

Received October 30, 2018, accepted November 14, 2018, date of publication December 7, 2018, date of current version January 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2885350

Pavo: A RNN-Based Learned Inverted Index, Supervised or Unsupervised?

WENKUN XIANG¹, HAO ZHANG¹, RUI CUI², XING CHU¹, KEQIN LI³, (Fellow, IEEE), AND WEI ZHOU¹

¹School of Software, Yunnan University, Kunming 650091, China

²Tencent Technology, Beijing 100081, China

³Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

Corresponding author: Wei Zhou (zwei@ynu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61762089, Grant 61663047, Grant 61863036, and Grant 61762092, and in part by the Science and Technology Innovation Team Project of Yunnan Province under Grant 2017HC012.

ABSTRACT The booms of big data and graphic processing unit technologies have allowed us to explore more appropriate data structures and algorithms with smaller time complexity. However, the application of machine learning as a potential alternative for the traditional data structures, especially using deep learning, is a relatively new and largely uncharted territory. In this paper, we propose a novel recurrent neural network-based learned inverted index, called Pavo, to efficiently and flexibly organize inverted data. The basic hash function in the traditional inverted index is replaced by a hierarchical neural network, which makes Pavo be able to well adapt for various data distributions while showing lower collision rate as well as higher space utilization rate. A particular feature of our approach is that a novel unsupervised learning strategy to construct the hash function is proposed. To the best of our knowledge, there are no similar results, in which the unsupervised learning strategy is employed to design hash functions, in the existing literature. Extensive experimental results show that the unsupervised model owns some advantages than the supervised one. Our approaches not only demonstrate the feasibility of deep learning-based data structures for index purpose but also provide benefits for developers to make more accurate decisions on both the design and the configuration of data organization, operation, and parameters tuning of neural network so as to improve the performance of information searching.

INDEX TERMS Learning index, RNN, hash function, LSTM.

I. INTRODUCTION

The rapid developments of GPU (Graphic Processing Unit) have allowed us to explore newer data structure and faster algorithm so as to extremely improve our capacity for organizing and searching data. In the meantime, some AI (Artificial Intelligence) technologies, such as the deep learning, have already shown their advantages in intelligent information processing. However, as fundamental parts of computer systems, the traditional data structures are designed more specifically to adapt for the CPU, where the data is organized according to a fixed pattern, regardless of various data distributions. Therefore, an intuitive question raises, could we design more appropriate data structures based on deep learning and GPU? Fortunately, one latest paper has provided some interesting results about using deep learning to build index structures, see [1]. This work inspires us to explore more possibilities.

It is well known that the inverted index is widely used as a form of data organization. The basic principle of inverted index is to aggregate and index according to the attributes of items such that similar items can be quickly found for the recommendation. Generally, companies construct hundreds of inverted indexes using the same hash function, regardless of various data distributions. However, with the advent of the era of big data, how to design an efficient and flexible data organization method to improve the space utilization rate of the inverted list has become a significant problem that needs to be solved urgently. Hence, this constructs the main motivation of us to explore a more intelligent inverted index which can adapt for different data distributions.

As a personalized reading product of Tencent, Daily Express relies on the powerful content output capabilities of Tencent News and Tencent Video, backed by QQ, WeChat and other large social networks, and quickly grows up to the

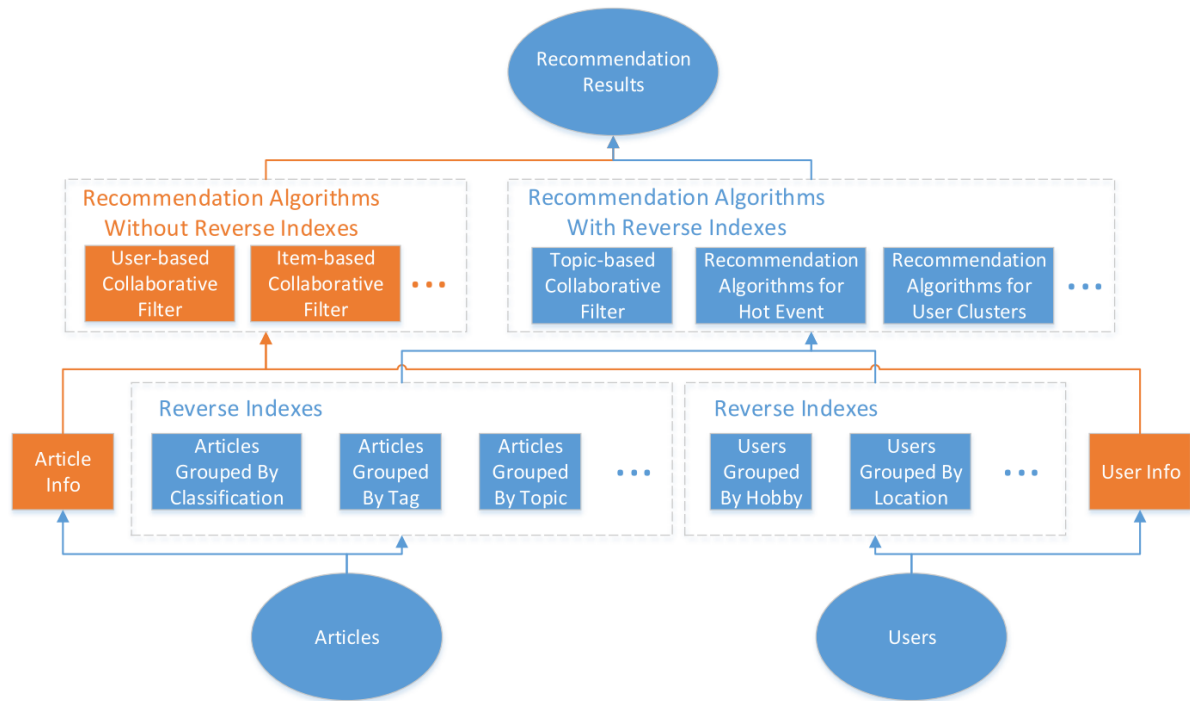


FIGURE 1. The framework of recommend systems in Daily Express of Tencent.

third largest information reading application in China within just a few years. As a real-time content recommendation system, how to find the content that users are interested in among tens of millions of resources is the main tasks of the system. In order to improve the accuracy of the recommendation, current Daily Express use commodity-based and interest-based collaborative filtering strategies shown as below.

