_{(r)} \circ \mathbf{b}_{(r)} \circ \mathbf{c}_{(r)} = [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \tag{6}$$

where matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ are the factor matrices in the CP decomposition. This decomposition process is illustrated in Fig.4(a), and in Fig.4(b), a frontal slice $\mathbf{X}_k$ can be written as

$$\begin{aligned}\mathbf{X}_k &= c_{k1}\mathbf{a}_{(1)} \circ \mathbf{b}_{(1)} + \cdots + c_{kR}\mathbf{a}_{(R)} \circ \mathbf{b}_{(R)} \\ &= \sum_{i=1}^{R} c_{ki}\mathbf{a}_{(i)} \circ \mathbf{b}_{(i)}\end{aligned} \tag{7}$$

where $c_{k1}, c_{k2}, \cdots, c_{kR}$ are the entries of the k-th row (i.e., $\mathbf{c}_k$) of the factor matrix $\mathbf{C}$.

As $\mathbf{A} = [\mathbf{a}_{(1)}, \cdots, \mathbf{a}_{(R)}]$ and $\mathbf{B} = [\mathbf{b}_{(1)}, \cdots, \mathbf{b}_{(R)}]$, according to (7), $\mathbf{X}_k$ can be rewritten as

$$\mathbf{M}_k = \mathbf{A}\mathbf{\Sigma}_k\mathbf{B}^T \tag{8}$$

where $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$ are the factor matrices in the CP decomposition, $\mathbf{\Sigma}_k = diag(\mathbf{c}_k)$, and $\mathbf{c}_k \in \mathbb{R}^{1 \times R}$ is the k-th row of the factor matrix $\mathbf{C}$. Let $\mathbf{D}^k = \mathbf{A}\mathbf{\Sigma}_k$, we have $\mathbf{X}_k = \mathbf{D}^k\mathbf{B}^T$. As $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{c}_k \in \mathbb{R}^{1 \times R}$, we have $\mathbf{\Sigma}_k \in \mathbb{R}^{R \times R}$ and thus $\mathbf{D}^k \in \mathbb{R}^{I \times R}$.

Let $\mathbf{d}_i^k$ and $\mathbf{b}_j$ denote the i-th row and j-th row of matrices $\mathbf{D}^k$ and $\mathbf{B}$. After factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ have been trained, the entry (i, j, k) can be recovered through

$$\hat{x}_{ijk} = \mathbf{d}_i^k \bullet \mathbf{b}_j \tag{9}$$

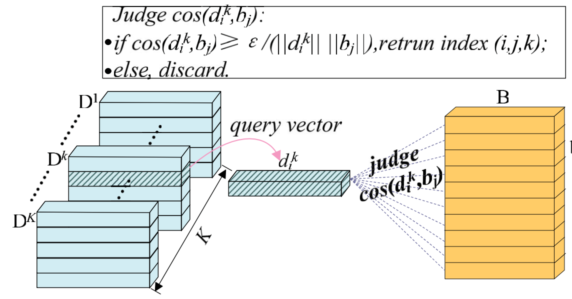

Judge cos(d_i^k,b_j):
•if cos(d_i^k,b_j)≥ ε /(||d_i^k|| ||b_j||),retrun index (i,j,k);
•else, discard.
D¹
Dᵏ
Dᴷ
K
query vector
d_i^k
judge
cos(d_i^k,b_j)
B
b_j

Fig. 5. Similarity search problem.

which is the dot product operation between $\mathbf{d}_i^k$ and $\mathbf{b}_j$.

According to Eq.(9), our problem of finding the large entry $\mathbf{a}_i \bullet \mathbf{b}_j \bullet \mathbf{c}_k > \varepsilon$ is transformed to a problem of finding pairs $(\mathbf{d}_i{}^k, \mathbf{b}_j)$ such that $\mathbf{d}_i{}^k \bullet \mathbf{b}_j \geq \varepsilon$. Obviously, we further have

$$\hat{x}_{ijk} = \mathbf{d}_i^k \bullet \mathbf{b}_j \geq \varepsilon \rightarrow \cos(\Theta(\mathbf{d}_i^k, \mathbf{b}_j)) \geq \frac{\varepsilon}{||\mathbf{d}_i^k||||\mathbf{b}_j||} \tag{10}$$

where $||\mathbf{v}|| = \sqrt{\sum_i \mathbf{v}_i^2}$ denotes the length (Euclidean norm) of vector $\mathbf{v}$, $\cos(\Theta(\mathbf{d}_i^k, \mathbf{b}_j))$ is the cosine similarity defined as the cosine of the angle $\Theta(\mathbf{d}_i^k, \mathbf{b}_j)$ between vectors $\mathbf{d}_i^k$ and $\mathbf{b}_j$, that is: $\cos(\Theta(\mathbf{d}_i^k, \mathbf{b}_j)) = \frac{\mathbf{d}_i^k \bullet \mathbf{b}_j}{||\mathbf{d}_i^k||||\mathbf{b}_j||}$.

Therefore, the large entry retrieval problem can be transformed to a cosine similarity search problem. As an entry x_{ijk} is considered large if $\cos(\Theta(\mathbf{d}_i^k, \mathbf{b}_j)) \geq \frac{\varepsilon}{||\mathbf{d}_i^k||||\mathbf{b}_j||}$, we can use each vector $\mathbf{d}_i^k$ to query against all $\mathbf{b}_j$ and check if there exists a vector pair $(\mathbf{d}_i^k, \mathbf{b}_j)$ that has the high cosine similarity value. As shown in Fig.5, we call $\mathbf{d}_i^k$ the query vector.

VI. Overview Solution

For a vector $\mathbf{d}_i^k$ to query, a straightforward strategy of finding similar vectors in $\mathbf{B}$ is to test all $\mathbf{b}_j$ it contains. However, this would result in a in high computation cost for a large-size tensor.

To reduce the computation cost, we propose to exploit Locality-sensitive Hashing (LSH). We will reorder and store vectors of $\mathbf{B}$ in LSH hash tables, and further propose a light-weight LSH query algorithm to quickly find similar vectors through simple hash calculations.

A. Locality-Sensitive Hash Families

Formally, an LSH family is defined as follows [34], [35]:

Definition 8 (Locality-Sensitive Hashing): A family of $\mathcal{H}$ is called (r, cr, p_1, p_2)-sensitive if for any two items $\mathbf{p}$ and $\mathbf{q}$,

1) if $Dist(\mathbf{p}, \mathbf{q}) \leq r$, then $Prob[h(\mathbf{p}) = h(\mathbf{q})] \geq p_1$
2) if $Dist(\mathbf{p}, \mathbf{q}) \geq cr$, then $Prob[h(\mathbf{p}) = h(\mathbf{q})] \leq p_2$

where $Dist(\mathbf{p}, \mathbf{q})$ calculates the distance between items $\mathbf{p}$ and $\mathbf{q}$, $h(\mathbf{v})$ is the hash value of the item $\mathbf{v}$ calculated by LSH hash function h(). Here $c > 1$ and $p_1 > p_2$.

Intuitively, a hash function is locality-sensitive if its probability of collision (by hashing two items to the same hash value) is higher for "nearby" items than for items that are "far apart". Two items are nearby if their distance is at most r, and they are far apart if their distance is at least cr, where $c > 1$ quantifies the gap between "near" and "far".

Based on LSH function, we could find the approximately similar neighbors of any item, simply by finding the bucket that it hashes to, and returning the other items in the bucket.

B. Cosine LSH

It is possible to define such locality-sensitive hash families using many different distance functions $Dist$, including the Cosine distance, Jaccard measure, the Hamming norm, and the l_1 and l_2 norms.

According to Eq.(10), given a vector $\mathbf{d}_i^k$, the aim of this paper is to quickly find the vector $\mathbf{b}_j \in \mathbf{B}$ that has a high cosine similarity to $\mathbf{d}_i^k$. We design our algorithm based on Cosine LSH as follows.

Definition 9 (Cosine Locality-Sensitive Hashing): A family of Cosine $\mathcal{H}$ is defined as

$$h(\mathbf{v}) = \operatorname{sgn}(\mathbf{v} \bullet \mathbf{w}) = \begin{cases} 1 & \mathbf{v} \bullet \mathbf{w} \geq 0 \\ 0 & \mathbf{v} \bullet \mathbf{w} < 0 \end{cases} \tag{11}$$

where $\mathbf{v}$ is the input item and $\mathbf{w}$ is a normal unit vector to denote a random hyperplane.

The basic idea of Cosine $\mathcal{H}$ is to choose a random hyperplane (defined by a normal unit vector $\mathbf{w}$) and uses the hyperplane to hash input items. $h(\mathbf{v}) = \{1 \ or \ 0\}$ depends on which side of the hyperplane $\mathbf{v}$ lies. It is not difficult to prove that [36], for two vectors $\mathbf{u}$, $\mathbf{v}$, $Pr[h(\mathbf{u}) = h(\mathbf{v})] = 1 - \frac{\theta(\mathbf{u},\mathbf{v})}{\pi}$, where $\theta(\mathbf{u}, \mathbf{v})$ is the angle between $\mathbf{u}$ and $\mathbf{v}$. Therefore, Cosine $\mathcal{H}$ is $(\theta, c\theta, p_1, p_2)$-sensitive, where $p_1 = 1 - \frac{\theta}{\pi}$, $p_2 = 1 - \frac{c\theta}{\pi}$, and $p_1 > p_2$.

Standard Cosine $\mathcal{H}$ in Eq.(11) only produces a single bit hash index. The probability for two vectors to have the same hashed bit value is proportional to the angle between them. Obviously, to find vector pair $(\mathbf{d}_i^k, \mathbf{b}_j)$ whose cosine similarity value exceeds the threshold $\frac{\varepsilon}{||\mathbf{d}_i^k|| \, ||\mathbf{b}_j||}$, it also depends on the length of $||\mathbf{d}_i^k||$ and $||\mathbf{b}_j||$. Therefore, different lengths of the vector pair have different angular requirements.

C. Overview of Quick Retrieving of Large Entries

A single hash function of Cosine $\mathcal{H}$ only generates one single bit hash index, and can not support the query for different angular levels required in our problem. Instead of using a single hash function, we use multiple hash functions in Cosine $\mathcal{H}$ to construct a set of hash tables.

