Distributed k-Core View Materialization and Maintenance for Large Dynamic Graphs

Hidayet Aksu, *Member, IEEE*, Mustafa Canim, Yuan-Chi Chang, *Senior Member, IEEE*, Ibrahim Korpeoglu, *Senior Member, IEEE*, and Özgür Ulusoy, *Member, IEEE*

Abstract—In graph theory, k-core is a key metric used to identify subgraphs of high cohesion, also known as the 'dense' regions of a graph. As the real world graphs such as social network graphs grow in size, the contents get richer and the topologies change dynamically, we are challenged not only to materialize k-core subgraphs for one time but also to maintain them in order to keep up with continuous updates. Adding to the challenge is that real world data sets are outgrowing the capacity of a single server and its main memory. These challenges inspired us to propose a new set of distributed algorithms for k-core view construction and maintenance on a horizontally scaling storage and computing platform. Our algorithms execute against the partitioned graph data in parallel and take advantage of k-core properties to aggressively prune unnecessary computation. Experimental evaluation results demonstrated orders of magnitude speedup and advantages of maintaining k-core incrementally and in batch windows over complete reconstruction. Our algorithms thus enable practitioners to create and maintain many k-core views on different topics in rich social network content simultaneously.

Index Terms—k-core, graph theory, distributed computing, dynamic social networks

1 Introduction

An ACM computing surveys article in 1984 began its introduction in the following words: *Graph theory is widely applied to problems in science and engineering. Practical graph problems often require large amounts of computer time* [1]. In today's graph applications, not only the graph size is larger, but also the data characterizing vertices and edges are richer and increasingly more dynamic, enabling new hybrid content and graph analysis. One key challenge to understanding large graph data is the identification of subgraphs of high cohesion, also known as "dense" regions.

This paper proposes scalable, distributed algorithms for k-core graph construction as well as its incremental and batch maintenance as dynamic changes are made to the graph. One critical aspect to understand large graph data is through the identification of "dense" areas in the graph which represent higher inter-vertex connectivity (or interactions in the case of a social network). In the literature, there is a growing list of subgraph density measures that may be suited in different application context. Examples include cliques, quasi-cliques [2], k-core, k-edge-connectivity [3], 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 [4], where other measures have complexity growing super-linearly or NP-hard.

For practical considerations, our focus is to identify and maintain k-core with fixed, large k values in particular. In contrast, a full k-core decomposition assigns a core number to every vertex in the graph. To understand "dense" areas in a graph, vertices with low core numbers do not contribute much and thus the computational expense of a full decomposition is not justified. Fig. 1 illustrates the degree distribution of nine published graph data sets, where partly due to their nature of power-law distribution, a significant percentage of graph vertices have low degrees and thus low core numbers. In addition to reduced cost in constructing k-core, it is also computationally less expensive to maintain it, compared to maintaining core numbers for large numbers of low degree vertices.

Real world graph data is not just about relationship topology but also the associated metadata attributes and possibly unstructured content. For example, a call graph contains not just the phone numbers, but also the duration, time of the day, geolocation, etc. In many practical applications graph data is stored in a distributed data store via sharded SQL or NoSQL technologies. This improves reliability, availability and performance. The data store continuously receives updates and may have other non-graph analytics executed along with graph analytics such as k-core. In addition, there are likely many projected graphs based on the metadata or content topic with snapshot or temporal evolution. There are various studies in the literature dealing with k-core construction in the presence of metadata. Giatsidis et al. in [5], [6] use co-authorship as edge weight in the graph. In [7], Wei and Ram consider organization of social bookmarking tags using k-core with tag weight as a metric. Chun et al. in [8] consider friends and their bidirectional relations on a graph. The paper

- *H. Aksu is with the Department of Computer Engineering, Bilkent University, Ankara 06836, Turkey. E-mail: haksu@cs.bilkent.edu.tr.*
- *M. Canim and Y.-C. Chang are with the IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Yorktown Heights, NY 10598. E-mail: {mustafa, yuanchi}@us.ibm.com.*
- *I. Korpeoglu and Ö. Ulusoy are with the Department of Computer Engineering, Bilkent University, Engineering Building, Ankara 06800, Turkey. E-mail: {korpe, oulusoy}@cs.bilkent.edu.tr.*

