Data & Knowledge Engineering 100 (2015) 133–147

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

# Efficient community identification and maintenance at multiple resolutions on distributed datastores☆

Hidayet Aksu [a,*], Mustafa Canim [b], Yuan-Chi Chang [b], Ibrahim Korpeoglu [a], Özgür Ulusoy [a]

[a] Department of Computer Engineering, Bilkent University, Ankara, Turkey
[b] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

ARTICLE INFO

*Article history:*
Received 16 September 2014
Received in revised form 6 May 2015
Accepted 2 June 2015
Available online 16 June 2015

*Keywords:*
Mining methods and algorithms
Distributed databases
Community identification
Big Data analytics
*k*-Core
HBase

ABSTRACT

The topic of network community identification at multiple resolutions is of great interest in practice to learn high cohesive subnetworks about different subjects in a network. For instance, one might examine the interconnections among web pages, blogs and social content to identify pockets of influencers on subjects like 'Big Data', 'smart phone' or 'global warming'. With dynamic changes to its graph representation and content, the incremental maintenance of a community poses significant challenges in computation. Moreover, the intensity of community engagement can be distinguished at multiple levels, resulting in a multi-resolution community representation that has to be maintained over time. In this paper, we first formalize this problem using the *k*-core metric projected at multiple *k*-values, so that multiple community resolutions are represented with multiple *k*-core graphs. Recognizing that large graphs and their even larger attributed content cannot be stored and managed by a single server, we then propose distributed algorithms to construct and maintain a multi-*k*-core graph, implemented on the scalable Big Data platform Apache HBase. Our experimental evaluation results demonstrate orders of magnitude speedup by maintaining multi-*k*-core incrementally over complete reconstruction. Our algorithms thus enable practitioners to create and maintain communities at multiple resolutions on multiple subjects in rich network content simultaneously.

## 1. Introduction

Multi-resolution community identification and evolution in a complex network have applications spanning multiple disciplines ranging from social science to physics. In recent years, the rise of very large, rich content networks re-ignited interests to the problem at the multi-*k*-core scale that poses computation challenges to early work with algorithm complexity greater than $O(n)$. A further distinction from the decade-old graph problem formulation is that multi-attributed content associated with vertices and edges must be included in creating, managing, interpreting and maintaining results. Thus the problem of community analysis is a hybrid of content and graph analysis on various subjects of interest. The problem is made even more complex with the observation that interactions with a community happen not just at one but also at multiple levels of intensity, which reflects in reality active to passive participants in a group. This results in multiple levels of resolution in

☆ A preliminary version [1] of this paper was presented at the IEEE 2nd International Congress on Big Data, 2013 (BigData 2013).
* Corresponding author.
*E-mail addresses:* hidayetaksu@gmail.com (H. Aksu), mustafa@us.ibm.com (M. Canim), yuanchi@us.ibm.com (Y.-C. Chang), korpe@cs.bilkent.edu.tr (I. Korpeoglu), oulusoy@cs.bilkent.edu.tr (Ö. Ulusoy).

http://dx.doi.org/10.1016/j.datak.2015.06.001
0169-023X/© 2015 Elsevier B.V. All rights reserved.

community identification. To make the solution practical, it is thus necessary to identify and maintain communities at multiple resolutions.

In this paper, we propose a set of algorithms built on the $k$-core metric to identify and maintain a content-projected community at multiple resolutions on an open-source Big Data platform, Apache HBase. We formulate the community identification problem by first projecting a subgraph by content topic of the social network interaction, such as a microblog or message, and then locating the "dense" areas in the subgraph which represent higher inter-vertex connectivity (or interactions in the case of a social network) at multiple resolutions. In the literature, there is a long list of subgraph density measures that may be suited in different application contexts. Examples include cliques, quasi-cliques [2], $k$-core, and $k$-edge-connectivity [3]. Among these graph density measures, $k$-core stands out as the least computationally expensive one that still gives reasonable results. An $O(n)$ algorithm is known to compute $k$-core decomposition in a graph with $n$ edges [4], where other measures have complexity growing super-linear or NP-hard.

The set of our proposed algorithms identify $k$-core subgraphs at multiple, fixed $k$ values and maintain the identified subgraphs incrementally over dynamic changes. These distributed algorithms run on a multi-server cluster with shared nothing partitioned graph data, managed by Apache HBase. The size of the social network graph and the rich content is only limited by storage space and not by main memory. Furthermore, identified communities at multiresolution are also retained and updated as changes come in. Our algorithms thus enable practitioners to monitor changes in communities on different topics and resolutions in rich social network content simultaneously, which main-memory based algorithms cannot achieve.

Previously, we studied the identification and maintenance of $k$-core subgraphs at a fixed $k$ value [5]. We also proposed algorithms to perform batch operations for maintenance purposes. The proposed approaches are quite effective when a constant $k$ value is used. On the other hand, when subgraphs at multiple resolutions are needed, one has to run separate instances of the algorithms for each $k$ value. In order to cope with this limitation, we made significant design changes in these algorithms to efficiently handle $k$-core subgraphs at multiple, fixed $k$ values. The new study proposes integrated algorithms for $k$-core construction, maintenance and the bulk processing of update operations. As we demonstrate in the experiments section, the new algorithms yield orders of magnitude speed up compared to the base case $k$-core construction. Note that, we study how $k$-core can be computed efficiently in distributed datastores such as Google BigTable, MegaStore, and Apache HBase which are widely used by web companies. In other words, we target the $k$-core computation problem in a specific context, and we acknowledge that $k$-core computation in specific setups, i.e. in-memory or topology-only centralized ones, might outperform multi-purpose general distributed datastores or distributed parallel databases which are designed to address high availability, fault-tolerance, scalability, and load balancing issues.

Our main contributions in this paper can be summarized as follows:

- We defined *multiresolution* community identification in a network as a multi-$k$-core problem and developed a distributed *construction* algorithm that exploits parallelism on a Big Data platform.
- In order to keep the materialized multiresolution representation up to date with incremental updates, we developed a distributed *maintenance* algorithm that also exploits parallelism.
- We further improved the maintenance algorithm with batch window refresh for practical applications with heavy updates. Batch update maintenance allows more expensive graph traversal steps to be aggregated for computational efficiency.
- We presented a robust implementation of our algorithms on top of Apache HBase, a horizontally scaling distributed storage platform through its coprocessor computing framework [6].

The remainder of the paper is organized as follows. We first provide a motivating example in Section 2, then we review prior work on community identification and $k$-core algorithms in Section 3. Section 4 describes our distributed multi-$k$-core construction algorithms in nave implementation and pruning techniques. Section 5 details our incremental maintenance algorithms for edge insertions and deletions. Experimental results are reported and discussed in Section 7. Finally, Section 8 concludes the paper and discusses future work.

## 2. Motivating example

Consider the following scenario as an example of why the distributed multi-$k$-core construction and maintenance algorithms we propose are needed in real life problems. Suppose that a data analytics company provides keyword based analytics services to its customers based on the retweet graph of Twitter data. The customers subscribe to the service by providing certain keywords along with the queries and the company provides a response whenever they want to get the results. To keep up with the growing size of the data and to manage the query load on the system, the graph is horizontally partitioned and stored on distributed computing nodes by the company. Moreover, to effectively respond to the computation load, the results of user queries are materialized and updated as the retweet graph changes. Depending on customer needs, both instant updates as well as batch updates are reflected to the maintained results.

The aforementioned scenario is quite realistic currently. As the popularity of social media sites increases, the demand for performing analytics on these large graphs grows dramatically. In the last few years, many web companies, such as "Followerwonk" [7], "SocialPing" [8], and "SimplyMeasured" [9], emerged to help customers make better marketing decisions based on the content of social media tools such as Twitter and Google+. These web companies have to deal with very large graphs to perform analytics. To keep up with the scale, storage and maintenance of these datasets efficiently, companies are compelled to use distributed data architectures. *Hence*, *previously used analytics*, *i.e.*, *community identification algorithms*, *need to be redesigned to work in such new data management environments*. We believe that the distributed algorithms presented in this paper can be leveraged on these large graph datasets to perform better analytics.