- Step (1): Choose m functions $h_1, h_2, \cdots, h_m$ uniformly at random (with replacement) from Cosine $\mathcal{H}$. For any item $\mathbf{b}_j$, place $\mathbf{b}_j$ in the bucket with label $g(\mathbf{b}_j) = (h_1(\mathbf{b}_j), h_2(\mathbf{b}_j), \cdots, h_m(\mathbf{b}_j))$.
- Step (2): Independently perform step (1) n times to construct n separate hash tables, with hash functions g_1, g_2, $\cdots$, g_n.

For each g_j, it consists of m basic cosine hash functions and can be denoted by $g_j = (h_{j1}, h_{j2}, \cdots, h_{jm})$. As each h_i outputs one "digit", each hash bucket in one table has an m-digit label.

In order to exploit hashing to find similar items, we would like items of higher similarity to fall into the same bucket, i.e., creating hash collisions, but ones not similar to avoid collisions. Step (1) concatenates m hash functions together to identify the hash bucket. If two "distant" items have a probability p_2 of collision with one hash function, their collision probability drops to p_2^m with the concatenation, which will reduce the rate of false positives in a query process.

However, if p_1 is the probability of two similar items collision with one hash function, the concatenation

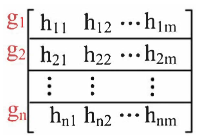


Fig. 6. Hash functions are organized in a hash generation matrix.

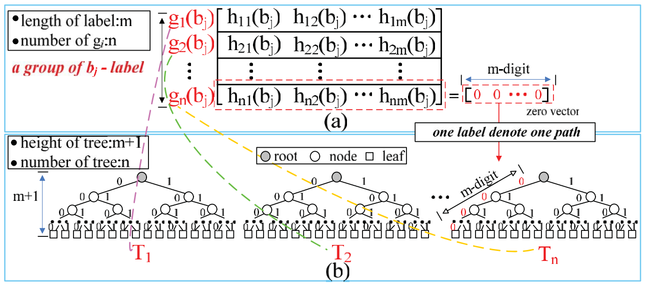


Fig. 7. The relationship between the LSH forest and the hash label: (a) hash table, (b) LSH forest.

of m functions also reduces the chance of collision between similar items and leads to the miss of similar items thus large entries in a query process. To improve the recall (i.e. the ratio between the number of similar vectors retrieved and the total number of similar vectors existing), step (2) constructs multiple hash tables for a given group of data to increase the chance of finding similar items. As long as the key is hashed to the same bucket in any of the tables, the similar item will be retrieved from the query.

Specifically, n such compound hash functions g_1, g_2, $\cdots$, g_n are constructed in step (2), each of which corresponds to one hash table. In Section VII, we will provide our algorithm to set parameters m and n. With Steps 1 and 2, we apply totally $n \times m$ hash operations to build the hash tables, and we organize these hash functions into a *hash generation matrix* as shown in Fig.6. Obviously, one row in the hash matrix generates an m-digit hash label, which corresponds to a hash bucket in one hash table. To build n hash tables in step (2), the hash generation matrix should have n rows, as shown in Fig.6. To help understand our solution, we build an LSH forest to present the hash tables. As shown in Fig.7, the LSH forest has n trees with the tree height being $m + 1$. Each tree represents one hash table. A square node in these trees denotes a leaf node, which corresponds to a hash bucket. For a given item, at each tree node, a hash function will be applied, and the binary hash value determines the tree will go for left or right children branch for this item. Thus, all trees in the forest are binary, and the hashed digits along the path from the root to the leaf can be used as the label of the leaf node.

Note that we only use the LSH forest to help introduce the operations in this paper, with the LSH forest virtually built. Physically, we only need to build the hash tables and directly map items to store and keys for query to the trees in the forest. Based on the LSH forest, we design our algorithm for the quick large-entry retrieving with the following three steps:

Preprocessing: re-order and store each $\mathbf{b}_j \in \mathbf{B}$ into n buckets (leaf nodes) in n different hash tables (trees) with their labels being $g_1(\mathbf{b}_j), g_2(\mathbf{b}_j), \cdots, g_n(\mathbf{b}_j)$.

Similarity search: for each $\mathbf{d}_i^k$, search all buckets (leaf nodes) $g_1(\mathbf{d}_i^k), g_2(\mathbf{d}_i^k), \cdots, g_n(\mathbf{d}_i^k)$ in n hash tables (trees).

Let $\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_l$ encountered therein be the candidate similar vectors. Obviously, $\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_l$ are the collision vectors of $\mathbf{d}_i^k$.

Large entry retrieving: For each candidate similar vector $\mathbf{z}_j$ found in step 2, calculate the dot operations $\mathbf{d}_i^k \bullet \mathbf{z}_j$, if $\mathbf{d}_i^k \bullet \mathbf{z}_j \geq \varepsilon$, return the corresponding entry and its entry value.

For a similarity search, the items matched in any bucket of $g_1(\mathbf{d}_i^k), g_2(\mathbf{d}_i^k), \cdots, g_n(\mathbf{d}_i^k)$ in the n hash tables are returned as the candidate similar vectors. As long as any of them meets the similarity search condition, the matched item is found. Thus building multiple hash tables can improve the query recall.

For each querying vector $\mathbf{d}_i^k$, instead of calculating the dot operations among all $\mathbf{d}_i^k$ and $\mathbf{b}_j$ pairs with J dot operations for all $1 \leq j \leq J$, only l dot operations are needed. Thus, the computation cost can be largely reduced.

VII. Building LSH Hash Tables

A. Challenge

Requirement: to achieve accurate similarity query and reduce the computation cost, building a good LSH hash tables (LSH forest) should guarantee that: 1) all $\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_l$ encountered in the hash buckets have high cosine similarity with the query vector $\mathbf{d}_i^k$; 2) no vectors with high cosine similarity are missing.

However, it is challenging to determine the parameters m and n that control the hash table building. With the hash table designed based on Cosine $\mathcal{H}$ which is $(\theta, c\theta, p_1, p_2)$-sensitive, to find the cosine similar neighbor for ${\mathbf{d}_i}^k$, we need to know the angular distance θ in order to select m and n. In other words, once we determine what constitutes a "nearby" point (distance less than θ) and what constitutes a far-away point (distance greater than $c\theta$), it is possible to construct a tuned index that returns approximately similar neighbors. However, different pairs (${\mathbf{d}_i}^k$, $\mathbf{b}_j$) have different vector lengths, thus different angular requirements to be identified that the corresponding entry is a large entry. So the LSH index calculated by a particular angular requirement θ of a single vector pair ${\mathbf{d}_i}^k$ and $\mathbf{b}_j$ cannot ensure all pairs can achieve a good performance. On the other hand, it would incur a high storage and computation cost to build the LSH index for every different θ.

To address the above issue, we first investigate the relationship between angle requirement and parameter setting, and then present our algorithms to build the hash tables and support flexible cosine similarity query with different θ requirements.

B. Building Hash Tables for a Given θ

The ball of radius θ centered at $\mathbf{d}_i^k$ is defined as $X(\mathbf{d}_i^k, \theta) = \{\mathbf{b}_j^* \in \mathbf{B} | \Theta(\mathbf{d}_i^k, \mathbf{b}_j^*) < \theta\}$. Let $Y(\mathbf{d}_i^k, \theta) = \{\mathbf{b}_j^\# \in \mathbf{B} | \Theta(\mathbf{d}_i^k, \mathbf{b}_j^\#) > c\theta\}$.

To satisfy the **Requirement** in Section VII-A, parameters m and n should be chosen to ensure that the following properties hold:

R_1: if $\mathbf{b}_j^* \in X(\mathbf{d}_i^k, \theta)$, then $g_p(\mathbf{b}_j^*) = g_p(\mathbf{d}_i^k)$ should be met for some $1 \leq p \leq n$ thus the similar item is not missed during a query.

R_2: if $\mathbf{b}_j^\# \in Y(\mathbf{d}_i^k, c\theta)$, then $g_p(\mathbf{b}_j^\#) \neq g_p(\mathbf{d}_i^k)$ for all $1 \leq p \leq n$.

Following we will demonstrate that if properties R_1 and R_2 hold, the search procedure works correctly. In our solution overview in Section VI-C, items in $g_1\left(\mathbf{d}_i^k\right) \cup g_2\left(\mathbf{d}_i^k\right) \cup \cdots \cup g_n\left(\mathbf{d}_i^k\right)$ are returned as the candidate similar vectors of $\mathbf{d}_i^k$ after the similarity search.

According to the property R_2, the buckets $g_p(\mathbf{d}_i^k)$ for all $1 \leq p \leq n$ cannot contain any items from $Y(\mathbf{d}_i^k, c\theta)$. According to the property R_1, every $\mathbf{b}_j^* \in X(\mathbf{d}_i^k, \theta)$ is contained in at least one bucket $g_p(\mathbf{d}_i^k)$ for some $1 \leq p \leq n$, so all similar items can be returned upon the search.

Obviously, when m and n are set to guarantee the properties R_1 and R_2, $g_1\left(\mathbf{d}_i^k\right) \cup g_2\left(\mathbf{d}_i^k\right) \cup \cdots \cup g_n\left(\mathbf{d}_i^k\right)$ contains only $\mathbf{b}_j$ such that $\Theta(\mathbf{d}_i^k, \mathbf{b}_j) \leq c\theta$.

Let property R_1 hold with a probability P_1. When R_1 holds, R_2 holds with a probability P_2. In the following Theorem 1, we will show both P_1 and P_2 are large when m and n are set to meet the two conditions for good search performance.

Theorem 1: Given angular requirement θ, setting $m = \log_{(p_1/p_2)} 2N$ and $n = (2N)^\rho$ guarantees that

- R_1 holds with the probability being at least $1 - \frac{1}{e}$
- when R_1 holds, R_2 holds with the probability being larger than $\frac{1}{2}$

where $p_1 = 1 - \frac{\theta}{\pi}$, $p_2 = 1 - \frac{c\theta}{\pi}$, $\rho = -\frac{ln(p_1)}{ln(p_1/p_2)}$, and N is the number of $\mathbf{b_j}$ in $\mathbf{B}$ that are needed to query against.

