Multi-resolution Social Network Community Identification and Maintenance on Big Data Platform

Hidayet Aksu
Department of Computer Engineering
Bilkent University, Ankara, Turkey
haksu@cs.bilkent.edu.tr

Mustafa Canim, Yuan-Chi Chang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{mustafa, yuanchi}@us.ibm.com

Ibrahim Korpeoglu, Özgür Ulusoy
Department of Computer Engineering
Bilkent University, Ankara, Turkey
{korpe,oulusoy}@cs.bilkent.edu.tr

***Abstract*—Community identification in social networks is of great interest and with dynamic changes to its graph representation and content, the incremental maintenance of 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. We then present 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 different topics in rich social network content simultaneously.**

***Keywords*-community identification; Big Data analytics; k-core; dynamic social networks; distributed computing**

I. Introduction

Community identification and evolution in a complex network has applications spanning multiple disciplines ranging from social science to physics. In recent years, the rise of very large, rich social networks re-ignited interests to the problem at the big data scale that poses computation challenges to early work with algorithm complexity greater than $O(n)$. In addition, many observed interactions with the community happen not just at one but multiple levels of intensity, which reflects in reality active to passive participants in a group.

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 as first projecting a subgraph by content topic of the social network interaction, such as 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 context. Examples include cliques, quasi-cliques [1], k-core, k-edge-connectivity [2], etc. Among these graph density measures, k-core stands out to be the least computationally expensive one that is still giving reasonable results. An $O(n)$ algorithm is known to compute k-core decomposition in a graph with n edges [3], 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 rich content is only limited by storage space and not by main memory. Furthermore, identified communities at multi-resolution are also persisted 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.

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

- We formulated *multi-resolution* community identification as a multi-k-core problem and developed a distributed multi-k-core *construction* algorithm that runs in parallel on big data platform.
- We further developed a distributed multi-k-core *maintenance* algorithm to keep the previously materialized multi-resolution community representation up to date with incremental updates.
- We presented a robust implementation of our algorithms on top of Apache HBase, a horizontally scaling distributed storage platform through its Coprocessor computing framework [4].

The rest of the paper is organized as follows. We first review prior work on community identification and k-core algorithms in Section II. Section III introduces the big data platform and programming framework. We define and introduce key k-core properties in Section IV. Section V describes our distributed multi-k-core construction algorithms in naïve implementation and pruning techniques. Section VI details our incremental maintenance algorithms for edge insertions and deletions. Experimental results are reported and discussed in Section VII. Finally, Section VIII concludes the paper and discusses future work.

II. 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 [5] [6] [7]. Challenges remain, however, to address both intensity and dynamicity of communities at 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 [8] for measuring group cohesion in social networks. Subsequently, Batagelj and Zaversnik (BZ) proposed a linear time algorithm to compute k-core [3]. The BZ algorithm first sorts the vertices in the increasing order of degrees and starts deleting the vertices with degree less than k. At each iteration, it needs to sort the vertices list to keep it ordered. Due to high number of random accesses to the graph, the algorithm can run efficiently only when the entire graph can fit into main memory of a single machine.

To tackle this problem, Cheng et al. in [9] proposed an external-memory solution which can spill into disk when the graph is too large to fit into main memory. The proposed algorithm, however, does not consider any distributed scenario where the graph resides on a large cluster of machines. A distributed k-core decomposition algorithm is introduced in [10] targeting a different computing platform than ours. They assume that each graph vertex can be located on a different computing node, similar to the nodes of a P2P network or a sensor network, which are good examples for distributed graph representations.

The k-core decomposition problem in a dynamic graph was first studied in [11], and an improved alternative was introduced by Li et al. in [12]. In [11], Miorandi et al. provide a statistical model for contacts among vertices and compute k-core decomposition as a tool to understand the influence of a spreader in diffusion of epidemics. k-core decomposition was recomputed at given time intervals using the BZ algorithm. The largest graph in those experiments, however, had only 300 vertices and 20K edges. We work with graphs of much bigger size. In [12], on the other hand, when a dynamic graph is updated, instead of recomputing k-core decomposition over the whole graph, the proposed algorithm tries to determine a minimal subgraph for which k-core decomposition might need to be recomputed. This approach, however, was reported for single server in-memory processing only, whose straightforward extension for distributed processing is far more costly.

