The Edge Weight Computation with MapReduce for Extracting Weighted Graphs

Yuhong Feng, Junpeng Wang, Zhiqiang Zhang, Haoming Zhong, Zhong Ming, Xuan Yang, and Rui Mao

Abstract—Automated weighted graph construction from massive data is essential to weighted graph theory based data mining processes, where the edge weight computation is time consuming or even fails to complete on a single machine when necessary resources are exhausted. In addition, existing work lacks of the measurement on the accuracy of the edge weights, which represents the graph accuracy and affects the following data mining results. This paper describes the classification, implementation and evaluation of edge weight computation algorithms with MapReduce Framework, which is a powerful parallel and distributed processing model. First, a classification of the edge weight computation algorithms is developed and how they can be applied on MapReduce is also discussed. Then we propose comprehensive measurements on the edge weight accuracy in terms of the number of edges, strength distribution, community structure, Hop-plot and effective diameters. Finally, a performance study has been conducted to evaluate these algorithms in terms of memory and disk usage, execution time and accuracy using a real massive social network application dataset. The results are presented and discussed. Our comparison results can help find out the most effective parallel and distributed edge weight computation algorithm for constructing a weighted graph for a given massive dataset.

Index Terms—Weighted graph extraction, edge weight computation, similarity measurement, MapReduce, massive data analysis

1 Introduction

Recent years have witnessed the rapid growth of application data and mining such data provides various desirable services benefiting our life in many ways. Many data mining methods are based on graph theory and graphs need to be constructed before they can be applied.

Graph construction for a given dataset includes graph extraction, transform, and load (ETL). Graph construction for massive data is a time-consuming and complicated task: (i) The raw application data grows exponentially; (ii) The number of entities and their features hidden in the raw data range from numbers to tens of thousands, meaning that multiple different topological graphs can be constructed from a particular dataset; and (iii) The management of graph construction workflow needs extensive computational domain knowledge, e.g., effective resource utilization, load balancing, and accurate graph delivering. This handicaps the data scientists from actual data analysis. In order to offload the complexities of graph construction from data scientist and help them focus on data analysis, a scalable framework Graphbuilder [1] using MapReduce [2] is recently open sourced, where MapReduce is a powerful parallel and distributed processing model for large scale data intensive applications. Graphbuilder provides a set of tools for automatically constructing graphs from raw data for various applications and it demonstrates its remarkable effectiveness on graph compression and partition, reducing memory consumption and achieving load balancing across the computational resources.

For the time being, there are no edge weights in the graphs extracted by Graphbuilder. A weighted graph consists of not only nodes and edges, but also edge weights, which measure the degree of the similarity between nodes and represent the closeness of the nodes' connections. Quite some widely used graph based data mining methods such as clustering [3] and collaborative filtering [4] are actually based on weighted graphs. In order to narrow down the research gap, our objective is to design algorithms for effective edge weight computation. The edge weight computation, i.e. the similarity measurement between nodes, over massive datasets is time consuming. How to parallel and scale up their computation on a single machine has been well reported in [5]. Massive data nowadays are often characteristic of *high volume*, *high dimension* and *high distribution*, which presents great challenges to the edge weight computation on a single host. MapReduce framework decomposes computing tasks into smaller ones and distributes them to execute on multiple distributed hosts simultaneously, which improves their efficiency and scalability.

Therefore, this paper describes the classification, implementation and evaluation of edge weight computation algorithms with MapReduce. First, to put the discussion into perspective, what graphs can be extracted from a given data is presented. Second, a classification of existing edge weight computation algorithms is developed and how they can be applied on MapReduce is also discussed. Third, we propose comprehensive measurements on the edge weight

- *Y. Feng, J. Wang, Z. Ming, X. Yang and R. Mao are with the College of Computer Science and Software Engineering and the Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, Shenzhen 518060, China. E-mail: {yuhongf, mingz, mao}@szu.edu.cn, wjp_2013@126.com, xyang0520@263.net.*
- *Z. Zhang is with the Department of Research and Development, Beansmile, Guangzhou, China. E-mail: padmazero@gmail.com.*
- *H. Zhong is with the Big Data Center, WeBank, Shenzhen, China. E-mail: zhonghm@gmail.com.*

Manuscript received 31 Jan. 2015; revised 6 Nov. 2015; accepted 14 Feb. 2016. Date of publication 29 Feb. 2016; date of current version 16 Nov. 2016.
Recommended for acceptance by J. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2536024

1045-9219 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

TABLE 1
A Sample of Facebook Application Data II

uid	app_id	app_id	app_id	app_id	app_id
1	1,523	1,544	1,567	1,580	1,594
2	1,544	1,567	1,580	1,599	

computation accuracy in terms of the number of edges, strength distribution, community structure, Hop-plot and effective diameters. Finally, a performance study has been conducted to evaluate these algorithms in terms of memory and disk usage, execution time and graph accuracy using a real massive social network dataset, i.e., Facebook application dataset. The results are presented and discussed. Our comparison results can help find out the most effective edge weight computation algorithm for extracting a weighted graph for a given dataset. In addition, the results of our proposed algorithms, i.e., the edge weights, can be used as one of the edge properties for the edge objects in Graphbuilder, thus extending it to support weighted graph extraction.

The rest of the paper is organized as follows: Section 2 gives an introduction on how graphs can be extracted from a given sample SN dataset and how edge weights can be computed, and then presents corresponding related work. Section 3 describes the classification of existing edge weight computation technologies and their implementations with MapReduce. Section 4 presents our performance study, reports and discusses our experimental results. Finally Section 5 concludes and outlines the future work.

2 Weighted Graph Extraction and Related Work

In general, the workflow for extracting weighted graphs includes three steps: identify nodes, extract and tokenize application specific features, compute edge weights. A set of real SN dataset, i.e., Facebook applications dataset II[1], collects the application installation or usage records in February 2008 [6] and is used to demonstrate what graphs can be extracted and how. The application data are recorded in the following format.

<uid> <app_id_1> <app_id_2>...<app_id_j>

where "uid" stands for an anonymized user Id, and "app_id_i" ($1 \leq i \leq j$) stands for an application Id. A sample of the facebook application data set II is illustrated in Table 1.

2.1 Node Identification and Feature Extraction

There can be multiple choices for selecting the appropriate data points as nodes, e.g., for Facebook application dataset II, either users or applications can be chosen as nodes.

When users are chosen as nodes, Facebook application dataset II can be graphically represented as a weighted graph, denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$, where $\mathcal{V}$ is the set of vertices denoting users, i.e., $\mathcal{V} = (u_0, u_1, \ldots, u_{n-1})$, and n is the number of users. A user can install and use multiple applications, e.g., the range of the number of applications used by a user in Facebook application dataset II is within the range $< 3, 773 >$ [7]. The applications installed by a user can be selected as features to characterize a user. Let $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ represents all the available applications, and m is the number of applications. Let the applications installed by a particular user, e.g., u_i, be denoted as $A(u_i) = \{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$, where k is the number of applications used by user u_i, and for any $a_{i_j} \in A(u_i)$, we have $a_{i_j} \in \mathcal{A}$. After node identification and feature extraction, the sample data in Table 1 can be represented in Table 2, where the value at line i and column j is denoted as $f(u_i, a_j)$. $f(u_i, a_j) = 1$ means that u_i installs a_j and $f(u_i, a_j) = 0$ means that u_i does not install a_j.

TABLE 2
Transformed Sample Data

uid	a_1 1,523	a_2 1,544	a_3 1,567	a_4 1,580	a_5 1,594	a_6 1,599
1	1	1	1	1	1	0
2	0	1	1	1	0	1

When applications are chosen as nodes, the dataset can be represented as another weighted graph, denoted as $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{W}')$, where $\mathcal{V}'$ is the set of vertices denoting applications and $\mathcal{V}' = \{a_1, a_2, \ldots, a_m\}$. An application can be installed by multiple users, and such users can be selected as features to characterize the application.

The number of dimensions and entries are two features representing the data complexity and data volume, respectively. Let the number of dimensions be denoted as $\mathcal{N}$. The number of its entries is the number of nodes, e.g., $|\mathcal{V}|^2$ for $\mathcal{G}$ and $|\mathcal{V}'|$ for $\mathcal{G}'$. When users are selected as nodes, from Table 2, we can have $\mathcal{N} = 6$ and the number of its entries is 2. Similarly, when applications are selected as nodes, we have $\mathcal{N} = 2$ and the number of its entries is 6. That is, different node selection can lead to different topological graphs and thus different analysis results. Facebook application dataset II collects application usage records for 297K users and 8,1000 applications. When users are selected as nodes, we have $\mathcal{N} = 8,1000$ and $|\mathcal{V}| = 297k$. While when applications are selected as nodes, we can have $\mathcal{N} = 297k$ and $|\mathcal{V}'| = 8,1000$, where we can see that dataset can be *high dimension* and *high volume*.

2.2 Edge Weight Computation

After nodes and features are identified, the edges and their weights for $\mathcal{G}$ and $\mathcal{G}'$ need to be settled down. For $\mathcal{G}$, there is an edge between user u_i and u_j (i.e., $(u_i, u_j) \in \mathcal{E}$) if user u_i and u_j have installed common applications. Similarly, for $\mathcal{G}'$, there is an edge between application a_i and a_j (i.e., $(a_i, a_j) \in \mathcal{E}'$) if application a_i and a_j have been installed by some common users. Meanwhile, each edge is associated with a weight representing the similarity between nodes. For example, for any $(u_i, u_j) \in \mathcal{E}$, its weight is denoted as $w_{i,j}$ and it is used to represent the similarity between the two sets of applications installed by users u_i and u_j, i.e., $A(u_i)$ and $A(u_j)$.