Manuscript received 22 Feb. 2013; revised 23 Nov. 2013; accepted 4 Dec. 2013. Date of publication 8 Jan. 2014; date of current version 29 Aug. 2014. Recommended for acceptance by E. Pitoura.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2013.2297918

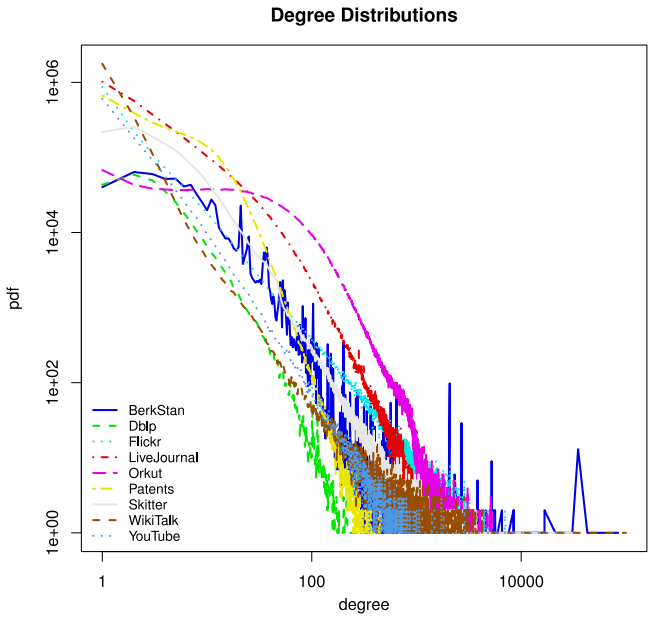

Fig. 1. Degree distribution of vertices in nine social network data sets on the log scale.

compares k-core of friendships and k-core of bidirectional activity relationships. As opposed to these studies, our proposed solution addresses the needs to maintain many k-cores without keeping them all on a single server or in memory.[1]

Our contributions in this paper can be summarized as follows:

- We developed and accelerated a distributed k-core construction algorithm through aggressive pruning of the graph that will not be in the final k-core subgraph.
- We developed a new k-core maintenance algorithm to keep the previously materialized subgraph up to date with incremental changes to the underlying graph. We developed pruning techniques to limit the scope of k-core updates in the face of edge insertions and deletions.
- We further improved the maintenance algorithm with batch window updates for practical applications. Batch update maintenance allows more expensive graph traversal steps to be aggregated for additional 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 [9]. Our system built on HBase stores graph data, including metadata and unstructured content, in the HBase tables. Our scalable algorithms read and write to these tables in a distributed, parallel manner for persistence and robustness. Since distributed graph processing associates with a certain overhead, we designed our algorithms carefully to minimize such overhead.

1. The readers are encouraged to read Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2013.2297918, for the detailed motivation behind our algorithms.

We recognize that depending on the specific k value, it is plausible that some may be more conveniently kept centralized, independent of other applications. In practice, however the k-core subgraph, once identified, can also be added as additional metadata to the vertices and edges of the distributed raw data. Such metadata is useful in conjunction with other analytics to weigh the k-core labeled vertices and edges differently or cross correlate k-core subgraphs from multiple topics. Our proposed implementation can accommodate both centralized and distributed maintenance by taking advantage of the flexible scale-out data store.

The rest of the paper is structured as follows. We first review the large body of prior work on k-core and parallel graph processing in Section 2. We next introduce our distributed graph computing framework implemented on top of Apache HBase and its coprocessor feature to set the context of algorithm presentation in Section 3. We formally define and introduce key k-core properties in Section 4. Section 5 describes our distributed k-core construction algorithms in naïve implementation and pruning techniques. Section 6 details our incremental maintenance algorithms for edge insertions and deletions. Section 7 makes further improvement for maintenance over batch window updates. Experimental results are reported and discussed in Section 8. Finally, Section 9 concludes the paper and discusses future work.