Parallel graph algorithms have a long history with high performance computing. Most early studies, however, targeted static graphs [13], [14]. More recent work implemented graph algorithms on MapReduce framework [15] and its open source implementation Apache Hadoop [16]. However, the iterative nature of many graph algorithms soon prompted many to realize that static data is needlessly shuffled between MapReduce tasks [17], [18], [19]. Pregel [20] thus proposed a new parallel graph programming framework following the bulk synchronous parallel (BSP) model and message passing constructs. Two Apache incubator projects, Giraph [21] and Hama [22], inspired by Pregel, are looking to implement BSP on top of Hadoop infrastructure.

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 an open-source big-data processing framework. Therefore, before getting to the details of our proposed algorithms, we first would like to briefly introduce in the next section the big data programming framework where our distributed k-core algorithms are implemented.

III. Big graph data analytics on Apache HBase

We model interactions between pairs of *objects*, including structured metadata and rich, unstructured textual content, in a graph representation materialized as an adjacency list known as edge table. An edge table is stored and managed as an ordered collection of row records in an *HTable* by Apache HBase [4]. Since Apache HBase is relatively new to the research community, we first describe its architectural foundation briefly to lay the context of its latest feature known as *Coprocessor*, which our algorithms make use of for graph query processing.

A. HBase and Coprocessors

Apache HBase is a non-relational, distributed data management system modeled after Google's BigTable [23]. HBase is developed as a part of the Apache Hadoop project and runs on top of Hadoop Distributed File System (HDFS). Unlike conventional Hadoop whose saved data becomes read-only, HBase supports random, fast insert, update and delete (IUD) access.

Fig. 1(a) depicts a simplified diagram of HBase with several key components relevant to this paper. An HBase cluster consists of master servers, which maintain HBase metadata, and region servers, which perform data operations. An HBase table, or HTable, may grow large and get split into multiple HRegions to be distributed across region servers. HTable split operations are managed by HBase by default and can be controlled via API also. In the example of Fig. 1(a), HTable 1 has four regions managed by region servers 4, 7 and 10 respectively, while HTable 2 has three

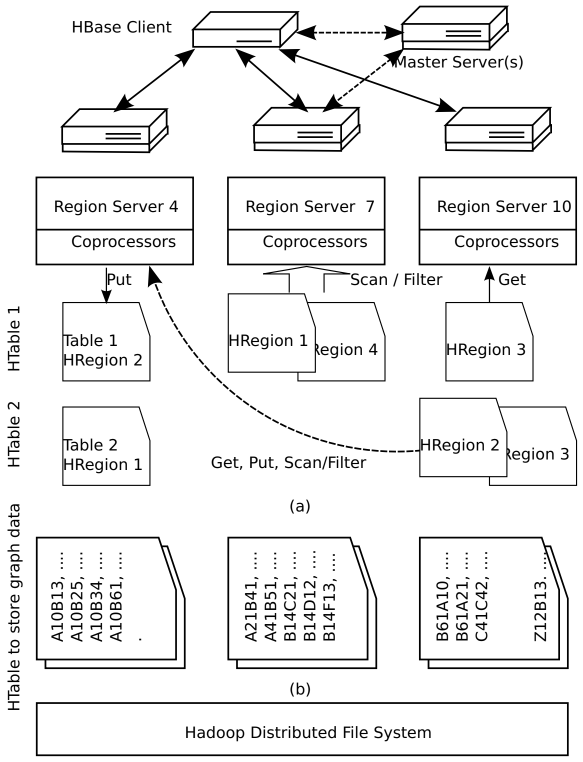


Figure 1. An HBase cluster consists of one or multiple master servers and region servers, each of which manages range partitioned regions of HBase tables. Coprocessors are user-deployed programs running in the region servers. They read and process data from local HRegion and can access remote data by remote calls to other region servers.

regions stored in region servers 4 and 10. An HBase client can directly communicate with region servers to read and write data. An HRegion is a single logical block of record data, in which row records are stored starting with a row key, followed by column families and their column values.