## 3. Related work

A wide range of applications from social science to physics need to identify communities in complex networks that share certain characteristics at various scales and resolutions [10–12]. Challenges remain, however, in addressing both the intensity and dynamicity of communities on a large scale. We thus focus on metrics and algorithms whose complexity is no greater than $O(n)$. The notion of $k$-core is first introduced in [13] for measuring group cohesion in social networks. Subsequently, Batagelj and Zaversnik (BZ) proposed a linear time algorithm to compute $k$-core [4]. The BZ algorithm first sorts the vertices into the increasing order of degrees and starts deleting the vertices with degrees less than $k$. At each iteration, it needs to sort the vertices list to keep it ordered. Due to the high number of random accesses to the graph, the algorithm can only run efficiently when the entire graph can fit into the main memory of a single machine. Recently, $k$-core is preferred as a key metric of measuring social interaction strength in many other studies [14–17].

On the other hand, to deal with the scalability concerns when data gets big, graph algorithms were implemented on the MapReduce framework [18] and its open source implementation Apache Hadoop [19–21]. By formulating common graph algorithms as iterations of matrix-vector multiplications, coupled with compression, [22] and [23] demonstrated significant speedup and storage savings, although such a formulation would prevent the inclusion of metadata and content as part of the analysis. The iterative nature of graph algorithms soon prompted many to realize that static data is needlessly shuffled between MapReduce tasks [24,21,25]. Pregel [26] thus proposed a new parallel graph programming framework following the bulk synchronous parallel (BSP) model and message passing constructs. In Pregel, vertices are assigned to distributed machines and only messages about their states are passed back and forth. In our work, we achieved the same objective through coprocessors. Pregel did not elaborate, however, about how to manage temporary data, if it is large, with a main memory implementation nor did it state if updates are allowed in its partitioned graph. Furthermore, by introducing a new framework, compatibility with MapReduce-based analytics is lost.

Our work learned from the strength and limitation of these algorithms and platforms to make progress in the areas of distributed big graph data processing and incremental multi-resolution maintenance. We implemented, tested, and analyzed our algorithms on the open-source Hadoop HBase Big Data processing framework. Therefore, before going into the details of our proposed algorithms, readers are encouraged to read the preliminary version of our paper [1] or some other reference about HBase to become familiar with the Hadoop Big Data programming framework where our distributed $k$-core algorithms are implemented.

## 4. Distributed multi-*k*-core construction

In this section, we first describe a nave distributed algorithm that constructs a $k$-core subgraph, then we propose a novel algorithm to compute a $k$-core graph for multiple $k$ values simultaneously. Table 1 summarizes the notations used in our pseudocode.

### *4.1. Base algorithm*

The base algorithm is an adaptation of the BZ algorithm to distributed processing for a fixed $k$ value. As described in Algorithms 1 and 2, the server side algorithm executes in parallel as HBase Coprocessors to scan partitioned graph data in the local regions

**Table 1**
Notations used in algorithms.

| | |
|---|---|
| $G$ | Dynamic graph partitioned into regions stored in multiple server nodes |
| $G_k$ | $k$-Core materialized view graph of $G$ |
| $G_{k_i}$ | Subgraph of $G_k$ holding $k$-core for core value $k_i$ |
| $k_{1 \ldots n}$ | Target core values in ascending order |
| $R_i$ | $i$'th region of graph stored on and processed by node $i$ |
| $N_i$ | $i$'th node storing region $i$ |
| $(X, Y) \leftarrow RC_f(R_i, S)$ | Remote call to function $f$ on region $i$ takes parameter $S$ and returns values $X$, $Y$ to client |
| $\{u, v\}$ | Graph edge from vertex $u$ to vertex $v$ |
| $R_i(G_A)$ | Region of graph $G_A$ processed by node $N_i$ |
| $T_A(C_X, C_Y)$ | Lookup table A with column $C_X$ and $C_Y$ |
| $d(u)$, $d_{G_{k_i}}(u)$ | Degree of vertex $u$ in $G$ and $G_{k_i}$ |
| $qnc_{k_i}(u)$ | Qualified neighbor count for vertex $u$ in $G_{k_i}$ with respect to next core value $k_{i+1}$ |

and delete those vertices with degrees less than $k$. The client side program monitors the parallel execution and issues iterations until the $k$-core is found. To compute the $k$-core graph for multiple $k$ each $k$ value separately.

**Algorithm 1.** Base $k$-core construction — client side.

```
Input:   Graph $G = (V, E)$,
          $k$: target core value
Output: $G_k$ the $k$-core graph

$G_k \leftarrow$ clone graph $G$
$doIterate \leftarrow true$
while $doIterate$ do
    for each region $i$ in $regions(G_k)$ do
        $anyEdgeDeleted_i \leftarrow RC_{Filter\ Out\ Edges}(R_i, G_k, k)$
    Wait RCs to complete
    $doIterate \leftarrow false$
    for each region $i$ in $regions(G_k)$ do
        $doIterate \leftarrow doIterate \| anyEdgeDeleted_i$
return $G_k$
```

**Algorithm 2.** Base $k$-core construction — node $N_i$ side.

```
Upon receiving $(anyEdgeDeleted) \leftarrow RC_{Filter\ Out\ Edges}(G_k, k)$
$anyEdgeDeleted \leftarrow false$
for each edge $\{u, v\} \in R_i(G_k)$ do
    if $d(u) < k$ then
        delete $\{u, v\}$ and $\{v, u\}$ from $G_k$
        $anyEdgeDeleted \leftarrow true$
Return $anyEdgeDeleted$
```

### 4.2. Multi-k-core construction

Our proposed algorithm computes $k$-core subgraphs for a list of distinct $k$ values. As stated in the notation, $k$ values are ordered and $k_i$ is the $i$'th $k$ value, e.g. $k_{1\ \dots\ 3} = \{15, 20, 30\}$. In the degenerate case, $k_0 = 0, G_{k_0} = G$. The algorithm starts with computing a $k$-core graph for $k_1$ and progressively moves up the index by reusing a previously found $k$-core subgraph.

The algorithms are described in Algorithms 3 and 4 for the client and server side, respectively. It first computes a $k$-core graph for $k_1$ using the Base algorithm. Next, the client invokes distributed parallel processing *Compute Core* at the server side to compute core values for vertices with a degree greater than or equal to $k_i$ and less than $k_{i+1}$. On the server side, it checks a vertex's degree count and decrements its neighbors' if their degree counts are greater than $k_{i+1}$. Iterations continue until all the parallel execution reported vertices in $G_{k_{i+1}}$ have been identified.