Assume that $\mathbf{b}_j^* \in X(\mathbf{d}_i^k, \theta)$; the proof is similar when $\mathbf{b}_j^* \notin X(\mathbf{d}_i^k, c\theta)$. Consider any point $\mathbf{b}_j^\# \in Y(\mathbf{d}_i^k, c\theta)$. Clearly, for every g_p with $1 \leq p \leq n$, we have

$$\begin{aligned} P_1 &= \Pr(\{[g_1(\mathbf{b}_j^*) = g_1(\mathbf{d}_i^k)] \cup \ldots \cup [g_n(\mathbf{b}_j^*) = g_n(\mathbf{d}_i^k)]\}) \\ &= 1 - (1 - p_1^m)^n \end{aligned} \tag{12}$$

where p_1^m is the probability of $g_t(\mathbf{b}_j^*) = g_t(\mathbf{d}_i^k)$ $(1 \leq t \leq n)$, as g_t consists of m basic cosine hash functions.

By substituting $m = \log_{(p_1/p_2)} 2N$ in p_1^m, we have

$$p_1^m = {p_1}^{\log_{(p_1/p_2)}(2N)} \tag{13}$$

Let ${p_1}^{\log_{(p_1/p_2)}(2N)} = f$, then we have

$$\begin{aligned} & {p_1}^{\log_{(p_1/p_2)}(2\mathrm{N})} = f \\ & \Leftrightarrow \log_{p_1} {p_1}^{\log_{(p_1/p_2)}(2N)} = \log_{p_1} f \\ & \Leftrightarrow \log_{(p1/p2)}(2N) = \log_{p1} f \\ & \Leftrightarrow (p_1/p_2)^{\log_{(p_1/p_2)}(2N)} = (p_1/p_2)^{\log_{p_1} f} \\ & \Leftrightarrow 2N = (p_1/p_2)^{\log_{p_1} f} \\ & \Leftrightarrow (2N)^{\log_{(p_1/p_2)}(p_1)} = \left((p_1/p_2)^{\log_{p_1} f}\right)^{\log_{(p_1/p_2)}(p_1)} = f \end{aligned} \tag{14}$$

Thus, we obtain $(2N)^{\log_{(p_1/p_2)}(p_1)} = f$. With ${p_1}^{\log_{(p_1/p_2)}(2N)} = f$ and $(2N)^{\log_{(p_1/p_2)}(p_1)} = f$, we have ${p_1}^{\log_{(p_1/p_2)}(2N)} = (2N)^{\log_{(p_1/p_2)}(p_1)}$ and

$$p_1^m = {p_1}^{\log_{(p_1/p_2)}(2N)} = (2N)^{\frac{\ln p_1}{\ln(p_1/p_2)}} = (2N)^{-\rho} \tag{15}$$

where $\rho = -\frac{ln(p_1)}{ln(p_1/p_2)}$. With (14), we can write (12) as follows

$$\begin{aligned} P_1 &= 1 - (1 - {p_1}^m)^n = 1 - (1 - (2N)^{-\rho})^{(2N)^\rho} \\ &= 1 - (1 + (-(2N)^{-\rho}))^{-(-(2N)^\rho)} \end{aligned} \tag{16}$$

Let $-(2N)^\rho = q$, then P_1 can be rewritten as $P_1 = \left(1+\frac{1}{q}\right)^{-q}$. Obviously, P_1 is a monotonically increasing function of q, where $q \in (-\infty, -1]$. Then we have $\lim_{q\to-\infty} 1 - (1+(\frac{1}{q}))^{-q} = 1 - \frac{1}{e}$. When $n = (2N)^\rho$, probability P_1 is at least $1 - \frac{1}{e}$.

To facilitate the finding of the probability P_2, we define the property W_2 as:

W_2: if $\mathbf{b}_j^\# \in Y(\mathbf{d}_i^k, c\theta)$, then $g_p(\mathbf{b}_j^\#) = g_p(\mathbf{d}_i^k)$ for some $1 \le p \le n$.

We can find the conditional probability P_2' for W_2 when R_1 is held:

$$\begin{aligned} {P_2}' &= \Pr(\ W_2 |\ R_1) = \frac{\Pr(W_2 \wedge R_1)}{\Pr(R_1)} \le \frac{\Pr(W_2)}{\Pr(R_1)} \\ &= \frac{\Pr(\{[g_1(\mathbf{b}_j^\#) = g_1(\mathbf{d}_i^k)] \cup \ldots \cup [g_n(\mathbf{b}_j^\#) = g_n(\mathbf{d}_i^k)]\})}{\Pr(\{[g_1(\mathbf{b}_j^*) = g_1(\mathbf{d}_i^k)] \cup \ldots \cup [g_n(\mathbf{b}_j^*) = g_n(\mathbf{d}_i^k)]\})} \\ &= \frac{1-(1-p_2^m)^n}{1-(1-p_1^m)^n} \\ &\le \frac{(1-p_2^m)^n}{(1-p_1^m)^n} = \left(\frac{1-p_2^m}{1-p_1^m}\right)^n \le \left(\frac{p_2^m}{p_1^m}\right)^n \end{aligned} \tag{17}$$

By substituting $m = \log_{(p_1/p_2)} 2N$ and $n = (2N)^\rho$ in (17), we can easily find that $P_2' \le \left(\frac{1}{2N}\right)^{(2N)^\rho}$. Therefore, P_2 can be written as:

$$\begin{aligned} P_2 &= 1 - \sum_{\mathbf{b}_j^\# \in Y(\mathbf{d}_i^k, \theta)} P_2' = 1 - \sum_{\mathbf{b}_j^\# \in Y(\mathbf{d}_i^k, \theta)} \left(\frac{1}{2N}\right)^{(2N)^\rho} \\ &\ge 1 - \left(\frac{1}{2}\right)^{(2N)^\rho} \ge \frac{1}{2} \end{aligned} \tag{18}$$

Thus, setting $m = \log_{(p_1/p_2)} 2N$ and $n = (2N)^\rho$ ensures that both P_1 and P_2 are large. The proof completes.

C. Pruning to Further Reduce Complexity

Large m and n will result in high hash computation cost for vector insertion and query of similar vectors. From Theorem 1, the parameters of the hash tables (i.e., m and n) are the function of the angular requirement θ. From Eq.(10), the determination of large entry is further impacted by the length of $||\mathbf{d}_i^k||$ and $||\mathbf{b}_j||$. This allows us to further prune vectors and reduce m and n.

As $\cos(\mathbf{d}_i^k, \mathbf{b}_j) \in [-1, 1]$, based on Eq.(10), the vector pair $\mathbf{d}_i^k$ and $\mathbf{b}_j$ do not need to be considered if $||\mathbf{d}_i^k||||\mathbf{b}_j|| < \varepsilon$. In this case, x_{ijk} is certainly not a large entry and we should require that $||\mathbf{d}_i^k||||\mathbf{b}_j|| \ge \varepsilon$ for the similarity search. Accordingly, we can prune some vectors and do not need to include them when building LSH hash tables.

Let $MaxLd$ and $MaxLb$ denote the maximum vector lengths of ${\mathbf{d}_i}^k$ and $\mathbf{b}_j$ respectively, where $1 \le i \le I$, $1 \le j \le J$, and $1 \le j \le K$. We can have the lower bound of $||{\mathbf{d}_i}^k||$ and $||\mathbf{b}_j||$:

$$||{\mathbf{d}_i}^k|| \ge \frac{\varepsilon}{||\mathbf{b}_j||} \ge \varepsilon({\mathbf{d}_i}^k) \overset{\text{def}}{=} \frac{\varepsilon}{MaxLb} \tag{19}$$

$$||\mathbf{b}_j|| \ge \frac{\varepsilon}{||{\mathbf{d}_i}^k||} \ge \varepsilon(\mathbf{b}_j) \overset{\text{def}}{=} \frac{\varepsilon}{MaxLd} \tag{20}$$

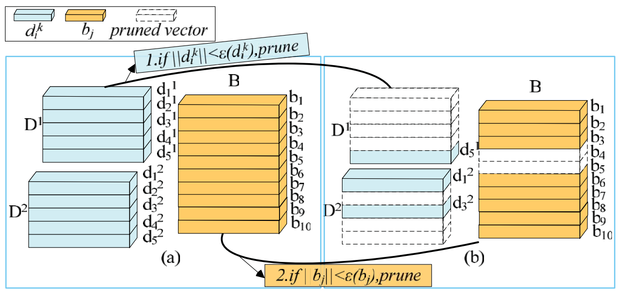


Fig. 8. Prune strategy: (a) vectors before pruning, (b) vectors after pruning.

If $||{\mathbf{d}_i}^k||$ is smaller than $\varepsilon({\mathbf{d}_i}^k)$, there are no vectors $\mathbf{b}_j$ that can satisfy ${\mathbf{d}_i}^k \bullet \mathbf{b}_j \ge \varepsilon$. So that we can prune it. Similarly, we can also prune $\mathbf{b}_j$ if its length is smaller than $\varepsilon(\mathbf{b}_j)$.

Fig.8 gives an example to illustrate the prune strategy, where $\varepsilon = 3$. There are 10 query vectors in two matrices: $\mathbf{D}^1 = \{\mathbf{d}_1^1(0.2, 0),\ \mathbf{d}_2^1(0.1, 0),\ \mathbf{d}_3^1(0.1, 0.2),\ \mathbf{d}_4^1(0.2, 0.1),\ \mathbf{d}_5^1(3, 0)\}$, and $\mathbf{D}^2 = \{\mathbf{d}_1^2(0, 3),\ \mathbf{d}_2^2(0, 0.2),\ \mathbf{d}_3^2(0, 2),\ \mathbf{d}_4^2(0.1, 0),\ \mathbf{d}_5^2(0.02, 0)\}$. There are also 10 vectors in matrix B: $\mathbf{b}_1(0, 11)$, $\mathbf{b}_2(2, 4)$, $\mathbf{b}_3(4, 3)$, $\mathbf{b}_4(0.5, 0)$, $\mathbf{b}_5(0, 0.5)$, $\mathbf{b}_6(5, 0)$, $\mathbf{b}_7(3, 0)$, $\mathbf{b}_8(0, 3)$, $\mathbf{b}_9(4, 0)$, and $\mathbf{b}_{10}(3, 4)$. According to Eq(19) and Eq(20), we obtain two thresholds $\varepsilon(\mathbf{b}_j) = 1$, $\varepsilon(\mathbf{d}_i^k) = 0.27$.

