Distributed k–core Decomposition and Maintenance in Large Dynamic Graphs

Sabeur Aridhi*
Aalto University
sabeur.aridhi@aalto.fi

Martin Brugnara
University of Trento
martin.brugnara@gmail.com

Alberto Montresor
University of Trento
alberto.montresor@unitn.it

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

Distributed processing of large, dynamic graphs has recently received considerable attention, especially in domains such as the analytics of social networks, web graphs and spatial networks. k-core decomposition is one of the significant figures of merit that can be analyzed in graphs. Efficient algorithms to compute k-cores exist already, both in centralized and decentralized setting. Yet, these algorithms have been designed for static graphs, without significant support to deal with the addition or removal of nodes and edges. Typically, this challenge is handled by re-executing the algorithm again on the updated graph. In this work, we propose distributed k-core decomposition and maintenance algorithms for large dynamic graphs. The proposed algorithms exploit, as much as possible, the topology of the graph to compute all the k-cores and maintain them in streaming settings where edge insertions and removals happen frequently. The key idea of the maintenance strategy is that whenever the original graph is updated by the insertion/deletion of one or more edges, only a limited number of nodes need their coreness to be re-evaluated. We present an implementation of the proposed approach on top of the AKKA framework, and experimentally show the efficiency of our approach in the case of large dynamic networks.

CCS Concepts

•Theory of computation → Dynamic graph algorithms; Distributed algorithms;

Keywords

Distributed k-core decomposition, k-core maintenance, Dynamic graphs, AKKA framework

*Work primarily done while the author was at the University of Trento.

DEBS '16, June 20-24, 2016, Irvine, California, USA

 ISBN 978-1-4503-4021-2/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2933267.2933299

1. INTRODUCTION

Over the last decade, the field of distributed processing of large graphs has attracted considerable attention. This field has been highly motivated, not only by the increasing size of graph data, but also by its huge number of applications. Such applications include the analysis of social networks [**?**, **?**], Web graphs [**?**], as well as spatial networks [**?**]. k-core decomposition is an important task that has been used to understand large graph data by identifying k-cores, which are a special family of maximally-induced subgraphs. Intuitively, a k-core is obtained by recursively removing all nodes of degree smaller than k, until the degree of all remaining vertices is larger than or equal to k. A node is said to have coreness k if it belongs to the k-core but not to the $(k+1)$-core [**?**]. The k-core decomposition has been used in several different domains including bioinformatics [**?**], graph visualization [**?**] and Internet structure analysis [**?**].

Several algorithms exist for k-core computation in static graphs, both in centralized and decentralized settings. Yet, modern graphs are growing dramatically and are becoming more and more dynamic, with an ever-increasing rate of node/edge additions or removals. In such environments, there is an urgent need for solutions that not only compute the k-core of large graphs, but are also able to maintain it in an efficient way while the data is constantly changing.

Our work is motivated by two factors. First, the size of the graphs is becoming so large, that makes it difficult to process with off-the-shelf, single machines. Second, and most important, the fact that the majority of the existing large graphs are already stored in a distributed way, either because they cannot be stored on a single machine due to their sheer size, or because they get processed and analyzed with decentralized techniques that require them to be distributed among a collection of machines. For these reasons, we identified the need of methods and techniques that can exploit as much as possible the existing topology of the graph data and perform the k-core decomposition in a cooperative way among the distribution nodes. Our solution is based on the idea of recomputing the coreness only for those nodes of the graph that are affected by the graph updates. The propagation of the effect is done first inside the partition that exists in a single node, and then across partitions by considering the *cut edges*, i.e., edges between nodes of different partitions. To the best of our knowledge, the proposed solution is the first that allows to consider graph streams and incremental changes while computing k-core decomposition

in graphs that are already stored in a distributed manner. More specifically, our contributions are the following:

- We present a distributed and streaming k-core decomposition algorithm for very large graphs that are partitioned and distributed across the nodes of a physically independent network of machines.
- We propose a maintenance strategy that deals with incremental changes on the graph by looking to the nodes that need to be updated in all the partitions and updating the coreness of only those nodes.
- We present an implementation of our algorithms on top of AKKA [?], a framework for building distributed and resilient message-driven applications. We experimentally evaluate the performance of the proposed approach on both real and synthetic datasets.

The remainder of the paper is organized as follows. Section 2 presents an overview of the related work and specifically those works that deal with the concept of distributed k-core decomposition. In Section 3, we define the problem of distributed k-core decomposition in large dynamic graphs. In Section 4, we present our incremental approach for the k-core maintenance in such graphs. In Section 5, we describe our experimental evaluation and we discuss our findings.

2. RELATED WORK

In this section we highlight the relevant literature in the field of k-core decomposition. We consider three research areas: (1) centralized algorithms, (2) distributed algorithms and (3) distributed and streaming algorithms for k-core decomposition and maintenance in dynamic graphs.