**Algorithm 3.** Multi $k$-core construction — client side.

```
Input:   Graph $G = (V, E)$,
            $k_{1...n}$: target core values
Output: $G_k$ the $k$-core graph

$G_k \leftarrow$ Base $k$-core construction( $G, k_1$)

Create new table $T_L(C_{degree})$
for each region $i$ in $regions(G_k)$ do
    $RC_{Compute\ Degrees}(R_i, G_k, T_L)$
Wait RCs to complete

$k_{n+1} \leftarrow infinity$
$next \leftarrow k_1$
for each $k_i$ in $k_{1...n}$ do
    while $next \geq k_i$ and $next < k_{i+1}$ do
        $next \leftarrow infinity$
        for each region $j$ in $regions(G_k)$ do
            $next_j \leftarrow RC_{Compute\ Core}(R_j, k_i, k_{i+1})$
        Wait RCs to complete
        for each region $j$ in $regions(G_k)$ do
            $next \leftarrow min(next, next_j)$
```

**Algorithm 4.** Multi $k$-core construction — node $N_i$ side.

```
Upon receiving $RC_{Compute\ Degrees}(G_k, T_L)$
for each vertex $u \in R_i(G_k)$ do
    compute $d_{G_k}(u)$ and put it into $T_L(C_{degree})$
return
Upon receiving $Compute\ Core(k_i, k_{i+1})$
$next \leftarrow infinity$
for each vertex $u \in R_i$ do
    if $d_{G_k}(u) \geq k_i$ and $d_{G_k}(u) < k_{i+1}$ then
        $core[\{u\}] \leftarrow k_i$
        for each vertex $v$ adjacent to $u$ do
            if $d_{G_k}(v) \geq k_{i+1}$ then
                $d_{G_k}(v) \leftarrow d_{G_k}(v) - 1$
                if $d_{G_k}(v) < k_{i+1}$ then
                    $next \leftarrow d_{G_k}(v)$
    if $d_{G_k}(u) \geq k_{i+1}$ then
        $next \leftarrow min(next, d_{G_k}(u))$
return $next$
```

## 5. Incremental multi-$k$-core maintenance

### 5.1. Edge insertion

With graph $G = \{V, E\}$ and its materialized multi-$k$-core subgraph $G_k = \cup_{i=1..n} G_{k_i}$ where $G_{k_i} = \{V_{k_i}, E_{k_i}\}$, we give the following edge insertion theorem.

**Theorem 1.** *Given a graph $G = \{V, E\}$ and its $k$-core subgraph $G_k = \cup_{i=1..n} G_{k_i}$, and an edge $\{u, v\}$ is inserted to $G$,*

- If both $u, v \in V_{k_n}$, then $G_{k_n}$ stays the same.
- If $u$ or $v$ or both $\in V_{k_i}$ and $i$ is maximal, i.e. $\nexists (j, k) | j > i, k > i, u \in V_{k_j}$ *and* $v \in V_{k_k}$, then the subgraph consisting of vertices in $\{w | w \in V_{k_i}, d_{G_{k_i}}(w) \geq k_{i+1},\ qnc_{G_{k_i}}(w) \geq k_{i+1}\}$, where every vertex is reachable from $u$ or $v$, may need to be updated to include additional vertices into $G_{k_{i+1}}$.

The intuition behind the theorem is that an edge insertion can at most increase the core number by one. And edge inserted into the highest $k$-core $G_{k_n}$ does not change the subgraph. However, an edge inserted into vertices in $G_{k_i}$ may push some vertices to $G_{k_{i+1}}$ but not further up in the hierarchy. Fig. 1 depicts this scenario, where a new edge and its update is always sandwiched between two rings of the $k$-core graph. Bounding by the two rings implies that our maintenance algorithm can exploit this property to minimize traversal.

To prove the theorem, we first prove the following lemma, a preliminary version of which is presented in [5].

**Lemma 1.** *If vertex $q$ is included in the $k_n$-core after the edge $\{u, v\}$ is inserted, then there exists at least one path originating from either $u$ or $v$ connecting to $q$ on which all vertices also have a core number greater than or equal to $k_n$ after the insertion of the new edge.*

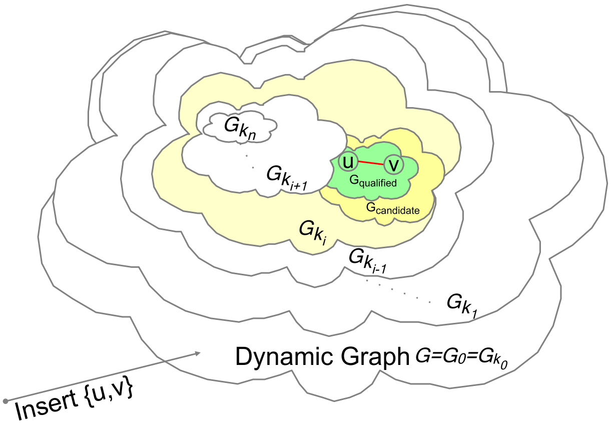

**Fig. 1.** Upon an edge $\{u, v\}$ insertion where $u$ or $v$ resides in $k_i$-core $G_{k_i}$, first tightly bounded $G_{candidate}$ graph is discovered exploiting maintained auxiliary information, then it is processed to compute $G_{qualified}$ subgraph qualifying for $k_{i+1}$-core.

**Proof.** Since $q$ was not in $V_{k_n}$ and is in $\tilde{V}_{k_n}$ after the edge insertion, its core number must have been increased from $k_{n-1}$ to $k_n$. The increase of $q$'s core number is due to one or more of its neighboring vertices whose core numbers also increased to $k_n$. The same logic applies to those neighbors and leads one or more connected paths to the vertices $u$ or $v$, where the graph topology is changed.

Using the above lemma, we now prove the edge insertion theorem by contradiction.

**Proof. Case 1.** If $u, v \in V_{k_n}$, the new edge $\{u, v\}$ is inserted into $E_{k_n}$ and there is no change to $V_{k_n}$.

**Case 2.** We prove by contradiction that a vertex $q$ in $G_{k_i}$, where $i$ is maximal, cannot be in the $k_{i+1}$-core, unless $q \in \{w | w \in V_{k_i}, D_G(w) \geq k_{i+1}, N_G^{k_{i+1}}(w) \geq k_{i+1}\}$ where all the vertices in the set are reachable by either $u$ or $v$. Suppose $q$ is in the $k_i$-core but $q \notin \{w | w \in V, D_G(w) \geq k_{i+1}, N_G^{k_{i+1}}(w) \geq k_{i+1}\}$ where all the vertices in the set are reachable by either $u$ or $v$. The above lemma states that there exists at least one path originating from either $u$ or $v$ connecting to $q$ on which all vertices also have a core number greater than or equal to $k_{i+1}$. By definition of $k$-core, the vertices on the path must have $D_G \geq k_{i+1}$ and $N_G^{k_{i+1}} \geq k_{i+1}$. Therefore, the vertices on the path to $q$ and including $q$ must also have $D_G \geq k_{i+1}$ and $N_G^{k_{i+1}} \geq k_{i+1}$. Therefore, $q$ must be in the subgraph expanded from $u$ and $v$.

Algorithms 5, 6 and 7 present the algorithms in detail. Several auxiliary counts are maintained for all vertices, $\forall\, v \in V$, its degree $d_{G_{k_i}}(v)$ and its qualifying neighbor count $qnc_{G_{k_i}}(v)$ for each maintained $k_i$. For each insert, the algorithm first looks for the maximal subgraph $G_{k_i}$ in which $u$ or $v$ is found. If any such $G_{k_i}$ graph is found for $i > 0$, a new edge is inserted and auxiliary information is updated. When $i$ is equal to $n$, which means both vertices are in the inner most core graph, no update is required so the algorithm terminates. If the *qnc* value for either vertex is no less than the next target $k_{i+1}$ value, then there is a possibility that $G_{k_{i+1}}$ will be updated because of the new edge. In this case, the algorithm searches the graph and marks a tightly bounded subgraph of vertices which needs to be updated. *Find Candidate Graph* subroutine in Algorithm 6 traverses the $G_{k_i}$ subgraph and returns the $G_{candidate}$ subgraph which covers the set of candidate edges that may be part of the $k_{i+1}$-core. The edges whose vertex $w$ satisfy the condition $d(w) \geq k_{i+1}$ and $qnc_{k_{i+1}}(w) \geq k_{i+1}$ are considered as candidate edges for $G_{k_{i+1}}$. *Partial KCore* in Algorithm 7 then processes the $G_{candidate}$ subgraph and returns the graph qualified for $k_{i+1}$ core into $G_{qualified}$.