As shown in Fig. 1, in the current developing group of Daily Express, many recommendation algorithms work together which leads to consume huge computing resources, especially there exists many trivial calculations in the process of computing similarity. Therefore, a large number of inverted indexes are built up to speed up similarity computing. And the number of these inverted index tables is roughly around 350.

Based on our long period of implementation and a large number of different business applications, we obtain some interesting observations as follows.

(1) The inverted index generally uses the hash function to construct the index where the hash function can find the location of the random key with $O(1)$ time complexity. Taking the popular in-memory database Redis as an example, Redis uses the Murmur hash function is used. When the maximum load factor is 1.0, a hash table with length 2^n is created for a data set with N pieces of data (n is a positive integer, $2^n > n > N > 2^{n-1}$). Although Murmur hash is an efficient hash function with a low collision rate, its average number of lookups is typically between 1.35 and 1.5 in a data set. While the ideal situation is that the data can be uniformly mapped by hash function, and the lowest average search time can be

reduced to 1.0., which indicates that the method exists some potential optimization space.

(2) The distribution of data has also a very significant impact on the outputs of the hash. In the english word list, the number of words, which start with “s” and the second place is “h”, is far more than the ones which start with “s” and the second place is “v”. It is well known that this kind of data is unevenly distributed in most of the data. However, the existing hash functions, such as Murmur hash, BKDRHash, etc., treat each character in the string separately, without considering the links between the characters. It will be helpful for us to evenly map the data to the hash table if we can previously learn the distribution of data in advance, and increase the distances between these similar data in the hash table.

(3) In most inverted indexes constructed by current recommendation systems, each index table has its own unique data distribution. Moreover, traditional indexing schemes often adopt the same hashing function (even if they can have different random number seeds). Such an indexing scheme may result in a lower collision rate in partial inverted tables, but is not universally applicable due to.... Therefore, customized inverted indexes are highly appreciated and can greatly reduce the possibility of conflict.

Based on the above discussions, we propose a hierarchical neural network based inverted index named as Pavo. The main contributions of this paper are reflected as follows:

- A novel RNN-based learned inverted index “Pavo” is proposed. To the best of our knowledge, it is the first

work to design an intelligent inverted index by means of RNN, which is used to split the data set into sub-data sets. In this way, the complexity of each model can be reduced while the index accuracy can be improved.

- ▶ Supervised and unsupervised learning strategies, in the learned inverted index, are constructed, respectively. Based on our experiments, it is found that unsupervised learning strategy can find a more uniform distribution and reduce the data collision rate in most data sets compared with the supervised inverted index. This is a new reveal after the seminal work [1].
- ▶ The framework and algorithms of Pavo are designed in detail, while the extensive experiments are conducted to evaluate the performance of Pavo. With different parameters and configurations, the efficiency and flexibility of Pavo are demonstrated.

The rest of the paper is organized as follows: The related works are introduced in Section II. Section III discusses the supervised and unsupervised model. The experimental results are shown and analyzed in Section IV. Finally, the paper is concluded in Section V.

II. RELATED WORK

Although using data structures in machine learning is not a new idea, the application of machine learning as a potential replacement for traditional data structures, especially using deep learning, is a relatively new and largely uncharted territory. Our work builds upon a wide range of previous outstanding research. In the following, we intend to outline several important interactions between machine learning and indexes. An inverted index is an index data structure consisting of a list of words or numbers, which is a mapping from documents [2]. The purpose of an inverted index is to efficiently generate a list of keyword vectors, in which hash functions are widely used. Afterwards, texts are stored in the data structure that allows for very efficient and fast full-text searches. For the moment, inverted index has been intensively studied and used in many different fields, such as search engine [3], information retrieval systems [4], bioinformatics [5], etc.. However, as far as we know there is no attempts to build an inverted index by using neural network instead of the hash functions in it. The explosion of workload complexity and the development of AI call for new approaches to achieve more efficient and intelligent computing. Machine learning has emerged as a powerful technique to address computer optimization. Most recently, researchers have been beginning to employ machine learning techniques for the optimization of indexes and hash functions. There has existed a lot of research on emulating locality-sensitive hash (LSH) functions, to build Approximate Nearest Neighborhood (ANN) indexes, ranging from supervised [6]–[11] and unsupervised [12]–[16] to semi-supervised settings [17]. These kinds of methods incorporate data-driven learning methods in the development of advanced hash functions. The principle of these works is a kind of “learn to

hash”, which means to learn the information of data distributions or class labels in order to guide the design of hash function. However, the basic construction of the hash function itself is not been changed. Therefore, those methods cannot be directly used to construct the fundamental data structures. As far as we know, paper [1] is the seminal work to develop a “learned index” which explores how neural networks, can be used to enhance, or even replace, traditional index structures. It provides a neural network based learned index to replace the B-Tree index and further discusses the differences between learned hash-map index and the traditional hash-map index. Moreover, experimental results show that the learned index owns significant advantages over traditional indexes. Inspired by the paper [1], we propose an RNN-based learned inverted index. Further, different with the supervised approach in the paper [1], an unsupervised scheme is studied. In addition, experiments are conducted and the results show that our unsupervised solution can find a more uniform distribution and reduce the data collision rate in most data sets.

III. THE FRAMEWORK OF RNN-BASED LEARNED INDEX

The index structure and the machine learning model are traditionally thought to be different approaches. The index structure is constructed in a fixed manner, and the machine learning model is established on the probability forecasting. However, the principles of these two methods both concern some kinds of spatial position locating or searching. The hash index can approximately be seen as a regression, because it works based on where the key prediction data is located, or a classification in which box the prediction data can be placed based on the key. Therefore, essentially hash indexes and neural networks possess some potential relationship. In this section, we present our RNN-based indexing framework, mainly focusing on analyzing the proposed Disperse Stage, Mapping stage, and comparing the differences between supervised and unsupervised learning strategies.

