Scalable Big Graph Processing in MapReduce

Lu Qin[†], Jeffrey Xu Yu[‡], Lijun Chang[§], Hong Cheng[‡], Chengqi Zhang[†], Xuemin Lin[§♮]

[†]Centre for Quantum Computation and Intelligent Systems, University of Technology, Sydney, Australia
[‡] The Chinese University of Hong Kong, China
[§] The University of New South Wales, Australia [♮]East China Normal University, China

[†]{lu.qin,chengqi.zhang}@uts.edu.au
[‡]{yu,hcheng}@se.cuhk.edu.hk [§]{ljchang,lxue}@cse.unsw.edu.au

ABSTRACT

MapReduce has become one of the most popular parallel computing paradigms in cloud, due to its high scalability, reliability, and fault-tolerance achieved for a large variety of applications in big data processing. In the literature, there are MapReduce Class $\mathcal{MRC}$ and Minimal MapReduce Class $\mathcal{MMC}$ to define the memory consumption, communication cost, CPU cost, and number of MapReduce rounds for an algorithm to execute in MapReduce. However, neither of them is designed for big graph processing in MapReduce, since the constraints in $\mathcal{MMC}$ can be hardly achieved simultaneously on graphs and the conditions in $\mathcal{MRC}$ may induce scalability problems when processing big graph data. In this paper, we study scalable big graph processing in MapReduce. We introduce a Scalable Graph processing Class $\mathcal{SGC}$ by relaxing some constraints in $\mathcal{MMC}$ to make it suitable for scalable graph processing. We define two graph join operators in $\mathcal{SGC}$, namely, $\mathcal{EN}$ join and $\mathcal{NE}$ join, using which a wide range of graph algorithms can be designed, including PageRank, breadth first search, graph keyword search, Connected Component (CC) computation, and Minimum Spanning Forest (MSF) computation. Remarkably, to the best of our knowledge, for the two fundamental graph problems CC and MSF computation, this is the first work that can achieve $O(\log(n))$ MapReduce rounds with $O(n + m)$ total communication cost in each round and constant memory consumption on each machine, where n and m are the number of nodes and edges in the graph respectively. We conducted extensive performance studies using two web-scale graphs *Twitter-2010* and *Friendster* with different graph characteristics. The experimental results demonstrate that our algorithms can achieve high scalability in big graph processing.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

Keywords

Graph; MapReduce; Cloud Computing; Big Data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593661.

1. INTRODUCTION

As one of the most popular parallel computing paradigms for big data, MapReduce [10] has been widely used in a lot of companies such as Google, Facebook, Yahoo, and Amazon to process a large amount of data in the order of tera-bytes everyday. The success of MapReduce is due to its high scalability, reliability, and fault-tolerance achieved for a large variety of applications and its easy-to-use programming model that allows developers to develop parallel data-driven algorithms in a distributed shared nothing environment. A MapReduce algorithm executes in rounds. Each round has three phases: *map*, *shuffle*, and *reduce*. The *map* phase generates a set of key-value pairs using a *map* function, the *shuffle* phase transfers the key-value pairs into different machines and ensures that key-value pairs with the same key arrive at the same machine, and the *reduce* phase processes all key-value pairs with the same key using a *reduce* function.

Motivation: In the literature, there are researches that define algorithm classes in MapReduce in terms of memory consumption, communication cost, CPU cost, and the number of rounds. Karloff et al. [23] give the first attempt in which the MapReduce Class ($\mathcal{MRC}$) is proposed. $\mathcal{MRC}$ defines the maximal requirements for an algorithm to execute in MapReduce, in the sense that if any condition in $\mathcal{MRC}$ is violated, running the algorithm in MapReduce is meaningless. Nevertheless, a better class is highly demanded to guide the development of more stable and scalable MapReduce algorithms. Thus, Tao et al. [41] introduce the Minimal MapReduce Class ($\mathcal{MMC}$) in which several aspects can achieve optimality simultaneously. A lot of important database problems including sorting and sliding aggregation can be solved in $\mathcal{MMC}$. However, $\mathcal{MMC}$ is still incapable of solving a large range of problems especially for those involved in graph processing, which is an important branch of big data processing. The reasons are twofold. First, a graph usually has some inherent characteristics which make it hard to achieve high parallelism. For example, a graph is usually unstructured and highly irregular, making the locality of the graph very poor [30]. Second, the loosely synchronised shared nothing computing structure in MapReduce makes it difficult to achieve high workload balancing and low communication cost simultaneously as defined in $\mathcal{MMC}$ when processing graphs (see Section 3 for more details). Motivated by this, in this paper, we relax some conditions in $\mathcal{MMC}$ and define a new class of MapReduce algorithms that is more suitable for scalable big graph processing.

Contributions: We make the following contributions in this paper.

(1) New class defined for scalable graph processing: We define a new class $\mathcal{SGC}$ for scalable graph processing in MapReduce. We aim at achieving three goals: *scalability*, *stability*, and *robustness*. *Scalability* requires an algorithm to achieve good speed-up w.r.t the

number of machines used. *Stability* requires an algorithm to terminate in bounded number of rounds. *Robustness* requires that an algorithm never fails regardless of how much memory each machine has. $\mathcal{SGC}$ relaxes two constraints defined in $\mathcal{MMC}$, namely, the communication cost on each machine, and the total number of rounds. For the former, we define a new cost, that balances the communication in a random manner where the randomness is related to the degree distribution of the graph. For the latter, we relax the $O(1)$ rounds defined in $\mathcal{MMC}$ to $O(\log(n))$ where n is the graph node number. In addition, we require the memory used in each machine to be loosely related to the size of the input data, in order to achieve high *robustness*. Such a condition is even stronger than that defined in $\mathcal{MMC}$. The *robustness* requirement is highly demanded by a commercial database system, with which a database administrator does not need to worry that the data grows too large to reside entirely in the total memory of the machines.

(2) Two elegant graph operators defined to solve a large range of graph problems: We define two graph join operators, namely, $\mathcal{NE}$ join and $\mathcal{EN}$ join. $\mathcal{NE}$ join propagates information from nodes to their adjacent edges and $\mathcal{EN}$ join aggregates information from adjacent edges to nodes. Both $\mathcal{NE}$ join and $\mathcal{EN}$ join can be implemented in $\mathcal{SGC}$. Using the two graph join operators, a large range of graph algorithms can be designed in $\mathcal{SGC}$, including PageRank, breadth first search, graph keyword search, Connected Component (CC) computation, and Minimum Spanning Forest (MSF) computation. Especially, for CC and MSF computation, it is non-trivial to solve them using graph operators in $\mathcal{SGC}$. To the best of our knowledge, for the two fundamental graph problems, this is the first work that can achieve $O(\log(n))$ MapReduce rounds with $O(n + m)$ total communication cost in each round and constant memory consumption on each machine, where n and m are the number of nodes and edges in the graph respectively. We believe our findings on MapReduce can also guide the development of scalable graph processing algorithms in other systems in cloud.

(3) Unified graph processing system: In all of our algorithms, we enforce the input and output of any graph operation to be either a node table or an edge table, with which a unified graph processing system can be designed. The benefits are twofold. First, the unified input and output make the system self-containable, on top of which more complex graph processing tasks can be designed by chaining several graph queries together. For example, one may want to find all connected components of the subgraph induced by nodes that are related to *photography* and *hiking*. This can be done by chaining three graph queries: a graph keyword search query, an induced subgraph query, and a connected component query, all of which are studied in this paper. Second, by chaining multiple queries with the unified input/output, more query optimization techniques can be developed and integrated into the system.

(4) Extensive performance studies: We conducted extensive performance studies using two real web-scale graphs *Twitter-2010* and *Friendster*, both of which have billions of edges. *Twitter-2010* has a smaller diameter with a skewed degree distribution, and *Friendster* has a larger diameter with a more uniform degree distribution. All of our algorithms can achieve high scalability on the two datasets.

Outline: Section 2 presents the preliminary. Section 3 introduces the scalable graph processing class $\mathcal{SGC}$ and two graph operators $\mathcal{NE}$ join and $\mathcal{EN}$ join in $\mathcal{SGC}$. Section 4 presents three basic graph algorithms in $\mathcal{SGC}$. Sections 5 and 6 show how to compute CC and MSF in $\mathcal{SGC}$ respectively. Section 7 evaluates the algorithms in $\mathcal{SGC}$ using extensive experiments. Section 8 reviews the related work and Section 9 concludes the paper.

2. PRELIMINARY

In this section, we introduce the MapReduce framework and review the two algorithm classes in MapReduce in the literature.

2.1 The MapReduce Framework

MapReduce, introduced by Google [10], is a programming model that allows developers to develop highly scalable and fault-tolerant parallel applications to process big data in a distributed shared nothing environment. A MapReduce algorithm executes in rounds. Each round involves three phases: *map*, *shuffle*, and *reduce*. Assuming that the input data is stored in a distributed file system as a set of key-value pairs, the three phases work as follows.

- *Map:* In this phase, each machine reads a part of the key-value pairs $\{(k^m_i, v^m_j)\}$ from the distributed file system and generates a new set of key-value pairs $\{(k^s_i, v^s_j)\}$ to be transferred to other machines in the *shuffle* phase.
- *Shuffle:* The key-value pairs $\{(k^s_i, v^s_j)\}$ generated in the map phase are shuffled across all machines. At the end of the shuffle phase, all the key-value pairs $\{(k^s_i, v^s_1), (k^s_i, v^s_2), \cdots\}$ with the same key k^s_i are guaranteed to arrive at the same machine.
- *Reduce:* Each machine groups the key-value pairs with the same key k^s_i together as $(k^s_i, \{v^s_1, v^s_2, \cdots\})$, from which a new set of key-value pairs $\{(k^r_i, v^r_j)\}$ is generated and stored in the distributed file system to be processed in the next round.

Two functions need to be implemented in each round: a *map* function and a *reduce* function. A *map* function determines how to generate $\{(k^s_i, v^s_j)\}$ from $\{(k^m_i, v^m_j)\}$ whereas a *reduce* function determines how to generate $\{(k^r_i, v^r_j)\}$ from $(k^s_i, \{v^s_1, v^s_2, \cdots\})$.

2.2 Algorithm Classes in MapReduce

In the literature, two algorithm classes have been introduced in MapReduce, in terms of disk usage, memory usage, communication cost, CPU cost, and number of MapReduce rounds. Let S be the set of objects in the problem and t be the number of machines in the system. The two classes are defined as follows.

MapReduce Class $\mathcal{MRC}$: The MapReduce Class $\mathcal{MRC}$ is introduced by Karloff et al. [23]. Fix a $\epsilon > 0$, a MapReduce algorithm in $\mathcal{MRC}$ should have the following properties:

- *Disk*: Each machine uses $O(|S|^{1-\epsilon})$ disk space. The total disk space used is $O(|S|^{2-2\epsilon})$.
- *Memory*: Each machine uses $O(|S|^{1-\epsilon})$ memory. The total memory used is $O(|S|^{2-2\epsilon})$.
- *Communication*: In each round, each machine sends/receives $O(|S|^{1-\epsilon})$ data. The total communication cost is $O(|S|^{2-2\epsilon})$.
- *CPU*: In each round, the CPU consumption on each machine is $O(poly(|S|))$, i.e., polynomial to $|S|$.
- *Number of rounds*: The number of rounds is $O(\log^i |S|)$ for a constant $i \geq 0$.