**Algorithm 5.** Edge insertion — node $N_i$ side

```
Input:    Graph $G = (V, E)$,
              $G_k$: the multi $k$-core graph,
              $\{u, v\}$: new edge,
              $k_{1...n}$: maintained core values
Output: the updated $k$-core graph

Auxiliary Update($G, u, v, k_{1...n}$)                      ▷ Update the auxiliary values
$i = min\{i | u \in G_{k_i}$ or $v \in G_{k_i}\}$
if $i > 0$ then                                             ▷ both vertices are in core graph
    insert edge $\{u, v\}$ and $\{v, u\}$ into $G_{k_i}$
    Auxiliary Update($G_k, u, v, k_{1...n}$)
if $i == n$ then
    return
if $d(u) < k_{i+1}$ or $d(v) < k_{i+1}$ then
    return
$G_{candidate} \leftarrow \emptyset$
if $qnc_{k_{i+1}}(u) \geq k_{i+1}$ or $qnc_{k_{i+1}}(v) \geq k_{i+1}$ then
    $G_{candidate} \leftarrow$ Find Candidate Graph($G_{k_i}, G_{k_{i+1}}, C, k_{i+1}, u$)
if $G_{candidate} \neq \emptyset$ then
    $G_{qualified} \leftarrow$ Partial KCore ($G_{candidate}, k_{i+1}$)
    $G_{k_{i+1}} \leftarrow G_{k_{i+1}} \cup G_{qualified}$
```

### 5.2. Edge deletion

We begin with the following edge deletion theorem, which mirrors the edge insertion theorem.

**Theorem 2.** *Given a graph $G = \{V, E\}$ and its k-core subgraph $G_k = \cup_{i=1..n} G_{k_i}$, and an edge $\{u, v\}$ is deleted from G,*

- If $\{u, v\} \notin E_{k_i}$, then $G_{k_i}$ does not change.
- If $\{u, v\} \in E_{k_i}$ and $i$ is maximal, then the subgraph consisting of vertices in $\{w | w \in V_{k_i}\}$, where every vertex is reachable from $u$ or $v$, may need to be updated to maintain edge deletion from $G_{k_i}$.

The intuition behind this theorem is that an edge deletion can at most decrease the core number by one and thus an edge deleted from $G_{k_i}$ may push some vertices from $G_{k_i}$ to $G_{k_{i-1}}$ but not further down in the hierarchy. Again, our algorithm exploits the property to minimize traversal.

Algorithm 8 implements the theorem on the server side. Edge deletion logic is similar to the edge insertion case. Upon receiving an edge deletion, it first finds out in which $k$-core graph this edge resides, say $G_{k_i}$. If it does not reside in any $k$-core, then the algorithm terminates. Otherwise, the *Update Coreness Cascaded* algorithm in Algorithm 9 starts with the vertex with $d_{G_{k_i}}$ less than $k_i$, and moves it to the lower $k$-core graph $G_{k_{i-1}}$. Then it recursively traverses the neighbors whose degrees in $G_{k_i}$ are now below $k_i$. The algorithm accelerates $k$-core re-computing by knowing, at each iteration, which vertices have changed their degrees. For the majority of cases where an edge deletion impacts a small fraction of vertices in the $k$-core, we have found this improved algorithm to be very effective.

**Algorithm 6.** Find candidate graph.

```
Input:   $G_{k_i}$: base $k$-core graph,
         $G_{k_{i+1}}$: target $k$-core graph,
         $C$: set of candidate edges,
         $k_j$: target core value,
         $u$: start vertex
Output: $C$: set of candidate edges

$Q \leftarrow$ new queue
$Q$.enqueue($u$)
mark($u$)
while $Q \neq \emptyset$ do
    $v \leftarrow Q$.dequeue()
    if $v$ is not local then remote request for edges of $v$
    for each vertex $w$ adjacent to $v$ in $G_{k_i}$ do
        if $\{v, w\} \notin C$ then
            if $d(w) \geq k_j$ and $qnc_{k_j}(w) \geq k_j$ then
                $C \leftarrow C \cup \{v, w\}$
                if $w \notin G_{k_{i+1}}$ then
                    $C \leftarrow C \cup \{w, v\}$
                    if $w$ is not marked then
                        $Q$.enqueue($w$)
                        mark($w$)
return $C$
```

**Algorithm 7.** Partial *KCore*.

```
Input:   $C$: set of candidate edges,
         $k_j$: target core value,
Output: $C$: the updated set of edges qualifying for $k$-core

changed $\leftarrow$ true
while changed do
    changed $\leftarrow$ false
    for each $\{u, v\} \in C$ do
        if $d_C(u) < k_j$ then
            delete $\{u, v\}$ and $\{v, u\}$ from $C$
            changed $\leftarrow$ true
return $C$
```

**Algorithm 8.** Edge Deletion — node $N_i$ side.

```
Input:   Graph $G = (V, E)$,
         $G_k$: the multi $k$-core graph,
         $\{u, v\}$: the edge to be deleted,
         $k_{1...n}$: maintained core values
Output: the updated $k$-core graph

Auxiliary Update($G$, $u$, $v$, $k_{1...n}$)                        ▹ Update the auxiliary values
$i = min\{i | u \in G_{k_i}$ or $v \in G_{k_i}\}$
if $i == 0$ then                                ▹ when edge is not in $G_k$, no change occurs
    return
delete $\{u, v\}$ and $\{v, u\}$ from $G_{k_i}$
Auxiliary Update($G_k$, $u$, $v$, $k_{1...n}$)
if $d_{G_{k_i}}(u) \geq k_i$ and $d_{G_{k_i}}(v) \geq k_i$ then
    return
if $d_{G_{k_i}}(u) < k_i$ then
    Update Coreness Cascaded($G_k$,$i$,$u$)
if $d_{G_{k_i}}(v) < k_i$ then
    Update Coreness Cascaded($G_k$,$i$,$v$)
```

**Algorithm 9.** Update coreness cascaded.

```
Input:  $G_k$: the multi $k$-core graph,
        $i$: maintained core value index,
        $u$: start vertex
Output: the updated $G_k$

$Q \leftarrow$ new queue
$Q$.enqueue($u$)
mark($u$)
while $Q \neq \emptyset$ do
    $v \leftarrow Q$.dequeue()
    $core[v] \leftarrow k_{i-1}$                              ▹ decrease vertex core value.
    for each vertex $w$ adjacent to $v$ in $G_{k_i}$ do
        if $k_{i-1} == 0$ then
            delete $\{v, w\}$ and $\{w, v\}$ from $G_{k_i}$
        if $d_{G_{k_i}}(w) < k_i$ then
            if $w$ is not marked then
                $Q$.enqueue($w$)
                mark($w$)
```

## 6. Batch multi-*k*-core maintenance

In an update-heavy workload, *k*-core does not need to be kept in lock steps with data updates and thus presents the opportunity to periodically maintain *k*-core in batch windows. Accumulating data updates and refreshing *k*-core in a batch bundles up expensive graph traversals and thus speeds up maintenance time, compared to maintaining each update incrementally.

In such a batch maintenance scenario, edge insertion and deletion incurs immediate updates to the auxiliary information, degree and QNC, while updates to the *k*-core subgraph are deferred. The system maintains a list of updates and flushes them based on update count or clocked window. In Algorithm 10, when the list is flushed, updates that cancel each other out are first removed from the list. Edge deletions, which typically incur a shorter graph traversal, are then treated next followed by edge insertions, which may include longer traversals. Regardless of the processing order, the net effect is the same.

**Algorithm 10.** Batch process — client side.

```
Input:  Graph $G = (V, E)$,
        $k_{1...n}$: maintained core values,
        $G_k$: $k$-core graph,
        $batchOperations$: list of operations stored in batch part
Output: the updated $k$-core graph

$deleteList \leftarrow$ choose delete operations from $batchOperations$
Perform Delete Traversals($G_k, deleteList, k_{1...n}$)
$insertList \leftarrow$ choose insert operations from $batchOperations$
Perform Insert Traversals($G, G_k, insertTraversals, k_{1...n}$)
```

Algorithm 11 presents the batch edge deletions in more detail. Edges in the deletion list *deleteList* are grouped and sent to the respective region's node, where each remote call returns a list of cascaded deletion requests. The client then regroups the requests.