A. THE FRAMEWORK OF RNN-BASED LEARNED INDEX

The ideal hash table requires to have efficient query efficiencies and space utilizations. Intuitively, it is worth to adopt an end-to-end neural network approach to simulate the whole hash function. In this kind of approach, for any arbitrarily complex data set distribution, the key values in theory can be sorted and inputted into a neural network with appropriate parameters. After a sufficient number of iterations, a very good space utilization rate can be obtained. However, with increasing parameters it is not only difficult for training, but also greatly reduces the efficiency of the search speed. As an index structure, finding data quickly and accurately is the most important issue, so we intend to use a hierarchical framework instead of a single large-scale neural network to build the index.

As Fig. 2 shows, we propose a hierarchical neural network to build the index. It firstly uses a RNN neural network to split the data set into sub-data sets. In this way, the complexity of each model is reduced and the index accuracy is also

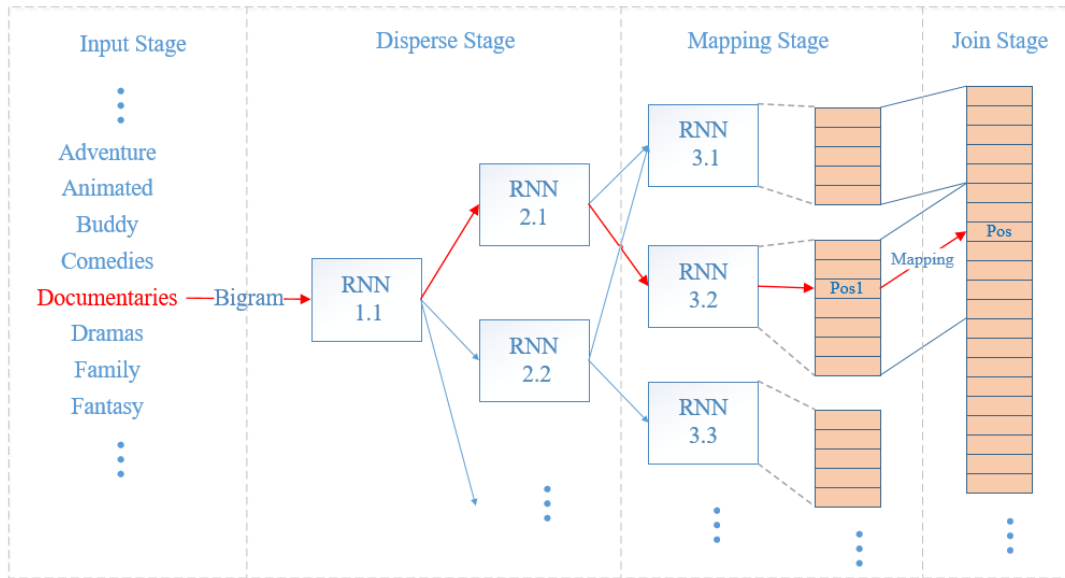


FIGURE 2. The framework of RNN-based learned index.

improved. Then, in the last step as shown in Join Stage, the data in the sub-data set is mapped from the smallest to the largest in the dictionary sequence to the hash table to complete the function of the entire hash model. Each RNN sub-model in each layer of the model randomly initializes parameters, so that the cyclic neural network learns different relationships between characters, splits the data set from multiple angles, and achieves a lower conflict rate.

As Fig. 2 shown above, the entire framework is divided into four stages: Input Stage, Disperse Stage, Mapping Stage and Join Stage. The supervised strategy is used in both Disperse Stage and Mapping Stage. And the unsupervised strategy can only be used in Mapping Stage.

1) INPUT STAGE

The input data is the key values (mainly string type) in the existing inverted list of Tencent Daily Express. We introduce the n -gram split method to preprocess the input data. One input data $I = k_n^{g1}, k_n^{g2}, \dots, k_n^{gs}$ is defined as a string fragments set from the same key-value string, where s is the window size and there is overlap between every two input fragments (i.e. $k_n^{gi} \cap k_n^{gi+1} \neq \emptyset$). Therefore the entire input data are $I^* = I_1, I_2, \dots, I_m$, where m is the size of the data set. In our approach, n is selected as 2, which means that it is processed in bigram mode.

2) DISPERSE STAGE

In this stage the split layer is composed of one or more RNN models. Our purpose is to encode preprocessed input strings and evenly distribute the codes in the vector space. The function of each model of the Disperse stage is to split the data set into multiple sub-data sets so that it is benefit for the smaller subsequent models to learn the mapping relationship.

For example, suppose we need to split the total spatial space c into r sub-space, where there are c/r sub-data sets. Two-tier Disperse stages are used in Fig. 2. The first layer splits the data set into l sub-data sets. Then there will be a l number of models in the second layer. Each model is responsible to a separate data set. Each model in the second layer then splits the respective data set into j sub-data sets, where $l*j = r$

3) MAPPING STAGE

The mapping layer is also implemented by multiple RNN models. The purpose of this layer is to map the sub-data sets output from the splitting stage into the local hash space, and find local hash location for each data in the output data set. We provide an unsupervised strategy in this mapping stage which makes our practice quite different from the first work [1].

4) JOIN STAGE

Finally, after the entire framework is setup, each RNN model in the mapping stage is serially connected from low to high, and the local hash space is connected to form the final hash table. Suppose we totally have i sub-data sets in the last layer of mapping stage, which means we have i mapping models in the mapping stage. Each sub-data set has d_i number of data, then the p -th position in the i -th local space will be mapped into the global hash table space:

$$Pos = \sum_{j=1}^{i-1} d_j + p$$

As a whole, the entire framework consists of multiple RNN models. Each RNN model consists of a single-layer LSTM layer and one or two full-connection layers. The entire framework is trained stage by stage from left to right, and multiple

models in each stage can be parallel trained. The first level in Disperse Stage uses the entire data set as training data, and the latter layer uses the sub-data set splitted from the previous layer. In the end, the data locates the position in the last layer of the Mapping Stage. During training, parameters are initialized randomly for each separate RNN model. Therefore, each RNN can extract different features from key values and split data sets from multiple dimensions.