Minimal MapReduce Class $\mathcal{MMC}$: The Minimal MapReduce Class $\mathcal{MMC}$ is introduced by Tao et al. [41] which aims to achieve outstanding efficiency in multiple aspects simultaneously. A MapReduce algorithm in $\mathcal{MMC}$ should have the following properties:

- *Disk*: Each machine uses $O(\frac{|S|}{t})$ disk space. The total disk space used is $O(|S|)$.
- *Memory*: Each machine uses $O(\frac{|S|}{t})$ memory. The total memory used is $O(|S|)$.
- *Communication*: In each round, each machine sends/receives $O(\frac{|S|}{t})$ data. The total communication cost is $O(|S|)$.
- *CPU*: In each round, the CPU consumption on each machine is $O(\frac{T_{seq}}{t})$, where T_{seq} is the time to solve the same problem on a single sequential machine.
- *Number of rounds*: The number of rounds is $O(1)$.

3. SCALABLE GRAPH PROCESSING

$\mathcal{MRC}$ defines the basic requirements for an algorithm to execute in MapReduce, whereas $\mathcal{MMC}$ requires several aspects to achieve optimality simultaneously in a MapReduce algorithm. In the following, we analyze the problems involved in $\mathcal{MRC}$ and $\mathcal{MMC}$ in graph processing and propose a new class $\mathcal{SGC}$ which is suitable for scalable graph processing in MapReduce.

We first analyze $\mathcal{MMC}$. Consider a graph $G(V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. A common graph operation is to exchange data among all adjacent nodes (nodes that share a common edge) in the graph G. The memory constraint in $\mathcal{MMC}$ requires that all edges/nodes are distributed evenly among all machines in the system. Let $E_{i,j}$ be the set of edges (u, v) in G such that u is in machine i and v is in machine j. The communication constraint in $\mathcal{MMC}$ can be formalized as follows:

$$\max_{1\leq i\leq t}(\Sigma_{1\leq j\leq t,j\neq i}|E_{i,j}|) \leq O((n+m)/t) \quad (1)$$

This requires minimizing $\max_{1\leq i\leq t}$ $(\Sigma_{1\leq j\leq t,j\neq i}$ $|E_{i,j}|)$ which is an NP-hard problem [13]. Furthermore, even if the optimal solution is computed, it is not guaranteed that min $(\max_{1\leq i\leq t}$ $(\sum_{1\leq j\leq t,j\neq i}$ $|E_{i,j}|)) \leq O(\frac{n+m}{t})$. Thus, $\mathcal{MMC}$ is not suitable to define a graph algorithm in MapReduce.

Next, we discuss $\mathcal{MRC}$. Since $\mathcal{MRC}$ defines the basic conditions that a MapReduce algorithm should satisfy, a graph algorithm in MapReduce is not an exception. However, like $\mathcal{MMC}$, a better class is always desirable to be defined for more stably and scalably graph processing in MapReduce. Given a graph $G(V, E)$ with n nodes and m edges, assume that $m \geq n^{1+c}$, the authors in [23] define a class based on $\mathcal{MRC}$ for graph processing in MapReduce, in which a MapReduce algorithm has the following properties:

- *Disk*: Each machine uses $O(n^{1+\frac{c}{2}})$ disk space. The total disk space used is $O(m^{1+\frac{c}{2}})$.
- *Memory*: Each machine uses $O(n^{1+\frac{c}{2}})$ memory. The total memory used is $O(m^{1+\frac{c}{2}})$.
- *Communication*: In each round, each machine sends/receives $O(n^{1+\frac{c}{2}})$ data. The total communication cost is $O(m^{1+\frac{c}{2}})$.
- *CPU*: In each round, the CPU consumption on each machine is $O(poly(m))$, i.e., polynomial to m.
- *Number of rounds*: The number of rounds is $O(1)$.

Such a class has a good property that the algorithm runs in constant rounds. However, it requires each machine to use $O(n^{1+\frac{c}{2}})$ memory, which can be large even for a dense graph. When the memory of each machine cannot hold $O(n^{1+\frac{c}{2}})$ data, the algorithm fails no matter how many machines are used in the system. Thus, the class is not scalable to handle a graph with large n.

3.1 The Scalable Graph Processing Class $\mathcal{SGC}$

We now explore a better class that is suitable for graph processing in MapReduce. We aim at defining a MapReduce class in which a graph algorithm has the following three properties:

- *Scalability:* The algorithm can always be speeded up by adding more machines.
- *Stability:* The algorithm stops in bounded rounds.
- *Robustness:* The algorithm never fails regardless of how much memory each machine has.

It is difficult to distribute the communication cost evenly among all machines for a graph algorithm in MapReduce. The main reason is due to the skewed degree distribution (e.g., power-law distribution) for a large range of real-life graphs, in which some nodes may have very high degrees. Hence, instead of using $O(\frac{m+n}{t})$ as the upper bound of communication cost per machine, we define a weaker bound, denoted as $\tilde{O}(\frac{m}{t}, D(G,t))$. Suppose the nodes are uniformly distributed among all machines, denote by V_i the set of nodes stored in machine i for $1 \leq i \leq t$, and let d_j be the degree of node v_j in the input graph, $\tilde{O}(\frac{m}{t}, D(G,t))$ is defined as:

$$\tilde{O}(\frac{m}{t}, D(G,t)) = O(\max_{1\leq i\leq t}(\Sigma_{v_j\in V_i} d_j)) \quad (2)$$

$$D(G,t) = \frac{t-1}{t^2}\Sigma_{v_j\in V} d_j^2 \quad (3)$$

	$\mathcal{MRC}$	$\mathcal{MMC}$	$\mathcal{SGC}$
Disk/machine	$O(n^{1+\frac{c}{2}})$	$O(\frac{n+m}{t})$	$O(\frac{n+m}{t})$
Disk/total	$O(m^{1+\frac{c}{2}})$	$O(n+m)$	$O(n+m)$
Memory/machine	$O(n^{1+\frac{c}{2}})$	$O(\frac{n+m}{t})$	$O(1)$
Memory/total	$O(m^{1+\frac{c}{2}})$	$O(n+m)$	$O(t)$
Communication/machine	$O(n^{1+\frac{c}{2}})$	$O(\frac{n+m}{t})$	$\tilde{O}(\frac{m}{t}, D(G,t))$
Communication/total	$O(m^{1+\frac{c}{2}})$	$O(n+m)$	$O(n+m)$
CPU/machine	$O(poly(m))$	$O(\frac{T_{seq}}{t})$	$\tilde{O}(\frac{m}{t}, D(G,t))$
CPU/total	$O(poly(m))$	$O(T_{seq})$	$O(n+m)$
Number of rounds	$O(1)$	$O(1)$	$O(\log(n))$

Table 1: Graph Algorithm Classes in MapReduce

Lemma 3.1: *Let x_i ($1 \leq i \leq q$) be the communication cost upper bound for machine i, i.e., $x_i = \sum_{v_j\in V_i} d_j$, the expected value of x_i, $E(x_i) = \frac{2m}{t}$, and the variance of x_i, $Var(x_i) = D(G,t)$.* □

The proof of Lemma 3.1 is omitted due to space limitation. Note that the variance of the degree distribution of G, denoted $Var(G)$, is $(\sum_{v_j\in V}(d_j - \frac{2m}{n})^2)/n = (n\Sigma_{v_j\in V} d_j^2 - 4m^2)/n^2$. For fixed t, n, and m values, minimizing $D(G,t)$ is equivalent to minimizing $Var(G)$. In other words, the variance of communication cost for each machine is minimized if all nodes in the graph have the same degree. We define the scalable graph processing class $\mathcal{SGC}$ below.

Scalable Graph Processing Class $\mathcal{SGC}$: A graph MapReduce algorithm in $\mathcal{SGC}$ should have the following properties:

- *Disk*: Each machine uses $O(\frac{m+n}{t})$ disk space. The total disk space used is $O(m+n)$. This is the minimal requirement, since we need at least $O(m+n)$ disk space to store the data.
- *Memory*: Each machine uses $O(1)$ memory. The total memory used is $O(t)$. This is a very strong constraint, to ensure the *robustness* of the algorithm. Note that the memory defined here is the memory used in the map and reduce phases. There is also memory used in the shuffle phase, which is usually predefined by the system and is independent of the algorithm.
- *Communication*: In each round, each machine sends/receives $\tilde{O}$ $(\frac{m}{t}, D(G,t))$ data, and the total communication cost is $O(m+n)$, where G is the input graph in the round.
- *CPU*: In each round, the CPU cost on each machine is $\tilde{O}$ $(\frac{m}{t}$, $D(G,t))$, where G is the input graph in the round. The CPU cost defined here is the cost spent in the map and reduce phases.
- *Number of rounds*: The number of rounds is $O(\log(n))$.

Discussion: For the memory constraint, $\mathcal{SGC}$ only requires each machine to use constant memory, that is to say, even if the total memory of the system is smaller than the input data, the algorithm can still be processed successfully. This is an even stronger constraint than that defined in $\mathcal{MMC}$. Nevertheless, we give the flexibility for the algorithm to run other query optimization tasks using the free memory, which can be orthogonally studied to our work. Given the constraints on memory, communication, and CPU, it is nearly impossible for a wide range of graph algorithms to be processed in constant rounds in MapReduce. Thus, we relax the $O(1)$ rounds defined in $\mathcal{MMC}$ to $O(\log(n))$ rounds, which is reasonable since $O(\log(n))$ is the processing time lower bound for a large number of parallel graph algorithms in the parallel random-access machines, and is practical for the MapReduce framework as evidenced by our experiments. The comparison of the three classes for graph processing in MapReduce is shown in Table 1.

3.2 Two Graph Operators in $\mathcal{SGC}$

We assume that a graph $G(V, E)$ is stored in a distributed file system as a node table V and an edge table E. Each node in the table has a unique id and some other information such as label and keywords. Each edge in the table has id_1, id_2 defining the source and target node ids of the edge, and some other information such as weight and label. We use the node id to represent the node if it is obvious. G can be either directed or undirected. For an undirected graph, each edge is stored as two edges (id_1, id_2) and (id_2, id_1). In the following, we introduce two graph operators in $\mathcal{SGC}$, namely, $\mathcal{NE}$ join, and $\mathcal{EN}$ join, using which a large range of graph problems can be designed.

3.2.1 $\mathcal{NE}$ Join

An $\mathcal{NE}$ join aims to propagate the information on nodes into edges, i.e., for each edge $(v_i, v_j) \in E$, an $\mathcal{NE}$ join outputs an edge $(v_i, v_j, F(v_i))$ (or $(v_i, v_j, F(v_j))$) where $F(v_i)$ (or $F(v_j)$) is a set of functions operated on v_i (or v_j) in the node table V. Given a node table V_i and an edge table E_j, an $\mathcal{NE}$ join of V_i and E_j can be formulated using the following algebra:

$$\Pi_{id_1,id_2,f_1(c_1)\to p_1,f_2(c_2)\to p_2,\cdots}(\sigma_{cond(c)}((V_i \to V) \\ \bowtie^{\mathcal{NE}}_{id,id'} (E_j \to E)))|C(cond'(c')) \to cnt \quad (4)$$

or equivalently in the following SQL form:

> **select** $id_1, id_2, f_1(c_1)$ **as** p_1, $f_2(c_2)$ **as** $p_2, \cdots$
> **from** V_i **as** V $\mathcal{NE}$ **join** E_j **as** E **on** $V.id = E.id'$
> **where** $cond(c)$
> **count** $cond'(c')$ **as** cnt

where each of c, c', c_1, c_2, $\cdots$ is a subset of fields in the two tables V_i and E_j, f_k is a function operated on the fields c_k, and *cond* and *cond*$'$ are two functions that return either *true* or *false* defined on the fields in c and c' respectively. id' can be either id_1 or id_2. The **count** part counts the number of *true*s returned by $cond'(c')$ and the number is assigned to a counter *cnt*, which is useful in determining a terminate condition for an iterative algorithm.