As the lengths of the query vectors $\mathbf{d}_1^1(0.2, 0)$, $\mathbf{d}_2^1(0.1, 0)$, $\mathbf{d}_3^1(0.1, 0.2)$, $\mathbf{d}_4^1(0.2, 0.1)$, $\mathbf{d}_2^2(0, 0.2)$, $\mathbf{d}_4^2(0.1, 0)$, $\mathbf{d}_5^2(0.02, 0)\}$ are less than $\varepsilon(\mathbf{d}_i^k)$, they are pruned. Similarly, as the lengths of $\mathbf{b}_4(0.5, 0)$ and $\mathbf{b}_5(0, 0.5)$ are less than $\varepsilon(\mathbf{b}_j)$, they are also pruned.

D. Building Hash Tables Using a Common Group of Hash Functions

From Theorem 1, as $p_1 = 1 - \frac{\theta}{\pi}$, $p_2 = 1 - \frac{c\theta}{\pi}$, and $\rho = -\frac{ln(p_1)}{ln(p_1/p_2)}$, obviously, the parameter m (corresponding to the length of the hash index and thus the tree height) and n (corresponding to the number of hash tables and the number of trees) of the LSH hash tables are the functions of θ and c. So we have

$$m = \log_{\frac{\pi-\theta}{\pi-c\theta}} 2N \tag{21}$$

and

$$n = (2N)^{-\frac{\ln(1-\theta/\pi)}{\ln((\pi-\theta)/(\pi-c\theta))}}, \tag{22}$$

and then

$$\theta \uparrow \Rightarrow \begin{cases} p_1 \downarrow \\ p_2 \downarrow \end{cases} \Rightarrow \begin{cases} p_1/p_2 \uparrow \\ \rho \downarrow \end{cases} \Rightarrow \begin{cases} m \downarrow \\ n \downarrow \end{cases} \tag{23}$$

That is, the tree height and the number of trees in the forest are the decreasing function of the angular requirement θ. Furthermore, according to (10), we have

$$\begin{aligned} \left\|\mathbf{d}_i^k\right\| \|\mathbf{b}_j\| \cos(\mathbf{d}_i^k, \mathbf{b}_j) \ge \varepsilon \Rightarrow \Theta(\mathbf{d}_i^k, \mathbf{b}_j) \\ \le \arg\cos\left(\frac{\varepsilon}{\left\|\mathbf{d}_i^k\right\| \|\mathbf{b}_j\|}\right) \end{aligned} \tag{24}$$

That is, given a pair $({\mathbf{d}_i}^k, \mathbf{b}_j)$, to identify that ${\mathbf{d}_i}^k$ and $\mathbf{b}_j$ are the similar vectors, the angle between these two vectors should less than $\arg\cos(\frac{\varepsilon}{\left\|\mathbf{d}_i^k\right\| \|\mathbf{b}_j\|})$. Therefore, given a

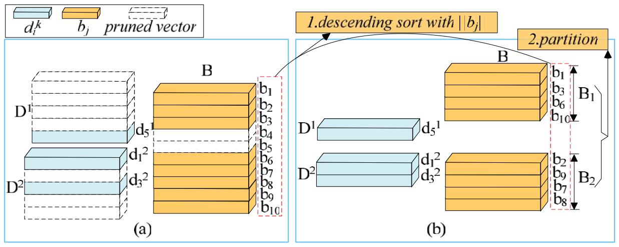
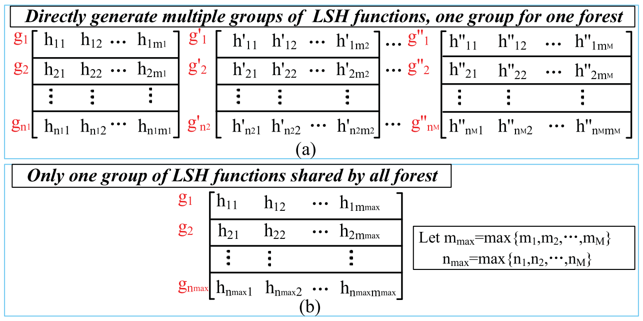


Fig. 9. Partition matrix $\mathbf{B}$: (a) vectors before partitioning, (b) vectors after partitioning.



Directly generate multiple groups of LSH functions, one group for one forest
Only one group of LSH functions shared by all forest
Let $m_{max}=\max\{m_1,m_2,\cdots,m_M\}$
$n_{max}=\max\{n_1,n_2,\cdots,n_M\}$
(a)
(b)

Fig. 10. Generate hash functions for the LSH forest.

pair ($\mathbf{d}_i{}^k$,$\mathbf{b}_j$), their angular requirement is denoted as $\theta = \arg\cos(\frac{\varepsilon}{||\mathbf{d}_i^k||||\mathbf{b}_j||})$. We further have

$$\left\|\mathbf{d}_i^k\right\| \|\mathbf{b}_j\| \uparrow \Rightarrow \theta = \arg\cos(\frac{\varepsilon}{\left\|\mathbf{d}_i^k\right\| \|\mathbf{b}_j\|}) \uparrow \Rightarrow \begin{cases} m \downarrow \\ n \downarrow \end{cases} \quad (25)$$

According to Eq.(25), m and n are the decreasing functions of ($||\mathbf{d}_i{}^k||||\mathbf{b}_j||$). Using $||\mathbf{d}_i{}^k||$ and $||\mathbf{b}_j||$ as the variables, we denote these two functions as $m = f_m(||\mathbf{d}_i{}^k||, ||\mathbf{b}_j||)$ and $n = f_n(||\mathbf{d}_i{}^k||, ||\mathbf{b}_j||)$, respectively.

Therefore, when the vector length $||\mathbf{d}_i{}^k||||\mathbf{b}_j||$ becomes large, m and n should be set to small values, and we can build a smaller forest with lower tree height and fewer trees. On the contrary, a small vector length $||\mathbf{d}_i{}^k||||\mathbf{b}_j||$ will result in large m and n values, and consequently a large forest with larger tree height and more trees.

In our design, all vectors are stored in the leaf nodes in the forest. Vectors in the matrix $\mathbf{B}$ have different vector lengths. If we store all vectors in one forest, it will require building a very large forest with the size determined by the smallest value $||\mathbf{d}_i{}^k||||\mathbf{b}_j||$. However, for those vector pairs with a large value, a small forest can be built to satisfy the requirement. Large forest means more hash computations, therefore, storing all vectors using a large forest may result in the cost from redundant computations to insert or query a vector for those pairs that have a large value $||\mathbf{d}_i{}^k||||\mathbf{b}_j||$. To reduce the cost, we propose to build multiple LSH forests with the following steps:

Step 1: Partitioning the matrix $\mathbf{B}$. We first sort the row vectors in matrix $\mathbf{B}$ according to their lengths in a descending order, then group the vectors of matrix $\mathbf{B}$ into M parts $\{\mathbf{B}_1, \mathbf{B}_2, \cdots, \mathbf{B}_M\}$, each consisting of the vectors of roughly similar length.

Step 2: Building a LSH forest for each part $\mathbf{B}_i$ with $1 \leq i \leq M$. Denoting $\mathbf{b}_i^l$ the vector with the smallest length in $\mathbf{B}_i$, we have $\mathbf{b}_i^l \geq \varepsilon(\mathbf{b}_j)$. To support similarity query from all $\mathbf{d}_i{}^k$, we use the low bound $\varepsilon(\mathbf{d}_i{}^k)$ in (19) and low bound $\mathbf{b}_i^l$ to set the tree height $m_i = f_m(||\varepsilon(\mathbf{d}_i{}^k)||, ||\mathbf{b}_i^l||)$ and the number of trees $n_i = f_n(||\varepsilon(\mathbf{d}_i{}^k)||, ||\mathbf{b}_i^l||)$ in the forest.

Step 3: Inserting vectors in $\mathbf{B}_i$ into the LSH forest corresponding to $\mathbf{B}_i$.

In Fig.9, after pruning, the remaining 8 vectors are first sorted, and then partitioned into two sub-matrices $\mathbf{B}_1 = \{\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_6, \mathbf{b}_{10}\}$ and $\mathbf{B}_2 = \{\mathbf{b}_2, \mathbf{b}_9, \mathbf{b}_7, \mathbf{b}_8\}$. After finding m_i and n_i for a LSH forest in Step 2, to insert vectors, a group of hash functions $g_1, g_2, \cdots, g_{n_i}$ with $g_j = (h_{j1}, h_{j2}, \cdots, h_{jm_i})$ should be applied to calculate the hash indexes of these vectors.

To store the row vectors in M sub-matrices, we need to build M LSH forests. However, directly building M LSH forests requires M groups of LSH functions, which would result in a high cost in both the maintenance of LSH hash functions and hash computations. In Fig.10(a), if we maintain one hash generation matrix for one LSH forest, we need M hash generation matrices to form M LSH forests.

From Section VI-B, one Cosine LSH only produces a single bit hash index, and we will exploit this feature to produce one common hash generation matrix that is shared by all M forests. More specifically, with $m_{max} = max\{m_1, m_2, \cdots, m_M\}$, $n_{max} = max\{n_1, n_2, \cdots, n_M\}$, we have the common hash generation matrix consisting of $n_{max} \times m_{max}$ hash functions from Cosine $\mathcal{H}$ (Fig.10(b)).

For each sub-matrix $\mathbf{B}_i$ with $1 \leq i \leq M$, we build the LSH forest (with parameters m_i and n_i) to store the vectors in $\mathbf{B}_i$ following two steps:

- Selecting the first n_i rows and first m_i columns from the common hash generation matrix as the hash generation matrix.
- For each vector $\mathbf{b}_j \in \mathbf{B}_i$, with the selected hash generation matrix, calculating the vector's LSH indexes in the forest to insert $\mathbf{b}_j$ into the leaf nodes.

In the example of Fig.9, the smallest length vectors in $\mathbf{B}_1$ and $\mathbf{B}_2$ are $\mathbf{b}_3$ and $\mathbf{b}_8$, respectively. The smallest-length query vector after pruning is $\mathbf{d}_3^2$. Using the pairs $(\mathbf{b}_3, \mathbf{d}_3^2)$ and $(\mathbf{b}_8, \mathbf{d}_3^2)$ to calculate the required tree height and the number of trees for sub-matrices $\mathbf{B}_1$ and $\mathbf{B}_2$, we have $m_1 = 2$, $n_1 = 3$ and $m_2 = 3, n_2 = 4$, thus we can get the common generation matrix with 4×3. Accordingly, as shown in Fig.11, we select two hash generation matrices (one is 3×2, another is 4×3) from the common hash generation matrix to calculate the hash indexes of vectors in $\mathbf{B}_1$ and $\mathbf{B}_2$. Accordingly, we have the hash indexes of vectors in $\mathbf{B}_1$ and $\mathbf{B}_2$. For example, the hash value of b_1 is $\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$, which includes three hash indexes 00, 01, and 00.