The red line in the Fig. 2 above describes a complete prediction process of the proposed framework: (1) The word, shown as “Documentaries”, is splitted into do, oc, cu,..., es using bigram, and are sent to the input layer RNN1.1 in the Disperse stage. The output of RNN1.1 is a number, which indicates the next model. In this case the calculated result is referring to RNN2.1 (2) The “Documentaries” word is transferred to RNN2.1 and recalculated using bigram. (3) Afterwards, the “Documentaries” word is transferred to RNN3.2 and the key Pos1 in the local hash space is located. (4) Pos1 is mapped to the global space in the entire hash table at last. Although the entire framework has a large number of parameters, in a key value prediction process, only one sub-model will be selected for calculation in each layer. Therefore, compared to a single model, the calculation amount of hierarchical model in the process of prediction is greatly reduced.

B. MODELS IN DISPERSE STAGE

In this subsection we describe the details of the supervised learning strategy in the Disperse Stage

The purpose of Disperse Stage is to encode preprocessed input strings, evenly separate the codes in the vector space, and then evenly split them into sub-data sets. Due to the fact that the key distribution in each sub-data set may be different. Therefore, we adopt multiple neural network modules to learn the key distribution. As shown in Fig. 2, we have RNN2.1, RNN2.2,... etc. Since each sub-model in Disperse Stage is randomly initialized, the neural network learns different relationships between characters, splits the data set from multiple angles, and achieves a lower conflict rate.

In the traditional hash index, the specific location of each data is determined by a hash function. Therefore, after the key has undergone operations such as multiplication and shifting, the hash functions modulo the length of the hash table to obtain the final position value. Generally speaking, only one fixed hash function can be used for an inverted index. However, no matter what kinds of hash functions are selected, it is impossible to give full consideration for all data distribution, so as to avoid conflicts. In the recommended applications field, in order to quickly read the inverted data, it often stores the inverted tables in the Redis, which uses a high-performance, low-impact Murmur hash to create indexes.

The neural networks in the Disperse Stage are applied supervised learning strategy. We need to give a standard position value for each key as a learning target, and use the gradient descent optimization parameter to reduce the gap between the standard position and the output position until the

model converges. For the convenience of comparison, we use the Murmur hash to construct standard values for training during the Disperse Stage.

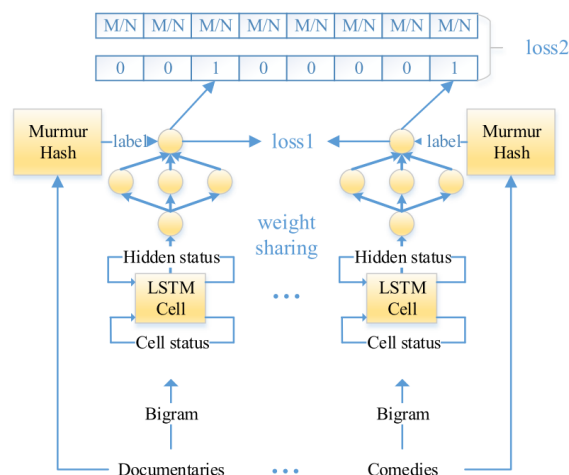


FIGURE 3. The supervised learning strategy in Disperse Stage.

As shown in the Fig. 3, we need to first determine the number M of models in the next layer according to actual requirements, that is, we need to split the data set into M copies. Each model contains a layer of LSTM and a layer of full connections, which can also be set to 2 layers. After inputting the key value sequences, LSTM first encodes the input string to obtain the feature vector F , then fits feature F through the first layer full-connection and finally output an integer value $i \in [0, M)$, which means that the piece of data is assigned to the i -th model in the next layer for processing. The training process uses Murmur hash to generate the target value and calculates the gap between the target value and the predicted value to optimize the network model.

Loss Function Selection: The purpose of Disperse Stage is to learn a series of parameters so as to find the next layer as best as possible to process the data. After long term exploration, two loss functions are used in our training process. First of all, we hope that the prediction value of the model can be as close as possible to the target value generated by Murmur hash. We measure this in terms of mean squared error, as shown in Eq. (1):

$$loss1 = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \tag{1}$$

In Eq. (1), y_i is the Murmur hash target value which shows as label in Fig. 3 and the \hat{y}_i the prediction value.

On the other hand, we hope that the amount of data handled by each model in the Disperse Stage will be as uniform as possible. In order to avoid the extremely imbalanced amount of data processed in the later layer, an additional loss2 is designed, which we call uniform distribution loss. During the calculation of the loss2, the data contained in each batch is firstly counted the amount of data k assigned in each slot

time, and then is computed to obtain the difference between k and the theoretical average distribution in each slot. This difference works as an optimization goal, as shown in Eq. (2):

$$loss1 = \sum_{j=1}^M \left(k_j - \frac{N}{M} \right)^2 \quad (2)$$

where N is the training batch size, M is the number of split slots (or number of next-level models) and $k(j)$ is the amount of data in the j -th slot for this batch of data.

During the training process, we integrate the above two loss functions as the overall optimization goal of the model:

$$L = (w_1 loss1 + w_2 loss2) \quad (3)$$

where w_1, w_2 are used to control the degree of fit hyperparameter. As Eq. (3) shows the overall loss is divided into two parts: the first part is to fit the Murmur hash and the second part is to reduce the gap between each slot and the theoretical average distribution.

C. MODELS IN MAPPING STAGE

In this subsection, we introduce our implementation of two mapping strategies: supervised mapping and unsupervised mapping.

The purpose in this stage is to map each piece of data into the local hash space, and try to avoid the conflict. Note here we don't care about the order of the mapped data. This is different from B-tree, where it is necessary to maintain the order of data. We also hope the conflict rate can be as small as possible or not at all perfectly.