**Algorithm 11.** Perform delete traversals — client side.

```
Input:  $G_k$: $k$-core graph,
        $deleteList$: list of edges to be deleted,
        $k_{1...n}$: maintained core values
Output: the updated $k$-core graph

$downgradeCoreList \leftarrow \emptyset$
while $deleteList \neq \emptyset$ or $downgradeCoreList \neq \emptyset$ do
    for each region $i$ in $regions(G_k)$ do
        $del_i \leftarrow$ from deleteList filter edges stored in $R_i$
        $down_i \leftarrow$ from downgradeCoreList filter vertices stored in $R_i$
        $\{cDel_i, cDown_i\} \leftarrow RC_{Handle\ Delete}(R_i, G_k, del_i, down_i, k_{1...n})$
    Wait RCs to complete
    $deleteList \leftarrow \emptyset$
    $downgradeCoreList \leftarrow \emptyset$
    for each region $i$ in $regions(G_k)$ do
        if $cDel_i \neq \emptyset$ then
            add $cDel_i$ to deleteList
        if $cDown_i \neq \emptyset$ then
            add $cDown_i$ to downgradeCoreList
```

Algorithm 12 describes the node side of edge deletions. The algorithm first receives the list *deleteList*, the list of edges to be deleted, and the list *downgradeList*, the list of vertices to be updated into one lower $k$ values in the maintained $k$-core list $k_{1 \ldots n}$. For each edge $\{u, v\}$ in the *deleteList* the edge is deleted first. To do that, all the outgoing edges of $u$ are deleted and the incoming edges are returned to the client as cascaded delete. We do not delete incoming edges in this remote call since those edges do not necessarily reside in the current region (edges are stored according to source vertex). If the degree of $u$ becomes smaller than the $k$ value of the core in which it currently resides, this vertex should be downgraded. Such vertices are added to the *downgradeList*. Each vertex in the *downgradeList* is moved from its current core value, let's say $k_i$, to one lower core value $k_{i-1}$. When the vertex becomes lower than the minimum maintained $k$ value, it is deleted from the materialized view and any cascaded delete is added to the *cascadedDeletes* list. After each core change, if any direct neighbor also needs to be updated, it is added to the *cascadedDowngrades* list to be processed in the next iteration.

**Algorithm 12.** Handle delete — node $N_i$ side.

```
Input:  $G_k$: $k$-core graph,
        $deleteList$: list of edges to be deleted,
        $downgradeList$: list of vertices to be downgraded,
        $k_{1 \ldots n}$: maintained core values
Output: $cascadedDeletes$ the cascaded delete list,
        $cascadedDowngrades$: the cascaded downgrade list

for each edge $\{u, v\}$ in $deleteList$ do
    delete $\{u, v\}$ from $G_k$
    $i \leftarrow$ core index for $core[u]$
    if $d_{G_{k_i}}(u) < k_i$ then
        $downgradeList \leftarrow downgradeList \cup \{u\}$
$cascadedDeletes \leftarrow \emptyset$
$cascadedDowngrades \leftarrow \emptyset$
for each vertex $\{u\}$ in $downgradeList$ do
    $i \leftarrow$ core index for $core[u]$
    $core[u] \leftarrow k_{i-1}$
    for each vertex $w$ adjacent to $u$ in $G_{k_i}$ do
        if $k_{i-1} == 0$ then
            delete $\{u, w\}$ from $G_{k_i}$
            $cascadedDeletes \leftarrow cascadedDeletes \cup \{w, u\}$
        if $d_{G_{k_i}}(w) < k_i$ then
            $cascadedDowngrades \leftarrow cascadedDowngrades \cup \{w\}$
return $\{cascadedDeletes, cascadedDowngrades\}$
```

Algorithm 13 presents batched edge insertion maintenance in detail. In essence, the independently launched graph traversal in each incremental maintenance is now aggregated into a single parallel graph traversal launched simultaneously from all the new edges. The algorithm first takes the list of edges *insertList*, and traverses them in parallel. All candidate edges discovered

**Algorithm 13.** Perform insert traversals — client side.

```
Input:  Graph $G = (V, E)$,
        $G_k$: $k$-core graph,
        $insertList$: list of vertices to be traversed,
        $k_{1 \ldots n}$: maintained core values
Output: the updated $k$-core graph

$G_{candidate_{1 \ldots n}} \leftarrow \emptyset$
while $insertList \neq \emptyset$ do                                          ▷ at each iteration
    for each region $i$ in $regions(G)$ do
        $bucket_i \leftarrow from\ insertList\ filter\ vertices\ stored\ in\ R_i$
        $qualifyingList_i \leftarrow RC_{Pruned\ MultiTraversal}(R_i, bucket_i, k_{1 \ldots n})$
    Wait RCs to complete
    $insertList \leftarrow \emptyset$
                                       ▷ Aggregate this turn results and compute next turn input
    for each region $j$ in $regions(G)$ do
        for each edge $\{u, v\}$ in $qualifyingList_j$ do
            $i \leftarrow$ core index for $core[u]$
            if $\{u, v\} \notin G_{candidate_i}$ then                           ▷ Select a vertex only once
                $G_{candidate_i} \leftarrow G_{candidate_i} \cup \{u, v\}$
                if $v \notin G_{k_{i+1}}$ then              ▷ do not go over vertices already in $G_{k_{i+1}}$
                    $G_{candidate_i} \leftarrow G_{candidate_i} \cup \{v, u\}$
                    $insertList \leftarrow insertList \cup \{v\}$                  ▷ continue traverse
for $i$ in $1 \ldots n$ do
    if $G_{candidate_i} \neq \emptyset$ then
        $G_{qualified_i} \leftarrow$ Partial KCore $(G_{candidate_i}, k_{i+1})$
        for each vertex $\{u\}$ in $G_{qualified_i}$ do
            $core[u] \leftarrow k_{i+1}$
```

during BFS traversals are kept in the client side in sets called $G_{candidate_{1\ldots n}}$ for posttraversal computation. Firstly, the client groups all vertices in the *insertList* according to their regions. A BFS operation is performed for each region by calling the $qualifyingList_i \leftarrow RC_{Pruned\ MultiTraversal}(R_i, bucket_i, k_{1\ \ldots\ n})$ remote call function. Each node handles the BFS iterations over its associated region and then returns the selected edge list to the client. The list *insertList* is cleared after all remote calls are made. An edge $\{u, v\}$ returned from the remote call is skipped if it already exists in the set $G_{candidate_{1\ldots n}}$, which means it has already been traversed previously. When the new edge $\{u, v\}$ is not connected to the next inner $k$-core subgraph, $v$ is inserted into the list *insertList* to be traversed in the next iteration.

Once the parallel traversal is done, the candidate lists $G_{candidate_{1\ldots n}}$ will be processed by the *Partial KCore* algorithm to compute each maintained $k$-core over the traversed graph.

The *Pruned MultiTraversal* algorithm described in Algorithm 6 runs on the node side and performs a single BFS iteration for the vertices in the *insertList* list. It selects the edges to the vertices with a QNC value greater than the next maintained core value.

**Algorithm 14.** Pruned MultiTraversal — node $N_i$ side.

```
Input:  Graph $G = (V, E)$,
          $insertList$: list of vertices to be traversed,
          $k_{1\ldots n}$: maintained core values
Output: $qualifyingList$: list of edges to qualifying neighbors

$returnList \leftarrow \emptyset$
for each vertex $u$ in $insertList$ do
    $i \leftarrow$ core index for $core[u]$
    for each vertex $w$ adjacent to $u$ in $G_{k_i}$ do
        if $d_{G_{k_i}}(w) \ge k_{i+1}$ and $qnc_{G_{k_i}}(w) \ge k_{i+1}$ then
            $qualifyingList \leftarrow qualifyingList \cup \{u, w\}$
return $qualifyingList$
```

## 7. Performance evaluation

We ran experiments to demonstrate the performance of our proposed multi-$k$-core construction algorithm and the performance of our proposed $k$-core maintenance algorithms on dynamic graphs. We show that recomputing the $k$-core subgraphs is much costlier than incrementally maintaining them in dynamic graphs where the edges are inserted and deleted.