Centralized algorithms. The first k-core decomposition algorithm was originally proposed by Batagelj and Zavernik (BZ) [?]. The main idea of the algorithm is to recursively delete vertices of degree less than k. It requires random access to the whole graph, which should therefore be kept in the main memory for the sake of performance. Cheng et al. [?] have proposed a strategy based on the BZ algorithm to handle graphs that do not fit into main memory. The proposed algorithm requires $O(k_{max})$ scans of the graph, where k_{max} is the largest coreness value of the graph.

Distributed algorithms. The problem of distributed k-core decomposition was first studied in [?] and a new algorithm for the computation of the k-coreness of a network was proposed. The proposed approach has been applied to two different computational models, one based on Pregel [?] and one based on a block-centric approach [?].

k-core decomposition and maintenance in dynamic graphs. Few works have studied the k-core decomposition problem from large dynamic graphs [?, ?, ?, ?, ?, ?]. Li, Yu and Mao [?] have presented a k-core maintenance approach in dynamic graphs. They proposed two pruning techniques to remove the nodes whose coreness is definitely unchanged after an update operation over the initial graph. When a dynamic graph is updated, the minimal subgraph for which k-core decomposition might have changed is computed, instead of re-computing everything from scratch. The proposed algorithm keeps track of core number for each vertex and upon an update provides the subgraph for which k-core decomposition needs to be updated. In [?], the authors present an incremental k-core decomposition algorithm for streaming graph data. The main idea of their approach is to first locate a small subgraph that contains the set of vertices whose coreness values have to be updated. Then it processes the located subgraph to incrementally maintain the coreness values of its vertices when a single edge is inserted or removed. In [?], the authors present a distributed incremental algorithm for k-core maintenance in large dynamic graphs. The presented approach uses HBase to store the graph data and hence to exploit the horizontally scaling of its distributed storage. The distributed algorithm constructs a k-core subgraph by progressively removing edges in parallel by remote calls on distributed nodes. It is worthwhile to mention that the approach presented in [?] uses a fixed k value and does not determine all the updated k-cores when dynamic changes are made to the graph.

Most of the above-cited solutions deal with core maintenance of large dynamic graphs. However, these approaches do not consider the case when the graph is too large to be kept in main memory or when the graph is already distributed across several machines. Only a few works include the core maintenance task in the case of large distributed graphs, which is the addressed issue in this paper.

3. PROBLEM FORMULATION

Given an undirected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, the concept of *k–core decomposition* [?] is condensed in the following two definitions:

DEFINITION 1. *A subgraph $G(C)$ induced by the set $C \subseteq V$ is a* k-core *if and only if $\forall u \in C : d_{G(C)}(u) \geq k$, and $G(C)$ is maximal, i.e., for each $\overline{C} \supset C$, there exists $v \in \overline{C}$ such that $d_{G(\overline{C})}(v) < k$.*

DEFINITION 2. *A node in G is said to have* coreness k *($k_G(u) = k$) if and only if it belongs to the k-core but not the $(k+1)$-core.*

$d_G(u)$ and $k_G(u)$ denote the degree and the coreness of u in G, respectively; in what follows, however, G can be dropped when it is clear from the context. The subgraph of G induced by C is defined as $G(C) = (C, E|C)$ where $E|C = \{(u, v) \in E : u \in C \vee v \in C\}$.

A k-core of a graph $G = (V, E)$ can be obtained by recursively removing all the vertices of degree less than k, until all vertices in the remaining graph have degree at least k. While such centralized solution is simple and works in linear time [?], the situation gets more complicated when the issues of dynamism and scale are considered; even more so when they are considered together. When new edges and nodes are added or removed, this may cause a cascading re-computation of the coreness of the nodes surrounding the newcomers, that can potentially span the entire graph. While re-computing the coreness of the entire graph is always an option, limiting the re-computation to as few nodes as possible is desirable. When the graph is large and cannot be stored or computed using a single machine, its vertex set can be partitioned into p disjoint partitions $\{V_1, \ldots, V_p\}$; in other words, $V = \cup_{i=1}^{p} V_i$ and $V_i \cap V_j = \emptyset$ for each i, j such that $1 \leq i, j \leq p$ and $i \neq j$. Such partitions induce p subgraphs $G_i = (V_i, E_i)$, where $E_i = E|V_i$. In such subgraphs,

an edge (u, v) is called a *frontier edge* of G_i if $u \in V_i$ and $v \in V_j \neq V_i$, i.e. the edge links a node in V_i with a node in a different partition. The set of frontier edges of a subgraph G_i is denoted F_i; clearly, $F_i \subseteq E_i$. The set of all frontier edges of a graph G is defined as $V_f = \cup_{i=1}^{p} F_i$ where F_i is the set of frontier edges of a subgraph G_i; clearly, $V_f \subseteq E$.