D. SUPERVISED MAPPING

In the Mapping Stage, when the loading factor is set to 1, the number of possible output values is equal to the amount of data. It is well known that there will be a considerable part of the hash space without data if we use hash function mapping, which causes space waste and increases the average number of searches. Here we choose to arrange the key values from the smallest to the largest, and place them one by one in the hash space to ensure that the key values and hash space locations are in one-to-one correspondence.

In the mapping layer, the dataset is the result of the splitting of the previous layer. The RNN used here is the same as the disperse stage. However, the label is different. The loss function is described as below:

$$loss = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i) \quad (4)$$

where N is the training batch size, \hat{y}_i is the output value of the recurrent neural network and y_i is the sorting result of the key value. As shows in the right side of Fig. 4, words are sorted and the labels are generated by the order number of the list. So the label of "Adventure" is 1, and the label of "Buddy" is 3, and so on.

Theoretically, if there are enough parameters in the neural network and after enough iterative optimization, it can ensure

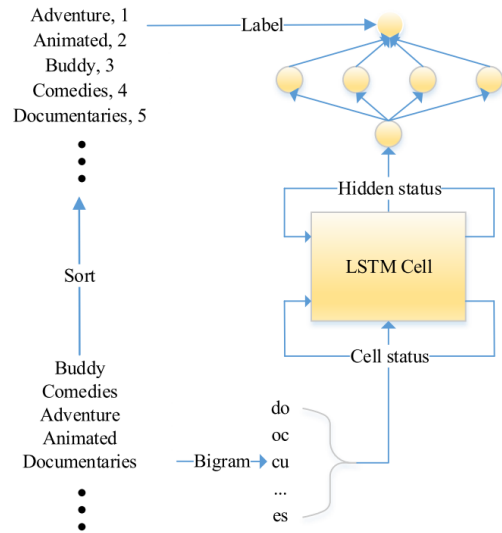


FIGURE 4. The supervised strategy in mapping stage.

that the neural network can perfectly learn this mapping relationship. However, in order to speed up the efficiency of the index, the amount of parameters in the neural network cannot be too much, so we need to balance the parameters and the conflict rate in the mapping layer.

E. UNSUPERVISED MAPPING

The previously supervised approach, which tags data according to a random hash function or a dictionary sequence of data key values, is essentially man-made method to specify the distribution of data. And it does not naturally satisfy the distribution of the data itself. Therefore, we consider whether it is possible to specify the location of each key and find the distribution that best fits the data automatically through the neural network.

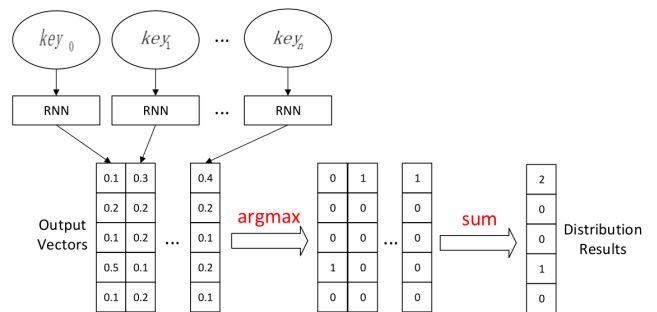


FIGURE 5. The unsupervised strategy in mapping stage.

As Fig. 5 shown, we propose an unsupervised neural network approach, which consider the process of locating the position of the key value on the hash table as a classification problem. The numbers of categories are equal to the length of the hash table. As shows in Fig. 5, the keys are preprocessed by bigram and inputted into the RNN networks. Each RNN

network outputs a vector and we take the position of the maximum value of the vector as the classification of the key value. Then it calculates the classification of each key value on the entire data set and sum the results to get the numbers of key values in each category. For example as shown in Fig. 5, after operating the key0, the first RNN network outputs a vector, which is the first column in the matrix. Then we select the maximum value, which is 0.5, in this column and setup it to 1. Other values in this column are set to 0. And so on key1 to key n . After this, we sum the values in each row to get the final categories distribution.

The loss function of the unsupervised approach is defined as:

$$\text{loss} = \frac{1}{N} \times Y \times (D - 1) \quad (5)$$

Among them, N is the data set size, Y is the output value of the neural network, and D is the numbers of key values in the category. Gradient descent is used to optimize the sum. When the number in each category equal 1, the loss value is 0, which means that it finds a hash model that makes the data set evenly distributed. The training algorithm is described as below:

Algorithm 1 Unsupervised Hash Function Training Strategy

```

Input: training sample Keys, iterations I
Output: trained Index
Training:
1 build a network model with one LSTM layer
  and one to two fully-connected layers
2 For i = 1 to I do
3   logits = network_model(Keys)
4   index = argmax(softmax(logits), axis=1)
5   hash_results = sum(index, axis=0);
6   loss = logits * hash_results;
7   back propagation to minimize loss;
8 return index

```

As above algorithm1 shows that it first establishes a recurrent neural network containing one LSTM layer and one or two full-connection layers with random parameters initialization. Keys are inputted during forward propagation, and the output logits (line 3) of the neural network is calculated, the output logits is a list of vectors shape of $[M, N]$, where M stands for the size of the dataset, and N stands for size of final hash table. For each output vector, finds the position of the largest value in the vector as the index value of the Key (line 4). Then accumulates all the vectors, results in the hash result of these keys in the i th iteration(line 5). After this the hash results is used to construct the loss function, and the gradient descent is used to optimize the loss function to get the final index model.

IV. EXPERIMENTS AND ANALYSIS

A. EXPERIMENT SETUP

The experiments were setup on a machine with 64GB main memory and one 2.6GHZ Intel(R) i7 processors. Two GTX1060 GPU card are installed and each of them has 16G

GPU memory. RedHat Enterprise Server 6.3 with Linux core 2.6.32 was installed, and we use tensorflow for experiments. For each experiment, we ran ten cases, where the median of the ten cases was used as the real performance.

In the experiments, we randomly generated data sets of three common distributions (uniform distribution, normal distribution, and long tail distribution) to test the learned index model and validate our results using a real data set. We hope to compare the advantages and disadvantages of learned index models with traditional hash functions when the data size go up to one million or more.