### 7.1. System setup and datasets

Graph data is stored in HBase and the algorithms are implemented as HBase Coprocessors where distributed parallelism is applicable. Table 2 shows how notations in algorithms are interpreted in HBase implementation. Our cluster consists of one master server and 13 slave servers, each of which is an Intel CPU based blade running Linux connected by a 10-gigabit Ethernet. We use a vanilla HBase environment running Hadoop 1.0.3 and HBase 0.94 with data nodes and region servers co-located on the slave servers. We configured HBase with a maximum 16 GB Java heap space and Hadoop with a 16 GB heap to avoid long garbage collection in the Java virtual machine. The HDFS (Hadoop File System) replication factor is set at the default three replicas. There was no significant interference from other workloads on the cluster during the experiments.

The datasets we used in the experiments were made available by Mislove et al. [27] and the Stanford Network Analysis Project [28]. We appreciate their generous offer to make the data openly available for research. For details, please see the references and we only briefly recap the key characteristics of the data in Table 3.

### 7.2. Experiments

We use multiple $k$ values to represent a community at multiple resolutions. For each social network dataset, we select three distinct $k$ values so that 4, 8 and 16% of the vertices in that dataset have a degree of at least $k$. The higher the $k$ value, the stronger or more tightly knit the communities are. Conversely, the lower the $k$ value, the weaker or more loosely connected the communities

**Table 2**
Mapping of graph notations in Table 1 to implementation in HBase.

| | |
|---|---|
| $G$ | HBase table holding graph edges partitioned into regions over multiple region servers |
| $G_k$ | HBase table holding $k$-core graph edges |
| $R_i$ | $i$'th region processed by coprocessor $N_i$ |
| $N_i$ | $i$'th coprocessor running on region $i$ |
| $(X, Y) \leftarrow RC_f(R_i, S)$ | Coprocessor function $f$ on region $i$ takes parameter $S$ and returns values $X$, $Y$ to client |
| $R_i(G_A)$ | Region of $G_A$ processed by coprocessor $N_i$ |
| $T_A(C_X, C_Y)$ | Table A created on HBase with column $C_X$ and $C_Y$ |

**Table 3**
Key characteristics of datasets in the experiments.

| Name | Vertex Count | Bidirectional Edge Count | Ref |
|---|---|---|---|
| Orkut | 3.1 M | 234 M | [27] |
| LiveJournal | 5.2 M | 144 M | [27] |
| Flickr | 1.8 M | 44 M | [27] |
| Patents | 3.8 M | 33 M | [28] |
| Skitter | 1.7 M | 22.2 M | [28] |
| BerkStan | 685 K | 13.2 M | [28] |
| YouTube | 1.1 M | 9.8 M | [27] |
| WikiTalk | 2.4 M | 9.3 M | [28] |
| Dblp | 317 K | 2.10 M | [28] |

**Table 4**
*k* values used in the experiments and the ratio of vertices with degree at least *k* in the corresponding graphs.

| Dataset — *k* values | 4% | 8% | 16% |
|---|---|---|---|
| Orkut | 263 | 183 | 123 |
| LiveJournal | 80 | 50 | 28 |
| Flickr | 65 | 24 | 9 |
| Patents | 28 | 21 | 15 |
| Skitter | 42 | 26 | 15 |
| BerkStan | 57 | 38 | 24 |
| WikiTalk | 5 | 3 | 2 |
| YouTube | 18 | 10 | 5 |
| Dblp | 25 | 16 | 10 |

are. Table 4 lists the chosen *k* values. We first run the *Base k-core construction* algorithm to measure the baseline *k*-core construction time for each dataset and *k* value. Then we run the *Multi-k-core construction* algorithm, which is described in Algorithms 3 and 4, for each dataset with all the chosen *k* values at once to measure the *k*-core construction for multiple *k* values. Fig. 2 shows the construction times for both algorithms. The speedup achieved by *Multi-k-core construction* algorithm is upper bounded by the number of distinct values which is 3 in this case. For larger datasets we observe that the algorithm achieved a higher speedup due to the redundant computation saved.

To evaluate the performance of the maintenance Algorithms 5 and 6, we first construct and materialize a *k*-core graph for selected multiple *k* values and under the three scenarios explained below we measure the average maintenance times.

1. In the *Extending window* scenario, a constant number of edges are continuously inserted into the original graph. We randomly choose 1000 vertices from the graph and exclude a random edge of each vertex at the beginning. Later we construct the *k*-core subgraph. Once the system is ready for changes we insert the excluded edges into the graph one by one while we maintain the *k*-core subgraph.

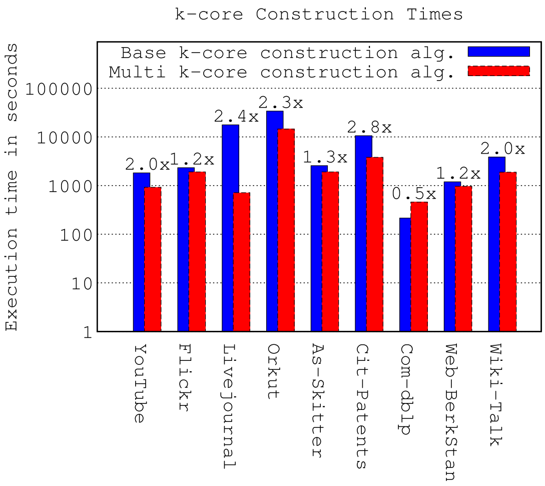

**Fig. 2.** *k*-Core construction times for base and multi *k*-core construction algorithms are shown for each dataset with three chosen *k* values. Relative speedup achievement of multialgorithm over Base algorithm is provided above each bar.

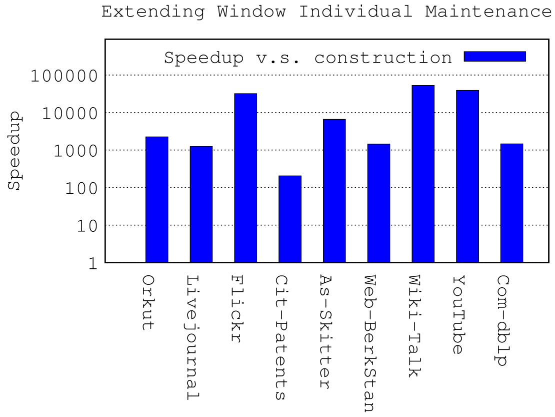

**Fig. 3.** *k*-Core maintenance algorithm speedup over construction algorithms for extending window scenario.

2 In the *Shrinking window* scenario, a constant number of edges are continuously deleted from the original graph. We first construct the *k*-core subgraph. Later, we randomly choose 1000 vertices from the graph to delete them one by one while we maintain the *k*-core subgraph.

3 In the *Moving window* (Mix) scenario, a constant number of edges are both inserted and deleted continuously. We choose 1000 random vertices from the graph and exclude a random edge of each vertex at the beginning. We use them for insertion. Next we construct the *k*-core subgraph. Once the system is ready for changes we keep inserting these edges while deleting a random edge from the original graph. So, each insertion is followed by a random deletion from the graph. By doing so we insert 1000 edges into the graph and delete 1000 edges from the graph while maintaining the *k*-core subgraph.