Using the selected hash functions, we build two LSH forests to store the vectors in $\mathbf{B}_1$ and $\mathbf{B}_2$, as illustrated in Fig.12. In Fig.12 (a), the forest with the tree height $m_1 + 1 = 3$ and the number of trees $n_1 = 3$ is applied to store vectors in $\mathbf{B}_1$. Fig.12 (b) contains the LSH forest to store vectors in $\mathbf{B}_2$, with the tree height $m_2 + 1 = 4$ and the number of trees $n_2 = 4$. With three indexes 00, 01, 00 in Fig.11, $\mathbf{b}_1 \in \mathbf{B}_1$ is inserted into three leaf nodes corresponding to 00, 01, 00 in three trees. To save the space, we do not need to really insert

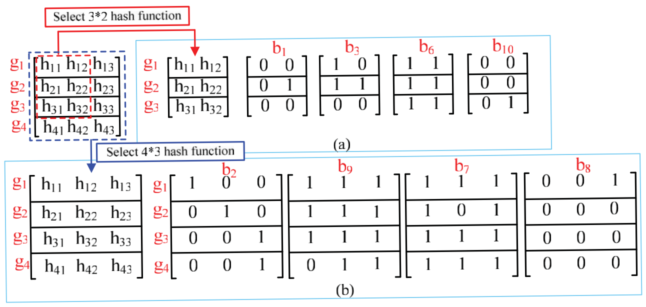


Fig. 11. Generate hash function from a common hash group: (a) select $3*2$ hash function, (b) select $4*3$ hash function.

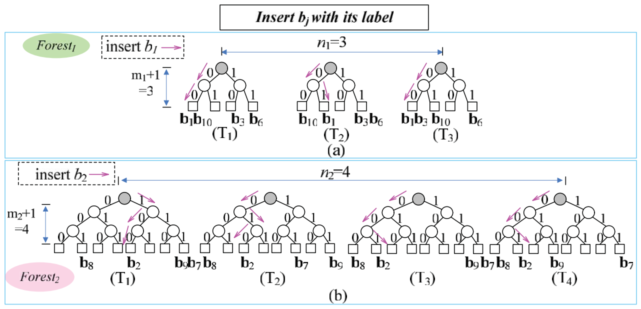


Fig. 12. Build LSH forests to store vectors in $\mathbf{B}_1$ and $\mathbf{B}_2$: (a) build LSH forest to store vectors in $\mathbf{B}_1$, (b) build LSH forest to store vectors in $\mathbf{B}_2$.

actual vectors, but only need to insert and store the vector IDs for the similarity query.

VIII. Similarity Search

We have translated the problem of large-entry search to the task of the similarity search. Given a query vector $\mathbf{d}_i^k$, we just need to find similar vectors $\mathbf{b}_j$ in $\mathbf{B}$ so that the vector pair $(\mathbf{d}_i^k, \mathbf{b}_j)$ has a high cosine similarity with $\cos(\mathbf{d}_i^k, \mathbf{b}_j) \geq \frac{\varepsilon}{||\mathbf{d}_i^k||||\mathbf{b}_j||}$.

With M LSH forests to store vectors in $\mathbf{B}_1, \mathbf{B}_2, \cdots, \mathbf{B}_M$, the similarity search should be performed against all the forests. For any sub-matrix $\mathbf{B}_i$, denoting the largest vector length in $\mathbf{B}_i$ as $\mathbf{b}_i^u$, the forest corresponds to the sub-matrix $\mathbf{B}_i$ should be searched following below steps:

Step 1 (Identifying the Hash Functions): We first find the search range (m_{iq}, n_{iq}) corresponding to the tree height $m_{iq} = f_m(||\mathbf{d}_i{}^k||, ||\mathbf{b}_i^u||)$ and the number of trees $n_{iq} = f_n(||\mathbf{d}_i{}^k||, ||\mathbf{b}_i^u||)$, and select the first n_{iq} rows and first m_{iq} columns from the common hash generation matrix.

Step 2 (Calculating the Bucket Index): With the selected hash generation matrix, we find the n_{iq} bucket indexes in the n_{iq} trees.

Step 3 (Returning the Similar Vectors): If the bucket indexes calculated correspond to the leaf nodes in the forest, our search algorithm returns the vectors of the leaf nodes; Otherwise, if the bucket indexes correspond to internal nodes, our search algorithm will return all vectors stored by leaf nodes corresponding to the descendants of these internal nodes.

In order to facilitate similarity queries from all $\mathbf{d}_i^k$, when storing vectors $\mathbf{b}_j \in \mathbf{B}_i$, we build the forest with the parameters m and n determined according to the low bounds $\varepsilon(\mathbf{d}_i{}^k)$ and $\mathbf{b}_i^l$ in $\mathbf{B}_i$. Instead, for the similarity search, we use the up

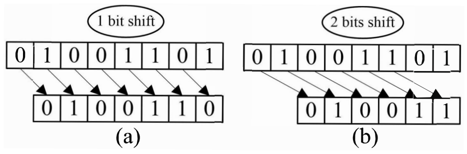


Fig. 13. Bit right-shifting operation.

bound $\mathbf{b}_i^u$ with the query vector $\mathbf{d}_i^k$ to identify the search range (m_{iq}, n_{iq}) to avoid missing the similar vectors. If $(m_{iq}, n_{iq}) < (m,\ n)$, the searching bucket indexes correspond to internal nodes in the forest.

Among M sub-matrices with M LSH forests, we denote $\mathbf{b}_{\min}^u = \min_{1\leq i\leq M} \mathbf{b}_i^u$. For a query vector $\mathbf{d}_i^k$, the largest searching range $(m_{\max}(\mathbf{d}_i^k),\ n_{\max}(\mathbf{d}_i^k))$ is determined according to the length $\mathbf{d}_i^k$ and $\mathbf{b}_{\min}^u$. The query hash indexes of $\mathbf{d}_i^k$ can be calculated using the hash functions taken from the first $n_{\max}(\mathbf{d}_i^k)$ rows and $m_{\max}(\mathbf{d}_i^k)$ columns of the common hash matrix, denoted as $g_1(\mathbf{d}_i^k), g_2(\mathbf{d}_i^k), \cdots, g_{n_{\max}(\mathbf{d}_i^k)}(\mathbf{d}_i^k)$.

In the following Theorem 2, we will show that for a given query vector $\mathbf{d}_i^k$, if we have the query hash indexes fall into the range $\left(m_{\max}\left(\mathbf{d}_i^k\right), n_{\max}\left(\mathbf{d}_i^k\right)\right)$, the corresponding query hash indexes for other forests can be easily deduced by simply performing bitwise right-shift of these indexes under $\left(m_{\max}\left(\mathbf{d}_i^k\right), n_{\max}\left(\mathbf{d}_i^k\right)\right)$. The bitwise right-shift operation is illustrated with an example in Fig.13, where a bitwise operation is performed over the binary representation of an integer. In this paper, we use right logic shift operation ($\gg$) to deduce hash indexes.

Theorem 2: Given the query vector $\mathbf{d}_i^k$, the hash indexes of $\mathbf{d}_i^k$ under the search range $(m_q,\ n_q)$ can be quickly calculated as follows

$$g_1(\mathbf{d}_i^k) >> (m_{\max}\left(\mathbf{d}_i^k\right) - m_q),$$

$$g_2(\mathbf{d}_i^k) >> (m_{\max}\left(\mathbf{d}_i^k\right) - m_q),$$

$$\cdots$$

$$g_{n_q}(\mathbf{d}_i^k) >> (m_{\max}\left(\mathbf{d}_i^k\right) - m_q).$$

where $g_1(\mathbf{d}_i^k), g_2(\mathbf{d}_i^k), \cdots, g_{n_q}(\mathbf{d}_i^k)$ are the hash indexes under the largest searching range $\left(m_{\max}\left(\mathbf{d}_i^k\right), n_{\max}\left(\mathbf{d}_i^k\right)\right)$.

The hash functions for searching range $\left(m_{\max}\left(\mathbf{d}_i^k\right), n_{\max}\left(\mathbf{d}_i^k\right)\right)$ are selected from the common hash generation matrix, as shown in Eq.(26).

$$\begin{bmatrix} h_{11} & \cdots & h_{1m_q} & \cdots & h_{1m_{\max}(d_i^k)} \\ h_{21} & \cdots & h_{2m_q} & \cdots & h_{2m_{\max}(d_i^k)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{n_q1} & \cdots & h_{n_qm_q} & \cdots & h_{n_qm_{\max}(d_i^k)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{n_{\max}(d_i^k)1} & \cdots & h_{n_{\max}(d_i^k)m_q} & \cdots & h_{n_{\max}(d_i^k)m_{\max}(d_i^k)} \end{bmatrix} \tag{26}$$

Using the hash generation matrix in Eq.(26), the hash index under $m_{\max}\left(\mathbf{d}_i^k\right)$ and $n_{\max}\left(\mathbf{d}_i^k\right)$ can be calculated as

$$g_j(\mathbf{d}_i^k) = (h_{j1}(\mathbf{d}_i^k), h_{j2}(\mathbf{d}_i^k), \cdots, h_{jm_{\max}(\mathbf{d}_i^k)}(\mathbf{d}_i^k)) \tag{27}$$

for $1 \leq j \leq n_{\max}\left(\mathbf{d}_i^k\right)$.

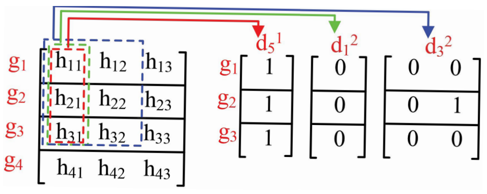


Fig. 14. Hash $\mathbf{d}_i^k$ one time for multiple LSH forests.