1. http://odysseas.calit2.uci.edu/doku.php/public:online_social_networks#facebook_applications

2. $|\mathcal{V}|$ refers to the cardinality of the set $\mathcal{V}$.

TABLE 3
Simplified Artificial Data

uid	applications id	id	id
u_1	a_1	a_2	a_4
u_2	a_2	a_3	a_4
u_3	a_1	a_2	a_3
u_4	a_2	a_3	

(a) The dataset.

app id	uid	uid	uid	uid
a_1	u_1	u_3		
a_2	u_1	u_2	u_3	u_4
a_3	u_2	u_3	u_4	
a_4	u_1	u_2		

(b) The transformed dataset.

(c) A weighted graph extracted from the dataset

Jaccard similarity (also named as Jaccard similarity coefficient or Jaccard index in literature), usually used as an indicator for similarity between two sets, measures their degree of the similarity based on the presence of common elements of two sets [8]. Therefore, Jacard similarity is used as an example to study how to compute the edge weights. For example, for any edge $(u_i, u_j) \in \mathcal{E}$, its weight, i.e., $w_{i,j}$, can be obtained by Equation (1)

$$w_{i,j} = \frac{|A(u_i) \cap A(u_j)|}{|A(u_i) \cup A(u_j)|} = \frac{|A(u_i) \cap A(u_j)|}{|A(u_i)| + |A(u_j)| - |A(u_i) \cap A(u_j)|}. \tag{1}$$

Based on Equation (1), we can have that $w_{i,j}$ is a number between 0 and 1. The bigger $w_{i,j}$ is, the more similar $A(u_i)$ and $A(u_j)$ are. Particularly, $w_{i,j} = 0$ means that there is no common application installed by user u_i and u_j, therefore, there should be no edge between u_i and u_j. On contrary, $w_{i,j} = 1$ means that the applications installed by user u_i and u_j are exactly the same.

The computation of $w_{i,j}$ includes the following three steps: (i) Compute the cardinality of each set, i.e., $|A(u_i)|$ and $|A(u_j)|$; (ii) Compute the cardinality of the set intersection of $A(u_i)$ and $A(u_j)$, i.e., $|A(u_i) \cap A(u_j)|$; and (iii) Obtain $w_{i,j}$ using Equation (1). For simplification, we use a simplified artificial data, as described in Table 3a, as an example to illustrate how to compute edge weights.

Assuming that the set of users using application a_i be represented as $U(a_i)$, we can have $a_i \in A(u_i') \cap A(u_j')$ if and only if $u_i' \in U(a_i)$ and $u_j' \in U(a_j)$. A new table can be created by exchanging the row and column in Table 3a, as shown in Table 3b. The together occurring two items in an entry of a table is called 2-itemset [9]. For example, u_1 and u_3 in Table 3b constitute a 2-itemset for the entry where $app_id = a_1$, where the 2-itemset is denoted as (u_1, u_3). The 2-itemset (u_1, u_3) occurs in both entries where $app_id = a_1$ and $app_id = a_2$, i.e., the frequency of the occurrence of (u_1, u_3) is 2. Let $f_{i,j}$ denote the frequency of the occurrence of the 2-itemset (u_i, u_j) in Table 3b, then we have $f_{1,3} = 2$. Meanwhile, from Table 3, we observe that $A(u_1) \cap A(u_3) = \{a_1, a_2\}$, i.e., $|A(u_1) \cap A(u_3)| = 2$. In more general, we can have the following observation as summarized in Equation (2)

$$|A(u_i) \cap A(u_j)| = f_{i,j}. \tag{2}$$

Based on Equation (2), the computation of $|A(u_i) \cap A(u_j)|$ in Table 3a can be regarded as the mining of the frequency of 2-itemsets in Table 3b. The objective of this paper is to compute the edge weights for massive data once the nodes and their features are given, as illustrated in Table 3c.

2.3 Related Work

This section surveys related work in Jaccard similarity measurement and frequent itemset mining algorithms using MapReduce. Mining the frequency of all 2-itemsets in large-scale datasets is a challenge. *Dimension reduction* and *frequent pattern tree (FP-tree for short) based data compression* technologies can be exploited to speedup the performance. Dimension reduction technologies reduce the dimension for reducing the computation complexity. FP-tree based data compression technologies exploit extended prefix-tree structure to compress data into more compact structure so as to make it fit into memory and avoid costly database scans. According to whether dimension reduction or FP-tree based data compression technologies are used, existing work can be classified into three main categories, including *Apriori-based algorithms* [10], *signature-based algorithms* [11], and *FP-tree-based algorithms* [12]. As the dataset size grows, the MapReduce version of each category has been proposed to meet the large-scale dataset challenge.

First, Apriori algorithm is the most popular algorithm for association rule mining, which was developed by Agrawal and Srikant in 1994. It uses frequent (k-1)-itemsets to generate candidate frequent k-itemsets, where database scan and pattern matching is used to collect the candidate itemsets. Experiment results demonstrate the improvement on the execution time and scalability of the parallel implementation of the Apriori-based algorithms using MapReduce [13], where each iteration of the algorithms usually includes two steps: (1) The Map function generates candidate items; and (2) The reduce function sums the frequency of all candidate item set appearance, prunes the infrequent itemsets and outputs the frequent ones.

Second, signature-based algorithms explore similarity preserving signatures for sets to measure the similarity of the high dimensional objects compactly. Hashes are one type of the most popular used signatures. Especially, MinHash is a scheme developed by Broder in 1997 for estimating the Jacarrd similarity between two sets [14], where the essence is to hash each element using multiple independent hash functions such that the same elements will have the same hash values. Let $\mathcal{H} = \{h^1, h^2, \ldots, h^{n_h}\}$ be a set of hash functions that map the members of any two sets to distinct integers, where n_h is the number of hash functions. For any $h^x \in \mathcal{H}$, let $min_{s \in S} h^x(s)$ represent the minimum hash value of h^x over all the elements in set S. It has been proved that the probability a hash function on two sets producing the same minimum values equals to their Jaccard similarity [14].

This property renders the fundamental basis for using the probability to estimate the Jaccard similarity. However, when hash collision occurs, i.e., more than one element in

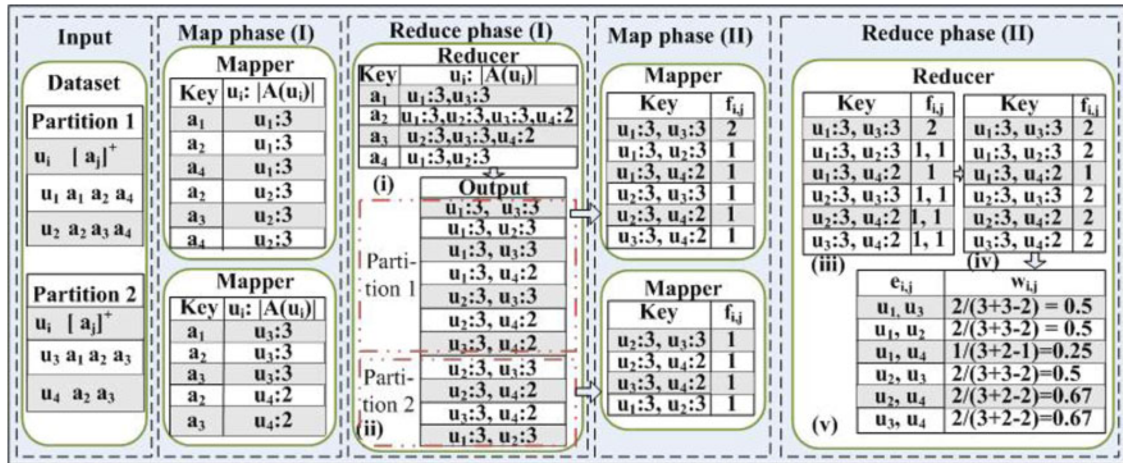


Fig. 1. Apriori-based edge weight computation over the example data.

the set links to the same hash value, there will be inaccuracy in the estimation. In order to increase the accuracy of the estimation, multiple independent hash functions are used and the average probability is used to avoid the relative error. Assuming ϵ is the given acceptable estimated error and δ is the relative error, in order to satisfy the requirement, the number of the hash functions n_h has to satisfy the constraint formalized in the following Equation (3) [15]

$$n_h \geq \frac{2+\epsilon}{\epsilon^2} \ln\left(\frac{2}{\delta}\right). \tag{3}$$

It has been accepted that setting $n_h = O(\frac{1}{\epsilon})$ can ensure that the estimated error of the MinHash is less than ϵ and the relative error is no more than δ [11], [16]. For example, when $\epsilon = 0.01$, then $n_h = 1/0.01 = 100$. Recalling that in Facebook application dataset II, the number of applications is 8,1000, which is much larger than 100. That is, MinHash scheme can map the large set of Fackbook application data into a much smaller one, reducing the dimension and therefore reducing the data size for further processing.

In general, signature-based Jaccard similarity measurement algorithms usually include two phase Map-Reduce computation. In the first phase, the Map function calculates the minhash signatures which reduce the dimension, and the Reduce function outputs the candidate itemsets. Then the second phase completes the rest of the computation. Comparing to the Apriori-based algorithms, though there are one more phase Map-Reduce in signature-based algorithms, the reduced complexity by dimension reduction expedites the following process, which makes it demonstrate remarkable efficiency for various applications such as fast sequence comparison [17] and privacy preserving record linkage [18].