We repeated these three scenarios with each dataset and measured their execution times. Figs. 3–5 plot the speedup through our incremental maintenance algorithms over recomputing *k*-core from scratch, for 9 different datasets. The *y*-axis shows the speedup in log-scale. For the extending, shrinking, and moving window scenarios and each dataset, the figures give the speedup of the incremental update approach with respect to from-scratch construction using the multi-*k*-core construction algorithm. As the figures show, three to five orders of magnitude speedup can be expected for the edge insertion workload. We observe different speedup values for different datasets. When compared with Table 3 and Table 4, there is no direct relation between the *k* values, dataset size and achieved speedup. The different datasets we used are from different networks, i.e., Com-dblp is a collaboration network, Orkut is a friendship network, and Flickr is an image sharing network. Their internal connection structure causes different maintenance costs. For instance homogenous and dense edge distribution would cause longer traversals in Algorithm 14 *Pruned MultiTraversal*, while heterogeneous and sparse edge distribution would result in shorter traversals. Thus, the internal edge distribution structure, which is different in each network, causes different maintenance speedups. Similar speedup factors are also observed for mixed edge insertions and deletions with a one-to-one ratio. A higher speedup of more than five orders of magnitude was achieved for the edge deletion only workload.

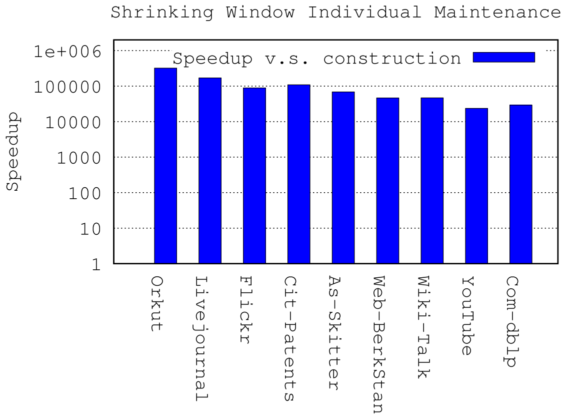

**Fig. 4.** *k*-Core maintenance algorithm speedup over construction algorithms for shrinking window scenario.

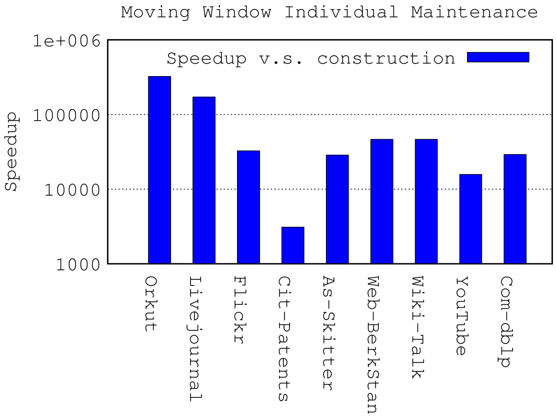

**Fig. 5.** *k*-Core maintenance algorithm speedup over construction algorithms for moving window scenario.

### 7.3. Batch maintenance experiments

In order to investigate the speedup provided by our batch update approach for maintaining the multi-*k*-core subgraph, we ran experiments on different datasets. To measure the performance improvement of the batch processing approach compared to individual updates and full reconstruction, we set up experiments for each dataset and for each update scenario described in Section 7. For each experiment, we used a 10 K batch size. Figs. 6–8 show batch processing speedup versus individual processing in *k*-core individual

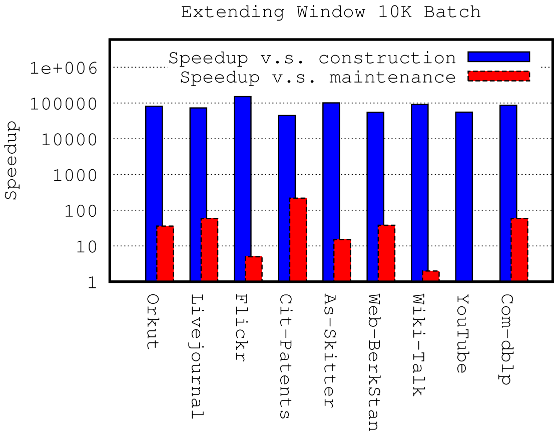

**Fig. 6.** 10 K sized batch maintenance speedups for extending window scenario.

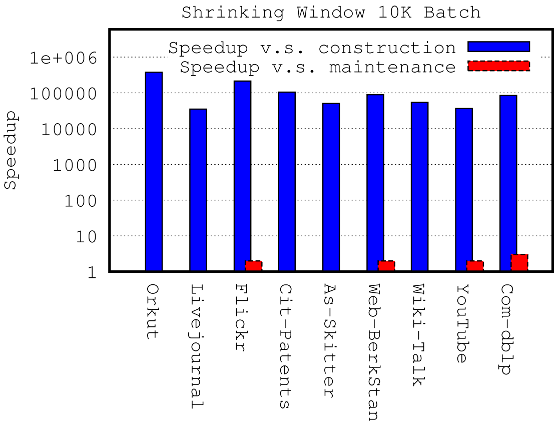

**Fig. 7.** 10 K sized batch maintenance speedups for shrinking window scenario.

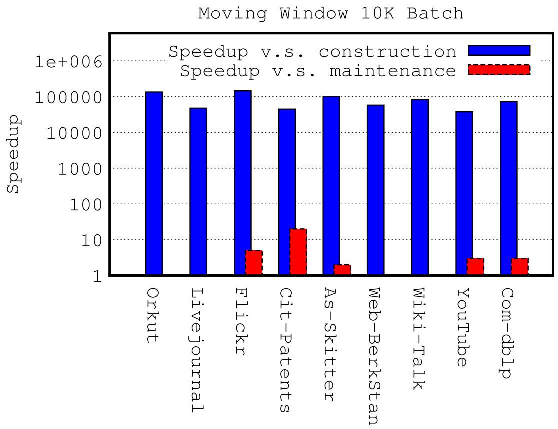

**Fig. 8.** 10 K sized batch maintenance speedups for moving window scenario.

maintenance and reconstruction for three different update scenarios and 9 different datasets. The speedup is shown in the $y$-axis in the log-scale. Each figure illustrates the batch maintenance speedup versus both individual maintenance and reconstruction.

For the Extending window scenario, in Fig. 6 we get a greater performance improvement of four to five orders of magnitude speedup when compared with the reconstruction case. We get up to 2 orders of magnitude speedup compared to individual maintenance. When compared with Fig. 3, we figured out that the batch window extending approach provides a more stable speedup ratio for different datasets. For instance, the Cit-Patent dataset shows the largest speedup in the batch maintenance case compared to individual maintenance, while it provides the smallest speedup in the individual maintenance case. In total, its speedup is close to the average speedup of other datasets. For the shrinking window scenario, the batch processing approach does not provide significant speedup, as the deletion cost in individual processing is already minimal, i.e., close to the auxiliary maintenance cost plus the base HBase update times. The moving window case provides speedups in between the extending and shrinking window cases, which is as expected considering that it is a mixture of insertion and deletion operations. The experiment results show that batch processing provides stable speedup for all scenarios and datasets with different sizes and topologies.

## 8. Conclusions

To the best of our knowledge this paper is the first to propose a horizontally scaling solution on the Big Data platform for multi-resolution social network community identification and maintenance. By using $k$-core as the measure of community intensity, we proposed multi-$k$-core construction and incremental maintenance algorithms and ran experiments to demonstrate orders of magnitude speedup with the aggressive pruning and fairly low maintenance overhead in the majority of graph updates at relatively high $k$-valued cores. We further extended algorithms to handle batch maintenance of a window of updates, which provides larger and more stable speedup for multi-$k$-core maintenance.

For the simplicity of the presentation, we left out the metadata and content associated with graph vertices and edges. In practice, a $k$-core subgraph is often associated with application context and semantic meaning. Our efficient maintenance algorithms now enable many practical applications to keep many $k$-core materialized views up-to-date and ready for user exploration.