$\mathcal{NE}$ join in MapReduce: The $\mathcal{NE}$ join operation can be implemented in MapReduce as follows. Let the set of fields used in V be c_v, and the set of fields used in E be c_e. In the *map* phase, for each node $v \in V$, the values in c_v with key $v.id$ are emitted as a key-value pair $(v.id, v.c_v)$. For each edge $e \in E$, the values in c_e with key $e.id'$ are emitted as a key-value pair $(e.id', e.c_e)$. In the *reduce* phase, for each node id, the set of key-value pairs $\{(id, v.c_v), (id, e_1.c_e), (id, e_2.c_e), \cdots\}$ can be processed as a data stream without loading the whole set into memory. Assuming that $(id, v.c_v)$ comes first before all other key-value pairs $(id, e_i.c_e)$ in the stream (this can be implemented as a secondary sort in MapReduce), the algorithm first loads $(id, v.c_v)$ into memory and then processes each $(id, e_i.c_e)$ one by one. For a certain $(id, e_i.c_e)$, the algorithm checks $cond(c)$. If $cond(c)$ returns *false*, the algorithm skips $(id, e_i.c_e)$ and continues to process the next $(id, e_{i+1}.c_e)$. Otherwise, the algorithm calculates all $f_j(c_j)$ and $cond'(c')$ from $(id, v.c_v)$ and $(id, e_i.c_e)$, outputs the selected fields as a single tuple into the distributed file system, and increases *cnt* if $cond'(c')$ returns *true*. It is easy to see that $\mathcal{NE}$ join belongs to $\mathcal{SGC}$.

3.2.2 $\mathcal{EN}$ Join

An $\mathcal{EN}$ join aims to aggregate the information on edges into nodes, i.e., for each node $v_i \in V$, an $\mathcal{EN}$ join outputs a node $(v_i, G(adj(v_i)))$ where $adj(v_i) = \{(v_i, v_j) \in E\}$, and G is a set of decomposable aggregate functions on the edge set $adj(v_i)$, where a decomposable aggregate function is defined as follows:

Definition 3.1: *(Decomposable Aggregate Function)* An aggregate function g_k is decomposable if for any dataset s, and any two subsets of s, s_1 and s_2, with $s_1 \cap s_2 = \emptyset$ and $s_1 \cup s_2 = s$, $g_k(s)$ can be computed using $g_k(s_1)$ and $g_k(s_2)$. □

Algorithm 1 PageRank$(V(id), E(id_1, id_2), d)$

1: $V_r \leftarrow \prod_{id,cnt(id_2)\to d,\frac{1}{|V|}\to r}(V \bowtie^{\mathcal{EN}}_{id,id_1} E)$;
2: **for** $i = 1$ **to** d **do**
3: $\quad E_r \leftarrow \prod_{id_1,id_2,\frac{r}{d}\to p}(V_r \bowtie^{\mathcal{NE}}_{id,id_1} E)$;
4: $\quad V_r \leftarrow \prod_{id,d,\frac{\alpha}{|V|}+(1-\alpha)sum(E_r.p)\to r}(V_r \bowtie^{\mathcal{EN}}_{id,id_2} E_r)$;
5: **return** $\prod_{id,r}(V_r)$;

Given a node table V_i and an edge table E_j, an $\mathcal{EN}$ join of V_i and E_j can be formulated using the following algebra:

$$\Pi_{id,g_1(c_1)\to p_1,g_2(c_2)\to p_2,\cdots}(\sigma_{cond(c)}((V_i \to V) \\ \bowtie^{\mathcal{EN}}_{id,id'} (E_j \to E)))|C(cond'(c')) \to cnt \quad (5)$$

or equivalently in the following SQL form:

> **select** id, $g_1(c_1)$ **as** p_1, $g_2(c_2)$ **as** $p_2, \cdots$
> **from** V_i **as** V $\mathcal{EN}$ **join** E_j **as** E **on** $V.id = E.id'$
> **where** $cond(c)$
> **group by** id
> **count** $cond'(c')$ **as** cnt

where each of c, c', c_1, c_2, $\cdots$ is a subset of fields in the two tables V_i and E_j, and id' can be either id_1 or id_2. The **where** part and **count** part are analogous to those defined in $\mathcal{NE}$ join. g_k is a decomposable aggregate function operated on the fields in c_k, by grouping the results using node ids as denoted in the **group by** part. Since the group by field is always the node id, we omit the group by part in Eq. 5 for simplicity.

$\mathcal{EN}$ join in MapReduce: The $\mathcal{EN}$ join operation can be implemented in MapReduce as follows. Let the set of fields used in V be c_v, and the set of fields used in E be c_e. The *map* phase is similar to that in the $\mathcal{NE}$ join. That is, for each node $v \in V$, the values in c_v with key $v.id$ are emitted as a key-value pair $(v.id, v.c_v)$, and for each edge $e \in E$, the values in c_e with key $e.id'$ are emitted as a key-value pair $(e.id', e.c_e)$. In the *reduce* phase, for each node id, the set of key-value pairs $\{(id, v.c_v), (id, e_1.c_e), (id, e_2.c_e), \cdots\}$ can be processed as a data stream without loading the whole set into memory. Assuming that $(id, v.c_v)$ comes first before all other key-value pairs $(id, e_i.c_e)$ in the stream, the algorithm first loads $(id, v.c_v)$ into memory and then processes each $(id, e_i.c_e)$ one by one. For each function g_k, since g_k is decomposable, $g_k(\{e_1, e_2, \cdots, e_i\})$ can be calculated using $g_k(\{e_1, e_2, \cdots, e_{i-1}\})$ and $g_k(\{e_i\})$. After processing all $(id, e_i.c_e)$, all the g_k functions are computed. Finally, the algorithm checks $cond(c)$. If $cond(c)$ returns *true*, it outputs the id as well as all the g_k values as a single tuple into the distributed file system and increases *cnt* if $cond'(c')$ returns *true*. It is easy to see that $\mathcal{EN}$ join belongs to $\mathcal{SGC}$.

4. BASIC GRAPH ALGORITHMS

The combination of $\mathcal{NE}$ join and $\mathcal{EN}$ join can solve a wide range of graph problems in $\mathcal{SGC}$. In this section, we introduce some basic graph algorithms, including PageRank, breadth first search, and graph keyword search, in which the number of rounds is determined by a user given parameter or a graph factor which is small and can be considered as a constant. We will introduce more complex algorithms that need logarithmic rounds in the worst case in the next sections, including connected component and minimum spanning forest computation.

PageRank. PageRank is a key graph operation which computes the rank of each node based on the links (directed edges) among them. Given a directed graph $G(V, E)$, PageRank is computed iteratively. Let the initial rank of each node be $\frac{1}{|V|}$, in iteration i, the rank of a

Algorithm 2 $\mathsf{BFS}(V(id), E(id_1, id_2, s)$

1: $V_d \leftarrow \prod_{id, id=s?0:\phi \to d}(V)$
2: **for** $i = 1$ **to** $+\infty$ **do**
3: $\quad E_d \leftarrow \prod_{id_1, id_2, d \to d_1}(V_d \bowtie^{\mathcal{NE}}_{id,id_1} E)$;
4: $\quad V_d \leftarrow \prod_{id, ((d=\phi \wedge min(d_1) \neq \phi)?i:d) \to d}(V_d \bowtie^{\mathcal{EN}}_{id,id_2} E_d) \mid C(d = i) \to n_{new}$;
5: $\quad$ **if** $n_{new} = 0$ **then break**;
6: **return** V_d;

Algorithm 3 $\mathsf{KWS}(V(id, t), E(id_1, id_2), \{k_1, \cdots, k_l\}, \mathsf{rmax})$

1: $V_r \leftarrow \prod_{id, k_1 \in t?(id,0):(\phi,\phi) \to (p_1, d_1), \cdots, k_l \in t?(id,0):(\phi,\phi) \to (p_l, d_l)}(V)$;
2: **for** $i = 1$ **to** rmax **do**
3: $\quad E_r \leftarrow \prod_{id_1, id_2, (p_1, d_1) \to (p_{e_1}, d_{e_1}), \cdots, (p_l, d_l) \to (p_{e_l}, d_{e_l})}(V_r \bowtie^{\mathcal{NE}}_{id,id_1} E)$;
4: $\quad V_r \leftarrow \prod_{id, amin(p_1, d_1, p_{e_1}, d_{e_1}+1) \to (p_1, d_1), \cdots, amin(p_l, d_l, p_{e_l}, d_{e_l}+1) \to (p_l, d_l)}(V_r \bowtie^{\mathcal{EN}}_{id,id_2} E_r)$;
5: **return** $\prod_{V_r.*}(\sigma_{d_1 \neq \phi \wedge \cdots \wedge d_l \neq \phi}(V_r))$;

node v is computed as $r_i(v) = \frac{\alpha}{|V|} + (1-\alpha) \sum_{u \in nbr_{in}(v)} \frac{r_{i-1}(u)}{d(u)}$, where $0 < \alpha < 1$ is a parameter, $nbr_{in}(v)$ is the set of in-neighbors of v in G, and $d(u)$ is the number of out-neighbors of u in G.

The PageRank algorithm in $\mathcal{SGC}$ is shown in Algorithm 1. Given the node table $V(id)$, the edge table $E(id_1, id_2)$, and the number of iterations d, initially, the algorithm computes the out-degree of each node using $V \bowtie^{\mathcal{EN}}_{id,id_1} E$, assigns an initial rank $\frac{1}{|V|}$ to each node, and generates a new table V_r (line 1). Then, the algorithm updates the node ranks in d iterations. In each iteration, the ranks of nodes are updated using an $\mathcal{NE}$ join followed by an $\mathcal{EN}$ join. In the $\mathcal{NE}$ join, the partial rank $p(v) = \frac{r(v)}{d(v)}$ for each node v is propagated to all its outgoing edges using $V_r \bowtie_{id,id_1} E$ and a new edge table E_r is generated (line 3). In the $\mathcal{EN}$ join, for each node v, the partial ranks $p(u)$ from all its incoming edges (u, v) are aggregated. The new rank is computed as $\frac{\alpha}{|V|} + (1-\alpha) \sum_{u \in nbr_{in}(v)}(p(u))$ using $V_r \bowtie^{\mathcal{EN}}_{id,id_2} E_r$, and V_r is updated with the new ranks (line 4).

Breadth First Search. Breadth First Search (BFS) is a fundamental graph operation. Given an undirected graph $G(V, E)$, and a source node s, a BFS computes for every node $v \in V$ the shortest distance (i.e., the minimum number of hops) from s to v in G.

The BFS algorithm in $\mathcal{SGC}$ is shown in Algorithm 2. Given a node table $V(id)$, an edge table $E(id_1, id_2)$, and a source node s, the algorithm computes for each node v the shortest distance $d(v)$ from s to v. Initially, a node table V_d is created with $d(s) = 0$ and $d(v) = \phi$ for $v \neq s$ (line 1). Next, the algorithm iteratively computes the nodes with $d(v) = i$ from nodes with $d(v) = i - 1$. Each iteration i is processed using an $\mathcal{NE}$ join followed by an $\mathcal{EN}$ join. The $\mathcal{NE}$ join propagates $d(u)$ into each edge (u, v) using $V_d \bowtie^{\mathcal{NE}}_{id,id_1} E$ and produces a new table E_d. The $\mathcal{EN}$ join updates all $d(v)$ based on the following rule:

(Distance Update Rule): In the i-th iteration of BFS, a node v is assigned $d(v) = i$ iff in the $(i-1)$-th iteration, $d(v) = \phi$ and there exists a neighbor u of v such that $d(u) \neq \phi$.