Given a graph $G(V, E)$, distributed in a number of partitions, and having the k-core decomposition already computed over it, we are interested in finding the coreness after a number of modifications (insertions or deletions) have taken place on the graph, without having to restart the computation from scratch. A by-product of such a maintenance solution is that the k-core can also be computed for the first time by running the k-core computation in each partition independently, and then considering the edges between the partitions as updates, and applying the maintenance approach that updates the coreness of every node into the right value considering the overall graph. After that, the k-core will be given by the nodes whose coreness is k.

4. K-CORE COMPUTATION

We assume that the graph is subdivided in multiple partitions, each of them assigned to a different worker. Inside each partition, a centralized algorithm to compute the coreness is run. At that point, we treat large-scale and dynamism in the same way: whenever a new edge is added to the graph, we first determine the set of candidate nodes (nodes whose coreness needs to be updated); then, we compute the correct values for the coreness of those nodes. The set of nodes to be updated may span multiple partitions, in particularly when frontier edges are added. The system overview of our approach is illustrated by Figure 1.

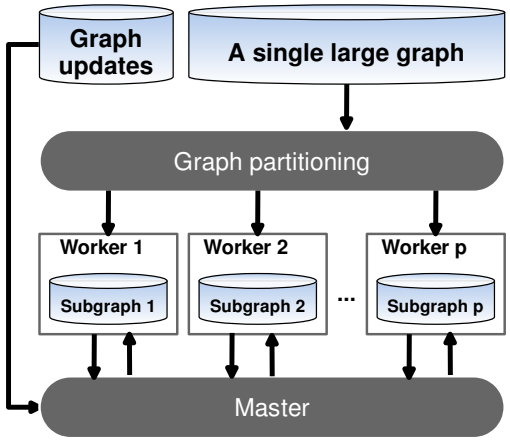


Figure 1: System overview.

As illustrated in Figure 1, each worker run a centralized algorithm to compute the coreness of nodes of its associated subgraph. The master worker orchestrates the execution of the update process after considering graph changes. Our approach operates in three computing modes:

- *M2W mode.* In this mode, message exchanges between the master and all workers are allowed. The master uses this mode in order to ask a distant worker to look for candidate nodes. The worker uses this mode in order to send the set of computed candidate nodes to the master.
- *W2W mode.* In this mode, message exchanges between workers are allowed. The workers use this mode in order to propagate the search for candidate nodes to one or more distant workers.
- *Local mode.* In this mode, only local computation is allowed.

Table 1: Notations

G	An undirected graph partitioned into p partitions
$k(u)$	Coreness of u
$N(u)$	Neighbors of u
V_f	The set of all frontier edges of G
kCore(G_i)	Executed by worker i and computes the coreness of all nodes of G_i
partitionID(u)	Partition associated to node u
visited(u)	Indicates whether the node u is visited or not while we are looking for reachable nodes.
p_i.f()	A remote call to the f() function on partition p_i

Our algorithm exploits Theorem 1, first stated and demonstrated by Li, Yu and Mao [**?**], that identifies what are the *candidate nodes* that may need to be updated whenever we add an edge:

THEOREM 1. *Let $G = (V, E)$ be a graph and (u, v) be an edge to be inserted in E, with $u, v \in V$. A node $w \in V$ is said to be a* candidate *to be updated based on the following three cases:*

- *If $k(u) > k(v)$, w is candidate if and only if w is k-reachable from u in the original graph G and $k = k(u)$;*
- *If $k(u) < k(v)$, w is candidate if and only if w is k-reachable from v in the original graph G and $k = k(v)$;*
- *If $k(u) = k(v)$, w is candidate if and only if w is k-reachable from either u and v in the original graph G and $k = k(u)$.*

A node w is *k-reachable* from u if there exists a path between u and w in the original graph such that all nodes in the path (including u and w) have coreness equal to $k = k(u)$. At this point, we can further prune the amount of possible nodes using Theorem 2.

THEOREM 2. *Let $G = (V, E)$ be a graph and let C be the set of candidates nodes after considering the new edge (u, v). Let $N(w)$ be the set of neighbors of w and let $X(w)$ be the number of neighbors of w such that $\forall w' \in N(w)$, $k(w') > k(w)$ or $w \in C$. Then, $\forall w \in C$, $X(w) \leq k(u)$ implies that the coreness of u is definitely unchanged.*

PROOF. After considering an edge to be inserted (u, v), the quantity $X(w)$ consists in the number of neighbors whose coreness values are larger than $k(w)$. We note that if $X(w) \leq k(w)$, the node w cannot belong to the $(k(u) + 1)$-core and thus, the coreness of w remains equal to $k(w)$. □

In order to increase the performance of our approach, the partitioning of the input graph need to be optimized in terms of having balanced partitions and a small number of frontier edges. In the following, we present basic algorithms for the distributed k-core decomposition task. Table 1 summarizes the notations used in our algorithms.