We provided a distributed implementation of the algorithms on top of Apache HBase, leveraging its horizontal scaling, range-based data partitioning, and the newly introduced coprocessor framework. Our implementation took full advantage of the distributed, parallel processing of the HBase Coprocessors. Building the graph data store and processing on HBase also benefits from the robustness of the platform and its future improvements.

## Acknowledgments

This research was sponsored by DARPA under agreements no. W911NF-11-C-0200 and W911NF-12-C-0028. We thank TUBITAK (The Scientific and Technological Research Council of Turkey) for supporting this work in part with project 113E274. The authors would like to thank the anonymous reviewers for their helpful comments.

## References

[1] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, O. Ulusoy, Multi-resolution social network community identification and maintenance on big data platform, Big Data (BigData Congress), 2013 IEEE International Congress on 2013, pp. 102–109, http://dx.doi.org/10.1109/BigData.Congress.2013.23 (doi:10.1109/BigData.Congress.2013.23).

[2] Z. Zeng, J. Wang, L. Zhou, G. Karypis, http://doi.acm.org/10.1145/1242524.1242530Out-of-core coherent closed quasi-clique mining from large dense graph databases, ACM Trans. Database Syst. 32 (2). http://dx.doi.org/10.1145/1242524.1242530 doi:10.1145/1242524.1242530. URL http://doi.acm.org/10.1145/1242524.1242530

[3] R. Zhou, C. Liu, J.X. Yu, W. Liang, B. Chen, J. Li, Finding maximal k-edge-connected subgraphs from a large graph, Proceedings of the 15th International Conference on Extending Database Technology, EDBT'12, ACM, New York, NY, USA 2012, pp. 480–491, http://dx.doi.org/10.1145/2247596.2247652 (http://dx.doi.org/10.1145/2247596.2247652 doi:10.1145/2247596.2247652. URL http://doi.acm.org/10.1145/2247596.2247652).

[4] V. Batagelj, M. Zaversnik, An o(m) algorithm for cores decomposition of networks, CoRR cs.DS/0310049

[5] H. AKSU, M. Canim, Y. Chang, I. Korpeoglu, O. Ulusoy, Distributed k-core view materialization and maintenance for large dynamic graphs, IEEE Trans. Knowl. Data Eng. (99) (2014) 1, http://dx.doi.org/10.1109/TKDE.2013.2297918 (doi:10.1109/TKDE.2013.2297918.).

[6] hbase.apache.org.

[7] followerwonk.com.

[8] socialping.com.

[9] simplymeasured.com.

[10] A. Lancichinetti, S. Fortunato, Community detection algorithms: a comparative analysis, Phys. Rev. E. 80 (5) (2009) 056117.

[11] L. Danon, A. Daz-Guilera, J. Duch, A. Arenas, Comparing community structure identification, J. Stat. Mech: Theory Exp. 2005 (09) (2005) (P09008).

[12] C. Tantipathananandh, T. Berger-Wolf, D. Kempe, A framework for community identification in dynamic social networks, Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'07, ACM, New York, NY, USA 2007, pp. 717–726, http://dx.doi.org/10.1145/1281192.1281269 (http://dx.doi.org/10.1145/1281192.1281269 doi:10.1145/1281192.1281269. URL http://doi.acm.org/10.1145/1281192.1281269).

[13] S.B. Seidman, Net work structure and minimum degreehttp://www.sciencedirect.com/science/article/pii/037887338390028X Soc. Networks 5 (3) (1983) 269–287, http://dx.doi.org/10.1016/0378-8733(83)90028-X (doi:10.1016/0378-8733(83)90028-X. URL http://www.sciencedirect.com/science/article/pii/037887 338390028X.).

[14] X. Shi, M. Bonner, L. Adamic, A.C. Gilbert, The very small world of the well-connected, SIGWEB Newsl. (Winter) 4 (10) (2009) 1–4, http://dx.doi.org/10.1145/1457507.1457511 http://dx.doi.org/10/1145/1457507.1457511 doi:10/1145/1457507.1457511. URL http://doi.acm.org/10/1145/1457507.1457511.

[15] F.D. Malliaros, M. Vazirgiannis, To stay or not to stay: modeling engagement dynamics in social graphs, Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management, CIKM'13, ACM, New York, NY, USA 2013, pp. 469–478, http://dx.doi.org/10.1145/2505515.2505561 http://dx.doi.org/10.1145/2505515.2505561 doi:10.1145/2505515.2505561. URL http://doi.acm.org/10.1145/2505515.2505561.

[16] D. Garcia, P. Mavrodiev, F. Schweitzer, Social resilience in online communities: the autopsy of Friendster, Proceedings of the First ACM Conference on Online Social Networks, COSN'13, ACM, New York, NY, USA 2013, pp. 39–50, http://dx.doi.org/10.1145/2512938.2512946 (http://dx.doi.org/10.1145/2512938.2512946 doi:10.1145/2512938.2512946. URL http://doi.acm.org/10.1145/2512938.2512946).

[17] W. Cui, Y. Xiao, H. Wang, W. Wang, Local search of communities in large graphs, Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD'14, ACM, New York, NY, USA 2014, pp. 991–1002, http://dx.doi.org/10.1145/2588555.2612179 (http://dx.doi.org/10.1145/2588555.2612179 doi:10.1145/2588555.2612179. URL http://doi.acm.org/10.1145/2588555.2612179).

[18] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113, http://dx.doi.org/10.1145/1327452.1327492 http://dx.doi.org/10.1145/1327452.1327492 doi:10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[19] hadoop.apache.org.

[20] J. Cohen, Graph twiddling in a mapreduce world, Comput. Sci. Eng. 11 (4) (2009) 29–41, http://dx.doi.org/10.1109/MCSE.2009.120.

[21] J. Lin, M. Schatz, Design patterns for efficient graph algorithms in mapreduce, Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG'10, ACM, New York, NY, USA 2010, pp. 78–85, http://dx.doi.org/10.1145/1830252.1830263 (http://dx.doi.org/10.1145/1830252.1830263 doi:10.1145/1830252.1830263. URL http://doi.acm.org/10.1145/1830252.1830263).

[22] U. Kang, C.E. Tsourakakis, C. Faloutsos, Pegasus: mining peta-scale graphs, Knowl. Inf. Syst. 27 (2) (2011) 303–325, http://dx.doi.org/10.1007/s10115-010-0305-0 (http://dx.doi.org/10.1007/s10115-010-0305-0 doi:10.1007/s10115-010-0305-0. URL http://dx.doi.org/10.1007/s10115-010-0305-0).

[23] U. Kang, H. Tong, J. Sun, C.-Y. Lin, C. Faloutsos, Gbase: a scalable and general graph management system, Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'11, ACM, New York, NY, USA 2011, pp. 1091–1099, http://dx.doi.org/10.1145/2020408.2020580 (http://dx.doi.org/10.1145/2020408.2020580 doi:10.1145/2020408.2020580. URL http://doi.acm.org/10.1145/2020408.2020580).

[24] Y. Bu, B. Howe, M. Balazinska, M.D. Ernst, Haloop: efficient iterative data processing on large clustershttp://dl.acm.org/citation.cfm?id=1920841.1920881 Proc. VLDB Endow 3 (1-2) (2010) 285–296 (URL http://dl.acm.org/citation.cfm?id=1920841.1920881).

[25] J. Huang, D. J. Abadi, K. Ren, Scalable sparql querying of large rdf graphs, Proc. VLDB Endow. 4 (11).

[26] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, Proceedings of the 2010 International Conference on Management of Data, SIGMOD'10, ACM, New York, NY, USA 2010, pp. 135–146, http://dx.doi.org/10.1145/1807167.1807184 (http://dx.doi.org/10.1145/1807167.1807184 doi:10.1145/1807167.1807184. URL http://doi.acm.org/10.1145/1807167.1807184).

[27] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel, B. Bhattacharjee, Measurement and analysis of online social networks, Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07), San Diego, CA, 2007.

[28] snap.stanford.edu/.