The rule can be easily implemented using $V_d \bowtie^{\mathcal{EN}}_{id,id_2} E_d$ (line 4) in which it also computes a counter n_{new} which is the number of nodes with $d(v) = i$. When $n_{new} = 0$, the algorithm terminates. It is easy to see that the number of iterations for Algorithm 2 is no larger than the diameter of the graph G. Thus the algorithm belongs to $\mathcal{SGC}$ if the diameter of the graph is small.

Graph Keyword Search. We now investigate a more complex algorithm, namely, keyword search in an undirected graph $G(V, E)$.

Algorithm 4 $\mathsf{CC}(V(id), E(id_1, id_2))$

1: $V_m \leftarrow \prod_{id, min(id, id_2) \to p}(V \bowtie^{\mathcal{EN}}_{id,id_1} E)$;
2: $V_c \leftarrow \prod_{V.id, V.p, cnt(V'.id) \to c}((V_m \to V) \bowtie^{\mathcal{EN}}_{id,p} (V_m \to V'))$;
3: $V_p \leftarrow \prod_{id, ((c=0 \wedge id=p)?min(id_2):p) \to p}(V_c \bowtie^{\mathcal{EN}}_{id,id_1} E)$;
4: **while** true **do**
5: $\quad V_s \leftarrow \mathsf{star}(V_p)$;
6: $\quad E_h \leftarrow \prod_{id_1, id_2, p \to p_1}(V_s \bowtie^{\mathcal{NE}}_{id,id_1} E)$;
7: $\quad V'_h \leftarrow \prod_{id, p, min(p, p_1) \to p_m}(\sigma_{s=1}(V_s \bowtie^{\mathcal{EN}}_{id,id_2} E_h))$;
8: $\quad V_h \leftarrow \prod_{V_s.id, (cnt(p_m)=0?V_s.p:min(p_m)) \to p}(V_s \bowtie^{\mathcal{EN}}_{id,p} V'_h)$;
9: $\quad V_s \leftarrow \mathsf{star}(V_h)$;
10: $\quad E_u \leftarrow \prod_{id_1, id_2, p \to p_1}(V_s \bowtie^{\mathcal{NE}}_{id,id_1} E)$;
11: $\quad V'_u \leftarrow \prod_{id, p, min(p_1 | p_1 \neq p) \to p_m}(\sigma_{s=1}(V_s \bowtie^{\mathcal{EN}}_{id,id_2} E_u))$;
12: $\quad V_u \leftarrow \prod_{V_s.id, (cnt(p_m)=0?V_s.p:min(p_m)) \to p}(V_s \bowtie^{\mathcal{EN}}_{id,p} V'_u)$;
13: $\quad V_p \leftarrow \prod_{V'.id, V.p}((V_u \to V) \bowtie^{\mathcal{NE}}_{id,p} (V_u \to V')) \mid C(V'.p \neq V.p) \to n_s$;
14: $\quad$ **if** $n_s = 0$ **then break**;
15: **return** V_p;

16: **Procedure** $\mathsf{star}(V_p)$
17: $V_g \leftarrow \prod_{V'.id, V'.p, V.p \to g, (V.p=V'.p?1:0) \to s}((V_p \to V) \bowtie^{\mathcal{NE}}_{id,p} (V_p \to V'))$;
18: $V'_s \leftarrow \prod_{V.id, V.p, and(V.s, V'.s) \to s}((V_g \to V) \bowtie^{\mathcal{EN}}_{id,g} (V_g \to V'))$;
19: $V_s \leftarrow \prod_{V'.id, V'.p, (V'.s=0?0:V.s) \to s}((V'_s \to V) \bowtie^{\mathcal{NE}}_{id,p} (V'_s \to V'))$;

20: **return** V_s;

Suppose for each $v \in V$, $t(v)$ is the text information included in v. Given a keyword query with a set of l keywords $Q = \{k_1, k_2, \cdots, k_l\}$, a keyword search [16, 17] finds a set of rooted trees in the form of $(r, \{(p_1, d(r, p_1)), (p_2, d(r, p_2)), \cdots, (p_l, d(r, p_l))\})$, where r is the root node, p_i is a node that contains keyword k_i in $t(p_i)$, and $d(r, p_i)$ is the shortest distance from r to p_i in G for $1 \leq i \leq l$. Each answer is uniquely determined by its root node r. rmax is the maximum distance allowed from the root node to a keyword node in an answer, i.e., $d(r, p_i) \leq \mathsf{rmax}$ for $1 \leq i \leq l$.

Graph keyword search can be solved in $\mathcal{SGC}$. The algorithm is shown in Algorithm 3. Given a node table $V(id, t)$, an edge table $E(id_1, id_2)$, a keyword query $\{k_1, k_2, \cdots, k_l\}$, and rmax, the algorithm first initializes a table V_r, where in each node v, for every k_i, a pair (p_i, d_i) is generated as $(id(v), 0)$ if k_i is contained in $v.t$, and (ϕ, ϕ) otherwise (line 1). Then the algorithm iteratively propagates the keyword information from each node to its neighbor nodes using rmax iterations. In each iteration, the keyword information for each node is first propagated into its adjacent edges using $\mathcal{NE}$ join, and then the information on edges is grouped into nodes to update the keyword information on each node using $\mathcal{EN}$ join. Specifically, the $\mathcal{NE}$ join generates a new edge table E_r, in which each edge (u, v) is embedded with keyword information $(p_1(u), d_1(u)), \cdots, (p_l(u), d_l(u))$ retrieved from node u using $V_r \bowtie^{\mathcal{NE}}_{id,id_1} E$ (line 3). In the $\mathcal{EN}$ join $V_r \bowtie^{\mathcal{EN}}_{id,id_2} E$ (line 4), each node updates its nearest node p_i that contains keyword k_i using an $amin$ function, which is defined as:

$$amin(\{(p_1, d_1), \cdots, (p_k, d_k)\}) = (p_i, d_i) | (d_i = min_{1 \leq j \leq k} d_j) \quad (6)$$

$amin$ is a decomposable since for any two sets s_1 and s_2 with $s_1 \cap s_2 = \emptyset$, the following equation holds:

$$amin(s_1 \cup s_2) = amin(\{amin(s_1), amin(s_2)\}) \quad (7)$$

After rmax iterations, for all nodes v in V_r, its nearest node p_i that contains keyword $k_i (1 \leq i \leq l)$ with distance $d_i = d(v, p_i) \leq \mathsf{rmax}$ is computed. The algorithm returns the nodes with $d_i \neq \phi$ for all $1 \leq i \leq l$ as the final set of answers (line 5).

5. CONNECTED COMPONENT

Given an undirected graph $G(V, E)$ with n nodes and m edges, a Connected Component (CC) is a maximal set of nodes that can

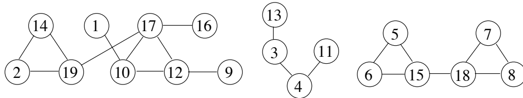


Figure 1: A Sample Graph $G(V, E)$

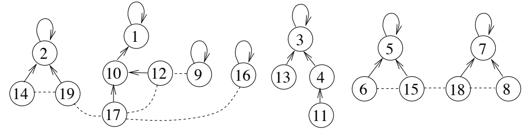


Figure 2: Initialize CC: Compute V_m

reach each other through paths in G. Computing all CCs of G is a fundamental graph problem and can be solved efficiently on a sequential machine using $O(n+m)$ time. However, it is non-trivial to solve the problem in MapReduce. Below, we briefly introduce the state-of-the-art algorithms for CC computation, followed by presenting our algorithm in $\mathcal{SGC}$.

5.1 State-of-the-art

We present three algorithms for CC computation in MapReduce: HashToMin, HashGToMin, and PRAM-Simulation. HashToMin and HashGToMin are two MapReduce algorithms proposed in [36], with a similar idea to use the smallest node in each CC as the representative of the CC, assuming that there is a total order among all nodes in G. PRAM-Simulation is to simulate the algorithm in the Parallel Random Access Machine (PRAM) model in MapReduce using the simulation method proposed in [23].

Algorithm HashToMin**:** Each node $v \in V$ maintains a set C_v initialized as $C_v = \{v\} \cup \{u|(u, v) \in E\}$. Let $v_{min} = \min\{u|u \in C_v\}$, the algorithm updates C_v in iterations until it converges. Each iteration is processed using MapReduce as follows. In the *map* phase, for each $v \in V$, two types of key-value pairs are emitted: (1) (v_{min}, C_v), and (2) $(u, \{v_{min}\})$ for all $u \in C_v$. In the *reduce* phase, for each $v \in V$, a set of key-value pairs are received in forms of $\{(v, C_v^1), \cdots, (v, C_v^k)\}$. The new C_v is updated as $\bigcup_{1 \leq i \leq k} C_v^i$. The HashToMin algorithm finishes in $O(\log(n))$ rounds[1], with $O(\log(n)(m+n))$ total communication cost in each round. The algorithm can be optimized to use $O(1)$ memory on each machine using secondary sort in MapReduce.

Algorithm HashGToMin**:** Each node $v \in V$ maintains a set C_v initialized as $C_v = \{v\}$. Let $C_{\geq v} = \{u|u \in C_v, u > v\}$ and $v_{min} = \min\{u|u \in C_v\}$, the algorithm updates C_v in iterations until it converges. Each iteration is processed using three MapReduce rounds. In the first two rounds, each round updates C_v as $C_v \cup \{u_{min}|(u, v) \in E\}$ in MapReduce. The third round is processed as follows. In the *map* phase, for each $v \in V$, two types of key-value pairs are emitted: (1) $(v_{min}, C_{\geq v})$, and (2) $(u, \{v_{min}\})$ for all $u \in C_{\geq v}$. In the *reduce* phase, for each $v \in V$, a set of key-value pairs are received in forms of $\{(v, C_v^1), \cdots, (v, C_v^k)\}$. The new C_v is updated as $\bigcup_{1 \leq i \leq k} C_v^i$. The HashGToMin algorithm finishes in $\tilde{O}(\log(n))$ (i.e., expected $O(\log(n))$) rounds, with $O(m+n)$ total communication cost in each round. However, it needs $O(n)$ memory for a single machine to hold a whole CC in memory. Thus, as indicated in [36], HashGToMin is not suitable to handle a graph with large n.

Algorithm PRAM-Simulation**:** The PRAM model allows multiple processors to compute in parallel using a shared memory. There are CRCW PRAM if concurrent writes are allowed, and CREW PRAM if not. In [23], a theoretical result shows that an CREW

[1]The result is only proved on a path graph in [36].

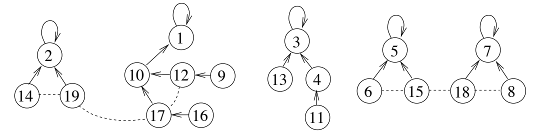


Figure 3: Singleton Elimination: Compute V_p

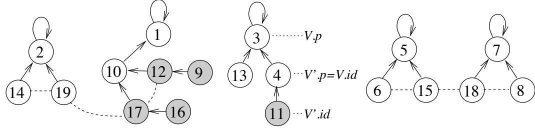


Figure 4: Star Detection Step 1: Compute V_g

PRAM algorithm in $O(t)$ time can be simulated in MapReduce in $O(t)$ rounds. For the CC computation problem, in the literature, the best result in CRWE PRAM is presented in [22] which computes CCs in $O(\log(n))$ time. However, it needs to compute the 2-hop node pairs which requires $O(n^2)$ communication cost in the worst case in each round. Thus, the simulation algorithm is impractical.