Algorithm 1 implements the distributed orchestration mechanism that first computes the coreness in each of the partitions, and then add the frontier edges one by one. This algorithm is run by a master worker under the *M2W* computing mode. Later, any kind of edge can be added, following the same approach.

Algorithm 1: Distributed k-core decomposition

```
foreach j ∈ {1, ..., P} do
    p_i.kCore(G_i)
foreach e = (u, v) ∈ V_f do
    C ← getCandidates(e)
    C ← pruneCandidatesInsert(C)
    foreach u ∈ C do
        p_u ← partitionID(u)
        p_u.k(u) ← p_u.k(u) + 1
```

The update process is composed of three steps. The first step consists in activating the *W2W* computing mode and identifying the set of candidate nodes, i.e., the set of nodes that may need to be updated (Algorithm 2). For each frontier edge (u, v), the current coreness of nodes u and v are compared. If the coreness of u (respectively v) is greater than the coreness of v (respectively u), then the set of candidate nodes consists of nodes that are k-reachable from v (respectively u), where $k = k(v)$ (respectively, $k(u)$). If the coreness of u is equal to the coreness of v, then the set of candidate nodes consists of the union of nodes that are k-reachable from u and from v, where $k = k(v) = k(u)$.

Algorithm 2: SET getCandidates (EDGE (u, v))

```
p_u ← partitionID(u)
p_v ← partitionID(v)
C ← ∅
if k(u) < k(v) then
    C ← p_u.reachable(u)
else if k(u) > k(v) then
    C ← p_v.reachable(v)
else
    C ← p_u.reachable(u) ∪ p_v.reachable(v)
return C
```

The *reachable* (u) function returns the set of nodes that are k-reachable from u, by performing a depth-first visit. The visit can span multiple partitions, meaning that the visit of a frontier edge can lead to the visit of a node in a different partition. The pseudo-code shown below illustrates the behavior of the visit; in the real implementation, the nodes identified as potential candidates are sent back to the master node that orchestrates the execution.

The second step consists in activating the *Local* computing mode of our approach and selecting the set of nodes that need to be updated from the set of candidate nodes. This step is achieved by applying the pruning strategy introduced in Theorem 2.

Finally, the third step consists in activating the *M2W* computing mode and updating the coreness values of the set of nodes computed in the second step.

It is important to highlight that the algorithms presented above aim to compute the distributed k-core decomposition in a large partitioned graph. The frontier edges of the original graph are considered one by one after computing the k-cores of the distributed graph partitions separately. The proposed approach deals with frontier edges as edge insertions in a dynamic graph. Consequently, the proposed algorithms can be simply used to deal with edge insertions in large dynamic graphs. Algorithm 5 implements the update process for edge insertion in a large dynamic graph.

Algorithm 3: SET reachable (NODE u)

```
SET C ← ∅
if visited(u) = false then
    C ← C ∪ {⟨u, k(u), N(u)⟩}
    visited(u) ← true
    p_u ← partitionID(u)
    foreach v ∈ N(u) do
        if k(v) = k(u) then
            p_v ← partitionID(v)
            if p_u = p_v then
                C ← C ∪ reachable(v)
            else
                C ← C ∪ p_v.reachable(v)
return C
```

Algorithm 4: pruneCandidatesInsert (SET C)

```
changed ← true
while changed do
    changed ← false
    foreach ⟨u, k(u), N(u)⟩ ∈ C do
        count ← 0
        foreach v ∈ N(u) do
            if ⟨v, k(v), N(v)⟩ ∈ C or k(v) > k(u) then
                count ← count + 1
        if count ≤ k(u) then
            changed ← true
            C ← C − {⟨u, k(u), N(u)⟩}
return C
```

Algorithm 5: updateInsertions (SET S)

```
foreach e ∈ S do
    C ← getCandidates(e)
    C ← pruneCandidatesInsert(C)
    foreach u ∈ C do
        p_u ← partitionID(u)
        p_u.k(u) ← p_u.k(u) + 1
```

The edge deletion task is slightly different from edge insertion. Algorithms 6 implement the update process for edge deletion in a large dynamic graph.

Algorithm 6 is run by a master worker under the *M2W* computing mode. It consists of three main steps. The first step consists in activating the *W2W* computing mode and identifying the set of nodes that may need to be updated after the deletion using Algorithm 2. The second step consists in activating the *Local* computing mode and selecting the set of nodes that need to be updated from the set of candidate nodes. The set of nodes with unchanged coreness

Algorithm 6: updateDeletions (SET S)

```
foreach e ∈ S do
    C ← getCandidates(e)
    C ← C − pruneCandidatesDelete(C)
    foreach u ∈ C do
        p_u ← partitionID(u)
        p_u.k(u) ← p_u.k(u) − 1
```

values is computed by Algorithm 7. We notice that for edge deletions the set of nodes that need to be updated is slightly different from the set computed by Algorithm 4.