We compare four kinds of data structures in our experiments:

1) BKDR HASH

BKDR Hash is an algorithm invented by Brian Kernighan, Dennis Ritchie, which is widely used for string processing.

2) MURMUR HASH

Murmur Hash is a non-cryptographic hash function suitable for general hash-based lookup. Redis uses the Murmur hash function as default.

3) SUPERVISED RNN-BASED INDEX

Our proposed RNN-based index, where the supervised algorithm used in both the mapping stage and the disperse stage.

4) UNSUPERVISED RNN-BASED INDEX

Our proposed RNN-based index, where the unsupervised algorithm used in the mapping stage, while it is still use supervised algorithm in the disperse stage. The Murmur hash and BKDR Hash are implemented in Python, and the intelligent indexes are implemented in Tensorflow and called directly in Python.

B. EXPERIMENT ANALYSIS

During the process of design and evaluation, it is observed that the size of the mapping layer has significant impact on the performance of the learned inverted index. To verify the observations, the average search time and the remaining space under different mapping size are obtained by experiments, where the data are generated with four different distributions (i.e. uniform distribution, normal distribution, long tail distribution and Tencent real inverted index). Here we choose the common character hash function BKDRHash and Murmur hash used in Redis as a comparison. Fig. 6(a) and Fig. 6(e) respectively reflect the average number of lookups and the remaining space for data sets of different mapping layers. As shows in fig. 6a, when the mapping size of supervised index is 100, the average number of search times is 1.0, which means that one-to-one mapping of data is implemented without conflict. Accordingly, the remaining space in the Fig. 6(e) is 0.0. Also, the average number of lookups for supervised index has risen slowly between size 200 and size 2000 in Fig. 6(a). However, it grows up sharply after 2000. But even at 5000, the lookup number is still lower than that

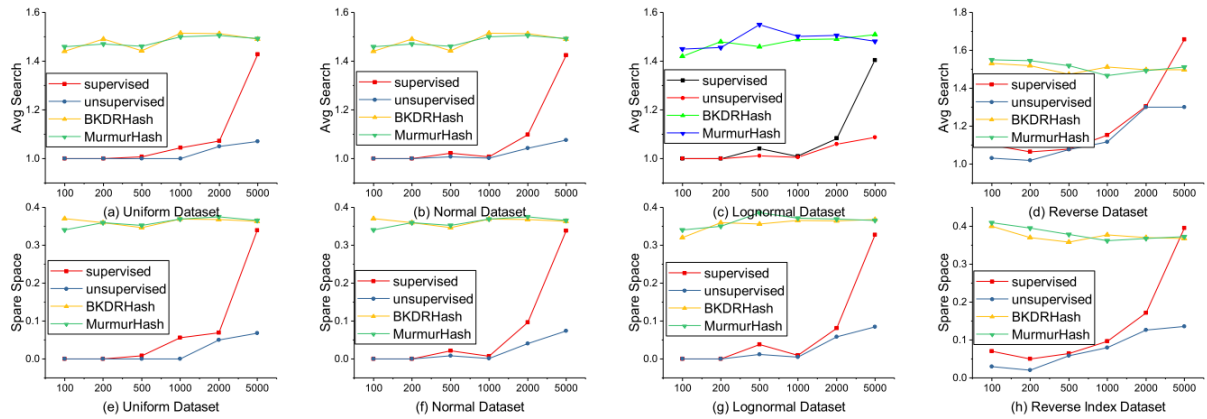


FIGURE 6. Evaluation of mapping layer's size with different data distributions.

of BKDRHash and Murmur hash. As for the unsupervised index in the same Fig. 6(a), its performance is relatively stable compared to the supervised model. Until 1000, the average number of searching is still 1.0, the corresponding space in the Fig. 6(e) is 0.0. At the size 5000, unsupervised index can still be well maintained at 1.05, and in the corresponding Fig. 6(e) the remaining space is 0.07, which is significantly lower than supervised index with 0.8 remaining space. Similarly, on the normal and long-tailed distribution datasets (Fig. 6(b), Fig. 6(c), Fig. 6(f), and Fig. 6(g)), supervised and unsupervised approaches reach the ideal value when the mapping size is less than 1000, where average number of searches is 1.0, and the remaining space is 0.0. When the mapping size is greater than 1000, supervised learning rises rapidly, while unsupervised learning grows much slower. In the real data set (Fig. 6(d), Fig. 7(h)) which has more complicated data distribution, the average search time of supervised and unsupervised index are slightly higher than these in Fig. 6(a)-Fig. 6(c), Fig. 6(e)-Fig. 6(g) between 100 and 1000. However, curve of supervised index rises faster after size 2000, and it even exceeds BKDRHash and Murmur hash at size 5000, which mean supervised index get worst performance at size 5000.

From Fig. 6, we come to know that both the supervised and unsupervised index have lower search time and higher space utilization when the mapping size less than 2000. Specifically, performance of the unsupervised index is better than the supervised one. It is also observed that the average search time of learned indexes is almost 1, which closes to the ideal state, when the mapping layer's size is less than 1000. Therefore, we choose 1000 mapping size in our final network.

The average number of lookups is an important indicator for evaluating the hash function. After selecting the mapping size 1000, we further test the average search time and give boxplots. From Fig. 7(a)-Fig. 7(d), we can see that the average search number of supervised or non-supervised strategies are significantly lower than the one of traditional BKDR Hash and Murmur hash. Among them, in the randomly

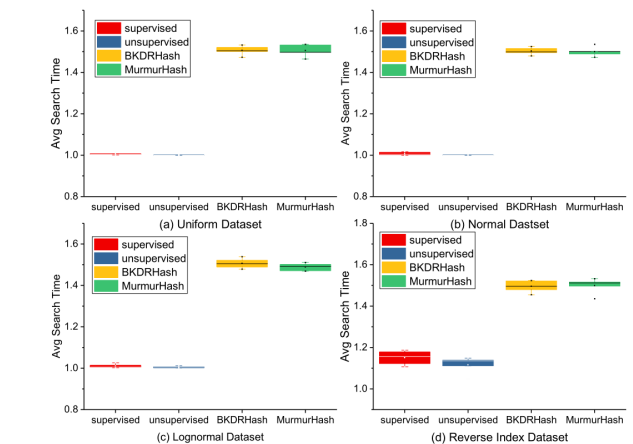


FIGURE 7. Comparison of the average number of lookups in different data distributions.