5.2 Connected Component in $\mathcal{SGC}$

We introduce our algorithm to compute CCs in $\mathcal{SGC}$. Conceptually, the algorithm shares similar ideas with most deterministic $O(\log(n))$ CRCW PRAM algorithms, such as [39] and [6], but it is a non-trivial adaption since each operation should be carefully designed using graph joins in $\mathcal{SGC}$. Our algorithm maintains a forest using a parent pointer $p(v)$ for each $v \in V$. Each rooted tree in the forest represents a partial CC. A *singleton* is a tree with one node, and a *star* is a tree of height 1. A tree is an isolated tree if there are no edges in E that connect the tree to another tree. The forest is iteratively updated using two operations: *hooking* and *pointer jumping*. Hooking merges several trees into a larger tree, and pointer jumping changes the parent of each node to its grandparent in each tree. When the algorithm ends, each tree becomes an isolated star that represents a CC in the graph.

Specifically, the algorithm first initializes a forest to make sure that no singletons exist except for isolated singletons. Then, the algorithm updates the forest in iterations. In each iteration, two hooking operations, namely, a conditional star hooking and an unconditional star hooking, followed by a pointer jumping operation are performed. The two hooking operations eliminate all non-isolated stars in the forest and the pointer jumping operation produces new stars to be eliminated in the next iteration. Our algorithm CC is shown in Algorithm 4, which includes five components: Forest Initialization (line 1-3), Star Detection (line 16-20), Conditional Star Hooking (line 5-8), Unconditional Star Hooking (line 9-12), and Pointer Jumping (line 13-14). We explain the algorithm using a sample graph $G(V, E)$ shown in Fig. 1.

Forest Initialization: The forest is initialized in three steps. (1) In the first step, a table V_m is computed, in which each node v finds the smallest node among its neighbors in G including itself as the parent $p(v)$ of v, i.e., $p(v) = min\{v \cup \{u|(u, v) \in E\}\}$. Such an operation guarantees that no cycles are created except for self cycles (i.e., $p(v) = v$). The operation can be done using $V \bowtie_{id,id_1}^{\mathcal{EN}} E$ as shown in line 1. (2) In the second step, we create a table V_c by counting the number of subnodes for each node in the forest, i.e., for each node v, $c(v) = |\{u|p(u) = v\}|$. This can be done using a self $\mathcal{EN}$ join $V_m \bowtie_{id,p}^{\mathcal{EN}} V_m$ where the second V_m is considered as an edge table since it has two fields representing node ids (line 2). (3) In the third step, we create V_p by eliminating all non-isolated singletons. A node v is a singleton, iff $c(v) = 0$ and $p(v) = v$. A non-isolated singleton v can be eliminated by assigning $p(v) = \min\{u|(u, v) \in E\}$, which can be done using

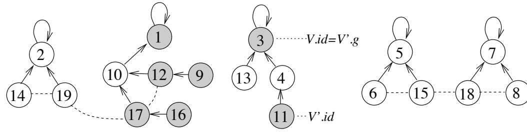


Figure 5: Star Detection Step 2: Compute V_s'

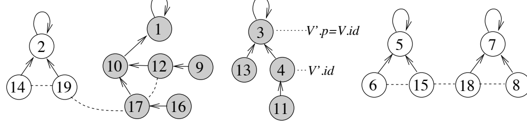


Figure 6: Star Detection Step 3: Compute V_s

$V_c \bowtie^{\mathcal{EN}}_{id,id_1} E$ (line 3). Obviously, no cycles (except for self cycles) are created in V_p.

For example, for the graph G shown in Fig. 1, the forest V_m is shown in Fig. 2, where solid edges represent parent pointers and dashed edges represent graph edges. $p(17) = 10$ since 10 is the smallest neighbor of 17 in G. $p(2) = 2$ since 2 has no neighbor which is smaller than 2. There are two singletons 9 and 16, which are eliminated in V_p as shown in Fig. 3, by pointing 9 to 12 and 16 to 17, since 12 is the smallest neighbor of 9 in G and 17 is the smallest neighbor of 16 in G.

Star Detection: We need to detect whether each node belongs to a star before hooking. We use $s(v) = 1$ to denote that v belongs to a star and $s(v) = 0$ otherwise. The star detection is done using three steps based on the following three filtering rules:

- *(Rule-1):* A node v does not belong to a star if $p(v) \neq p(p(v))$.
- *(Rule-2):* A node v does not belong to a star if $\exists u$, such that u does not belong to a star and $p(p(u)) = v$.
- *(Rule-3):* A node v does not belong to a star if $p(v)$ does not belong to a star.

It is guaranteed that after applying the three rules one by one in order, all non-stars are filtered. We now introduce how to apply the three rules using graph join operators. (Rule-1) For each node v, we find its grandparent $g(v) = p(p(v))$, and assign 1 or 0 to $s(v)$ depending on whether $p(v) = g(v)$. This can be done using a self join $V_p \bowtie^{\mathcal{NE}}_{id,p} V_p$ as shown in line 17. (Rule-2) A node v belongs to a star after applying Rule-2 if $s(v) = 1$ and for all u such that $g(u) = v$, $s(u) = 1$. Thus, we use an aggregate function $and(s(v), s(u))$ which is a boolean function and returns 1 iff $s(v) = 1$ and $\forall g(u) = v, s(u) = 1$. This can be done using a self join $V_g \bowtie^{\mathcal{EN}}_{id,g} V_g$ for V_g created in Rule-1 as shown in line 18. (Rule-3) For each node v, we compute $s(p(v))$ and assign $s(v) = 0$ if $s(p(v)) = 0$. This can be done using a self join $V_s' \bowtie^{\mathcal{NE}}_{id,p} V_s'$ for V_s' created in Rule-2 as shown in line 19.

For example, Fig. 4 shows V_g by applying Rule-1 on V_p shown in Fig. 3. The grey nodes are those detected as non-star nodes, i.e., $s(v) = 0$. For node 11, it does not belong to a star as $(g(11) = 3) \neq (p(11) = 4)$. Fig. 5 shows V_s' by applying Rule-2 on V_g. Two new nodes 1 and 3 are filtered as non-star nodes. For node 3, $s(3) = 0$ since there exists node 11 with $g(11) = 3$ and $s(11) = 0$. Fig. 6 shows V_s by applying Rule-3 on V_s'. Three nodes 10, 13 and 4 are filtered. For node 4, $s(4) = 0$ since its parent node 3 has $s(3) = 0$. In V_s, all non-star nodes are filtered, and 3 stars rooted at 2, 5 and 7 are detected.

Conditional Star Hooking: In a conditional star hooking, for any node v which is the root of a star (i.e., $p(v) = v$ and $s(v) = 1$), the parent of v is updated to $min\{\{p(v)\} \cup \{u | \exists(x, y) \in E$, s.t. $p(x) = u$ and $p(y) = v\}\}$. In other words, v is hooked to a new parent u, if u is no larger than v, and the tree that u lies in is connected to the star that v lies in through an edge (x, y) with $p(x) = u$ and $p(y) = v$. The operation ensures that $p(v)$ is no larger than v in order to make sure that no cycles (except for self cycles) are created. After the hooking, it is guaranteed that there are no edges that connect two stars. Conditional star hooking is done in three steps. (1) Create a new edge table E_h by embedding $p(x)$ to each edge $(x, y) \in E$ using $V_s \bowtie^{\mathcal{NE}}_{id,id_1} E$ (line 6), where V_s is the forest with all stars detected (line 5). (2) Create a table V_h', in which for each node y such that y is in a star, $p_m(y) = min\{\{p(y)\} \cup \{p(x) | (x, y) \in E\}\}$ is computed. This can be done using $\sigma_{s=1}(V_s \bowtie^{\mathcal{EN}}_{id,id_2} E_h)$ (line 7). (3) Create a table V_h, in which the parent of each node v is updated to $min\{p_m(y) | p(y) = v\}$ if such p_m exists using $V_s \bowtie^{\mathcal{EN}}_{id,p} V_h'$ (line 8).

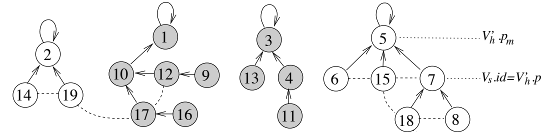


Figure 7: Conditional Star Hooking: Compute V_h

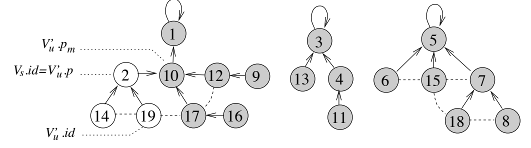


Figure 8: Unconditional Star Hooking: Compute V_u

For example, for V_s shown in Fig. 6 with all stars detected, there exists an edge $(x, y) = (15, 18)$ with $u = p(x) = 5$ and $v = p(y) = 7$. Since 7 is in a star and $5 < 7$, 7 is hooked to a new parent 5 by assigning $p(7) = 5$ as shown in Fig. 7. Note that 5 is also in a star in V_s, however, since $7 > 5$, we cannot hook 5 to 7 by assigning $p(5) = 7$ after which a cycle is created.

Unconditional Star Hooking: Unconditional star hooking is similar to conditional star hooking by dropping the condition that a node v should be hooked to a parent u with $u \leq v$. It is done using the similar three steps (line 10-12) with the only difference on the second step, which calculates $p_m(y)$ as $min\{p(x) | (x, y) \in E$ and $p(x) \neq p(y)\}$, instead of $min\{\{p(y)\} \cup \{p(x) | (x, y) \in E\}\}$. We add a condition $p(x) \neq p(y)$ to avoid hooking a star to itself in order to make sure that all non-isolated stars are eliminated. Unconditional star hooking does not create cycles (except for self cycles) due to the fact that after conditional star hooking, there is no edge that connects two stars.

For example, for the forest V_h shown in Fig. 7, there is only one star rooted at node 2. There exists an edge $(x, y) = (19, 17)$ with $u = p(x) = 2$ and $v = p(y) = 10$ and $u \neq v$, so 2 is hooked to a new parent 10 as shown in Fig. 8 with no stars existing.

Pointer Jumping: Pointer jumping changes the parent of each node to its grandparent in the forest V_u generated in unconditional star hooking by assigning $p(v) = p(p(v))$ for each node v. This can be done using a self join $V_u \bowtie^{\mathcal{NE}}_{id,p} V_u$ (line 13). In pointer jumping, we also create a counter n_s which counts the number of nodes with $p(v) \neq p(p(v))$. When $n_s = 0$, all stars in V_u are isolated stars and the algorithm terminates with each star represents a CC (line 14). For example, for the forest V_u computed in unconditional star hooking, after pointer jumping, the new forest V_p is shown in Fig. 9 with two stars with roots 3 and 5 generated.

The following theorem shows the efficiency of Algorithm 4. Due to lack of space, the proof is omitted.

Theorem 5.1: *Algorithm 4 stops in $O(\log(n))$ iterations.* □

The comparison of algorithms HashToMin, HashGToMin, and our algorithm CC is shown in Table 2 in terms of the memory consumption per machine, total communication cost per round, and the number of rounds, in which our algorithm is the best in all factors.

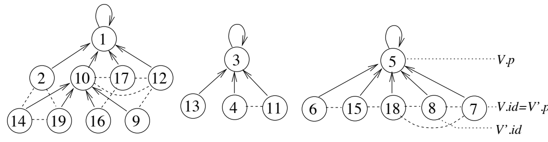


Figure 9: Pointer Jumping: Compute V_p

	HashToMin	HashGToMin	CC
Memory/machine	$O(1)$	$O(n)$	$O(1)$
Communication/round	$O(\log(n)(n+m))$	$O(n+m)$	$O(n+m)$
Number of rounds	$O(\log(n))$	$\tilde{O}(\log(n))$	$O(\log(n))$

Table 2: CC Computation Algorithms in MapReduce

6. MINIMUM SPANNING FOREST

Given a weighted undirected graph $G(V, E)$ of n nodes and m edges, with each edge $(u, v) \in E$ assigned a weight $w((u, v))$, a Minimum Spanning Forest (MSF) is a spanning forest of G with the minimum total edge weight. We also use $(u, v, w((u, v)))$ to denote an edge. Although MSF can be efficiently computed on a sequential machine using $O(m + n\log(n))$ time, it is non-trivial to solve the algorithm in MapReduce.