Algorithm 7: pruneCandidatesDelete (SET C)

```
changed ← true
while changed do
    changed ← false
    foreach ⟨u, k(u), N(u)⟩ ∈ C do
        count ← 0
        foreach v ∈ N(u) do
            if ⟨v, k(v), N(v)⟩ ∈ C or k(v) > k(u) then
                count ← count + 1
        if count < k(u) then
            changed ← true
            C ← C − {⟨u, k(u), N(u)⟩}
return C
```

The last step of the edge deletion task consists in activating the *M2W* computing mode and updating the coreness values of the nodes that need to be updated.

5. EXPERIMENTS

We have performed an extensive set of experiments to evaluate the effectiveness and efficiency of our approach on a number of different real and synthetic datasets. Additional and more detailed information about our datasets and our experiments in general can be found in the following link: http://db.disi.unitn.eu/pages/dkcore/.

5.1 Experimental data

Since we are interested in the core of graph data, the characteristic properties of our datasets are the number of nodes, edges, the diameter, the average clustering coefficient and the maximum coreness. These properties for the datasets that we have used in our work are indicated in Table 2. We have used two groups of datasets: (1) real-world datasets, made available by the Stanford Large Network Dataset collection [**?**] and (2) synthetic datasets, created by a graph generator based on the Nearest Neighbor model [**?**], that builds undirected graphs with power-law degree distribution with exponent between 1.5 and 1.75, matching that of online social networks.

5.2 Experimental environment

We have implemented our approach on top of the AKKA framework, a toolkit and runtime for building highly concurrent, distributed, resilient message-driven applications. In order to evaluate the performance of our approach, we used a cluster of 17 `m3.medium` instances on Amazon EC2. Each `m3.medium` instance contained 1 virtual 64-bit CPU, 3.75 GB of main memory, and a 8 GB of local instance storage. We also implemented two existing approaches for k–core decomposition in large dynamic graphs. First, we implemented Li et al.'s approach [**?**] and we run it on a machine equipped with two Intel(R) Xeon(R) E5-2440 CPUs (2.40GHz) and 192 GB of memory. Second, we implemented the HBase-based approach of Aksu et al. [**?**] and we run it on a cluster of 9 `m3.medium` instances on Amazon EC2 (1 master and 8 slaves).

5.3 Experimental protocol

In order to simulate dynamism in each dataset, we consider two update scenarios. For each scenario, we measure the performance of the system to update the core numbers of all the nodes in the considered graph after insertion/deletion of a constant number of edges.

- In the *inter-partition* scenario, we either delete or insert 1000 random edges connecting two nodes belonging to *different* partitions.
- In the *intra-partition* scenario, we either delete or insert 1000 random edges connecting two nodes belonging to *the same* partition.

We consider three figures of merit to evaluate our approach.

First, we measure the average insertion time (AIT) and the average deletion time (ADT) in the two proposed scenarios. We also compare the results of our algorithm with existing solutions for k–core decomposition in large dynamic graphs including Li et al.'s approach [**?**] and Aksu et al.'s approach [**?**].

Second, we study the data communications and networking. In this context, we measure the amount of exchanged data needed to compute the task of k-core decomposition.

Third, we study the scalability of our approach with respect to the number of machines in our cluster. In this context, we vary the number of worker machines and we measure the average insertion/deletion time for each update scenario.

5.4 Experimental results

Speedup. Table 3 illustrates the results obtained with both the real and the synthetic datasets. For each dataset, we measure the number of frontier edges and we record the average insertion time (AIT) and the average deletion time (ADT) over the 1000 insertions/deletions for both *inter-partition* and *intra-partition* scenarios. To generate the results of Table 3, we randomly partition the graph dataset into 8 partitions.

As shown in Table 3, we observe that in the *intra-partition* scenario, the values of the average insertion/deletion time are much smaller than those in the *inter-partition* scenario. This can be explained by the fact that the inserted/deleted edges in the *intra-partition* scenario are internal ones. Consequently, the amount of data to be exchanged between the distributed machines in the case of internal edges is smaller, in most cases, than the amount of exchanged data in the case of edges of the *inter-partition* scenario (i.e., frontier edges). During the k-core maintenance process after insertion/deletion of an internal edge, there is always the chance of not having to visit distributed workers/partitions other than the partition that holds the internal edge.