2 Related Work

k-core decomposition on a single machine. Extracting dense regions in large graphs has been a critical problem in many applications. Among the solutions proposed, k-core decomposition became a very popular one and many studies have been conducted on k-core decomposition on graphs efficiently [10], [11], [12], [13], [14]. k-core decomposition has been used in many applications such as network visualization [15], [16], [17], [18], [19], [20], internet topology analysis [21], [22], [23], social networks [24], [25], and biological networks [26], [27], [28]. The notion of k-core is first introduced in [15] for measuring group cohesion in social networks. The approach introduced generates subgraphs iteratively that has higher cohesion. This approach has been very popular for characterizing and comparing network structures. Although the concept of k-core is first introduced in [15] a well known algorithm for computing k-core decomposition is first proposed by Batagelj and Zaversnik (BZ) [4]. 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 the vertices list ordered. Due to high random accesses to the graph, the algorithm can run efficiently if the entire graph fits in main memory of a single machine. To tackle this problem Cheng et al. in [29] 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 large cluster of machines.

Distributed k-core decomposition: A distributed k-core decomposition algorithm is introduced in [30] targeting a different computing platform. In this paper it is assumed

that each graph vertex is located on a different computing node similar to P2P networks or sensor networks. In our case, however, we horizontally partition a large graph and keep each large partition on a different computing node. Each of these nodes may store millions or billions of edges. Therefore we never make an assumption that each graph partition will fit into main memories of computing nodes and we keep them on disks. As opposed to our algorithms, in [30], it is assumed that everything is held in the memories in computing nodes. The third important point is that in [30], only the number of iterations required to compute k-core decomposition is reported but not real execution times. In this paper however, we provide real execution times for our experiments conducted on large real graphs.

None of the papers mentioned so far targets k-core maintenance in dynamic graphs where the data does not fit into main memories of computing nodes.

k-core decomposition in dynamic graphs: k-core decomposition in dynamic graphs was first studied in [31] and an improved alternative was introduced by Li and Yu in [32]. In [31], Miorandi and De Pellegrini provide a statistical model for contacts among vertices and compute k-core decomposition as a tool to understand the spreaders' influence in diffusion of epidemics. k-core decomposition was recomputed at given time intervals using the BZ algorithm. The largest graph in those experiments had 300 vertices and 20K edges. This approach is not feasible for large dynamic networks where k-core recomputation likely will take a long time. In [32], Li and Yu addressed the problem of efficiently computing the k-core decomposition in dynamic graphs. The main idea is that when a dynamic graph is updated, instead of recomputing k-core decomposition over the whole graph, their algorithm tries to determine a minimal subgraph for which k-core decomposition might get changed. The proposed coloring based algorithm keeps track of core number for each vertex and upon an update provides the subgraph for which k-core decomposition needs to be updated. This approach was reported for single server in-memory processing only and a straightforward extension of the algorithm for distributed processing is far more costly. Also, in this paper we propose algorithms for batch window updates which could provide greater performance improvement compared to performing updates step by step. To our knowledge, our work is the first one proposing algorithms for performing batch window updates for the maintenance of k-core subgraphs.

Other parallel graph algorithms: Early studies in parallel graph algorithms targeted static graphs [1], [33]. In the recent years studies in this field gained momentum again due to the growing popularity of social media tools. By formulating common graph algorithms as iterations of matrix-vector multiplications, coupled with compression, [34] and [35] demonstrated significant speedup and storage savings, although such 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 [36], [37]. In Pregel [38], 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. Recently, the Trinity graph engine was introduced for distributed in-memory graph processing [39]. Its capability is more suited for one-time batch processing while lacking the support for incremental updates and rich metadata.