6.1 State-of-the-art

We introduce two algorithms in $\mathcal{MRC}$, namely, OneRoundMSF and MultiRoundMSF. OneRoundMSF is proposed in [23] and MultiRoundMSF is proposed in [25].

Algorithm OneRoundMSF**:** Fix a number k, the algorithm partitions V into k equally sized subsets randomly, i.e., $V = V_1 \cup V_2 \cup \dots \cup V_k$ with $V_i \cap V_j = \emptyset$ for $i \neq j$. Then $k(k-1)/2$ graphs $G_{i,j}$ for $1 \leq i < j \leq k$ are created. Each $G_{i,j}$ is a subgraph of G induced by nodes $V_i \cup V_j$. Next, the MSF of each $G_{i,j}$, $M_{i,j}$ is computed in parallel in $k(k-1)/2$ machines. Finally, all $M_{i,j}$ are merged in a single machine as a new graph H, and the MSF of H is computed as the MSF of G. The algorithm can be processed using one round of MapReduce, and it requires H with size $O(n^{1+\frac{c}{2}})$ to fit in the memory of a single machine assuming that $m \geq n^{1+c}$.

Algorithm MultiRoundMSF**:** OneRoundMSF does not work efficiently since every node is duplicated $k-1$ times. MultiRoundMSF proposed in [25] improves OneRoundMSF using multiple rounds of MapReduce. In each round, the edges E are partitioned into l equally sized subsets randomly, i.e., $E = E_1 \cup E_2 \cup \dots \cup E_l$ with $E_i \cap E_j = \emptyset$ for $i \neq j$. The MSF of each E_i, T_i is computed in parallel in l machines, and the new E is assigned $T_1 \cup T_2 \cup \dots \cup T_l$. The algorithm stops when $|E| \leq n^{1+\epsilon}$ for a constant ϵ and the MSF of E is computed in a single machine as the MSF of G. The algorithm requires a single machine to have $O(n^{1+\epsilon})$ memory.

6.2 Minimum Spanning Forest in $\mathcal{SGC}$

Suppose there is a total order among all edges as follows. For any two edges $e_1 = (u_1, v_1, w_1)$ and $e_2 = (u_2, v_2, w_2)$, $e_1 < e_2$ iff one of the following conditions holds: (1) $w_1 < w_2$, (2) $w_1 = w_2$ and $min(u_1, v_1) < min(u_2, v_2)$, and (3) $w_1 = w_2$, $min(u_1, v_1) = min(u_2, v_2)$, and $max(u_1, v_1) < max(u_2, v_2)$.

Our algorithm is based on the Sollin's Algorithm [5] for MSF computation, in which the following lemma plays a key role.

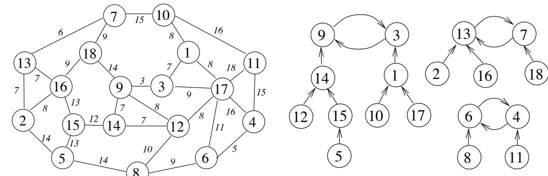
Lemma 6.1: *For any $V_s \subseteq V$, the smallest edge in $\{(u, v) | u \in V_s, v \notin V_s\}$ is in the MSF.* □

Our algorithm MSF shares similar ideas with Algorithm 4 for CC computation. We maintain a forest using parent pointers. Trees in the forest are merged to form larger trees in iterations, and the algorithm terminates when all trees in the forest become isolated stars. In each iteration, the forest is updated using two operations, namely, hooking and pointer jumping. Hooking eliminates all stars by merging them into other trees and pointer jumping decreases the depth of the trees to generate new stars. The algorithm is different from Algorithm 4 mainly in three aspects:

- *Hooking Strategy:* Different from CC computation, in MSF, a star cannot be arbitrarily hooked to a tree as long as there is an edge connecting them. Instead, a star can only be hooked to a tree using an edge that is minimum among all edges leaving the star, as indicated in Lemma 6.1.
- *Cycle Breaking:* The above hooking strategy may produce cycles among multiple nodes. We need a strategy to break all such cycles without breaking any tree apart.
- *MSF Maintenance:* Instead of maintaining the forest defined by parent pointers, we also need to maintain the MSF which is another forest different from the forest defined by parent pointers.

Algorithm 5 MSF$(V(id), E(id_1, id_2, w))$

1: $V_p \leftarrow \prod_{id, min((id_1, id_2, w)) \rightarrow e_m, e_m.id_2 \rightarrow p}(V \bowtie^{\mathcal{EN}}_{id, id_1} E)$;
2: $E_t \leftarrow \prod_{e_m}(\sigma_{e_m \neq \phi}(V_p))$;
3: **while** true **do**
4: $\quad V_b \leftarrow \prod_{V'.id, ((V'.id = V.p \wedge V'.id < V.id) ? V'.id : V.id) \rightarrow p}((V_p \rightarrow V) \bowtie^{\mathcal{NE}}_{id, p} (V_p \rightarrow V'))$;
5: $\quad V_c \leftarrow \prod_{V'.id, V.p}((V_b \rightarrow V) \bowtie^{\mathcal{NE}}_{id, p} (V_b \rightarrow V')) \mid C(V'.p \neq V.p) \rightarrow n_s$;
6: $\quad$ **if** $n_s = 0$ **then break**;
7: $\quad V_s \leftarrow \mathsf{star}(V_c)$;
8: $\quad E_m \leftarrow \prod_{(id_1, id_2, w) \rightarrow e, p \rightarrow p_1}(V_s \bowtie^{\mathcal{NE}}_{id, id_1} E)$;
9: $\quad V_m \leftarrow \prod_{id, p, amin(e, p_1 | p_1 \neq p) \rightarrow (e_m, p_m)}(\sigma_{s=1}(V_s \bowtie^{\mathcal{EN}}_{id, id_2} E_m))$;
10: $\quad V_p \leftarrow \prod_{V_s.id, (cnt(p_m)=0 ? (\phi, V_s.p) : amin(e_m, p_m)) \rightarrow (e_m, p)}(V_s \bowtie^{\mathcal{EN}}_{id, p} V_m)$;
11: $\quad E_t \leftarrow E_t \cup (\prod_{e_m}(\sigma_{e_m \neq \phi}(V_p)))$;
12: **return** E_t;



(a) Graph $G(V, E)$ (b) Forest Init: Compute V_p
Figure 10: A Sample Graph and Forest Initialization

The algorithm MSF is shown in Algorithm 5. We introduce MSF in terms of Forest Initialization (line 1-2), Cycle Breaking (line 4), Pointer Jumping (line 5-6), and Edge Hooking (line 7-11). We explain the algorithm using a sample graph $G(V, E)$ shown in Fig. 10(a)

Forest Initialization: Suppose we use $p(v)$ to denote the parent pointer of each node $v \in V$, and use edge table E_t to maintain the edges in MSF. In the initialization step, for each node $v \in V$, the algorithm finds its minimum adjacent edge $(u, v) \in E$, hooks v to u using $p(v) = u$, and adds (u, v) to the MSF E_t by Lemma 6.1. The hooking can be done using $V \bowtie^{\mathcal{EN}}_{id, id_1} E$ (line 1). Let V_p be the forest after the hooking, it is guaranteed that no singletons exist in V_p except for isolated singletons. It is possible that in V_p, cycles of multiple nodes can be formed by parent pointers, however, the following lemma shows a good property of V_p using which a cycle breaking method can be applied efficiently.

Lemma 6.2: *Each cycle in V_p is with length no larger than* 2. □

For example, for the graph shown in Fig. 10(a), after forest initialization, the forest V_p is shown in Fig. 10(b). Node 2 is hooked to 13 since the edge $(2, 13, 7)$ is the smallest edge among all adjacent edges of 2 in G. Note that node 13 is also hooked to node 2 by

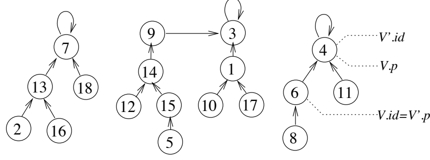


Figure 11: Cycle Breaking: Compute V_b

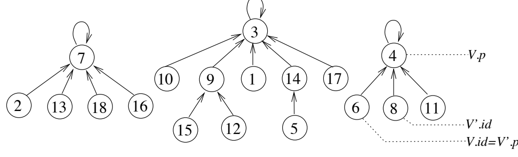


Figure 12: Pointer Jumping: Compute V_c

the same edge $(2, 13, 7)$, thus a cycle of size 2 is formed by nodes 2 and 13. All the edges in V_p are added to the MSF E_t.

Cycle Breaking: Cycle breaking breaks the cycles to make sure that there are no cycles except for self cycles in the forest. According to Lemma 6.2, a cycle of length 2 can be easily detected if there is a node v with $p(v) \neq v$ and $p(p(v)) = v$. We can eliminate such cycles using the following rule:

(Cycle Breaking Rule): For a node v with $p(v) \neq v$ and $p(p(v)) = v$, if $p(v) > v$, then assign v to $p(v)$.

By applying the rule, we create a table V_b with no cycles (except for self cycles) using a self join $V_p \bowtie^{\mathcal{NE}}_{id,p} V_p$ as shown in line 4.

For example, for the forest V_p in Fig. 10(b), the node 4 has $p(4) = 6 \neq 4$ and $p(p(4)) = 4$. Since $p(4) > 4$, by applying the cycle breaking rule, $p(4)$ is updated to 4. The new forest V_b with no cycles of length larger than 1 is shown in Fig. 11.

Pointer Jumping: Pointer jumping is analogous to that in Algorithm 4, which creates a table V_c by changing the parent of each node v to its grandparent, using $p(v) = p(p(v))$ by a self join $V_b \bowtie^{\mathcal{NE}}_{id,p} V_b$ as shown in line 5. Again, in pointer jumping, we create a counter n_s to count the number of non-star nodes. When $n_s = 0$, the algorithm terminates and outputs E_t as the MSF of G.

For example, after pointer jumping, the forest V_b in Fig. 11 is changed to the forest V_c in Fig. 12. The parent of node 8 changes to its grandparent 4, and two new stars with roots 7 and 4 are created.

Edge Hooking: Edge hooking aims to eliminate all stars (except for isolated stars) in the forest V_c . Suppose we create a table V_s in line 7 with all stars detected using the same procedure star in Algorithm 4. In edge hooking, for any node v which is the root of a star (i.e., $p(v) = v$ and $s(v) = 1$), let $(x, y, w) = \min\{(x', y', w')|p(y') = v, p(x') \neq p(y')\}$, then v is assigned a new parent $u = p(x)$ after hooking. Edge hooking is done in three steps. (1) Create a new edge table E_m by embedding $p(x)$ to each edge $(x, y, w) \in E$ using $V_s \bowtie^{\mathcal{NE}}_{id,id_1} E$ (line 8). (2) Create a table V_m, in which for each node y such that y is in a star, $p_m(y) = p(x)$ with $e_m(y) = (x, y, w) = min\{(x', y', w')|p(y') = v, p(x') \neq p(y')\}$ is computed. This can be done with an aggregate function $amin((x, y, w), p(x)|p(x) \neq p(y))$ using the $\mathcal{EN}$ join $\sigma_{s=1}$ ($V_s \bowtie^{\mathcal{EN}}_{id,id_2} E_m$) (line 9). (3) Create a table V_p, in which the parent of each node v is updated to $p_m(y)$ with $(x, y, w) = min\{e_m(y) \mid p(y) = v\}$ if such p_m exists using $V_s \bowtie^{\mathcal{EN}}_{id,p} V_m$ (line 10). The corresponding edge $e_m = min\{e_m(y)|p(y) = v\}$ is added to the MSF table E_t (line 11) by Lemma 6.1 if it exists. It is guaranteed that Lemma 6.2 still holds on the V_p created in edge hooking.