HBase's Coprocessor feature was introduced to selectively push computation to the server where user deployed code can operate on the data directly without communication overheads for performance benefit. The Endpoint Coprocessor (CP) is a user-deployed program, resembling database stored procedures, that runs natively in region servers. It can be invoked by an HBase client to execute at one or multiple target regions in parallel. Results from the remote executions can be returned directly to the client, or inserted into other HTables in HBase, as exemplified in our algorithms.

Fig. 1(a) depicts common deployment scenarios for Endpoint CP to access data. A CP may scan every row from the start to the end keys in the HRegion or it may impose filters to retrieve a subset in selected rows and/or selected columns. Note that the row keys are sorted alphanumerically in ascending order in the HRegion and the scan results preserve the order of sorted keys. In addition to reading local data, a CP may be implemented to behave like an HBase client. Through the Scan, Get, Put and Delete methods and their bulk processing variants, a CP can access other HTables hosted in the HBase cluster.

B. Graph Processing on HBase

We map the rich graph representation $G = \{V, E, M, C\}$ defined in Section IV to an HTable. We first format the vertex identifier $v \in V$ into a fixed length string $pad(v)$. Extra bytes are padded to make up for identifiers whose length is shorter than the fixed length format. The row key of a vertex v is its padded id $pad(v)$. The row key of an edge $e = \{s, t\} \in E$ is encoded as the concatenation of the fixed length formatted strings of the source vertex $pad(s)$, and the target vertex $pad(t)$. The encoded row key thus will also be a fixed length string $pad(s) + pad(t)$. This encoding convention guarantees a vertex's row always immediately proceeds the rows of its outbound edges in an HTable. Our graph algorithms exploit the strict ordering to join ranges of two tables. Fig. 1(b) includes a simple example of encoded graph table, whose partitioned HRegions are shown across three servers. In this table, a vertex is encoded as a string of three characters such as 'A10', 'B13', 'B25', 'A21', etc. A row key encoded like 'A10B13' represents a graph edge from vertex 'A10' to 'B13'.

k-core algorithms in Sections V and VI are implemented in several HBase Coprocessors to achieve maximal parallelism. Take degree computation as an example. Multiple instances of Coprocessors scan the graph data table's local partitions in parallel and then insert vertices' degrees into another HBase table. When a non-local edge is to be deleted, a Coprocessor instance issues the row delete message to the remote HBase region server, which deletes the edge. Our algorithms are optimized to minimize the message exchanges by achieving as much processing in the local partition as possible.

IV. Preliminaries

We define a rich graph representation G

$$G = \{V, E, M[V, E], C[V, E]\} \quad (1)$$

where V is the set of vertices, E is the set of edges, $M[V, E]$ and $C[V, E]$ are the structured metadata and unstructured content respectively. The paper simplified its description by including all vertices in the k-core computation while in practice, our system can be used to construct and maintain multiple k-core subgraphs on different metadata topics and context simultaneously.

The problem of k-core subgraph identification is formally defined as follows:

Definition 1: A subgraph $G_k = \{V_k, E_k\}$ induced from G where $V_k \subset V$, $E_k \subset E$, is a k-core if and only if $\forall v \in V_k$, its degree, $D_{G_k}(v)$ to the other vertices in G_k is greater than or equal to k. G_k is the maximum subgraph in G with this property.

Definition 2: The core number of a vertex, v, is the maximum k where $v \in V_k$ and $v \notin V_{k+1}$.

From the definitions, we can deduce the following lemmas, which are used extensively in our algorithms to prune the search space.