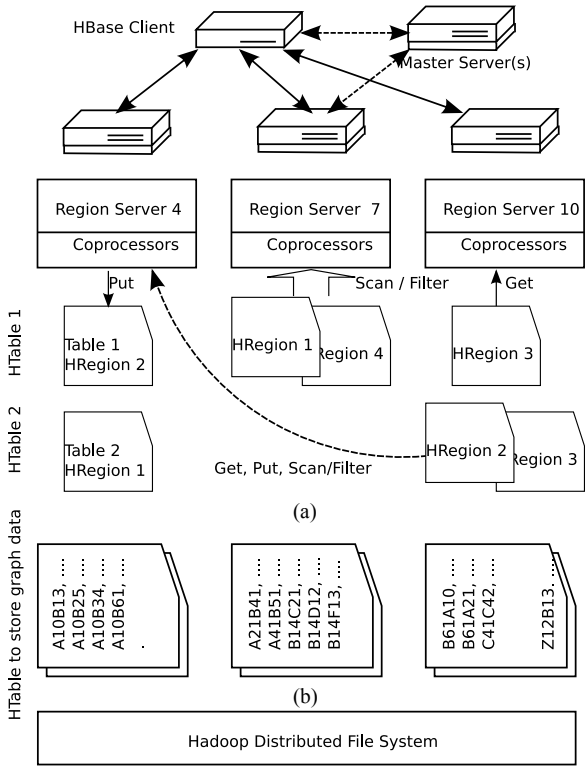

Fig. 2. An HBase cluster consists of one or multiple master servers and region servers, each of which manages range partitioned regions of HBase tables. Coprocessors are user-deployed programs running in the region servers. They read and process data from local HRegion and can access remote data by remote calls to other region servers.

3 Algorithm Implementation on Apache HBase

We model interactions between pairs of *objects*, including structured metadata and rich, unstructured textual content, in the graph representation materialized as 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 [9]. Since Apache HBase is relatively new to the research community, we included descriptions about its architecture and the *Coprocessor* computing framework in Fig. 2 and Appendix C, available in the online supplemental material.

Apache HBase is a non-relational, distributed data management system modeled after Google's BigTable [40]. Written in Java, 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 at the granularity of row records, mimicking transactional databases. Prominent HBase partitioners include Facebook [41] and many others [42].

Fig. 2a depicts a simplified architectural 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. In the example of Fig. 2a, HTable 1 has four regions managed by region servers 4, 7 and 10 respectively while HTable 2 has three regions stored in region servers 4 and 10. After consulting with the master server, an HBase client can directly communicate with region servers to read and write data.

An HRegion is a single logical block of record data, which is physically materialized into multiple HFiles stored in HDFS for availability. Within each HRegion, row records are organized with their keys sorted in alphanumeric order. This sorted order is always preserved after new row insertions. Each HRegion thus has a start (the lowest) key and an end (the highest) key. Our algorithms take advantage of range partitioning to reduce the amount of data shuffling.

We map the rich graph representation $G = \{V, E, M, C\}$ defined in Section 4 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 padding aims to preserve the natural representation of the id's for other applications and avoids id remapping.

The row key of a vertex v is its padded id $pad(v)$. The row key of an edge $e = \{u, v\} \in E$ is encoded as the concatenation of the fixed length formatted strings of the source vertex $pad(u)$, and the target vertex $pad(v)$. The encoded row key thus will also be a fixed length string $pad(u) + pad(v)$. 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. Respective metadata $M[V, E]$ and content $C[V, E]$ are stored in the columns. Fig. 2b 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', with fanout of four, to another vertex 'B13'. This layout retains minimal clustering, only a vertex and its immediate outbound edges are stored consecutively. Our current work does not attempt to partition or cluster the graph data, although we can adopt partitioning techniques such as [43]. Note that, we use the terms partition and region interchangeably.

k-core algorithms in the paper are implemented as 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 for subsequent computing. When an edge needs to be deleted, a coprocessor instance issues the row delete message to a possibly remote HBase region server, which holds the current row. Our algorithms are optimized to minimize the messaging exchanges by achieving as much processing in the local partition as possible.