For example, for V_c in Fig. 12, there exists an edge $(x, y) = (15, 16)$ with $u = p(x) = 9$ and $v = p(y) = 7$. Since 7 is the root of a star and $(15, 16, 13) = \min\{(x', y'w')|p(y') = 7, p(x') \neq 7\} = min\{(10, 7, 15), (9, 18, 14), (15, 16, 13), (5, 2, 14)\}$, 7 is hooked to 9 as shown in Fig. 13. Similarly, 4 is also hooked to 9. The edges $(15, 16, 13)$ and $(12, 8, 10)$ are added to the MSF E_t.

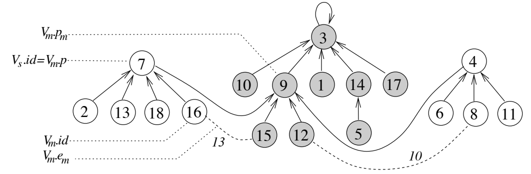


Figure 13: Edge Hooking: Compute V_p

	OneRoundMSF	MultiRoundMSF	MSF
Memory/machine	$O(n^{1+\frac{c}{2}})$	$O(n^{1+\epsilon})$	$O(1)$
Communication/round	$O(m^{1+\frac{c}{2}})$	$O(n+m)$	$O(n+m)$
Number of rounds	$O(1)$	$O(\frac{\log_n(m)-1}{\epsilon})$	$O(\log(n))$

Table 3: MSF Computation Algorithms in MapReduce

The following theorem shows the efficiency of Algorithm 5. Due to lack of space, the proof is omitted.

Theorem 6.1: *Algorithm 5 stops in $O(\log(n))$ iterations.* □

The comparison of algorithms OneRoundMSF, MultiRoundMSF, and our algorithm MSF is shown in Table 3 in terms of memory consumption per machine, total communication cost per round, and the number of rounds. As we will show later in our experiments, the high memory requirement of OneRoundMSF and MultiRoundMSF becomes the bottleneck for the algorithms to achieve high scalability when handling graphs with large n.

7. PERFORMANCE STUDIES

In this section, we show our experimental results. We deploy a cluster of 17 computing nodes, including one master node and 16 slave nodes, each of which has four Intel Xeon 2.4GHz CPUs and 15GB RAM running 64-bit Ubuntu Linux. We implement all algorithms using Hadoop (version 1.2.1) with Java 1.6. We allow each node to run three mappers and three reducers concurrently, each of which uses a heap size of 2048MB in JVM. The block size in HDFS is set to be 128MB, the data replication factor of HDFS is set to be 3, and the I/O buffer size is set to be 128KB.

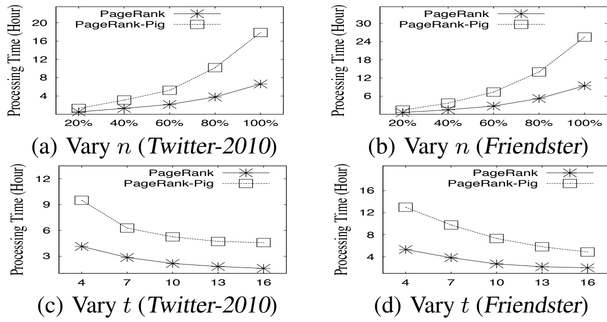
Datasets: We use two web-scale graphs *Twitter-2010*[2] and *Friendster*[3] with different graph characteristics for testing.

- *Twitter-2010* contains 41,652,230 nodes and 1,468,365,182 edges with an average degree of 71. The maximum degree is 3,081,112 and the diameter of *Twitter-2010* is around 24.
- *Friendster* contains 65,608,366 nodes and 1,806,067,135 edges with an average degree of 55. The maximum degree is 5,214 and the diameter of *Friendster* is around 32.

Algorithms: Besides the five algorithms PageRank (Algorithm 1), BFS (Algorithm 2), KWS (Algorithm 3), CC (Algorithm 4), and MSF (Algorithm 5), we also implement the algorithms for PageRank, BFS, and graph keyword search using the join operations supported by Pig (http://pig.apache.org/) on Hadoop, denoted PageRank-Pig, BFS-Pig and KWS-Pig respectively. Since the algorithms for PageRanks, BFS, and graph keyword search are rather simple, i.e., for each algorithm, only two MapReduce jobs are needed in each iteration for both Pig and our implementation, the main difference between Pig and our implementation is how the join operation is implemented. In Pig, the join operation is implemented using a load-and-join manner where each key-value pair is accessed for more than once in the reducer, and in our implementation, the join operation is implemented as a streaming

[2] http://law.di.unimi.it/webdata/twitter-2010/

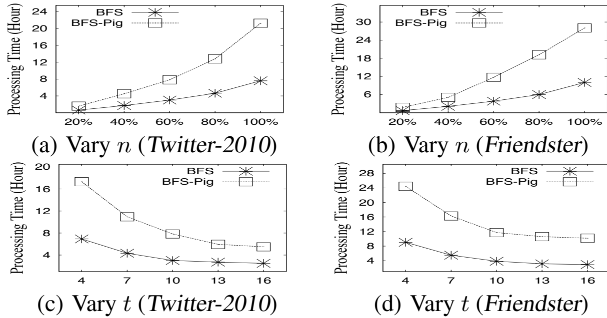
[3] http://snap.stanford.edu/data/com-Friendster.html



(a) Vary n (*Twitter-2010*) (b) Vary n (*Friendster*)

(c) Vary t (*Twitter-2010*) (d) Vary t (*Friendster*)

Figure 14: PageRank



(a) Vary n (*Twitter-2010*) (b) Vary n (*Friendster*)

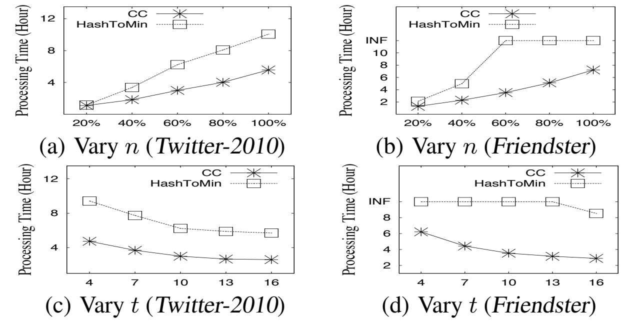
(c) Vary t (*Twitter-2010*) (d) Vary t (*Friendster*)

Figure 15: Breadth First Search

manner as introduced in Section 3.2 where each key-value pair is accessed for only once in the reducer. For CC computation, we implement HashToMin and HashGToMin as introduced in Section 5 by integrating all the optimization techniques introduced in [36], and for MSF computation, we implement OneRoundMSF and MultiRoundMSF as introduced in Section 6. All the four algorithms OneRoundMSF, MultiRoundMSF, HashToMin, and HashGToMin, and all our algorithms proposed in this paper are implemented by java using plain MapReduce in Hadoop without using Pig or other MapReduce based translators.

Parameters: For each dataset, we extract subgraphs of 20%, 40%, 60%, 80% and 100% nodes of the original graph with a default value of 60%. We also vary the number of slave nodes t in the cluster from 4 to 16 with a default value of 10. We test both processing time and maximum communication cost on each machine. The curves for the communication cost are very similar to those for processing time, thus we only show the processing time due to lack of space. We set the maximum running time to be 36 hours. If a test does not stop in the time limit, or fails due to out of memory exception, we will denote the processing time using INF. For PageRank, we consider each graph as a directed graph and for other algorithms, we consider each graph as an undirected graph.

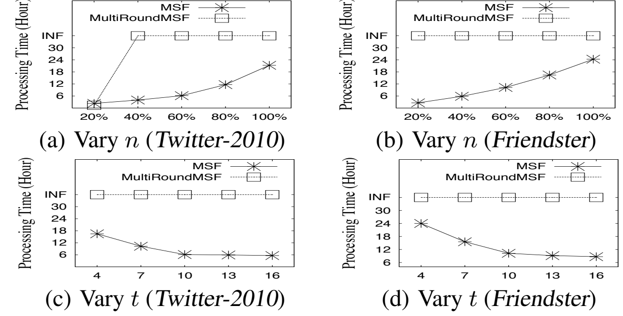
Exp-1: PageRank. We test PageRank . We set the default number of iterations d to be 6. The testing results are shown in Fig. 14. Fig. 14 (a) and Fig. 14 (b) demonstrate the curves for *Twitter-2010* and *Friendster* respectively when varying the size of the graph from 20% to 100%. The time for both PageRank and PageRank-Pig increases when the size of the graph increases. PageRank-Pig is 1.5 to 3 times slower than PageRank. This is because Pig is not optimized for graph processing using the techniques in $\mathcal{NE}$ join and $\mathcal{EN}$ join introduced in Section 3. Fig. 14 (c) and Fig. 14 (d) show the results on *Twitter-2010* and *Friendster* respectively when varying the number of slave nodes t in the cluster from 4 to 16. When t increases, the processing time for both PageRank and PageRank-Pig decreases. PageRank can achieve high scalability in both *Twitter-2010* and *Friendster*. PageRank-Pig is 2.3 times



(a) Vary n (*Twitter-2010*) (b) Vary n (*Friendster*)

(c) Vary t (*Twitter-2010*) (d) Vary t (*Friendster*)

Figure 16: Connected Component



(a) Vary n (*Twitter-2010*) (b) Vary n (*Friendster*)

(c) Vary t (*Twitter-2010*) (d) Vary t (*Friendster*)

Figure 17: Minimum Spanning Forest

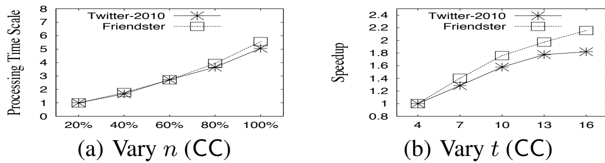
slower than PageRank on average. We also vary the number of iterations d from 2 to 10. The processing time for both PageRank and PageRank-Pig increases linearly with the number of iterations. The curves are not shown due to lack of space.

Exp-2: Breadth First Search. Fig. 15 shows the testing results for BFS and BFS-Pig by setting the depth of BFS to be 6. The curves on *Twitter-2010* and *Friendster* when varying the graph size are shown in Fig. 15 (a) and Fig. 15 (b) respectively. The results are similar to those in PageRank. As shown in Fig. 15 (c) and Fig. 15 (d), when increasing t from 4 to 16, the processing time for both BFS and BFS-Pig decreases, and the time decreases more sharply when t is smaller. This is because when t is smaller, each machine will spend more time on shuffling and sorting data on disk, which is costly. We also vary the depth of BFS d from 2 to 10. The curves are omitted since the processing time for both BFS and BFS-Pig increases linearly with d.

Exp-3: Graph Keyword Search. We randomly generate some keywords in both *Twitter-2010* and *Friendster*, and test KWS and KWS-Pig using a keyword query with 3 keywords and $\mathsf{rmax} = 3$ by default. The trends by varying n and t on both *Twitter-2010* and *Friendster* are similar to those in BFS, since in graph keyword search, the keyword information on each node is propagated to its neighbors in a BFS manner in iterations. We also vary rmax from 1 to 5. Again, the processing time for both KWS and KWS-Pig increases linearly with rmax. Due to space limitation, we omit the figures for graph keyword search in the paper.