generated uniform distribution Fig. 7(a), the average number for supervised and unsupervised strategies is almost 1.0, while BKDR Hash and Murmur hash are both around 1.5. And this trend is basically maintained in the normal distribution and long-tail distribution data sets. In the real data set, BKDR Hash and Murmur hash still remained around 1.5, while the average number of supervised and unsupervised strategies rose slightly up to 1.15. In the three datasets from Fig. 7(a)-Fig. 7(c), the learning indexes (i.e. supervised and unsupervised) fluctuate very light and the results are extremely stable. In the real inverted index data set on Fig. 7(d), the fluctuate effect of traditional hash function is basically the same as the previous three simulated data sets, however the learning models are more volatile. This shows that for a regular distribution of Fig. 7(a)-Fig. 7(c), neural network can learn well, but the distribution of real data is more complex and the fitting ability of neural network will decrease. This also suggests that a potential method is to increase the number of parameters in the neural network to reduce the average number of search times and free space of

the model in large data sets, but the cost of increasing the amount of parameters will increase the training and testing time of the model. The results of Fig. 7(a)-Fig. 7(d) also show that unsupervised index has less fluctuation than supervised index, which means that unsupervised learning strategy has better abilities to adapt the data distribution. Neural network parameters are important indicators that affect the speed of network training and final searching. We further evaluate the relationship between the network parameter amount and the average number of lookups. Because in Fig. 6, when the data is 1000, the average number of lookups under the three ideal distribution conditions is almost the same, so we only select the long tail distribution and the real data for testing. In the ideal long-tail distribution map Fig. 8(a), it can be seen that when the model parameter is 2000, the average number of supervised index reaches 1.4, indicating that the conflict rate is relatively high. While unsupervised index only has 1.08 at this time. With the increasing of parameters, the conflict rate of supervised index is significantly reduced. At 8500 point, supervised learning was slightly higher than unsupervised index. When the model parameters reached 18,000, supervised index reaches the ideal value of 1.0. The average search times of unsupervised index is very low at first. When the parameter reaches 4000, it already get its ideal value of 1.0. and then remain its curve trend afterwards. In the real data Fig. 8(b), unsupervised index shows a lower conflict rate at the beginning, and the average number of lookups is already close to 1.0 at 2000 point. In supervised index, when the number of parameters is 2000, the average number of searches is still about 1.4. At a parameter of 10,000, both supervised and unsupervised indexes achieve good results close to 1.0. Although the parameter amount is 18000 in the ideal distribution Fig. 8(a), the average number of lookup times for supervised index is lower. But when the parameter is 8500, the model size will be enlarged 2.1 times ($18000/8500=2.1$), and the training time of the model is also increased. Therefore, according to the test results of Fig. 8, in the subsequent experiments, we choose 10000 of our parameters in the real dataset as showed in Fig. 8(b), and we choose 8500 parameters under the ideal distribution as showed in Fig. 8(a).

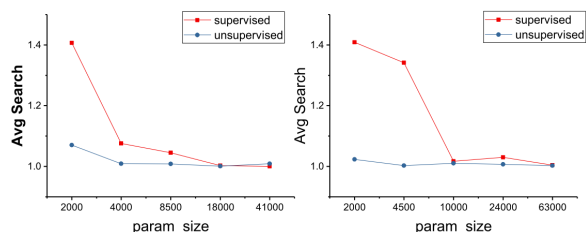


FIGURE 8. Evaluation of neural network parameters in the mapping layer.

The effect of mapping layer will be affected by the size of the data set, so in the Disperse Stage it is necessary to split the data as evenly as possible to ensure the learning ability of mapping layer. Therefore, in this set of experiments,

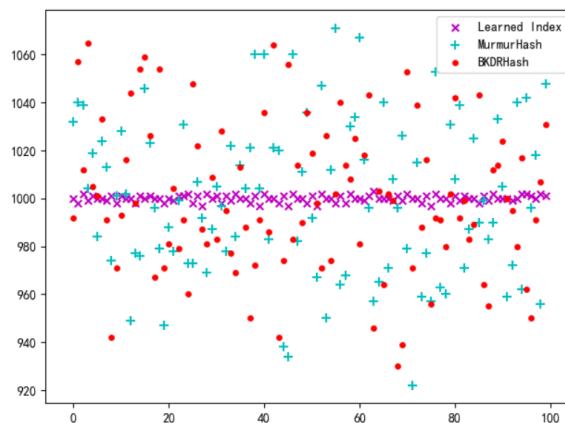


FIGURE 9. Evaluation of performance of Disperse Stage.

we analyze the performance of Disperse Stage, where one hundred thousand data are split into 100 parts by different methods. In Fig. 9 abscissa represents 1 to 100 data, and ordinates represent the number of data in each part. As Fig. 9 shows, the maximum value of Murmur hash is 1070 and the minimum value is 921. And the fluctuation is very large. The highest value of BKDR hash is 1066 and the lowest value is 928, which slightly better than Murmur hash. While if we look at our proposed disperse layer, its performance is very stable. The highest value is 1004 and the lowest value is 997. The amount of data split by it is close to 1000. This experiment shows that the learned index structure can well divide the data evenly.

We conduct a series of experiments for comparing the RNN-based index with BKDR hash and Murmur hash based indexes. 1 million randomly generated long tail distributed data are used to construct different indexes. Table 1 shows the average search number of learning index is slightly higher than 1, while the average search number of both Murmur hash and BKDR hash based indexes are near 1.5. the unsupervised index gains the smallest spare space with only 0.20%. The spare space of supervised index is about 4.4%, which is slight higher than unsupervised index. While Murmur hash based index is 37%. This shows that, comparing to Murmur hash based index, unsupervised index save space 99.46% ($(37-0.2)/37=99.46$). Table 1 also shows that the RKDR hash

TABLE 1. Evaluation of performance of RNN-based learned index.