Table 2: Experimental data

Dataset	Type	♯ Nodes (N)	♯ Edges (M)	⊘	Avg. CC	Max(k)
DS1	Synthetic	10,000	70,622	4	0.3977	33
DS2	Synthetic	20,000	144,741	4	0.3935	38
DS3	Synthetic	50,000	365,883	4	0.3929	42
DS4	Synthetic	100,000	734,416	4	0.3908	46
ego-Facebook	Real	4,039	88,234	8	0.6055	115
email-Enron	Real	36,692	183,831	11	0.4970	43
roadNet-TX	Real	1,379,917	1,921,660	1,054	0.0470	3
roadNet-CA	Real	1,965,206	2,766,607	849	0.0464	3
com-LiveJournal	Real	3,997,962	34,681,189	17	0.2843	296
soc-LiveJournal	Real	4,847,571	68,993,773	16	0.2742	318

Table 3: Experimental results

Dataset	Number of frontier edges	AIT (ms)		ADT (ms)	
		inter	intra	inter	intra
DS1	61,803 (87.51%)	27	6	20	4
DS2	126,720 (87.54%)	39	16	27	9
DS3	320,318 (87.54%)	42	10	32	8
DS4	643,189 (87.57%)	30	10	25	8
ego-Facebook	77,253 (87.55%)	38	15	32	10
email-Enron	161,055 (87.61%)	32	8	28	6
roadNet-TX	1,681,830 (87.51%)	28	9	25	7
roadNet-CA	2,420,674 (87.49%)	30	12	26	10
com-LiveJournal	30,348,426 (87.50%)	256	30	205	27
soc-LiveJournal	59,916,050 (86.84%)	579	27	499	25

Figure 2 presents a comparison of our approach with both the sequential approach proposed by Li et al. and the HBase-based approach proposed by Aksu et al. in terms of average insertion/deletion time. For our approach, we used 9 `m3.medium` instances on Amazon EC2 (1 acting as a master and 8 acting as workers). For the HBase-based approach, we used 9 `m3.medium` instances on Amazon EC2 (1 master node and 8 slave nodes).

We notice that the Li et al.'s approach produces better results in terms of average insertion/deletion time for almost all small datasets. This can be explained by the communication cost of our approach compared to Li et al.'s approach which performs in-memory and centralized computing. For road network and LiveJournal datasets, our approach performs much faster that Li et al.'s approach with both *inter-partition* and *intra-partition* scenarios. It is also important to mention here that the Li et al.'s approach has failed to deal with LiveJournal datasets using one of the `m3.medium` instances used for the evaluation of our approach due to lack of memory.

As shown in Figure 2, our approach allows much better results compared to the HBase-based approach for almost all datasets. It is noteworthy to mention that the presented runtime values of the HBase-based approach correspond to the maintenance time of only one fixed k value core ($k = max(k)$ in our experimental study). This means that, for each dataset, the maintenance process of the HBase-based approach needs to be repeated $max(k)$ times in order to achieve the same results as our approach.

Data communications and networking. In order to study data communications and networking, we begin by examining the amount of exchanged data between the distributed machines. Then, we study the impact of the partitioning method and the number of graph partitions on the amount of exchanged data between the master node and the worker nodes. Figure 3 shows the average value of the amount of exchanged data between the master node and the worker nodes. The amount of exchanged data is shown in log-scale. For each dataset, we present the mean value of the exchanged data. As illustrated in Figure 3, the amount of data to be exchanged in the *intra-partition* scenario is much smaller than the amount of exchanged data in the *inter-partition* scenario.

In order to study the impact of the number of partitions on the amount of exchanged data, we show in Figure 4, for each number of partitions, the mean value of the exchanged data and the standard deviation value which corresponds to the error bar. This standard deviation gives a general idea of how the values of the exchanged data are concentrated around the mean value. We note that the amount of exchanged data is inversely proportional to the number of partitions in almost all datasets (see Figure 4).

Scalability. To study the scalability of our approach and to show the impact of the number of worker machines on the maintenance task runtime in the case of large-scale networks, we measured the average insertion/deletion time of our approach for each number of worker machines. We presents these results in Figure 5.

As illustrated in Figure 5, our approach scales up as the number of worker machines increases. In fact, the average insertion/deletion time of our approach is inversely proportional to the number of such machines.

6. CONCLUSIONS

The paper deals with the problem of distributed k-core decomposition in large dynamic networks. Most of the existing approaches solve the problem of k-core maintenance for graphs that can fit into the main memory of one single machine. They do not consider the cases of already distributed graphs and graphs that can not fit into one single machine. In this paper, we have introduced an efficient distributed and streaming k-core decomposition approach for large and dynamic networks. Our approach deals with graph changes/updates by selecting only the nodes of a subgraph of the original graph that really need to update their core numbers. We implemented our approach on top of AKKA framework, a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications. By running experiments on a variety of both real and syn-