The hash functions for searching range (m_q, n_q) are shown in Eq.(28).

$$\begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1m_q} \\ h_{21} & h_{22} & \cdots & h_{2m_q} \\ \vdots & \vdots & \vdots & \vdots \\ h_{n_q1} & h_{n_q2} & \cdots & h_{n_qm_q} \end{bmatrix} \tag{28}$$

Using the hash generation matrix in Eq.(28), the hash index under the searching range $(m_q,\, n_q)$ is calculated as

$$t_j(\mathbf{d}_i^k) = (h_{j1}(\mathbf{d}_i^k), h_{j2}(\mathbf{d}_i^k), \cdots, h_{m_q}(\mathbf{d}_i^k)) \tag{29}$$

for all $1 \le j \le n_q$.

According to the Cosine LSH function defined in Section VI-B, the hash functions (for example $h_{j2}(\mathbf{d}_i^k)$) in both Eq.(27) and Eq.(29) generate one bit value (1 or 0). By comparing $t_j(\mathbf{d}_i^k)$ with $g_j(\mathbf{d}_i^k)$, the first m_q hash values are the same. Therefore, we have

$$t_j(\mathbf{d}_i^k) = g_j(\mathbf{d}_i^k) >> (m_{\max}(\mathbf{d}_i^k) - m_q) \tag{30}$$

The proof completes.

We use Fig.15 as an example to illustrate our algorithm on the quick search of similar vectors. Following Fig.8, after the pruning, there are three query vectors $\mathbf{d}_5^1$, $\mathbf{d}_1^2$, and $\mathbf{d}_3^2$ remained. We take $\mathbf{d}_3^2$ as an example to illustrate our similarity query process. As the largest length vector in $\mathbf{B}_1$ and $\mathbf{B}_2$ are $\mathbf{b}_1$ and $\mathbf{b}_2$, the search range in the LSH forests of $\mathbf{B}_1$ and $\mathbf{B}_2$ can be calculated using $(\mathbf{d}_3^2,\, \mathbf{b}_1)$ and $(\mathbf{d}_3^2,\, \mathbf{b}_2)$, and we have $(m_{1q} = 1,\, n_{1q} = 2)$ and $(m_{2q} = 2,\, n_{2q} = 3)$.

As $m_{2q} > m_{1q}$ and $n_{2q} > n_{1q}$, we only calculate the query hash indexes for the largest search range $(m_{2q} = 2,\, n_{2q} = 3)$ that corresponds to the second forest in Fig.15(b). From the bit shifting, we can easily obtain the query hash indexes for the first forest in Fig.15(a). For both forests, the search ranges do not reach the tree height and the number of trees of the forest. Therefore, the hash indexes can only point to internal nodes in the forest. We can easily obtain the candidate similar vectors by moving down the leaf nodes from the matched internal nodes. From the first forest in Fig.15(a), we find two candidate similar vectors $\mathbf{b}_1$ and $\mathbf{b}_{10}$. We also find two candidate similar vectors $\mathbf{b}_2$ and $\mathbf{b}_8$ from the second forest in Fig.15(b). After merging the two results, we obtain all the candidate similar vectors $\mathbf{b}_1$, $\mathbf{b}_{10}$, $\mathbf{b}_2$ and $\mathbf{b}_8$.

IX. Algorithm Analysis

A. Time Complexity

To reduce the computation cost of large entry retrieving, we transform the problem of retrieving large entries based on dot-product to the problem of query of similar items based on LSH forests. As our solution has two major processes, hash-table building and similarity query, we analyze the time complexity based on these two.

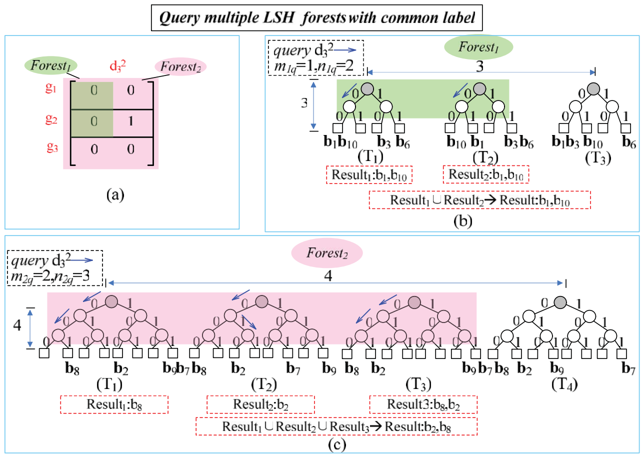


Fig. 15. Query multiple LSH forests: (a) select hash index from a common hash matrix, (b) query LSH forest 1, (c) query LSH forest 2.

1) Hash Table Building: To facilitate similarity query, we build LSH tables to reorder and store vectors in $\mathbf{B}$. As shown in Section VII-D, vectors in $\mathbf{B}$ are divided into M parts $\{\mathbf{B}_1,\, \mathbf{B}_2,\, \cdots,\, \mathbf{B}_M\}$. For each part $\mathbf{B}_i$, totally $m_i \times n_i$ hash functions are applied to find the hash indexes of each vector in $\mathbf{B}_i$, which incurs the total computation cost of $O(m_i \times n_i)$. To reduce the hash table building cost, in Section VII-C, we apply a pruning strategy to exclude the vectors that are not needed to store as they certainly do not correspond to the large entries. As a result, the total number of vectors to insert is less than J, the total number of items in B. As a result, the total hash computation cost of inserting all vectors into the hash tables is at most $O(\sum_{i=1}^{M}((J/M) \times m_i \times n_i))$ if the vectors are uniformly distributed among the M parts.

2) Similarity Query: We have M groups of LSH hash tables to store $\{\mathbf{B}_1, \mathbf{B}_2, \cdots, \mathbf{B}_M\}$. One group of LSH hash table stores one part and also corresponds to one LSH forest. To find the similar vector for the vector $\mathbf{d}_i^k$ to query, we can only calculate the hash indexes once for one forest, while deducing the hash indexes for other forests through simple bit shifting. Thus the hash computation cost is $O(m_{\max}(\mathbf{d}_i^k) \times n_{\max}(\mathbf{d}_i^k))$. As our pruning strategy in Section VII-C not only is able to prune the vector set in $\mathbf{B}$ but also prune the query vector set $\{\mathbf{d}_i^k | 1 \le i \le I, 1 \le k \le K\}$, the number of query vectors is at most $I \times K$, so the computation cost is at most $O(I \times K \times m_{\max}(\mathbf{d}_i^k) \times n_{\max}(\mathbf{d}_i^k))$.

After collecting all returned vectors in the hash buckets of all hash tables, we have a set of candidate similar vectors. For each candidate similar vector $\mathbf{b}_j$ found in step 2, calculating the dot operations $\mathbf{d}_i^k \bullet \mathbf{b}_j$, and the corresponding entry and its entry value will be returned if $\mathbf{d}_i^k \bullet \mathbf{b}_j \ge \varepsilon$. For each querying vector $\mathbf{d}_i^k$, instead of calculating the dot operations among all $\mathbf{d}_i^k$ and $\mathbf{b}_j$ pairs for $1 \le j \le J$, only $l \ll J$ dot operations are needed where l is the number of similar items found from the hashing tables. Thus, the computation cost can be largely reduced. In the simulation part, we will shown that our method based on LSH hashing can bring significantly large speed gain compared to the direct dot operations.

B. Space Complexity

Although we use LSH forest to represent hash tables for the convenience of illustrating the similarity query process,

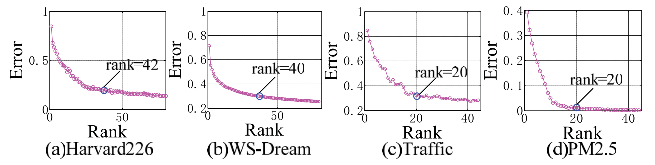


Fig. 16. Train rank: (a) Harvard226, (b) WS-Dream, (c) Traffic, (d) PM2.5.

the LSH forests are virtually built from the basic hash tables rather than physically exist. In our method, only the hash tables are physically built and stored.

Moreover, instead of storing a vector itself in the hash table, only the vector ID needs to be stored in the hash table. For the vectors in $\mathbf{B}_i$, as we build n_i hash tables to store these vectors, n_i copies of vector IDs are required to store in the hash table. Moreover, not all vectors in original matrix $\mathbf{B}$ are required to store because of our pruning strategy. The space complexity of storing vector IDs is at most $O(\sum_{i=1}^{M}((J/M)\times n_i))$ if the vectors are uniformly distributed among the M parts. The space complexity is not high.

X. Performance Evaluation

Four public data sets are used to evaluate the performance of our proposed FLTER-LSH:

- **WS-DREAM** [37] records the traffic volume between 142 users and 4500 Web services over 64 consecutive time slices, at an interval of 15 minutes. The top-k entries in WS-DREAM are the k user-service pairs whose throughputs are the largest.
- **Harvard226** [38] contains measurement data of application-level RTTs between 226 Azureus clients collected in 72 hours. The top-k entries in Harvard226 are the client pairs that have the k largest RTT latencies.
- **PM 2.5** [39] includes PM 2.5 air condition data collected every one hour in the time span of 2014-05-01 to 2015-04-30 from 437 monitoring locations in 43 cities. The top-k entries in PM2.5 denote the locations that have the worst air quality with k largest PM2.5 values.
- **Traffic** [40] includes traffic speed data collected from 142 road segments in Manhattan (New York City) every five minutes from 04:00 AM to 23:55 PM everyday during the time span from 2017-11-29 to 2018-01-11. The top-k entries in Traffic denote the road segments that have the k largest speeds.

Although we use network monitoring as an example to illustrate our algorithm, it is general and will find many practical applications. Therefore, besides two network monitoring data sets WS-DREAM and Harvard226, we also utilize one sensor network data set PM 2.5 and one transportation data set Traffic to evaluate our algorithm.

As rank R impacts the recovery accuracy [41], [42], to identify the proper rank setting for different data sets, we vary the rank R in Fig.16 with the metric **Error** $= \frac{\sqrt{\sum_{(i,j,k)}(m_{i,j,k}-\hat{m}_{i,j,k})^2}}{\sqrt{\sum_{(i,j,k)}(m_{i,j,k})^2}}$ where m_{ijk} and $\hat{m}_{ijk}$ denote the raw data and the recovered data at (i,j,k)-th element of $\mathscr{M}$, $1\le i\le I$, $1\le j\le J$ and $1\le k\le K$.