Finally, in order to avoid the huge candidate sets generated in Apriori-based algorithms, FP-tree-based algorithms, e.g., FP-Growth [12], find frequent 1-itemsets first, order them in frequency descending order, construct the FP-tree where more frequent items are more likely to be shared, then recursively mine the FP-tree and grow frequent itemsets obtained so far. In order to enable the FP-tree-based algorithms work over the MapReduce, a group based algorithm [19] has been proposed to partition the dataset in a way that the following construction of the FP-trees and the mining of the frequent itemsets can be carried out independently in parallel.

Meanwhile, many related technologies have been proposed recently to further improve the performance issues raised in MapReduce based Jaccard similarity measurement or frequent itemset mining, e.g., efficient data partition for load balancing [20], intermediate information cache [21], parallelized incremental mining [22], appropriate sampling for dimension independent similarity measurement [11], and integrating GPU for MapReduce programming [23]. It is noteworthy that the above technologies can be used to improve the performance of the three category edge weight computation algorithms, but they are out the scope of this paper. To simplify the description and discussion, it is not assumed that they are in place in the algorithms' design and experimental comparison study with MapReduce in the rest of the paper.

3 Edge Weight Computation Algorithms Using MapReduce

Based on which category frequent itemset mining algorithm is used, we design three algorithms for the edge weight computation using MapReduce: *Apriori-based algorithm*, *signature-based algorithm*, and *FP-tree-based algorithm*. The example dataset in Table 3 is used to illustrate how edge weights are obtained by using each algorithm, where the sample dataset are partitioned into two subsets and each of which has two entries, as illustrated in the input of Fig. 1.

3.1 Apriori-Based Algorithm

The Apriori-based edge weight computation algorithm includes two pass scan over the dataset. The first pass obtains the cardinality of the application set of each node, e.g., $|A(u_i)|$ for any u_i in the example data, as described in the map phase (I) of Fig. 1. Meanwhile, the transform of the dataset by exchanging the contents of rows and columns is automatically done by the shuffle process of the MapReduce framework, in a way where the column is composed of a sequence of tuples $(uid : |A(u_i)|)$, as shown in the reduces of Fig. 1i. For each entry of the result obtained by the shuffle process, the reducers will enumerate all the 2-itemsets in the format of $(u_i : |A(u_i)|, u_j : |A(u_j)|)$ and output them, as

TABLE 4
Apriori-Based Algorithm for the Edge Weight Computation

$p\#$	Mapper	Reducer
1th	Input: $\{u_i, (a_{i_1}, a_{i_2}, \ldots, a_{i_k})\}$ Output: $\{(a_j, u_i : \|A(u_i)\|)\}$ 1) for each u_i in the partition 2) if $(a_j \in A(u_i))$ then 3) emit$(a_j, u_i : \|A(u_i)\|)$;	Input: $\{(a_j, u_{i_1} : \|A(u_{i_1})\|, \ldots, u_{i_k} : \|A(u_{i_k})\|)\}$ Output: $\{(u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|)\}$ 1) for each a_j in the partition 2) if $(u_i \in U(a_j))$ && $(u_{i'} \in U(a_j))$ then 3) output $(u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|)$;
2th	Input: $\{(u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|)\}$ Output: $\{((u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|), f_{i,i'})\}$ 1) for each $(u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|)$ 2) $f_{i,i'} = f_{i,i'} + 1$ 3) end for 4) emit each $((u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|), f_{i,i'})$	Input: $\{((u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|), f^1_{i,i'}, \ldots, f^k_{i,i'})\}$ Output: $\{((u_i, u_{i'}), w_{i,i'})\}$ 1) for each $(u_i : \|A(u_i)\|, u_{i'} : \|A(u_{i'})\|)$ 2) $f_{i,i'} = \sum_{j=1}^{j=k} f^j_{i,i'}$; $w_{i,i'} = \frac{f_{i,i'}}{\|A(u_i)\| + \|A(u_{i'})\| - f_{i,i'}}$ 3) output $((u_i, u_{i'}), w_{i,i'})$ 4) end for

shown in the reduces of Fig. 1ii. Alternatively, the output of the reducer can also be composed of tuples with an application id and the sequence uids associated with the cardinality of its application set, i.e., $(app_id, u_{i_1} : |A(u_{i_1})|, \ldots, u_{i_k} : |A(u_{i_k})|)$. However, this implementation will consume more time since the concatenation of the $(u_i : |A(u_i)|)$ for each application is time consuming with 297 K users in Facebook application dataset II.

The second pass of the algorithm first reads the output of the first pass and counts how often each 2-itemset $(u_i : |A(u_i)|, u_j : |A(u_j)|)$ occurs, as shown in Map phase (II) of Fig. 1. Then reducers in Reduce phase (II) will perform the following workflow: (1) Accumulate and summarize the frequency for each 2-itemset, as shown in Figs. 1 iii and 1 iv. Based on Equation (2), the obtained frequency of each 2-itemset, e.g., $(u_i : |A(u_i)|, u_j : |A(u_j)|)$, is equal to $|A(u_i) \cap A(u_j)|$. (2) Compute the weight between any two nodes using Equation (1), as shown in Fig. 1v. (3) Finally proceed to output the weights. In all, Table 4 describes Apriori-based edge weight computation algorithm for graph $\mathcal{G}$ when users are selected as nodes, where $p\#$ denotes the pth round of the scan pass over the dataset.

3.2 Signature-Based Algorithm

The signature-based edge weight computation algorithm using MinHash scheme includes 2 pass scan over the dataset. The first pass of the algorithm figures out the node pairs having the same minimum value for each hash function, where the mappers in Map phase (I) performs the following workflow.

1. Calculate the hash values for the input data by hashing each application using the chosen hash functions, as illustrated in Map phase (I) (i) in Fig. 2.
2. Select the minimum hash values for each user and hash function taking into account whether the user has installed the applications. For any u_i, let $min_{a_j \in A(u_i)} h^x(a_j)$ be the minimum hash value of h^x over all elements in $\mathcal{A}$ and $f(u_i, a_j) \neq 0$, i.e., the application a_j has been installed by the user u_i. As illustrated in the Map phase (I) (ii) in Fig. 2, we have $min_{a_j \in A(u_i)} h^1(a_j) = 0$ and $min_{a_j \in A(u_i)} h^2(a_j) = 1$ for u_1.
3. Transform obtained minimum hash values into the locality aware minimum hash values for each hash function and user by concatenating the location of a hash function, i.e., the ith hash function, with a hyphen and then the corresponding minimum hash values for each hash function and the node. For example, as illustrated in the Map phase (I) (iii) in Fig. 2, the output is denoted as $(x_min_{a_j \in A(u_i)} h^x(a_j), u_i)$, where x denotes the xth hash function.
4. Emit the key-value tuples $(x_min_{a_j \in A(u_i)} h^x(a_j), u_i)$.

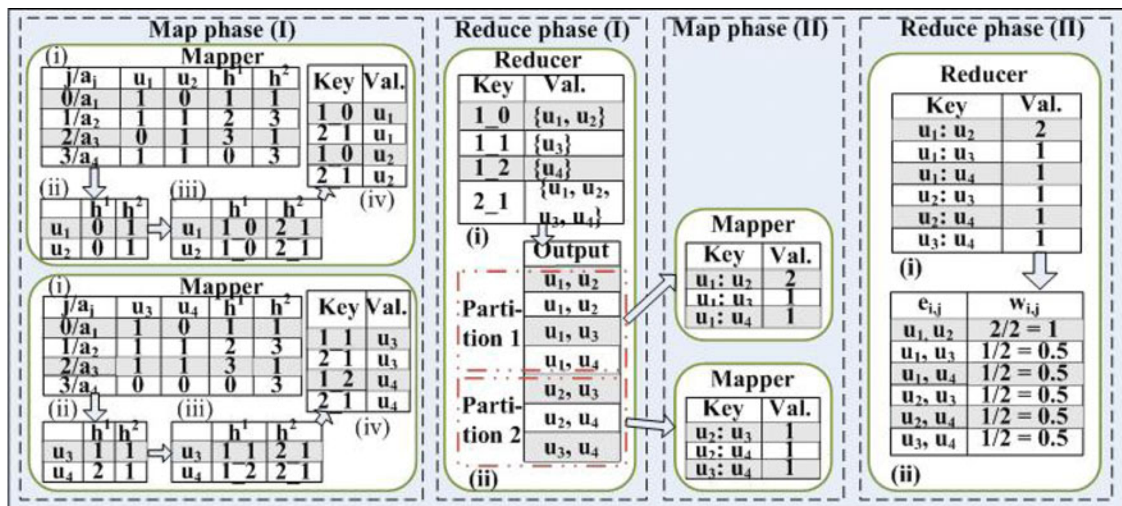


Fig. 2. Signature-based edge weight computation over the example data.

TABLE 5
Signature-Based Algorithm for the Edge Weight Computation

$p\#$	Mapper	Reducer
1th	Input:$\{u_i, (a_{i_1}, a_{i_2}, \ldots, a_{i_k})\}$ Output:$\{(x_min_{a_j \in A(u_i)} h^x(a_j), u_i)\}$ 1) for each $a_i \in \mathcal{A}$ 2) $\forall h^x \in \mathcal{H}$, calculate $h^x(a_i)$ 3) for each u_i 4) for each $h^x \in \mathcal{H}$ 5) calculate $min_{a_j \in A(u_i)} h^x(a_j)$ 6) emit $(x_min_{a_j \in A(u_i)} h^x(a_j), u_i)$ 7) end for	Input:$(x_min_{a_j \in A(u_i)} h^x(a_j), u_{i_1}, \ldots, u_{i_k})$ Output:$\{(u_i, u_{i'})\}$ 1) for each $x_min_{a_j \in A(u_i)} h^x(a_j)$ 2) if $(min_{a_j \in A(u_i)} h^x(a_j) = min_{a_j \in A(u_{i'})} h^x(a_j))$ then 3) emit $(u_i, u_{i'})$
2th	Input:$\{(u_i, u_{i'})\}$; Output:$\{(u_i : u_{i'}, f_{i,i'})\}$ 1) for each $(u_i, u_{i'})$ 2) $f_{i,i'} = f_{i,i'} + 1$ 3) emit each $(u_i : u_{i'}, f_{i,i'})$	Input:$\{(u_i : u_{i'}, f^1_{i,i'}, \ldots, f^{i_k}_{i,i'})\}$; Output:$\{((u_i, u_{i'}), w_{i,i'})\}$ 1) for each $(u_i : u_{i'})$ 2) $f_{i,i'} = \sum_{y=1}^{y=i_k} f^y_{i,i'}$; $w_{i,i'} = \frac{f_{i,i'}}{n_h}$; output $(u_i : u_{i'}, w_{i,i'})$ 3) end for

The reducers accumulate the uids for each locality aware MinHash value, enumerate and output the user pairs $(u_i, u_{i'})$ having the same locality aware MinHash value, as illustrated in Reduce phase (I) in Fig. 2.