Lemma 1: $\forall v \in V_k$, $D_G(v) \geq k$

We further define $N_G^k(v)$ as the number of neighbors of the vertex v in G, whose degree is greater than or equal to k, i.e. $N_G^k(v) = |\{w|(w,v) \in E, D_G(w) \geq k\}|$. In later sections, we sometimes refer to $N_G^k(v)$ as Qualifying Neighbor Count (QNC) or shorthand as $qnc_k(v)$.

Lemma 2: $\forall v \in V_k$, $N_G^k(v) \geq k$

V. Distributed Multi k-core Construction

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

Table I
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
$k_{1...n}$	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
$(X) \leftarrow RC_f(R_i, S)$	Remote call to function f on region i takes parameter S and returns value X 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}

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

B. 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. $k_{1...3} = \{15, 20, 30\}$. In the degenerate case, $k_0 = 0, G_{k_0} = G$. The algorithm starts with computing k-core graph for k_1 and progressively moves up the index by reusing previously found k-core subgraph.

Algorithm 1 Base k-core construction- Client Side

```
Input:  Graph $G = (V, E)$,
        $k$: target core value
Output: $G_k$ the $k$-core graph
1: $G_k \leftarrow$ clone graph $G$
2: $doIterate \leftarrow true$
3: while $doIterate$ do
4:     for each region $i$ in $regions(G_k)$ do
5:         $anyEdgeDeleted_i \leftarrow RC_{Filter\ Out\ Edges}(R_i, G_k, k)$
6:     Wait RCs to complete
7:     $doIterate \leftarrow false$
8:     for each region $i$ in $regions(G_k)$ do
9:         $doIterate \leftarrow doIterate || anyEdgeDeleted_i$
10: return $G_k$
```

Algorithm 2 Base k-core construction- Node N_i Side

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

The algorithms are described in Algorithms 3 and 4 for the client and server side, respectively. It first computes 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 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
1: $G_k \leftarrow$ Base k-core construction( $G, k_1$)

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

6: $k_{n+1} \leftarrow infinity$
7: $next \leftarrow k_1$
8: for each $k_i$ in $k_{1...n}$ do
9:     while $next \geq k_i$ and $next < k_{i+1}$ do
10:        $next \leftarrow infinity$
11:        for each region $j$ in $regions(G_k)$ do
12:            $next_j \leftarrow RC_{Compute\ Core}(R_j, k_i, k_{i+1})$
13:        Wait RCs to complete
14:        for each region $j$ in $regions(G_k)$ do
15:            $next \leftarrow min(next, next_j)$
```

VI. Incremental multi k-core Maintenance

A. 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 without proof due to space limitation.

Algorithm 4 Multi k-core construction- Node N_i Side