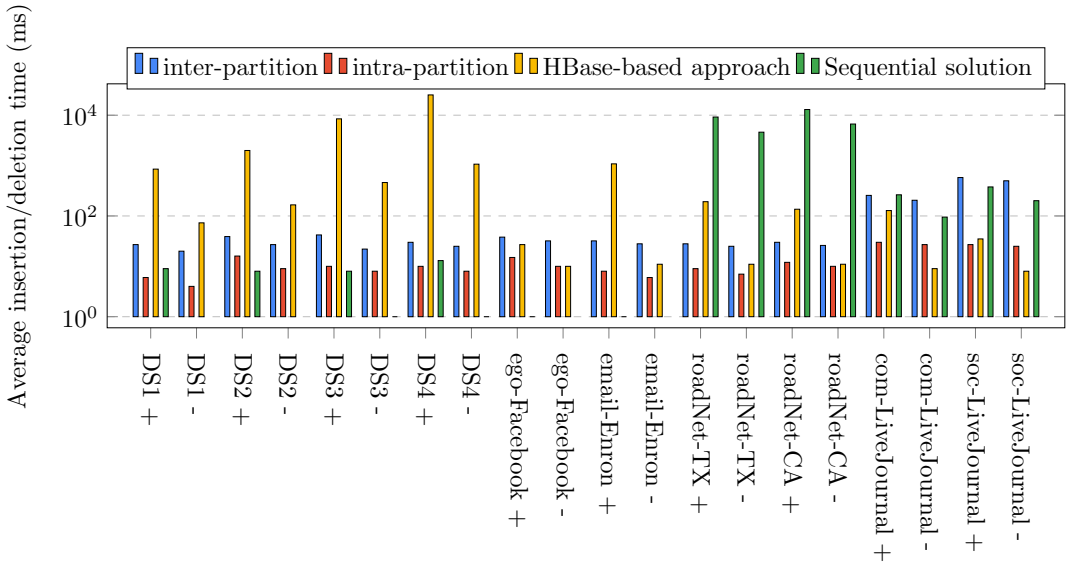


Figure 2: Average insertion/deletion time.

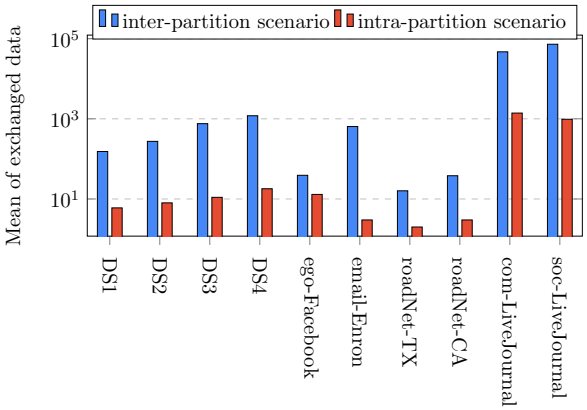


Figure 3: Amount of exchanged data.

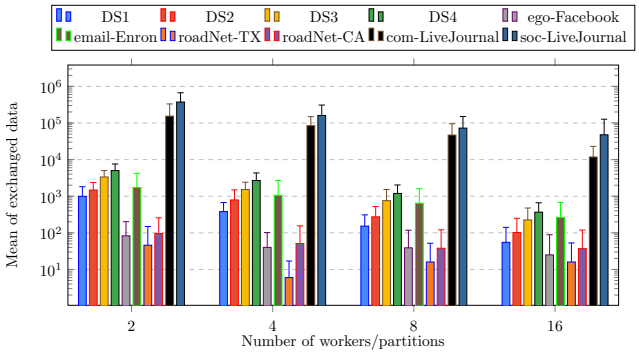


Figure 4: Impact of the number of partitions on the amount of exchanged data.

thetic datasets, we have shown that the proposed method is interesting in the case of very large graphs with a very satisfactory performance and scalability for large graphs.

Acknowledgements

This research was partially supported by EIT Digital Project "Sensemaking Service: Entity Linking for Big Linked Data" - Activity Num. 16197 - 2016.

7. REFERENCES

[1] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and O. Ulusoy. Distributed k -core view materialization and maintenance for large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2439–2452, 2014.

[2] J. I. Alvarez-Hamelin, A. Barrat, A. Vespignani, and et al. K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media*, 3(2):371, 2008.