Dataset	Hash Model	Search Time(ns)	Avg Search	Spare Space	Reduction	
Log Normal Dataset	Learned Index Structure	Supervised mapping layer	73987	1.0483	4.41%	88.08%
		Unsupervised mapping layer	72181	1.002	0.20%	99.46%
	Murmur hash based Index	20637	1.50646	37.00%	0.00%	
	BKDRHash based Index	8610	1.50263	36.90%	0.27%	

based index has the smallest search time with 8610ns, and the Murmur hash based index is the second small search time with 20637ns. The search time of unsupervised index is 72181ns, which is 3.5 (72181/20637) times more than Murmur hash. The reason is that learned index is composed of multi layers network, which unavoidably spends more time to locating the data. Comparing to the supervised index, the unsupervised index save search time 2.4 % ((73987-72181)/93987).

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel RNN-based learned inverted index, which uses hierarchical neural network to simulate hash functions. Experimental results show that both the supervised and unsupervised approaches have lower collision rate as well as higher space utilization, compared with the traditional hash functions. Although the conflict rates both in the supervised learning and the unsupervised learning index model are very low on some data sets, the unsupervised learning model can better fit the relatively complex data and is less affected by the outliers. When the data distribution is unknown, unsupervised learning can train better models with the same model parameter quantities. Although the search time of learned index is at least 3-4 times more than the traditional hash function based index on our python implementation, considering the quick developed GPU, there is strong evidence to show that neural network based index is promising in future.

REFERENCES

- [1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manage. Data*, Houston, TX, USA, Jun. 2018, pp. 489–504, doi: [10.1145/3183713.3196909](https://doi.org/10.1145/3183713.3196909).
- [2] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Trans. Database Syst.*, vol. 23, no. 4, pp. 453–490, Dec. 1998.
- [3] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, p. 6, Jul. 2006, doi: [10.1145/1132956.1132959](https://doi.org/10.1145/1132956.1132959).
- [4] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 6, pp. 734–749, Jun. 2005.
- [5] A. Morgulis, G. Coulouris, Y. Raytzelis, T. L. Madden, R. Agarwala, and A. A. Schäffer, "Database indexing for production MegaBLAST searches," *Bioinformatics*, vol. 24, no. 16, pp. 1757–1764, Jun. 2008.
- [6] B. Kulis, P. Jain, and K. Grauman, "Fast similarity search for learned metrics," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 12, pp. 2143–2157, Dec. 2009.
- [7] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang, "Supervised hashing with kernels," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Providence, RI, USA, Jun. 2012, pp. 2074–2081.
- [8] M. Norouzi, D. Fleet, and R. Salakhutdinov, "Hamming distance metric learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1061–1069.
- [9] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Anchorage, AK, USA, Jun. 2008, pp. 1–8.
- [10] L. Fan, "Supervised binary hash code learning with Jensen Shannon divergence," in *Proc. IEEE Int. Conf. Comput. Vis.*, Dec. 2013, pp. 2616–2623.
- [11] F. Shen, C. Shen, W. Liu, and H. T. Shen, "Supervised discrete hashing," in *Proc. IEEE Int. Conf. Comput. Vis. Pattern Recognit.*, Boston, MA, USA, Jun. 2015, pp. 37–45.
- [12] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in Neural Information Processing Systems*, vol. 21, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds. Cambridge, MA, USA: MIT Press, 2009, pp. 1753–1760.
- [13] W. Liu, J. Wang, S. Kumar, and S.-F. Chang, "Hashing with graphs," in *Proc. ICML*, Bellevue, WA, USA, 2011, pp. 1–8.
- [14] Y. Gong and S. Lazebnik, "Iterative quantization: A procrustean approach to learning binary codes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Colorado Springs, CO, USA, Jun. 2011, pp. 817–824.
- [15] W. Kong and W.-J. Li, "Isotropic hashing," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1655–1663.
- [16] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik, "Angular quantization based binary codes for fast similarity search," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1646–1654.
- [17] J. Wang, S. Kumar, and S.-F. Chang, "Semi-supervised hashing for large-scale search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 12, pp. 2393–2406, Dec. 2012.



WENKUN XIANG received the B.S. and M.S. degrees in software engineering from Yunnan University, in 2015 and 2018, respectively. He is currently a Researcher with Tencent Technology, Beijing. His current research interests include machine learning, big data computing, and recommender system.



HAO ZHANG is currently pursuing the graduate degree with Yunnan University. His main research interests include machine learning, deep learning, and bioinformatics.



RUI CUI received the B.S. and M.S. degrees in computer application technology from Northwestern Polytechnical University, in 2007 and 2010, respectively. She is currently a Senior Researcher of recommendation algorithm with Tencent. She is mainly devoted to the research and development of short video personalized recommendation of daily express.



XING CHU received the B.E. degree from the Kunming University of Science and Technology, in 2011, the M.E. degree from Hunan University, China, in 2014, and the Ph.D. degree in control theory and engineering from the Ecole Centrale de Lille, in 2017. He is currently a Lecturer with the School of Software, Yunnan University, China. His research interests lie in the distributed cooperative control of multi-agent/robot systems and intelligent automobile.



KEQIN LI is currently a SUNY Distinguished Professor of computer science with the State University of New York. He has published over 590 journal articles, book chapters, and refereed conference papers. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU–graphic processing unit hybrid and

cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, the Internet of Things, and cyber-physical systems. He was a recipient of several best paper awards. He is currently serving or has served on the editorial boards of the *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, the *IEEE TRANSACTIONS ON COMPUTERS*, the *IEEE TRANSACTIONS ON CLOUD COMPUTING*, the *IEEE TRANSACTIONS ON SERVICES COMPUTING*, and the *IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING*.



WEI ZHOU received the Ph.D. degree from the University of Chinese Academy of Sciences. He is currently a Full Professor with the Software School, Yunnan University. His current research interests include the distributed data intensive computing and bio-informatics.

...