```
1: Upon receiving $RC_{Compute\ Degrees}(G_k, T_L)$
2: for each vertex $u \in R_i(G_k)$ do
3:     compute $d_{G_k}(u)$ and put it into $T_L(C_{degree})$
4: return
5: Upon receiving $Compute\ Core(k_i, k_{i+1})$
6: $next \leftarrow infinity$
7: for each vertex $u \in R_i$ do
8:     if $d_{G_k}(u) \geq k_i$ and $d_{G_k}(u) < k_{i+1}$ then
9:         $core[\{u\}] \leftarrow k_i$
10:        for each vertex $v$ adjacent to $u$ do
11:            if $d_{G_k}(v) \geq k_{i+1}$ then
12:                $d_{G_k}(v) \leftarrow d_{G_k}(v) - 1$
13:                if $d_{G_k}(v) < k_{i+1}$ then
14:                    $next \leftarrow d_{G_k}(v)$
15:    if $d_{G_k}(u) \geq k_{i+1}$ then
16:        $next \leftarrow min(next, d_{G_k}(u))$
17: return $next$
```

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 core number by one. An edge inserted to the highest k-core G_{k_n} does not change the subgraph. However, an edge inserted to vertices in G_{k_i} may push some vertices to $G_{k_{i+1}}$ but not further up in the hierarchy. Figure 2 depicts this scenario, where a new edge and its update is always sandwiched between two rings of k-core graph. Bounding by the two rings implies that our maintenance algorithm can exploit this property to minimize traversal.

Algorithms 5, 6 and 7 present the algorithms in detail. There are several auxiliary counts 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 $i > 0$, 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 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 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

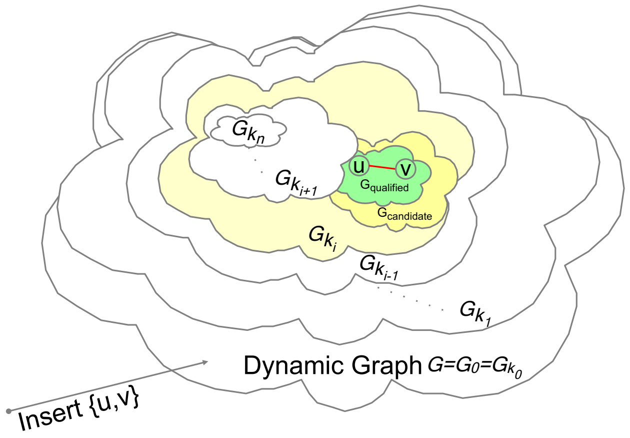


Figure 2. Upon an edge $\{u, v,\}$ insertion where u or v resides in k_i-core G_{k_i}, first tightly bounded $G_{candidate}$ graph is discovered exploiting maintained auxiliary information, then it is processed to compute $G_{qualified}$ subgraph qualifying for k_{i+1}-core.

candidate edges for $G_{k_{i+1}}$. *Partial KCore* in Algorithm 7 then processes $G_{candidate}$ 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
1: Auxiliary Update($G$, $u$, $v$, $k_{1...n}$)          ▷ Update the auxiliary values
2: $i = min\{i | u \in G_{k_i}\ or\ v \in G_{k_i}\}$
3: if $i > 0$ then                                        ▷ both vertices are in core graph
4:     insert edge $\{u, v\}$ and $\{v, u\}$ into $G_{k_i}$
5:     Auxiliary Update($G_k$, $u$, $v$, $k_{1...n}$)
6: if $i == n$ then
7:     return
8: if $d(u) < k_{i+1}$ or $d(v) < k_{i+1}$ then
9:     return
10: $G_{candidate} \leftarrow \emptyset$
11: if $qnc_{k_{i+1}}(u) \geq k_{i+1}$ or $qnc_{k_{i+1}}(v) \geq k_{i+1}$ then
12:     $G_{candidate} \leftarrow$Find Candidate Graph($G_{k_i}$, $G_{k_{i+1}}$,$C$,$k_{i+1}$, $u$)
13: if $G_{candidate} \neq \emptyset$ then
14:     $G_{qualified} \leftarrow$ Partial KCore ($G_{candidate}$, $k_{i+1}$)
15:     $G_{k_{i+1}} \leftarrow G_{k_{i+1}} \cup G_{qualified}$
```

B. 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}.

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

1: $Q \leftarrow new\ queue$
2: $Q.enqueue(u)$
3: $mark(u)$
4: while $Q \neq \emptyset$ do
5:     $v \leftarrow Q.dequeue()$
6:     if $v$ is not local then remote request for edges of $v$
7:     for each vertex $w$ adjacent to $v$ in $G_{k_i}$ do
8:         if $\{v, w\} \notin C$ then
9:             if $d(w) \geq k_j$ and $qnc_{k_j}(w) \geq k_j$ then
10:                $C \leftarrow C \cup \{v, w\}$
11:                if $w \notin G_{k_{i+1}}$ then
12:                    $C \leftarrow C \cup \{w, v\}$
13:                    if $w\ is\ not\ marked$ then
14:                        $Q.enqueue(w)$
15:                        $mark(w)$
16: 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