The second pass of the algorithm computes the edge weights, whose workflow is similar to that of reducers of Reduce phase (II) of the Apriori-based algorithm, excepts that the weights of the edges are computed using the average probability of the set of hash functions on two sets producing the same minimum values. That is, the weights are obtained by the frequency of the 2-itemsets against the number of hash functions, i.e., $w_{i,i'} = \frac{f_{i,i'}}{n_h}$, as shown in the reduce phase (II) (ii) in Fig. 2. In summary, Table 5 describes the signature-based edge weight computation algorithm using MinHash scheme.

3.3 FP-Tree-Based Algorithm

In order to enable the parallel and independent FP-tree-based 2-itemset mining over MapReduce, we exploit the group based partitioning algorithm proposed in [19] for the FP-tree-based edge weight computation algorithm, which includes 3 pass scan over the datasets. The mappers of the first pass obtain the frequency of the 1-itemset for each user. The reducers then group users according to their 1-itemset frequencies. For example, there are two frequency values for the uids in the example dataset, i.e., 3 and 2. The uids with frequency 3 are categorized to group 2 and those with frequency 2 are categorized to group 1. Let $g(u_i)$ represent the group id of u_i, the reducers of the first pass output tuples $(u_i : |A(u_i)| : g(u_i), a_{i_1} \ldots a_{i_k})$.

In the second pass, the mappers transform the dataset as the previous two algorithm and emit sequences of tuples $(a_k, u_i : |A(u_i)| : g(u_i))^*$, then the tuples are sorted during the shuffle and merge stage according to the group id in descending order. If there are multiple users have the same group id, then the users are ordered in descending order according to their 1-itemset frequency. Again, if there are multiple users have the same group id and 1-itemset frequency, then the users are ordered according to their uids in ascending order. The reducers in Reduce phase (II) aggregate and output the sorted tuple sequence $(u_{i_1} : |A(u_{i_1})| : g(u_{i_1}), \ldots, u_{i_l} : |A(u_{i_l}| : g(u_{i_l}))$ for each application, as shown in the Reduce phase (II) of Fig. 3.

In the third pass, the mappers categorize all the tuples in its partition to appropriate groups, where the key is a group id and the value is the tuples. In general, for any tuple

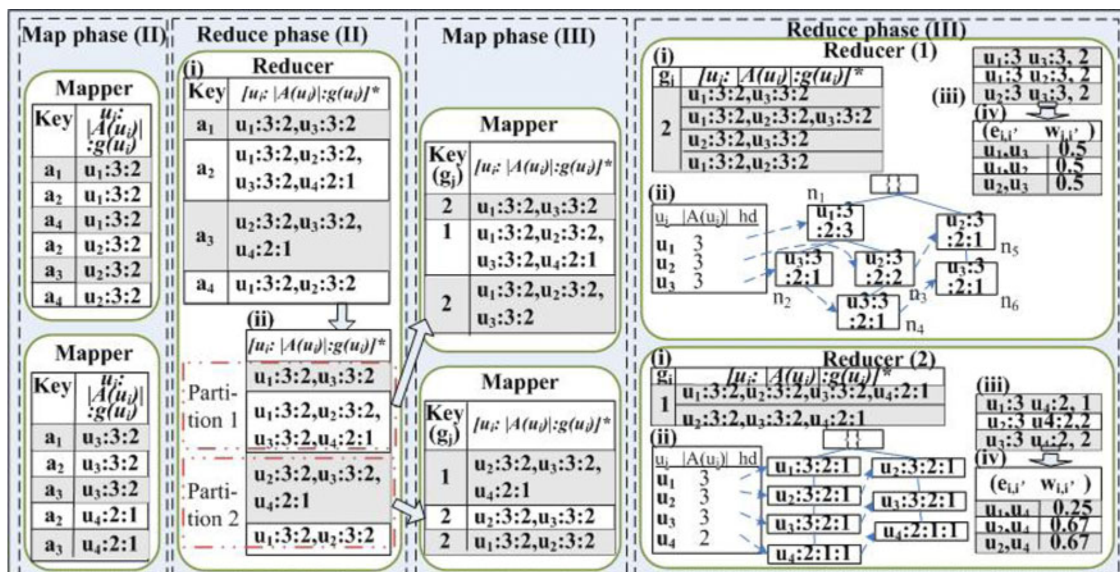


Fig. 3. FP-tree-based edge weight computation over the example data.

TABLE 6
FP-Tree-Based Algorithm for the Edge Weight Computation

p#	Mapper	Reducer
2th	Input:{$u_i, < a_{i_1}, a_{i_2}, \ldots, a_{i_k} >$} Output:{$(a_k, u_i : \lvert A(u_i)\rvert : g_j)$} 1) for each u_i in the partition 2) assign groud id g_j to u_i 3) for each $((a_k \in A(u_i))$ 4) emit $(a_k, u_i : \lvert A(u_i)\rvert : g_j)$ 5) end for	Input:$(a_k, u_{i_1} : \lvert A(u_{i_1})\rvert : g_{j_1}, \ldots, u_{i_l} : \lvert A(u_{i_l}\rvert : g_{j_m})$ Output:$(u_{i'_1} : \lvert A(u_{i'_1})\rvert : g_{j'_1}, \ldots, u_{i'_l} : \lvert A(u_{i'_l})\rvert : g_{j'_m})$ 1) for each a_k in the partition 2) order $(u_{i_1} : \lvert A(u_{i_1})\rvert : g_{j_1}, \ldots, u_{i_l} : \lvert A(u_{i_l})\rvert : g_{j_m})$ 3) output the ordered $(u_{i'_1} : \lvert A(u_{i'_1})\rvert : g_{j'_1}, \ldots, u_{i'_l} : \lvert A(u_{i'_l})\rvert : g_{j'_m})$ 4) end for
3th	Input:$(u_{i_1} : \lvert A(u_{i_1})\rvert : g_{j_1}, \ldots, u_{i_m} : \lvert A(u_{i_m}\rvert : g_{j_m})$ Output:$(g_{j_k}, u_{i_1} : \lvert A(u_{i_1})\rvert : g_{j_1}, \ldots, u_{i_k} : \lvert A(u_{i_k})\rvert : g_{j_k})$ 1) for each tuple sequence 2) $T = (u_{i_1} : \lvert A(u_{i_1})\rvert : g_{j_1}, \ldots, u_{i_m} : \lvert A(u_{i_m}\rvert : g_{j_m})$ 3) while T is not empty 4) g_{j_m} is T's rightmost group id 5) emit (g_{j_k}, T) 6) remove tuples with g_{j_k} from T 7) end while 8) end for	Input:$(g_{j_k}, (u_{i_1} : \lvert A(u_{i_1})\rvert : g_{j_1}, \ldots, u_{i_k} : \lvert A(u_{i_k})\rvert : g_{j_k}))$ Output:{$((u_i, u_{i'}), w_{i,i'})$} 1) Construct the header table $\mathcal{H}$ 2) Construct the local FP-tree 3) for each $((u_i \in \mathcal{H})$ && $(g(u_i) == g_{j_k}))$ 4) for any $u'_i \in p(u_i)$ && $(u'_i \neq u_i))$ 5) $f_{i,i'} = f_{i,i'} + f(u_i))$ 6) update tuple $(u_i : \lvert A(u_i)\rvert u'_i : \lvert A(u'_i)\rvert, f_{i,i'})$ 7) endfor 8) output each $(u_i, u_{i'}, w_{i,i'} = \frac{f_{i,i'}}{\lvert A(u_i)\rvert+\lvert A(u_{i'})\rvert-f_{i,i'}})$ 9) endfor

sequence $T = (u_{i_1} : |A(u_{i_1})| : g(u_{i_1}), \ldots, u_{i_l} : |A(u_{i_l})| : g(u_{i_l}))$, the mapper first locates its right-most group id, i.e., $g(u_{i_l})$, and outputs a key-value $g(u_{i_l}), (u_{i_1} : |A(u_{i_1})| : g(u_{i_1}), \ldots, u_{i_l} : |A(u_{i_l})| : g(u_{i_l}))$. Then the tuples with the group id which equals to $g(u_{i_l})$ will be removed from T. If T is empty, the mapper will be fed with the next tuple sequence. Otherwise, the mapper will continue to output the key-values on T. For example, for the tuple sequence $T = (u_1 : 3 : 2, u_2 : 3 : 2, u_3 : 3 : 2, u_4 : 2 : 1)$, the rightmost group id is 1, the mapper first outputs the key-value $(1, (u_1 : 3 : 2, u_2 : 3 : 2, u_3 : 3 : 2, u_4 : 2 : 1))$. Then we have $T = T - (u_4 : 2 : 1) = (u_1 : 3 : 2, u_2 : 3 : 2, u_3 : 3 : 2)$. Since T is not empty, the mapper continues to carry out the same operation on T and it outputs another key-value $(2, (u_1 : 3 : 2, u_2 : 3 : 2, u_3 : 3 : 2))$. Now, we have $T = T - (u_1 : 3 : 2, u_2 : 3 : 2, u_3 : 3 : 2) = \{\}$. Then, the mapper completes its execution or is fed with another tuple sequence if any.