In Fig.16, as expected, the error decreases with the increase of rank, as an under-estimated rank R makes the CP decomposition far from capturing the full structure of the data sets.

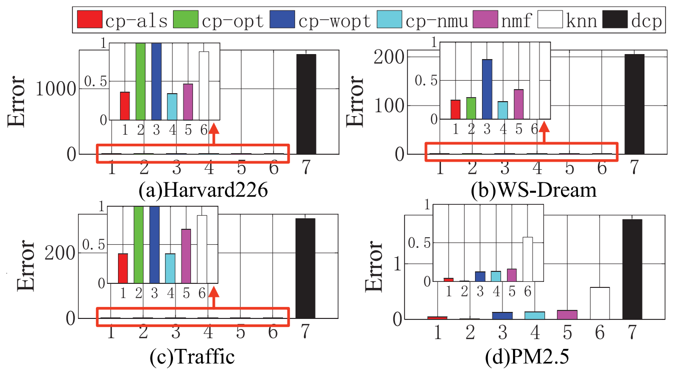


Fig. 17. Data recovery performance under different inferring algorithms: (a) Harvard226, (b) WS-Dream, (c) Traffic, (d) PM2.5.

After R reaches 20 (PM 2.5), 20 (Traffic), 40 (WS-DREAM), and 42 (Harvard226), further increasing R will not bring much gain in reducing the recovery error. Therefore, we set R to 20 (PM 2.5), 20 (Traffic), 40 (WS-DREAM), and 42 (Harvard226) in the rest of experiments.

To retrieve large entries with partial samples in a tensor, the tensor factorization parameters (factor matrices) should be first trained, then we can apply our FLTER-LSH to return the large entries. This paper does not depend on specific factor matrix training process. We implement five tensor completion algorithms CP_{nmu} [43], CP_{opt} [44], CP_{als} [45], CP_{wopt} [46], and DCP [21] to find the best factor matrices with good missing data recovery performance.

Besides the above five tensor completion algorithms, we also implement two classical inferring methods, NMF [25] (Non-negative Matrix Factorization) and KNN [47] (K-Nearest-Neighbor) to compare the recovery performance.

Fig.17 compares the recovery performance with 20% missing data. Although we use the best rank setting for DCP, its recovery accuracy is very poor compared with other CP-based tensor completion algorithms. Among other six algorithms (CP_{nmu}, CP_{opt}, CP_{als}, CP_{wopt}, KNN, and NMF), CP_{als} achieves the best performance with the lowest error for most of the data sets. In following experiments, we only present the experiment results of our FLTER-LSH with the factor matrices trained by CP_{als}.

A. Parameter Setting

Three performance metrics (precision, recall, computation time) are utilized to evaluate FLTER-LSH. We set the large entry threshold ε as the top 0.01% entry values in the data sets. The three metrics are defined as follows.

In the field of information retrieval, precision is the fraction of retrieved entries that are relevant to the query (31), as shown at the bottom of the next page.

In the field of information retrieval, Recall is the fraction of the relevant entries that are successfully retrieved (32), as shown at the bottom of the next page.

Computation time: the average number of millisconds taken to retrieve the large entries.

All experiments are run on a Microstar workstation, which is equipped with two Intel (R) Xeon (R) E5-2620 CPUs with 2GHz processor, 24 Cores and 32 GB RAM. We insert a timer to all schemes implemented.

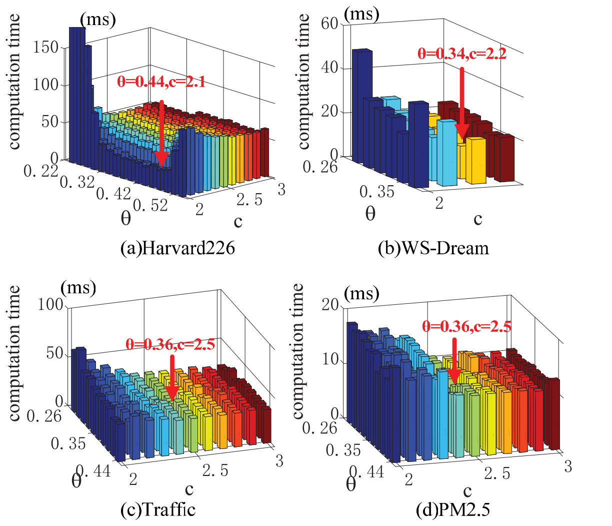


Fig. 18. Impact of the size of parameters θ and c.

In Section VII, we have known that parameters m and n impact the retrieval performance. As our main goal is to quickly retrieve the large entries, we run experiments to set proper m and n to achieve the fastest speed.

According to Eq.(21) and Eq.(22) in Section VII, we know that the value m and n are further determined by parameters θ and c. To find the proper m and n that can retrieve the top-k largest entries with the fast speed, we search for the proper θ and c. Fig.18 shows the speed of large entry retrieving by varying θ and c. We find when $\theta = 0.44$ and $c = 2.1$ for Harvard226, $\theta = 0.34$ and $c = 2.2$ for WS-Dream, $\theta = 0.36$ and $c = 2.5$ for Traffic, $\theta = 0.36$ and $c = 2.5$ for PM2.5, our FLTER-LSH achieves the fast speed of large entry retrieving.

According to Eq.(21) and Eq.(22), given θ and c, we can calculate and set parameters m and n as follows: $m = 2$ and $n = 5$ for Harvard226, $m = 5$ and $n = 9$ for WS-Dream, $m = 3$ and $n = 3$ for Traffic, $m = 2$ and $n = 3$ for PM2.5, respectively.

Moreover, given θ and c, according to $p_1 = 1 - \frac{\theta}{\pi}$, $p_2 = 1 - \frac{c\theta}{\pi}$, p_1 and p_2 can also be identified as: $p_1 = 0.86$ and $p_2 = 0.71$ for Harvard226, $p_1 = 0.89$ and $p_2 = 0.76$ for WS-Dream, $p_1 = 0.89$ and $p_2 = 0.71$ for Traffic, $p_1 = 0.89$ and $p_2 = 0.71$ for PM2.5.

B. Performance Under Large Entry Retrieving

As we are not aware any existing work studies the large entry retrieving problem in tensor field, to compare with our FLTER-LSH, we implement a straightforward solution (denoted as straightforward in Fig.19 and Fig. 20) where all un-measurement entries are recovered first before large entry retrieval.

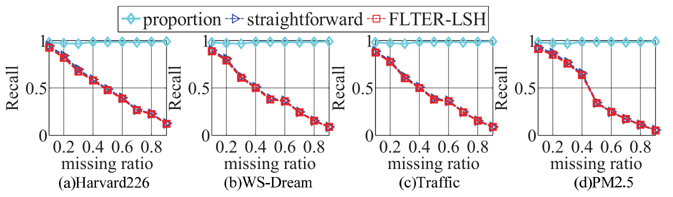


Fig. 19. Recall.

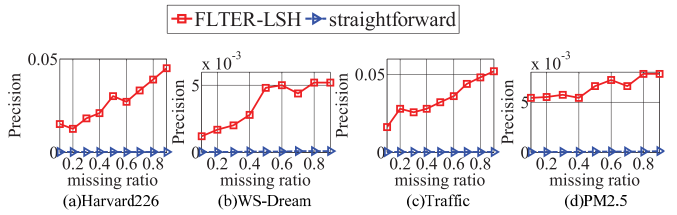


Fig. 20. Precision.

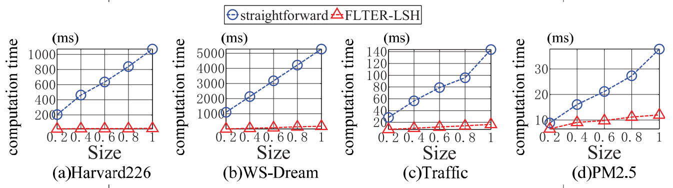


Fig. 21. Computation time under different tensor size.

In Fig.19 and Fig.20, we investigate recall and precision at different data missing ratios. From Fig.19, as the missing ratio increases, the recall of both straightforward solution and our FLTER-LSH decrease. Under all the missing ratio, our FLTER-LSH achieves nearly the same recall as that of the straightforward solution.

From Fig.20, we find the precision under our FLTER-LSH is always larger than that under the straightforward solution, which demonstrates the good large entry retrieving performance of our FLTER-LSH. This is because our hash table can effectively filter the small entries, which leads high precision in finding the large entries.

Therefore, compared with the straightforward solution, our FLTER-LSH can achieve higher accuracy in large entry retrieving with the same recall with large precision. Moreover, in Fig.21, we will show that by filtering out small entries before performing the dot production to recover the large entries, our FLTER-LSH achieves significantly faster speed to retrieve the large entries than the straightforward solution under the same tensor size. Our FLTER-LSH can be at least 60 times faster than the straightforward solution.

$$\textbf{Precision} = \frac{|\{existing\ large\ \text{entries}\} \cap \{retrieved\ \ \text{large entries}\}|}{|\{retrieved\ \text{large entries}\}|} \tag{31}$$

$$\textbf{Recall} = \frac{|\{existing\ large\ \text{entries}\} \cap \{retrieved\ \text{large entries}\}|}{|\{existing\ large\ \text{entries}\}|} \tag{32}$$

C. Speed and Scalability Testing

To test the scalability of FLTER-LSH, we change the tensor size by adding in data sequentially. In Fig.21, x-axis denotes the size ratio of the large tensor holding the whole data set. As expected, with the increase of tensor size, the computation time under both algorithms increase. However, the computation time gap between the two becomes larger with the increase of the tensor size. The computation time FLTER-LSH remains small even though tensors become larger.

Combining Fig.19, Fig.20 and Fig.21, we can conclude that our FLTER-LSH can achieve much better accuracy in large entry retrieving with significantly fast speed.

XI. Conclusion

Given incomplete measurement data, this paper aims to efficiently retrieve large entries. Taking full advantage of the multilinear structures, we can apply tensor completion to first recover the un-measurement/missing data and then find the large entries. However, recovering all un-measurement/missing data before retrieving will incur high time complexity when the size of the tensor is large. Instead, we propose to first find the candidate locations of large entries and only recover the corresponding entries. To achieve the goal, we transform the large entry retrieving problem to a cosine similarity searching problem. We propose to exploit LSH hash tables to reorder the factor vectors so that the vectors with small cosine distances are placed into the same hash bucket. We also propose a query algorithm to quickly find the similar vector of a query one so they can be applied together to find a large entry. Several novel techniques are proposed in above two processes, including the LSH table representation with the LSH forest, good hash table building to support flexible search of cosine similarity, and bit-shifting-based fast similarity query. We have conducted extensive experiments on 4 real world datasets to evaluate the performance. Compared with the solution based on conventional tensor completion, our technique is up to 60 times faster.