1: $changed \leftarrow true$
2: while $changed$ do
3:     $changed \leftarrow false$
4:     for each $\{u, v\} \in C$ do
5:         if $d_C(u) < k_j$ then
6:             delete $\{u, v\}$ and $\{v, u\}$ from $C$
7:             $changed \leftarrow true$
8: return $C$
```

The intuition behind this theorem is that an edge deletion can at most decrease 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 edge insertion case. Upon receiving an edge deletion, it first finds out in which k-core graph this edges resides, say G_{k_i}. If it does not reside in any k-core, then the algorithm terminates. Otherwise, *Update Coreness Cascaded* algorithm described in Algorithm 9 starts with the vertex with $d_{G_{k_i}}$ less than k_i, 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.

VII. 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 it in dynamic graphs where edges are inserted and deleted.

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

1: Auxiliary Update($G$, $u$, $v$, $k_{1...n}$)        ▷ Update the auxiliary values
2: $i = min\{i | u \in G_{k_i}\ or\ v \in G_{k_i}\}$
3: if $i == 0$ then        ▷ when edge is not in $G_k$, no change occurs
4:     return
5: delete $\{u, v\}$ and $\{v, u\}$ from $G_{k_i}$
6: Auxiliary Update($G_k$, $u$, $v$, $k_{1...n}$)
7: if $d_{G_{k_i}}(u) \geq k_i$ and $d_{G_{k_i}}(v) \geq k_i$ then
8:     return
9: if $d_{G_{k_i}}(u) < k_i$ then
10:    Update Coreness Cascaded($G_k$,$i$,$u$)
11: if $d_{G_{k_i}}(v) < k_i$ then
12:    Update Coreness Cascaded($G_k$,$i$,$v$)
```

Algorithm 9 Update Coreness Cascaded

```
Input:  $G_k$: the multi $k$-core graph,
        $k_{1...n}$: maintained core values,
        $u$: start vertex
Output: the updated $G_k$

1: $Q \leftarrow new\ queue$
2: $Q.enqueue(u)$
3: $mark(u)$
4: while $Q \neq \emptyset$ do
5:     $v \leftarrow Q.dequeue()$
6:     $core[v] \leftarrow k_{i-1}$        ▷ decrease vertex core value.
7:     for each vertex $w$ adjacent to $v$ in $G_{k_i}$ do
8:         if $k_{i-1} == 0$ then
9:             delete $\{v, w\}$ and $\{w, v\}$ from $G_{k_i}$
10:        if $d_{G_{k_i}}(w) < k_i$ then
11:            if $w\ is\ not\ marked$ then
12:                $Q.enqueue(w)$
13:                $mark(w)$
```

A. System Setup and Datasets

Graph data is stored in HBase and the algorithms are implemented as HBase Coprocessors where distributed parallelism is applicable. Table II 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 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 maximum 16 GB Java heap space and Hadoop with 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 Milove et al. [24] and the Stanford Network Analysis Project [25]. We appreciate their generous offer to make the data openly available for research. For details,

Table II
MAPPING OF GRAPH NOTATIONS IN TABLE I 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
$(X) \leftarrow RC_f(R_i, S)$	Coprocessor function f on region i takes parameter S and returns value X 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 III
KEY CHARACTERISTICS OF DATASETS IN THE EXPERIMENTS

Name	Vertex Count	Bidirectional Edge Count	Ref
Orkut	3.1 M	234 M	[24]
LiveJournal	5.2 M	144 M	[24]
Flickr	1.8 M	44 M	[24]
Patents	3.8 M	33 M	[25]
Skitter	1.7 M	22.2 M	[25]
BerkStan	685 K	13.2 M	[25]
YouTube	1.1 M	9.8 M	[24]
WikiTalk	2.4 M	9.3 M	[25]
Dblp	317 K	2.10 M	[25]

please see the references and we only briefly recap the key characteristics of the data in Table III.