Finally, the input data for each reducer in reduce phase (III) is the tuples categorized to a group id, as illustrated in Fig. 3i. Based on the input data, reducers first construct a header table and a FP-tree for the frequent items and patterns, as depicted in Fig. 3ii. Each entry in the header table is a tuple $(u_i, |A(u_i)|, link(u_i))$, where $link(u_i)$ links all the nodes whose uid is u_i in the FP-tree. The entries in the header table are ordered in descending order according to their 1-itemset frequency. A node in the constructed FP-tree is denoted as $n_j = (u_i : |A(u_i)| : g(u_i) : f(u_i))$, where $f(u_i)$ represents the occurrence frequency of the pattern $(root = \{\}, \ldots, u_i : |A(u_i)| : g(u_i))$ in reducer's input data. For example, the occurrence frequency of pattern $(u_1 : 3 : 2, u_2 : 3 : 2)$ in the input data of reducer 1 is 2, corresponding $f(u_2)$ is 2 and thus we have $n_3 = (u_2 : 3 : 2 : 2)$. The constructed FP-trees have the following features: (i) A node with bigger $|A(u_i)|$ is closer to the root and more likely to be shared; and (ii) From each node on $link(u_i)$ to the root constructed a series of patterns, denoted as $p(u_i)$. For example, in Reducer (1), we have $p(u_2) = \{(u_2 : 3 : 2 : 2, u_1 : 3 : 2 : 3), (u_2 : 3 : 2 : 1)\}$.

After the FP-tree is constructed, the reducer will complete the following tasks: (1) Mine the frequencies for the 2-itemsets, which is similar to the traditional FP-growth algorithm. The difference is that the frequencies of the tuples with the same group id will not be counted when the group id of the tuple is not the same as the one assigned to the reducer. For example, the reducer 2 is assigned with $g_i = 1$, although tuples $(u_1 : 3 : 2 : 1)$ and $(u_2 : 3 : 2 : 1)$ construct a pattern in the FP-tree, their frequency will not be counted since their group id are 2. (2) Compute edge weights and output them, whose computation is similar to the one used in Apriori-based algorithm. Table 6 describes the FP-tree-based edge weight computation algorithm.

The FP-tree-based algorithm speeds up the performance of the 2-itemset frequency mining, but it takes extra execution time for the grouping process and the FP-tree construction. The signature-based algorithm reduces the data dimension and therefore expedites the computation. However, it obtains different weights for edge weights, e.g., the weights for edges (u_1, u_2),(u_1, u_4),(u_2, u_4), and (u_3, u_4) are different from the ones obtained by the other two algorithms, as shown in Figs. 1, 2 and 3. How much such differences may affect the later data analysis based on the extracted graphs? To the best of our knowledge, little work has reported on such measurement. In order to evaluate the performance of the above three category edge computation algorithms, an empirical comparison study by applying them to extract graphs for Facebook applications dataset II will be given in the next section.

4 Performance Study

The efficiency of an edge weight computation algorithm can be measured by the *memory and disk usage*, *execution time*, and *accuracy* of the extracted graph. First, memory and disk usage measures how much memory and disk an algorithm consumes during its execution. Higher memory and disk usage limits the scalability of the algorithm on larger datasets and reduces the number of concurrent jobs. Second, execution time is used to measure how much time the edge weight computation takes, the shorter the execution time, the more efficient an algorithm is. Finally, accuracy is used

TABLE 7
Experimental Dataset Features

$N(u)$	$N(a)$	$M(a,u)$	$M(u,a)$
10k	4825	644	8615
20k	6311	644	17131
30k	7106	644	25695
40k	7854	644	34275
50k	8436	644	42883
60k	8866	644	51452
70k	9256	774	60043
80k	9682	774	68534
120k	10914	774	102628
160k	11741	774	136644
200k	12415	774	170735

(a) 10k - 200k entries

$N(u)$	1000	2000	3000	4000
$M(u,a)$	862	1727	2592	3448
$N(u)$	5000	6000	7000	8000
$M(u,a)$	4326	5201	6058	6924

(b) 1k - 8k entries

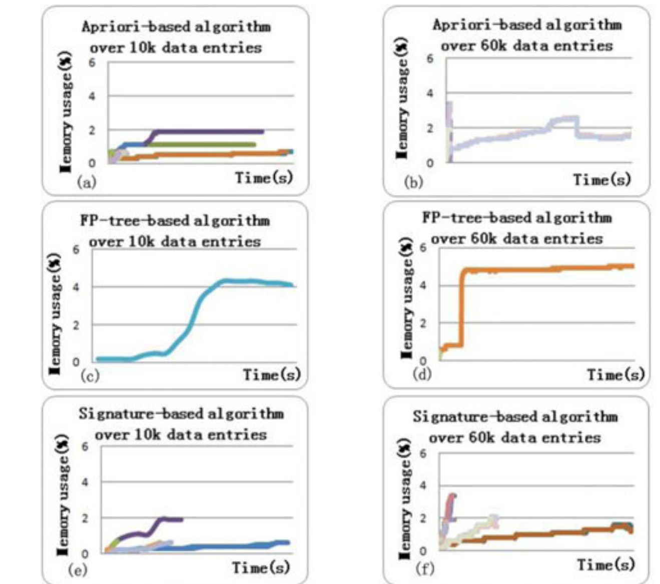


Fig. 4. Memory usage.

to measure the quality of the extracted graphs. Metrics characterizing graphs include *number of nodes*, *number of edges*, nodes' *strength distribution*, *community structure*, and *hop-plot* [24], [25], [26]. The meaning of the first two metrics are straight forward. The strength distribution is the probabilities of the nodes' strength, where the strength of a node is the sum of the weights of the edges connecting to the node. Comparing the nodes' strength distribution of two graphs measures their similarity in vertex importance in connectivity as well as the weights of the links [26]. A community is regarded as a set of similar nodes, i.e., nodes in different communities are dissimilar. Comparing the community structure of two graphs measures their similarity in their clumpiness, which can be calculated by computing the inconsistency rates of detected communities from two graphs. Finally, hop-plot is the sum of the total neighborhood size $N(h)$ for h hops starting from each node, which can be further used to calculate the expansion, eccentricity or effective diameter of the graph. Particularly, effective diameter is the minimum number of hops in which a given fraction (e.g., 90 percent) of all connected pairs of nodes can reach each other. Comparing the Hop-plot and effective diameter of two graphs measures their similarity in neighborhood structure and eccentricity. The number of nodes is assumed to be the same in our graph extraction, therefore, we will measure the accuracy of the extracted graphs in terms of *number of edges*, nodes' *strength distribution*, *community structure* and *hop-plot and effective diameter*.

Our set-up is based on a cluster consisting of 18 DOWN TC4600 blades, each of which has a 2-core 2.4 GHz processor, 2× 300 GB hard drivers, 64 GB of RAM and a network card with 1,000-gigabit Ethernet ports. We used OpenJDK 1.8.0 and Hadoop 1.2.1 to compile and run the codes. We experiment with two sets of datasets extracted from Facebook applications dataset II. The first set includes 11 subset datasets with size ranging from 10 to 200k, whose features are summarized in Table 7a, including the number of users (denoted as $N(u)$), the number of applications (denoted as $N(a)$), the maximum number of applications used by a user (denoted as $M(a,u)$), and the maximum number of users who install an application (denoted as $M(u,a)$). The second set is shown in Table 7b, where the number of the data entries are chosen from 1 to 8 k. As discussed in Section 2, we can at least extract two graphs from these two sets of data, one uses users as nodes, and the other uses applications as nodes. The experimental results are obtained by applying the three algorithms over the datasets.

4.1 Memory and Disk Usage

Figs. 4a and 4f depict the memory usage of the three algorithms over the 10k-entry and 60k-entry datasets respectively on a particular node during the last phase Reduce in the cluster, where users are chosen as nodes and different lines represent different tasks. We can see that all the algorithms consume more memory as the number of dataset entries increases. Meanwhile, we can also notice that both Apriori-based algorithm and signature-based algorithm have multiple concurrent executing tasks on a node, where the peak memory usage for each task lasts for very short time.

However, for the FP-tree-based algorithm, there is only one executing task on the node. In addition, the memory used for that particular task keeps its peak memory usage for quite a long time since the reducer keeps the tree in memory for the 2-itemset frequency mining. This will reduce the number of concurrent executing tasks on the node when the size of the dataset increases.

Fig. 5 depict the overall disk usage of the last phase of Map and Reduce of the three algorithms over the first set of data, where we can see that the Apriori-based consumes the most hard disk and it fails to complete the edge weight computation for dataset larger than 60k entries and the signature based algorithm fails to complete the edge weight computation for dataset larger than 160k entries. The failure of the execution lies in two reasons: (1) Each blade in our experiment has two 300 GB hard drivers, one of which is for backup. Therefore each task on one blade can consume less than 300 GB disk space. Once the output of the results requires more hard disk, it will fail to complete its

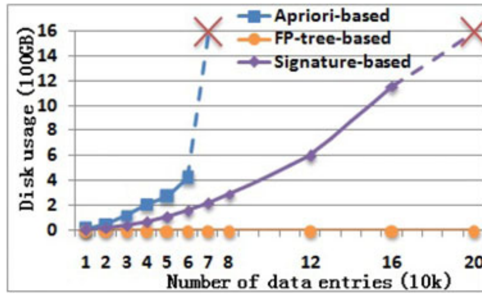


Fig. 5. Disk usage.

execution. (2) From Figs. 1 and 2 we can see that the mappers in the second Map phase in Apriori-based and signature-based algorithms output the intermediate 2-itemsets (i.e., an edge), where a particular edge can appear in the output of multiple mappers and the number of edges are actually very high.