References

[1] A. Sivaraman *et al.*, "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 44–57.

[2] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational IP networks: Methodology and experience," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 30, pp. 257–270, Aug. 2000.

[3] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas. (IMC)*, 2004, pp. 201–206.

[4] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "Detector: A topology-aware monitoring system for data center networks," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Santa Clara, CA, USA: USENIX Association, Jul. 2017, pp. 55–68.

[5] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft, "Structural analysis of network traffic flows," in *Proc. Joint Int. Conf. Meas. Model. Comput. Syst. (SIGMETRICS/PERFORMANCE)*, 2004, pp. 61–72.

[6] Y. Vardi, "Network tomography," *J. Amer. Statist. Assoc.*, vol. 91, no. 433, pp. 365–377, 1996.

[7] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proc. ACM IMW*, 2002, pp. 71–82.

[8] M. Roughan, Y. Zhang, W. Willinger, and L. Qiu, "Spatio-temporal compressive sensing and internet traffic matrices (extended version)," *IEEE/ACM Trans. Netw.*, vol. 20, no. 3, pp. 662–676, Jun. 2012.

[9] G. Gürsun and M. Crovella, "On traffic matrix completion in the internet," in *Proc. ACM Conf. Internet Meas. Conf. (IMC)*, 2012, pp. 399–412.

[10] Y.-C. Chen, L. Qiu, Y. Zhang, G. Xue, and Z. Hu, "Robust network compressive sensing," in *Proc. 20th Annu. Int. Conf. Mobile Comput. Netw.*, Sep. 2014.

[11] K. Xie *et al.*, "Accurate recovery of internet traffic data: A sequential tensor completion approach," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 793–806, Apr. 2018.

[12] K. Xie, C. Peng, X. Wang, G. Xie, and J. Wen, "Accurate recovery of internet traffic data under dynamic measurements," in *Proc. IEEE INFOCOM*, May 2017, pp. 1–9.

[13] K. Xie *et al.*, "Accurate recovery of missing network measurement data with localized tensor completion," *IEEE/ACM Trans. Netw.*, vol. 27, no. 6, pp. 2222–2235, Dec. 2019.

[14] K. Xie *et al.*, "Accurate recovery of internet traffic data under variable rate measurements," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1137–1150, Jun. 2018.

[15] M. Ashraphijuo and X. Wang, "Fundamental conditions for low-CP-rank tensor completion," *J. Mach. Learn. Res.*, vol. 18, pp. 1–29, Jan. 2017.

[16] Q. Zhao, L. Zhang, and A. Cichocki, "Bayesian CP factorization of incomplete tensors with automatic rank determination," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 9, pp. 1751–1763, Sep. 2015.

[17] S. Gandy, B. Recht, and I. Yamada, "Tensor completion and low-n-rank tensor recovery via convex optimization," *Inverse Problems*, vol. 27, no. 2, p. 025010, 2011.

[18] W. Shao, "Tensor completion," M.S. thesis, Dept. Comput. Sci., Universitat des Saarlandes, Saarbrücken, Germany, 2012.

[19] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the CP format," *Parallel Comput.*, vol. 57, pp. 222–234, Sep. 2015.

[20] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *Proc. IEEE Int. Conf. Data Mining*, Dec. 2014.

[21] K. Xie, J. Tian, X. Wang, G. Xie, J. Wen, and D. Zhang, "Efficiently inferring top-k elephant flows based on discrete tensor completion," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 2170–2178.

[22] B. W. Bader, "Tensor decompositions and applications?" *Siam Rev.*, vol. 51, no. 3, pp. 455–500, 2009.

[23] E. J. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information," *IEEE Trans. Inf. Theory*, vol. 52, no. 2, pp. 489–509, Feb. 2006.

[24] J. Haupt, W. U. Bajwa, M. Rabbat, and R. Nowak, "Compressed sensing for networked data," *IEEE Signal Process. Mag.*, vol. 25, no. 2, pp. 92–101, Mar. 2008.

[25] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2001, pp. 556–562.

[26] J.-F. Cai, E. J. Candès, and Z. Shen, "A singular value thresholding algorithm for matrix completion," *SIAM J. Optim.*, vol. 20, no. 4, pp. 1956–1982, 2008.

[27] Z. Wen, W. Yin, and Y. Zhang, "Solving a low-rank factorization model for matrix completion by a nonlinear successive over-relaxation algorithm," *Math. Programm. Comput.*, vol. 4, no. 4, pp. 333–361, Dec. 2012.

[28] E. J. Candès and B. Recht, "Exact matrix completion via convex optimization," *Found. Comput. Math.*, vol. 9, no. 6, pp. 717–772, Apr. 2009.

[29] R. Keshavan, A. Montanari, and S. Oh, "Matrix completion from a few entries," *IEEE Trans. Inf. Theory*, vol. 56, no. 6, pp. 2980–2998, Jun. 2009.

[30] N. D. Sidiropoulos, L. D. Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Trans. Signal Process.*, vol. 65, no. 13, pp. 3551–3582, Jul. 2017.

[31] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, Aug. 2009.

[32] L. De Lathauwer, "Blind separation of exponential polynomials and the decomposition of a tensor in rank-(L_r,L_r,1) terms," *SIAM J. Matrix Anal. Appl.*, vol. 32, no. 4, pp. 1451–1474, 2011.

[33] B. W. Bader and T. G. Kolda, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Trans. Math. Softw.*, vol. 32, no. 4, pp. 635–653, Dec. 2006.

[34] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. 13th Annu. ACM Symp. Theory Comput. (STOC)*, 1998, pp. 604–613.

[35] 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 (SCG)*, 2004, pp. 253–262.

[36] K. Eshghi and S. Rajaram, “Locality sensitive hash functions based on concomitant rank order statistics,” in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2008, pp. 221–229.
[37] Z. Zheng and M. R. Lyu, “Ws-dream: A distributed reliability assessment mechanism for web services,” Tech. Rep., 2008, pp. 392–397.
[38] J. Ledlie, P. Gardner, and M. I. Seltzer, “Network coordinates in the wild,” in *Proc. 4th Symp. Netw. Syst. Design Implement.*, Cambridge, MA, USA, Apr. 2007, pp. 299–311.
[39] Y. Zheng *et al.*, “Forecasting fine-grained air quality based on big data,” in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2015, pp. 2267–2276.
[40] *The New York City Traffic Data Collections*. Accessed: 2018. [Online]. Available: http://flowmap.nyctmc.org/weborb4/flowmap
[41] Q. Zhao, L. Zhang, and A. Cichocki, “Bayesian CP factorization of incomplete tensors with automatic rank determination,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 9, pp. 1751–1763, Sep. 2015.
[42] E. S. Allman, P. D. Jarvis, J. A. Rhodes, and J. G. Sumner, “Tensor rank, invariants, inequalities, and applications,” *SIAM J. Matrix Anal. Appl.*, vol. 34, no. 3, pp. 1014–1045, 2013.
[43] B. W. Bader and T. G. Kolda, “Efficient MATLAB computations with sparse and factored tensors,” *SIAM J. Sci. Comput.*, vol. 30, no. 1, pp. 205–231, Dec. 2007.
[44] E. Acar, T. G. Dunlavy, and M. Daniel, and Kolda, “A scalable optimization approach for fitting canonical tensor decompositions,” *J. Chemometrics*, vol. 25, no. 2, pp. 67–86, 2011.
[45] B. W. Bader *et al.*, “MATLAB tensor toolbox version 2.5,” Tech. Rep., Jan. 2012.
[46] E. Acar, D. M. Dunlavy, T. G. Kolda, and M. Mørup, “Scalable tensor factorizations for incomplete data,” *Chemometrics Intell. Lab. Syst.*, vol. 106, no. 1, pp. 41–56, 2011.
[47] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp. 21–27, Jan. 1967.

Kun Xie (Member, IEEE) received the Ph.D. degree in computer application from Hunan University, Changsha, China, in 2007. She is currently a Professor with Hunan University. Her research interests include network measurement, network security, big data, and AI.

Jiazheng Tian is currently pursuing the Ph.D. degree with Hunan University. His research interest includes network measurement.

Xin Wang (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Columbia University, USA. She is currently an Associate Professor with the Department of Electrical and Computer Engineering, The State University of New York at Stony Brook, USA. Her research interests include algorithm and protocol design in wireless networks and communications, mobile and distributed computing, and networked sensing and detection. She received the NSF Career Award in 2005 and the ONR Challenge Award in 2010.

Gaogang Xie (Senior Member, IEEE) received the B.S. degree in physics and the M.S. and Ph.D. degrees in computer science from Hunan University in 1996, 1999, and 2002, respectively. He is currently a Professor with the Computer Network Information Center (CNIC), Chinese Academy of Sciences (CAS), and the University of Chinese Academy of Sciences, and the Vice President of CNIC. His research interests include internet architecture, packet processing and forwarding, and internet measurement.

Jiannong Cao (Fellow, IEEE) received the Ph.D. degree in computer science from Washington State University, Pullman, WA, USA, in 1990. He is currently the Otto Poon Charitable Foundation Professor in data science and the Chair Professor of distributed and mobile computing with the Department of Computing, The Hong Kong Polytechnic University (PolyU), Hong Kong. His research interests include parallel and distributed computing, wireless sensing and networks, pervasive and mobile computing, and big data.

Hongbo Jiang (Senior Member, IEEE) received the Ph.D. degree from Case Western Reserve University in 2008. He was a Professor at the Huazhong University of Science and Technology. He is now a Professor with the College of Computer Science and Electronic Engineering, Hunan University. His research concerns computer networking, especially algorithms and protocols for wireless and mobile networks.

Jigang Wen received the Ph.D. degree in computer application from Hunan University, China, in 2011. He was a Research Assistant with the Department of Computing, The Hong Kong Polytechnic University, from 2008 to 2010. His research interests include wireless networks and mobile computing and high-speed network measurement and management.