B. 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 percent of the vertices in that dataset have a degree of at least k. The higher the k value, the stronger or tightly knit the communities are. Conversely, the lower the k value, the weaker or loosely connected the communities are. Table IV lists the chosen k values. We first run *Base k-core construction* algorithm to measure the baseline k-core construction time for each dataset and k value. Then we run *Multi k-core construction* algorithm, which is described in Algorithms 3 and 4, for each dataset with all chosen k values at once to measure k-core construction for multiple k values. Figure 3 shows the construction times for both algorithms. Speedup achieved by *Multi k-core construction* algorithm is upper bounded by the number of distinct values which is 3 in this case. We observe that, for larger datasets the algorithm achieved higher speedup due to the redundant computation saved.

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

1) In *Insertion* scenario, 1000 randomly chosen edges are inserted into the graph. Those random edges are selected from the graph and deleted before materialized k-core graph is constructed.
2) In *Deletion* scenario, 1000 randomly chosen edges are deleted from the graph.
3) In *Mix* scenario, Insertion and Deletion scenarios are run simultaneously where one insertion is followed by one deletion.

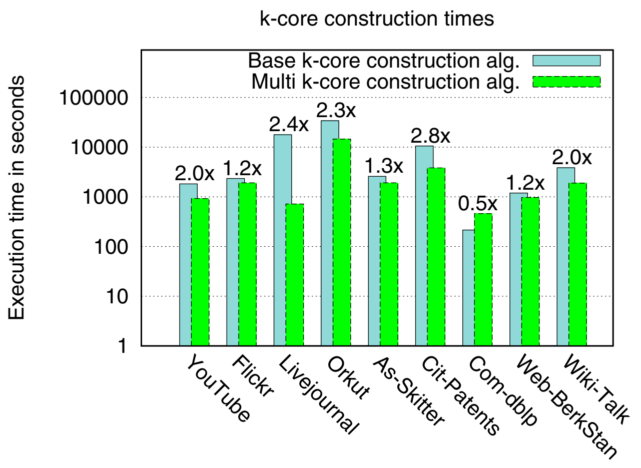


Figure 3. k-core construction times for Base and Multi k-core construction algorithms are shown for each dataset with three chosen k values. Relative speedup achievement of Multi algorithm over Base algorithm is provided above each bar.

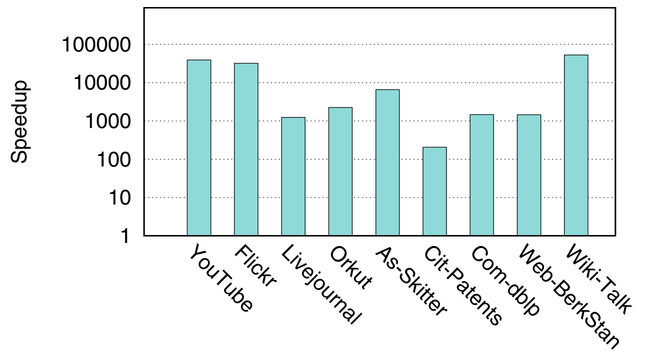
We repeated these three scenarios with each dataset and measured their execution times. Fig. 4 plots the speedup through our incremental maintenance algorithms over re-computing k-core from scratch, for 9 different datasets. The y-axis shows the speedup in log-scale. For insertion, deletion, mix scenarios and each dataset, the figure gives the speedup of incremental update approach with respect to from-scratch construction using the multi k-core construction algorithm. As the figure shows, three to five orders of magnitude speedup can be expected for edge insertion workload. Similar speedup factors are also observed for mixed edge insertions and deletions with one to one ratio. Higher speedup, more then five orders of magnitude was achieved for edge deletion only workload. Note that storing a new edge in HBase without maintenance algorithm took 3 ms on the average.