For a given dataset, let the number of edges of the extracted graphs using Apriori-based algorithm, signature-based algorithm, and FP-tree-based algorithm be represented as E_a, E_s and E_f respectively, then the inconsistency rates between edge numbers of the extracted graphs using Apriori-based algorithm and the ones using signature-based algorithm can be calculated using $\Delta_{a,s} = \frac{|E_a - E_s|}{E_a}$, where n is the number of nodes. The numbers of edges of each extracted graphs and their inconsistency rates are recorded in Table 8, where (1) is for the ones when users are chosen as nodes and (2) is for the ones when applications are chosen as nodes. We can see that the Apriori-based and FP-based algorithms obtain the same number of edges while the signature-based algorithm obtains 40-70 percent more edges for the first set of data and 20-45 percent more edges for the second set of data. Considering the scale of the dataset is large, we only output edges whose weights are greater than a threshold. When the threshold is 0.5, the weights of edge u_1, u_4 in Figs. 1 and 3 are 0.25 and edge u_1, u_4 won't be output by the Apriori-based and FP-tree based algorithms. The collision of hash values in minHash scheme will increase the similarity of such nodes whose similarity are actually very low, e.g., the weight of edge u_1, u_4 in Fig. 2 is 0.5, which is greater than that in Figs. 1 and 3, and then edge u_1, u_4 will be output by the signature-based algorithm. That is why the signature-based algorithm outputs more edges. In our set-up, the threshold is 0.6, we still can see that the number of edges output is very high and it increases quickly when users are selected as nodes. From Table 8, we can see that the Apriori-based and FP-based algorithms obtain the same number of edges while the signature-based algorithm obtains more edges. However, from Fig. 3, we can see that the mappers in the second Map phase in FP-tree-based algorithm donot output the intermediate 2-itemsets, which makes it consume the least disk space.

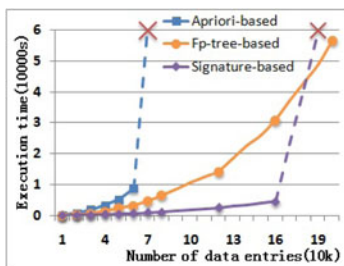
TABLE 8
Number of Edges of Extracted Graphs

Data		Apriori E_a	FP-tree E_f	Signature E_s	Signature $\Delta_{a,s}$
(1)	10k	230,990	230,990	343,190	48.6%
	20k	659,394	659,394	1,052,383	59.6%
	30k	1,321,659	1,321,659	2,175,868	64.6%
	40k	2,280,857	2,280,857	3,778,677	65.7%
	50k	3,441,583	3,441,583	5,740,634	66.8%
	60k	5,414,681	5,414,681	8,763,268	61.8%
(2)	10k	3,228	3,228	4,031	24.8%
	20k	2,434	2,434	3,227	32.6%
	30k	1,843	1,843	2,577	39.8%
	40k	1,763	1,763	2,488	41.1%
	50k	1,674	1,674	2,387	42.6%
	60k	1,566	1,566	2,234	42.7%

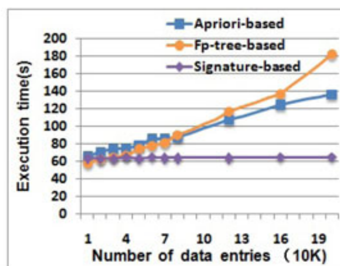
4.2 Execution Time

Figs. 6a and 6b depict the execution time of the three algorithms over the first dataset. When users are chosen as nodes, the execution time gets longer as the number of data entries gets larger. Particularly, the execution time of the Apriori-based algorithm exponentially grows against the number of data entries, which is much larger and grows much faster than those of the other two algorithms and it fails to complete its execution on dataset with the number of entries larger than 60 k because of short of disk spaces.

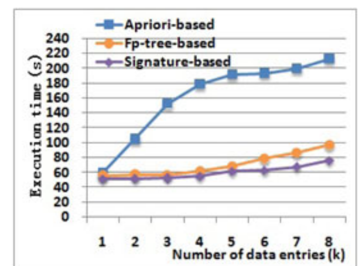
The signature-based algorithm achieves the shortest execution time once the disk space can meet its requirement. Though the FP-tree-based algorithm computes the edge weights by mining the frequency of the 2-itemset in the in-memory tree, it have one more phase of MapReduce for grouping. In addition, after the grouping algorithm completes its execution, there must be groups having the whole data tuples without any removal, e.g., g_1 in Reducer (2) in Fig. 3. The data dimension of such sub-dataset can be very high, whose value is positively correlated to the number of data entries. Therefore, as the number of the input data entries increases, corresponding data dimension increases, which makes the edge computation over this group of data much heavier than the others. In addition, the FP-tree



(a) Users as nodes (I)



(b) Applications as nodes



(c) User as nodes (II)

Fig. 6. Execution time.

constructed over this group of data in Reduce phase (II) in Fig. 3 is much deeper and wider than others, which consumes more memory and more execution time. Hadoop sets a memory usage limitation, i.e., VMem, for every computational node. When the memory consumed by a task exceeds VMem, it will be killed and migrated to be executed on another computational node. This can avoid that an exceptional large memory consumption caused by a task may handicap the necessary memory requirement of the others. But this runtime migration prolongs the execution time of the task. That is why once the number of the data entries increase to a certain value (e.g, in our experimental setup, 80 k), the execution time of the FP-tree-based algorithm grows faster. This makes the task over the sub-dataset with group id g_1 the performance bottleneck of the FP-tree-based algorithm and it can be even worse. Once none of the computation node can satisfy the task's memory requirement, it will also fail and the FP-tree-based algorithm will thus fail to complete the execution.

However, when applications are chosen as nodes, there is no obvious difference on the execution time among the three algorithms, where the small difference comes from the fluctuating delay of the data transmission over the network. As discussed in Section 3, the frequencies of the 2-itemsets are computed over the transformed datasets. Therefore, the bigger the number of users who install an application, i.e., $M(u, a)$, the more time the computation takes. As the number of data entries increases from 10 to 200 k, the value of $M(u, a)$ increases almost linearly from 8,615 to 170,735, as shown in Table 7a. However, for the same dataset, when applications are chosen as nodes, the values of $M(a, u)$ are 644 or 774 as the number data entries increases. That is why the execution time of these three algorithms does not always increase with the number of the data entries. Therefore, we have that the execution time of the edge weight computation over a dataset are mainly affected by the nodes' maximum feature dimension in its transformed dataset.

We can further verify our observation by carrying out another experiment over the second dataset with smaller number of data entries so that the value of their $M(u, a))$ is around the size of that of $M(a, u)$ in Table 7a. As shown in Table 7b, when the number of the data entries are chosen from 1 to 8k, the values of their $M(u, a)$ are within the scope $[862, 6924]$. Meanwhile, Fig. 6c shows that their corresponding execution time are all within the scope $[40s, 220s]$, which is consistent with those results shown in Fig. 6b.

4.3 Accuracy

There is no data reduction or compression during the graph extraction when the Apriori-based algorithm is applied. Therefore, the metrics characterizing the extracted graphs using Apriori-based algorithm are accurate and they can be used as a reference for accuracy measurement for the other two algorithms. The numbers of edges of each extracted graphs and their inconsistency rates have been presented in Table 8.

4.3.1 *Strength distribution*

First, we describe how we obtain the difference of the strength distribution of two graphs. For any node u_i in the extracted graph, let s_i represent the its strength and initially $s_i = 0$. Then the strength of each node can be calculated as follows: for all $(u_i, u_j) \in \mathcal{E}$, $s_i = s_i + w_{i,j}$ and $s_j = s_j + w_{i,j}$. Let $INT(s_i)$ obtain the biggest integer which is not larger than s_i, $m_d = \max_{u_i \in \mathcal{V}}(INT(s_i))$, and p_j represent the number of nodes with strength s_i and $INT(s_i) = j$. Then p_j can be calculated as follows: for all $u_i \in \mathcal{V}, p_{INT(s_i)} = p_{INT(s_i)} + 1$. The probability of the appearance of nodes with strength s_i and $INT(s_i) = j$ in G can be calculated as $Gp'_j = p_j/n$, where n is the number of nodes in the graph G. Let the inconsistency rates of the strength distribution of two graph G and G' be represented as $\Delta_s(G, G')$, then $\Delta_s(G, G')$ can be calculated as follows: $\Delta_s(G, G') = \frac{\sum |Gp'_i - G'p'_i|}{m_d}$.

Our experimental results show that the inaccuracy rates of strength distribution of the FP-tree-based algorithm against the Apriori-based one is always 0, meaning that it achieves 100 percent accuracy for measuring vertex importance in connectivity and link weights. The inconsistency rates of the strength distribution of signature-based algorithm against the Apriori-based one over the first set of dataset are depicted in Table 7a and 7b, where we can see that the inconsistency rates with users as nodes are almost an order of magnitude less than those with applications. In addition, we can see that the inconsistency rates are all very low and lower than 6 percent. This result is consistent with inconsistency rates between edge numbers of the extracted graphs using Apriori-based algorithm and the ones using signature-based algorithm, as presented in Table 8.