Exp-4: Connected Component. We test three algorithms: CC, HashToMin, and HashGToMin. Since HashGToMin fails in almost all test cases due to the out of memory exception, we only show the testing results for HashToMin and CC.

Fig. 16 (a) shows the curves when varying the size of the graph in *Twitter-2010*. When the number of nodes n in the graph is 20%, HashToMin and CC have similar performance. However, when n increases, the processing time of HashToMin increases more sharply than CC. The reasons are twofold. First, HashToMin generates more intermediate results than CC, which increase both the



(a) Vary n (CC) (b) Vary t (CC)

Figure 18: Degree Distribution Test

communication cost and the cost for sorting data on each machine. This is consistent with the theoretical results shown in Table 2. Second, HashToMin needs to sort all the edges in the graph in order to achieve constant memory consumption on each machine, which is very costly. Fig. 16 (b) shows the testing results on *Friendster* when varying n. HashToMin cannot stop in the time limit when n increases to 60% while CC still keeps stable. The reason is due to the large intermediate results generated by HashToMin. Note that *Friendster* is a graph with a large diameter and uniform degree distribution. In such a graph, the number of iterations used in HashToMin is large. Since the communication cost of HashToMin in each round is linear with respect to the number of iterations used, a large number of intermediate results are generated and shuffled by HashToMin in each round, making HashToMin inefficient.

The testing results when varying t on *Twitter-2010* and *Friendster* are shown in Fig. 16 (c) and Fig. 16 (d) respectively. In *Twitter-2010*, the processing time for both CC and HashToMin decreases when t increases. When t increases to 13, the time for HashToMin decreases more slowly than CC. HashToMin is 1.5 to 2.2 times slower than CC in all test cases. When varying t in *Friendster*, HashToMin cannot stop in the time limit until t reaches 16 due to the large number of intermediate results generated, while the processing time of CC decreases stably when t increases.

Exp-5: Minimum Spanning Forest. In our last set of experiments, we compare three algorithms: MSF, OneRoundMSF, and MultiRoundMSF. OneRoundMSF runs out of memory for all test cases, thus we only show the testing results for MultiRoundMSF and MSF. We assign each edge a random weight uniformly in $(0, 1)$. We also tested other methods of assigning edge weights (e.g., using other distributions). We find that the weight assignment method has small effect on the efficiency of the algorithms.

The testing results are shown in Fig. 17, in which Fig. 17 (a) and Fig. 17 (b) show the curves when varying n and Fig. 17 (c) and Fig. 17 (d) show the curves when varying t. Not surprisingly, MultiRoundMSF runs out of memory in most cases, since it requires a single machine to have a memory size super-linear to the number of nodes in the graph. In Fig. 17 (b), when increasing n from 20% to 100%, the processing time of MSF increases near linearly, demonstrating the high scalability of MSF. As shown in Fig. 17 (c) and Fig. 17 (d), once MultiRoundMSF runs out of memory, the problem cannot be solved by adding more machines in the system, while the performance of MSF can be improved stably when the number of slaves increases from 4 to 16.

Exp-6: Degree Distribution Test. We test the impact of degree distribution to the efficiency of the algorithms proposed in this paper. Recall that the datasets *Twitter-2010* and *Friendster* have different degree distributions. *Twitter-2010* has a skewed degree distribution with maximum degree 3,081,112, and *Friendster* has a more uniform degree distribution with maximum degree 5,214. We test the CC algorithm on both *Twitter-2010* and *Friendster*. First, we take the processing time for $n = 20\%$ as 1, and test the relative processing time scale when varying n from 20% to 100% on both *Twitter-2010* and *Friendster*. The result is shown in Fig. 18 (a). The curves for processing time scale on *Twitter-2010* and *Friendster* are similar, which indicates that the processing time scale when varying n is not largely impacted by the degree distribution. Next, we take the processing time for $t = 4$ as 1, and test the speedup of the algorithm when varying t from 4 to 16 on both *Twitter-2010* and *Friendster*. The result is shown in Fig. 18 (b). When t is large, the speedup of CC on *Twitter-2010* increases slower than *Friendster*, because the nodes with very high degree in *Twitter-2010* become the bottleneck for the speedup, while the speedup for *Friendster* still keeps increasing stably when $t = 16$. For other proposed algorithms, we can get similar result. Due to space limitation, we only show the result for the CC algorithm in the paper.

8. RELATED WORK

MapReduce Framework: MapReduce [10] is a big data processing framework that has become the de facto standard used throughout both industry and academia. In industry, Google has developed a MapReduce system on the distributed key-value store BigTable on top of GFS (Google File System). Yahoo has developed a MapReduce system Hadoop and a high level data flow language Pig based on a distributed key-value store HBase on top of HDFS (Hadoop Distributed File System). Facebook has developed a SQL-like data warehouse infrastructure Hive based on Hadoop using a key-value store Cassandra. Microsoft has developed two languages, Scope and DryadLINQ, based on the distributed execution engine Dryad. Amazon has developed a key-value store called Dynamo.

In academia, Hadoop++ by Dittrich et al. [11] improves the performance of Hadoop using user-defined functions. Column-based storage is studied by Floratou et al. [14] and hybrid-based storage is studied by Lin et al. [28] to speed up MapReduce tasks. HAIL is proposed by Dittrich et al. [12] to improve the upload pipeline of HDFS. Query optimization in MapReduce is discussed by Jahani et al. [18] and Lim et al. [27]. Mulit-query and iterative query optimization in MapReduce are studied by Wang et al. [45] and Onizuka et al. [34] respectively. Cost analysis of MapReduce is given by Afrati et al. [3]. Some other works focus on solving a specific type of query in MapReduce. For example, set similarity joins in MapReduce are studied by Vernica et al. [44] and Metwally and Faloutsos [32]. Theta joins in MapReduce are discussed by Okcan and Riedewald [33] and Zhang et al. [48]. Multiway joins are optimized by Afrati et al. [4]. KNN joins in MapReduce are proposed by Lu et al. [29]. A survey on distributed data management and processing using MapReduce is given by Li et al. [26].

Graph Processing Systems in Cloud: Many graph processing systems are developed in order to deal with big graphs. One representative such system is Pregel [31], and its open source implementation Giraph and HAMA based on Hadoop. Pregel takes a vertex-centric approach and implements a bulk synchronous parallel (BSP) computation model [43], which targets towards iterative computation on graphs. HipG [24] improves BSP by using asynchronous messages to avoid synchronization. Microsoft Research Labs develop a distributed in-memory based graph processing engine called Trinity [38, 47] based on a hypergraph model. PowerGraph [15] is a distributed graph processing system that is optimized to process power-law graphs. Distance oracle is studied in [35]. Giraph++ is proposed in [42] to take graph partitioning into consideration when processing graphs. Workload balancing for graph processing in cloud is discussed in [37].

Graph Processing in MapReduce: Many graph algorithms including triangles/rectangles enumeration, k-cliques computation, barycentric clustering, and components finding in MapReduce are discussed in [9]. In [25], several techniques are proposed to reduce the size of the input in a distributed fashion in MapReduce, and the techniques are applied for several graph problems such

as minimum spanning trees, approximate maximal matchings, approximate node/edge covers, and minimum cuts. Triangle counting in MapReduce is optimized in [40]. Diameter and radii computations in MapReduce are studied in [20]. Personalized PageRank computation in MapReduce is studied in [7]. Matrix multiplication based graph mining algorithms in MapReduce are discussed in [19, 21]. Densest subgraph computation in MapReduce is studied in [8]. Subgraph instances enumeration in MapReduce is proposed in [2]. Algorithms for connected components computation in MapReduce in logarithmic rounds are discussed in [36]. Maximum clique computation in MapReduce is studied in [46]. Recursive query processing such as transitive closure computation in MapReduce is discussed in [1].

Algorithm Classes in MapReduce: Algorithm classes in MapReduce are studied by Karloff et al. [23] and Tao et al. [41], both of which have been introduced in details in Section 2. Note that by defining a new class for graph processing in MapReduce, we focus on the scalability issues that can be commonly achieved by a class of graph algorithms in MapReduce. The query optimization techniques introduced in the MapReduce framework can be orthogonally studied to our work by case-to-case analyses. Each of the algorithms studied in this paper can certainly benefit from further optimization using techniques such as multi-query optimization [45] and iterative query optimization [34] which are beyond the main scope of research in this paper.

9. CONCLUSIONS

In this paper, we study scalable big graph processing in MapReduce. We review previous MapReduce classes, and propose a new class $\mathcal{SGC}$ to guide the development of scalable graph processing algorithms in MapReduce. We introduce two graph join operators using which a large range of graph algorithms can be designed in $\mathcal{SGC}$. Especially, for two fundamental graph algorithms CC computation and MSF computation, we improve the state-of-the-art algorithms both in theory and practice. We conducted extensive performance studies using real web-scale graphs to show the high scalability achieved for our algorithms in $\mathcal{SGC}$.

Acknowledgements. Lu Qin was support by ARC DE140100999. Jeffrey Xu Yu was supported by grant of the RGC of Hong Kong SAR, No. CUHK 418512. Hong Cheng was supported by grant of the RGC of Hong Kong SAR, No. CUHK 411310 and 411211. Xuemin Lin was supported by NSFC61232006, NSFC61021004, ARC DP110102937, ARC DP120104168, and ARC DP140103578.

10. REFERENCES

[1] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *Proc. of EDBT'11*, 2011.

[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *Proc. of ICDE'13*, 2013.

[3] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4), 2013.

[4] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[6] B. Awerbuch and Y. Shiloach. New connectivity and msf algorithms for shuffle-exchange network and pram. *IEEE Trans. Computers*, 36(10), 1987.

[7] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *Proc. of SIGMOD'11*, 2011.

[8] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.

[9] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI'04*, 2004.

[11] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1), 2010.

[12] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[13] U. Elsner. Graph partitioning - a survey. *Technical Report SFB393/97-27, Technische Universitat Chemnitz*, 1997.

[14] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4, 2011.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12*, 2012.

[16] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *Proc. of SIGMOD'07*, 2007.

[17] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33(1), 2008.

[18] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *PVLDB*, 4, 2011.

[19] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *Proc. of KDD'11*, 2011.

[20] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2), 2011.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proc. of ICDM'09*, 2009.

[22] D. R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the erew pram. *SIAM J. Comput.*, 28(3), 1999.

[23] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proc. of SODA'10*, 2010.

[24] E. Krepska, T. Kielmann, W. Fokkink, and H. E. Bal. Hipg: parallel processing of large-scale graphs. *Operating Systems Review*, 45(2), 2011.

[25] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proc. of SPAA'11*, 2011.

[26] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3), 2014.

[27] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *PVLDB*, 5(11):1196–1207, 2012.

[28] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proc. of SIGMOD'11*, 2011.

[29] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.

[30] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1), 2007.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

[32] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8), 2012.

[33] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proc. of SIGMOD'11*, pages 949–960, 2011.

[34] M. Onizuka, H. Kato, S. Hidaka, K. Nakano, and Z. Hu. Optimization for iterative queries on mapreduce. *PVLDB*, 7(4), 2013.

[35] Z. Qi, Y. Xiao, B. Shao, and H. Wang. Toward a distance oracle for billion-node graphs. *PVLDB*, 7(1), 2013.

[36] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Proc. of ICDE'13*, 2013.

[37] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *Proc. of ICDE'13*, 2013.

[38] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *Proc. of SIGMOD'13*, 2013.

[39] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1), 1982.

[40] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. of WWW'11*, 2011.

[41] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *Proc. of SIGMOD'13*, 2013.

[42] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3), 2013.

[43] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[44] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proc. of SIGMOD'10*, 2010.

[45] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3), 2013.

[46] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *Proc. of ICDE'13*, 2013.

[47] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *PVLDB*, 6(4), 2013.

[48] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.