VIII. CONCLUSIONS

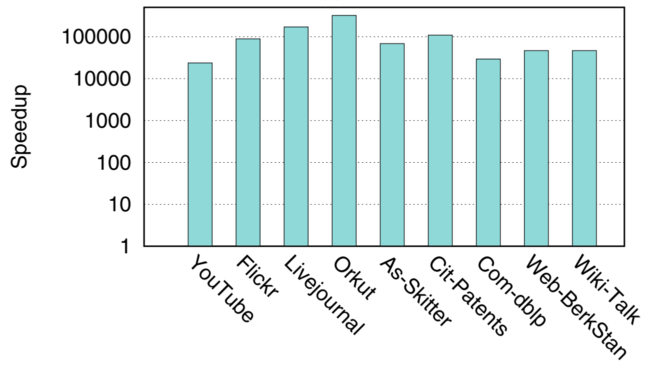
To the best of our knowledge, this paper is the first to propose a horizontally scaling solution on the big data platform for multi-resolution social network community identification and maintenance. By using k-core as the measure of community intensity, we proposed multi-k-core construction and incremental maintenance algorithms and

Table IV
k VALUES USED IN THE EXPERIMENTS AND THE RATIO OF VERTICES WITH DEGREE AT LEAST k IN THE CORRESPONDING GRAPHS

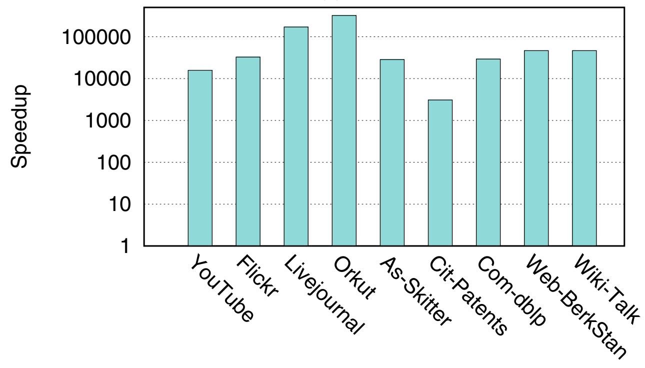
Datase - k values	4%	8%	16%
Orkut	263	183	123
LiveJournal	80	50	28
Flickr	65	24	9
Patents	28	21	15
Skitter	42	26	15
BerkStan	57	38	24
WikiTalk	5	3	2
YouTube	18	10	5
Dblp	25	16	10



(a)



(b)



(c)

Figure 4. k-core maintenance speedups under a) Insertion b) Deletion c) Mix workloads.

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.

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 fully took advantage of 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.

References

[1] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Out-of-core coherent closed quasi-clique mining from large dense graph databases," *ACM Trans. Database Syst.*, vol. 32, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1242524.1242530

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

[3] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[4] hbase.apache.org.

[5] A. Lancichinetti and S. Fortunato, "Community detection algorithms: a comparative analysis," *Physical Review E*, vol. 80, no. 5, p. 056117, 2009.

[6] L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas, "Comparing community structure identification," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2005, no. 09, p. P09008, 2005.

[7] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe, "A framework for community identification in dynamic social networks," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 717–726. [Online]. Available: http://doi.acm.org/10.1145/1281192.1281269

[8] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269 – 287, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/037887338390028X

[9] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *ICDE*, 2011, pp. 51–62.

[10] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-core decomposition," in *PODC*, 2011, pp. 207–208.

[11] D. Miorandi and F. De Pellegrini, "K-shell decomposition for dynamic complex networks," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2010 Proceedings of the 8th International Symposium on*. IEEE, 2010, pp. 488–496.

[12] R. Li and J. Yu, "Efficient core maintenance in large dynamic graphs," *arXiv preprint arXiv:1207.4567*, 2012.

[13] J. Greiner, "A comparison of parallel algorithms for connected components," in *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '94. New York, NY, USA: ACM, 1994, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/181014.181021

[14] M. J. Quinn and N. Deo, "Parallel graph algorithms," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 319–348, Sep. 1984. [Online]. Available: http://doi.acm.org/10.1145/2514.2515

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

[16] hadoop.apache.org.

[17] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920881

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

[19] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proc. VLDB Endow.*, vol. 4, no. 11, Sep. 2011.

[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[21] incubator.apache.org/giraph.

[22] incubator.apache.org/hama.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[24] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[25] snap.stanford.edu/.