4.3.2 *Community Structure*

Before measuring the inconsistency rates of detected communities, we first introduce a clustering algorithm using minimum spanning tree (MST) [27] over the obtained graphs for community detection. Given a weighted graph, the MST based clustering algorithm constructs MSTs, then it removes edges with weights that are bigger than the predefined threshold. The process is repeated until k clusters are detected. The detected clusters form a set of communities. In order to apply the MST based clustering algorithm over the obtained graphs, the weight between two nodes need to be revised using Equation (4)

$$w'_{i,j} = 1 - w_{i,j}. \tag{4}$$

In this context, the graph accuracy can be measured by the inconsistency rates between the two community sets. To be specific, if two nodes are in the same community for set 1, while they are classified into different communities for set 2, then an inconsistency occurs between set 1 and set 2. The inconsistency rate between two community set is defined as the number of the occurrence of such inconsistency against the number of the pairs of any two nodes. More specifically, suppose the correct set of communities detected from a dataset is denoted as $C = \{C_1, C_2, \ldots, C_k\}$. Assuming that another detected set of communities to be measured is denoted as $C' = \{C'_1, C'_2, \ldots, C'_m\}$, note that here k and m can be different. For any node $u \in \bigcup_{i=1}^{k} C_i$, suppose $u \in C_x$ and $u \in C'_y$, then for any node $u' \in C_x$ and $u' \notin C'_y$, an inconsistency occurs since u and u' is in the same community for set C while they are not in the same one for set C'. Meanwhile, for and node $u'' \in C'_y$ and $u'' \notin C_x$, another

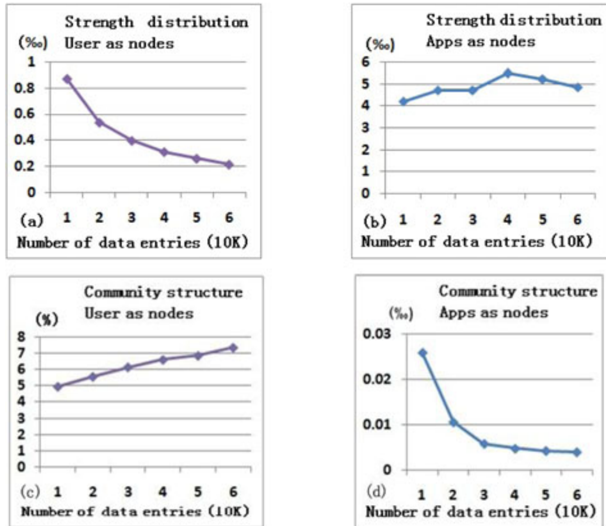


Fig. 7. Inconsistency rates of the signature-based algorithm against Apriori-based algorithm.

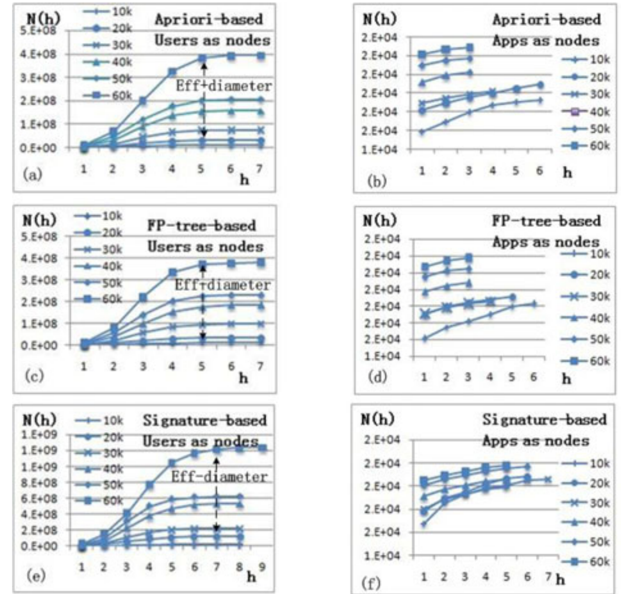


Fig. 8. Hop-plot and effective diameters.

inconsistency occurs since u and u'' are not in the same community for set C while they are in the same community C'_y for set C'. Let the inconsistency rate between C and C' be denoted as $\Delta_c(C, C')$, the total number of the inconsistency be denoted as $\sum(C, C')$, and n be the number of nodes. During the summation of inconsistency rates of any two nodes, the same inconsistency are counted twice. Meanwhile, for n nodes, there are $C_n^2 = n * (n-1)/2$ pair of two nodes. Therefore, $\Delta_c(C, C')$ can be calculated using Equation (5)

$$\Delta_c(C, C') = \frac{\sum(C, C')/2}{n * (n-1)/2} = \frac{\sum(C, C')}{n * (n-1)}. \tag{5}$$

Our experimental results show that the inaccuracy rate of the FP-tree-based algorithm against the Apriori-based one is always 0, meaning that it achieves 100 percent accuracy for community structure. The inaccuracy rates of the signature-based algorithm against the Apriori-based one over the first set of dataset are depicted in Figs. 7c and 7d, where we can see that the inconsistency rates with users as nodes are much larger than those with applications. In addition, when the number of the data entries gets larger, the changing of the inconsistency rates with users as nodes is quite different from the one with applications as nodes.

When users as chosen as nodes, the inconsistency rates increase as the number of the data entries get larger. As depicted in the Map phase (I) of Fig. 2, the bigger the number of a node's feature dimension is, the more corresponding hash values there will be. Then the probability of two nodes having the same minimum hash value will get smaller, i.e., the inaccuracy rates get smaller. Recall that when users are chosen as nodes, the number of applications, i.e, $\mathcal{N} = 8, 1{,}000$. In addition, the maximum number of applications used by as user, i.e., $M(a, u)$, is 644 for all datasets ranging from 5 to 25 k, as shown in Table 7a. Larger number of data entries means more users and bigger probability of the occurrence of hash value collisions, thus incurring bigger inconsistency rates.

When applications are chosen as nodes, the inconsistency rates decrease as the number of the data entries gets larger. Now, the rows of the hash values in the Map phase (I) of Fig. 2 will be denoted using users. When the number of entries increase, more users will be included into the hash value calculation. This will enlarge the scope of the hash values since they are location-aware. In addition, the value of maximum number of users who install an application, i.e., $M(u, a)$, increases as the number of the data entries gets larger, as shown in Table 7 (a). Both of these two data characteristics enlarge the scope of the hash values and make them scattered among the enlarging uids, reducing the probability of the occurrence of hash value collisions and thus the inconsistency rates.

4.3.3 Hop-Plot and Effective Diameter

For any two nodes $u_i \in G$ and $u_j \in G$, let $dist(u_i, u_j)$ represent the number of edges on the shortest path from u_i to u_j. Then the sum of total neighborhood size $N(h)$ for h hops can be calculated using the neighborhood function defined in Equation (6)

$$N(h) = |(u_i, u_j) : u_i \in V, u_j \in V, dist(u_i, u_j) \leq h|. \tag{6}$$

The exact computation of $N(h)$ is too expensive for large disk resident graphs. Fast and memory-efficient approaches like ANF [28], HyperANF [29] and FlajoletMartin (FM) sketh based approaches [30], [31] have been proposed for approximating the neighborhood function on large disk resident graphs with small errors. In our experiment, we compare the $N(h)$ of the graphs extracted using the signature-based and FP-tree-based algorithm against those using Apriori-based algorithm to compare their similarity, i.e., accuracy. In our experiment, we choose the source code of ANF[3] to compute the $N(h)$ for the graphs. HyperANF and FM sketh based approaches may outperform ANF in terms speed and scalability, but they produce similar approximation accuracy.

Figs. 8a-8f shows the Hop-plot and effective diameters (Eff-diameter in the figures for short) for two extracted

3. http://www.cs.cmu.edu/ christos/software.html

graphs of the three algorithms, where we can see that Apriori-based and FP-tree-based algorithms obtain similar results, while signature-based algorithm obtains different ones. For example, when users are chosen as nodes, the effective diameters of Apriori-based and FP-tree-based algorithms are around 5, but the effective diameter of the signature-based algorithm is around 7.

4.4 Summary

In summery, our experimental results have the following three important findings for edge weight computation, which can be useful for selecting the most appropriate algorithms for the edge weight computation for extracting weighted graphs for a given dataset.

- The edge weight computation time, i.e., execution time, is mainly affected by the maximum feature dimension of the nodes, where the signature-based algorithm obtains the shortest execution time and the Apriori-based algorithm obtains the longest one.
- During the computation, the FP-tree-based algorithm consumes the most memory while the other two algorithms consume more disk space. It will be better to choose Apriori-based and signature-based algorithms when the execution environment has limited memory; and it will be better to choose the FP-tree-based algorithm when the execution environment has limited disk space.
- The accuracy of the graph for the Apriori-based and FP-tree-based algorithms are 100 percent correct. When only the strength distribution is concerned, the signature-based algorithm can obtain fast and high precise result. When community structure is concerned, the accuracy depends on the data characteristics as well as the users' requirement. When the nodes' feature dimension and the maximum nodes' dimension are fixed, the accuracy decreases as the number of entries increases. While when nodes' feature dimension and the maximum nodes' dimension increases as the number of data entries increases, the accuracy can increase. When Hop-plot and effective diameters are concerned, it will be better choose Apriori-based or FP-tree-based algorithm.

5 Conclusion

Extracting weighted graphs from raw dataset is one of the indispensable preprocessing tasks for graph based data mining and machine learning. Recent years have witnessed the rapid growth of data from various applications, e.g., social networks, which presents great challenges to the edge weight computation for weighted graphs. In addition, existing work lacks of the measurement on the accuracy of the edge weights, which represents the graph accuracy and affects the following mining and learning results.