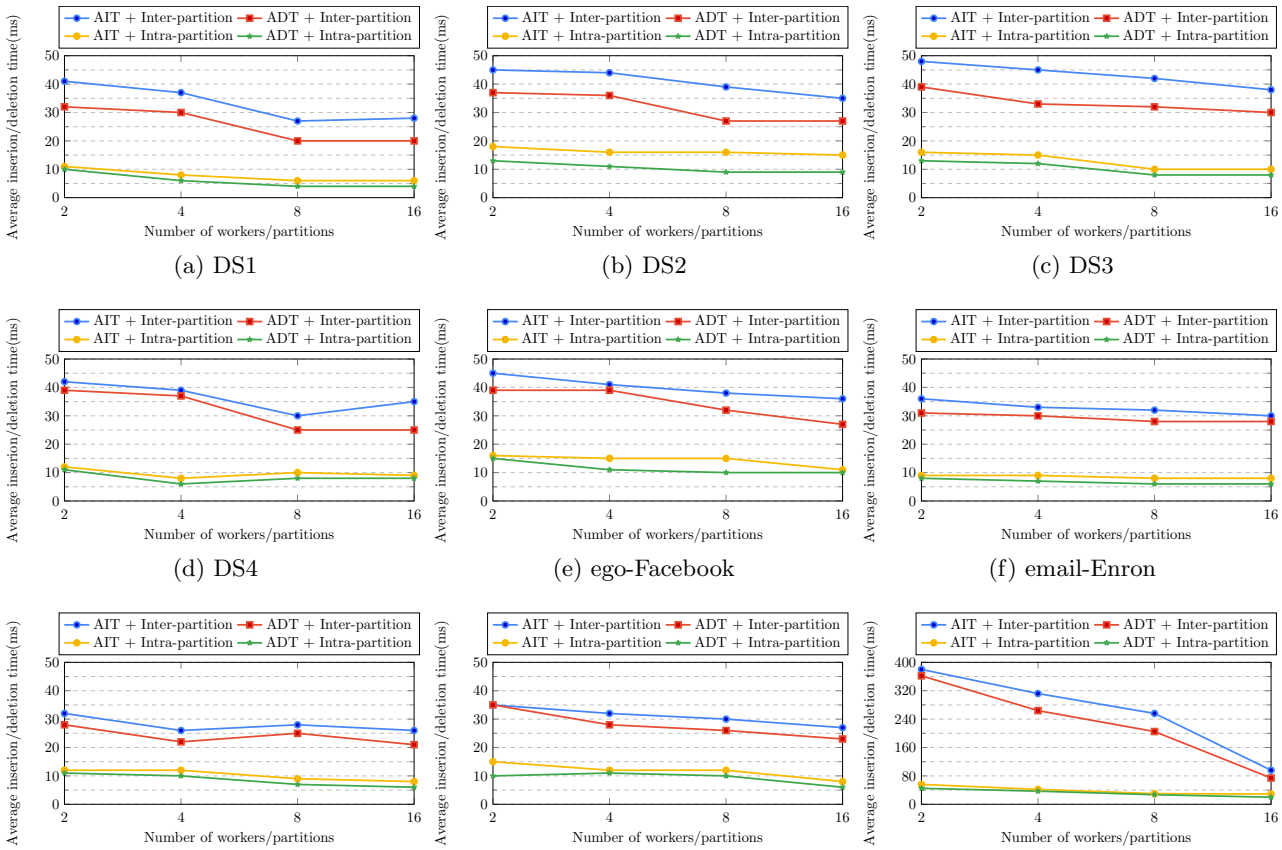
[3] J. I. Alvarez-Hamelin, M. G. Beiró, and J. R. Busch. Understanding edge connectivity in the Internet through core decomposition. *Internet Mathematics*, 7(1):45–66, 2011.

[4] G. Bader and C. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(1), 2003.

[5] V. Batagelj and M. ZaverÅąnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.

[6] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *Proc. of the 27th Int. Conf. on Data Engineering*, ICDE '11, pages 51–62, Washington, DC, USA, 2011. IEEE Computer Society.

[7] C. Giatsidis, D. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *Proc. of the Int. Conf. on*



(a) DS1 (b) DS2 (c) DS3

(d) DS4 (e) ego-Facebook (f) email-Enron

(g) roadNet-TX (h) roadNet-CA (i) com-LiveJournal

Figure 5: Effect of the number of workers on the average insertion time (AIT) and the average deletion time (ADT).

Advances in Social Networks Analysis and Mining, ASONAM'11, pages 87–93, July 2011.

[8] P. Jakma, M. Orczyk, C. S. Perkins, and M. Fayed. Distributed k-core decomposition of dynamic graphs. In *Proc. of the 2012 ACM CoNEXT Student Workshop*, New York, NY, USA, 2012. ACM.

[9] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[10] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2453–2465, 2014.

[11] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the ACM Int. Conf. on Management of Data*, SIGMOD'10, pages 135–146. ACM, 2010.

[12] D. Miorandi and F. De Pellegrini. K-shell decomposition for dynamic complex networks. In *Proc. of the 8th Int. Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks)*, WiOpt'10, pages 488–496, May 2010.

[13] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, 2013.

[14] R. Patuelli, A. Reggiani, P. Nijkamp, and F.-J. Bade. The evolution of the commuting network in Germany: Spatial and connectivity patterns. *Journal of Transport and Land Use*, 2(3), 2010.

[15] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proc. of the 19th Int. Conf. on World Wide Web*, WWW'10, pages 861–870, New York, NY, USA, 2010. ACM.

[16] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, Apr. 2013.

[17] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[18] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. van Wijk, J.-D. Fekete, and D. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Computer Graphics Forum*, 30(6):1719–1749, 2011.

[19] D. Wyatt. *Akka Concurrency*. Artima Inc., USA, 2013.

[20] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.