This paper carries out a systematic study on edge weight computation algorithms for extracting weighted graphs with MapReduce Framework over a real social network dataset, i.e., Facebook application data II. For a given data set, there can be more than one graphs be extracted. This paper describes the weighted graphs extracted from facebook application data II, classifies edge weight computation algorithms, presents their design and implementation with the MapReduce framework, measures and discusses their effectiveness. Particularly, this paper also propose comprehensive measurements on edge weight computation accuracy in terms of the number of edges, strength distribution, community structure, Hop-plot and effective diameters, regarding the graph based data mining applications. Our experimental results can be useful for making decision on selecting the most appropriate algorithms for the edge weight computation for extracting weighted graphs for a given dataset.

Finally, our current experiment results also demonstrate that the group based dataset partition scheme for the FP-tree-based algorithm can produce unbalanced data partition, our on-going research is investigating how to balance the FP-trees for edge weight computation with MapReduce framework. In addition, The effectiveness on other tools such as graph compression and partition has only been proved on graphs without edge weights their effectiveness on graphs with edge weights may need further study. But this is out of the research scope of this paper.

Acknowledgments

The authors would like to thank Prof. X. Cheng-zhong for his valued comments on improving the paper. We would also like to appreciate the support by the Shenzhen Science and Technology Foundation (JCYJ20150324140036842, JCYJ2015 0529164656096JCYJ201418193546117 and JCYJ2014050917 2609174), National Natural Science Foundation of China (61103001,61170077, and 61202377), National Key Technology Research and Development Program of the Ministry of Science and Technology of China (2014BAH28F05), the Guangdong Province Key Laboratory Project (2012A061400024), the Guangdong Natural Science Foundation (2014A030313553), the National High Technology Joint Research Program of China (2015AA015305), Science and Technology Planning Project of Guangdong Province (2013B090500055), National High Technology Joint Research Program of China (2015AA015305) and NSF-China and Guangdong Province Joint Project (U1301252). Z. Ming and R. Mao are the corresponding authors.

References

[1] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: Scalable graph ETL framework," in *Proc. 1st Int. Workshop Graph Data Manage. Exp. Syst*., 2013, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2484425.2484429

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[3] V. Nair and S. Dua, "Folksonomy-based ad hoc community detection in online social networks," *Soc. Netw. Anal. Mining*, vol. 2, no. 4, pp. 305–328, 2012.

[4] H. Kautz, B. Selman, and M. Shah, "Referral web: Combining social networks and collaborative filtering," *Commun. ACM*, vol. 40, no. 3, pp. 63–65, Mar. 1997.

[5] Y. M. Roberto J. Bayardo and R. Srikant, "Scaling up all pairs similarity search," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 131–140.

[6] A. Nazir, S. Raza, D. Gupta, C.-N. Chuah, and B. Krishnamurthy, "Network level footprints of facebook applications," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf*., 2009, pp. 63–75. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644901.

[7] P. Nancy and R. G. Ramani, "Frequent pattern mining in social network data (facebook application data)," *Eur. J. Sci. Res.*, vol. 79, pp. 531–540, 2012.

[8] P. Jaccard, "Nouvelles recherches sur la distribution florale," *Bulletin De La Societe De Vaud Des Sci. Naturflles*, vol. 44, pp. 223–270, 1908.

[9] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce," in *Proc. 6th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2012, pp. 76-1–76-8. [Online]. Available: http://doi.acm.org/10.1145/2184751.2184842.

[10] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.

[11] R. B. Zadeh and A. Goel, "Dimension independent similarity computation," *J. Machine Learning Res.*, vol. 14, no. 1, pp. 1605–1626, 2013.

[12] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Mining Knowl. Discovery*, vol. 8, no. 1, pp. 53–87, Jan. 2004.

[13] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel implementation of Apriori algorithm based on mapreduce," in *Proc. 13th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel Distrib. Comput.*, Aug. 2012, pp. 236–241.

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

[15] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang, "Finding interesting associations without support pruning," *IEEE Trans. Knowl. Data Eng.*, vol. 13, no. 1, pp. 64–78, Jan./Feb. 2001.

[16] C. H. C. Teixeira, A. Silva, and W. Meira. Jr., "Min-hash fingerprints for graph kernels: A trade-off among accuracy, efficiency, and compression," *J. Inf. Data Manage.*, vol. 3, no. 3, pp. 227–242, 2012.

[17] J. Drew and M. Hahsler, "Strand: Fast sequence comparison using mapreduce and locality sensitive hashing," in *Proc. 5th ACM Conf. Bioinf., Comput. Biol. Health Informat.*, 2014, pp. 506–513.

[18] D. Karapiperis and V. S. Verykios, "A distributed framework for scaling up LSH-based computations in privacy preserving record linkage," in *Proc. 6th Balkan Conf. Infor.*, 2013, pp. 102–109.

[19] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: Parallel FP-growth for query recommendation," in *Proc. ACM Conf. Recommender Syst.*, 2008, pp. 107–114. [Online]. Available: http://doi.acm.org/10.1145/1454008.1454027.

[20] M. J. C. Rares Vernica and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 495–506.

[21] D. Huang, Y. Song, R. Routray, and F. Qin, "Smart cache: An optimized mapreduce implementation of frequent itemset mining," in *Proc. IEEE Int. Conf. Cloud Eng.*, Mar. 2015, pp. 16–25.

[22] X. Wei, Y. Ma, F. Zhang, M. Liu, and W. Shen, "Incremental FP-growth mining strategy for dynamic threshold value and database based on mapreduce," in *Proc. IEEE 18th Int. Conf. Comput. Supported Cooperative Work Des.*, May 2014, pp. 271–276.

[23] M. Tiwary, A. Sahoo, and R. Misra, "Efficient implementation of Apriori algorithm on HDFS using GPU," in *Proc. Int. Conf. High Perform. Comput. Appl.*, Dec. 2014, pp. 1–7.

[24] D. F. Nettleton, "Data mining of social networks represented as graphs," *Comput. Sci. Rev.*, vol. 7, pp. 1–34, 2013.

[25] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surveys*, vol. 38, no. 1, pp. 1–69, 2006.

[26] A. Ioannis and T. Eleni, "Statistical analysis of weighted networks," *Discr. Dyn. Nature Soc.*, vol. 2008, p. 16, 2008.

[27] C. Zhong, D. Miao, and P. Fränti, "Minimum spanning tree based split-and-merge: A hierarchical clustering method," *Inf. Sci.*, vol. 181, no. 16, pp. 3397–3410, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2011.04.013.

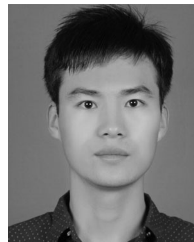
[28] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, "ANF: A fast and scalable tool for data mining in massive graphs," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2002, pp. 81–90.

[29] P. Boldi, M. Rosa, and S. Vigna, "HyperANF: Approximating the neighbourhood function of very large graphs on a budget," in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 625–634.

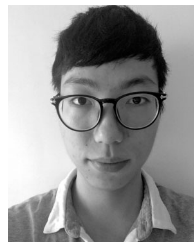
[30] R.-H. Li, J. X. Yu, X. Huang, H. Cheng, and Z. Shang, "Measuring the impact of MVC attack in large complex networks," *Inf. Sci.*, vol. 278, pp. 685–702, 2014.

[31] R.-H. Li and J. Yu, "Triangle minimization in large networks," *Knowl. Inf. Syst.*, vol. 45, no. 3, pp. 617–643, 2015.

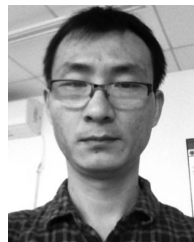
Yuhong Feng received the BS and PhD degrees from the University of Science and Technology of China, Hefei, China, and Singapore Nanyang Technological University, Singapore, respectively. She is currently an associate professor at the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. She was an assistant researcher at SIAT, CAS and a postdoctoral fellow at Hong Kong Polytechnic University, Hong Kong. Her research interests include workflow management, cloud computing, and data mining.



Junpeng Wang is currently working toward the masters' degree at the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include data analysis and Cloud computing.

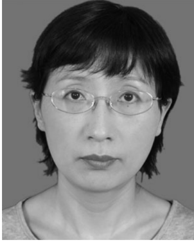


Zhiqiang Zhang received the BS degree from the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. He is currently a software developer at Beansmile, Guangzhou, China. His research interests include Web technologies, cloud computing, machine learning, and data mining.



Haoming Zhong received the BS and PhD degrees from University of Science and Technology of China, Hefei, China, and Singapore Nanyang Technological University, Singapore, respectively. He is currently a senior data architect in Big Data Center, WeBank, Shenzhen, China. He was the director of software architect at Moody's Analytics (San Francisco Bay Area). His research interests include artificial intelligence, data mining, and financial data analytics.

Zhong Ming is currently a professor at the College of Computer and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include Cloud computing, software engineering and embedded systems. He is also a member of a council and senior member of the Chinese Computer Federation.



Xuan Yang received the BS degree from the Xidian University, Xi'an, China, and, the MS and PhD degrees from the Xian Jiaotong University, Xi'an. She is currently a professor at the College of Computer and Software Engineering, Shenzhen University, Shenzhen, China. She has published more than 80 papers. Her research interests include intelligent information processing, image processing & analysis, and pattern recognition.

Rui Mao received the BS and PhD degrees in computer science from the University of Science and Technology of China, Hefei, China, and the University of Texas at Austin, Austin, TX, USA, respectively. He is currently an associate professor at the College of Computer and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include universal data management and analysis in metric space, and high performance computing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.