Note that, k-core view maintenance algorithms depend on raw graph and possibly metadata, hence keeping a small k-core view result in a centralized location would still require working with the raw graph on each update. On the other hand, if the view is stored as additional metadata as part of the raw graph data, the improved affinity helps not only incremental maintenance but also the consumption by other analytics that weights the input of k-core. A concrete example of a distributed social graph with metadata is provided at Appendix B, available in the online supplemental material.

4 Preliminaries

We define a rich graph representation G

$$G = \{V, E, M[V, E], C[V, E]\}, \tag{1}$$

where V is the set of vertices, E is the set of edges, $M[V, E]$ is the structured metadata associated with a vertex or an edge, and $C[V, E]$ is the unstructured context respectively. We simplified the description in this paper by including all vertices in the k-core computation while in practice, our system is used to construct and maintain multiple k-core subgraphs projected over different metadata 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. $\forall v \in V_k$, $d_G(v) \geq k$.

Proof. By definition, $d_{G_k}(v) \geq k$. Since $v \in G_k \subset G$, $d_G(v) \geq d_{G_k}(v)$. Thus, $d_G(v) \geq k$. □

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. $\forall v \in V_k$, $N_G^k(v) \geq k$.

Proof. By Lemma 1, we know that every vertex in V_k has degree greater than or equal to k. Since by definition, a vertex in V_k has at least k neighbors, we thus deduct that it must have at least k neighbors whose degree is greater than or equal to k, i.e., $N_G^k(v) \geq k$. □

See Appendix D.1.1, available in the online supplemental material, for an illustration on the relationship between a vertex's core number, its degree in the entire graph.

5 Distributed k-Core Construction

In this section, we first describe a naïve distributed algorithm that constructs a k-core subgraph by progressively removing edges in parallel with the help of remote calls running on server nodes. As indicated earlier, the given graph data is partitioned to server nodes, hence the computed k-core subgraph will also be partitioned. Next, we describe how to

TABLE 1
Notations Used in Algorithms

G	Dynamic graph partitioned into regions stored in multiple nodes
G_k	k-core materialized view graph of G
$partitions(G_A)$	Partition list of graph G_A
P_i	i'th partition of graph stored on and processed by node i
N_i	i'th node storing partition i
$(X) \leftarrow RC_f(P_i, S)$	Remote call to function f on partition i takes parameter S and returns value X to client
$\{u, v\}$	Graph edge from vertex u to vertex v
$P_i(G_A)$	Partition of graph G_A processed by node N_i
$T_A(C_X, C_Y)$	Lookup table A with columns C_X and C_Y
$d(u), d_{G_k}(u)$	Degree of vertex u in G and G_k
$qnc_k(u)$	Qualified Neighbor Count for vertex u in G with respect to core value k

improve the base algorithm with an early pruning technique. The proposed improvement reduces the message traffic between the computing nodes dramatically and yields significant speedup as we demonstrate with the experiments.

Our k-core construction algorithms alter the BZ algorithm [4] by leaping to the fixed k value directly in a distributed computing environment where graph data is partitioned and remote references are expensive. As described in Section 3, edges are sorted and clustered by their source vertex ids. The degree of a vertex thus can be computed locally by node. Nodes request edges stored on remote servers by sending messages. Table 1 summarizes notations used in our pseudocodes.

Algorithm 1 Base distributed k-core construction algorithm (Base k-Core) - 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 partition P_i in $partitions(G_k)$ **do**
5: $anyEdgeDeleted_i \leftarrow RC_{Filter\ Out\ Edges}(P_i, G_k, k)$
6: $doIterate \leftarrow$ if any RC return edge delete
7: **return** G_k

5.1 Base Algorithm

The base algorithm, as described in Algorithms 1 and 2, runs at server nodes and the client coordinates the execution of these remote servers. Each node scans its own partition and deletes those edges incident to the vertices with degrees lower than k. Unlike the BZ algorithm where the vertices can be immediately sorted by their degrees in memory, our distributed algorithm relies on iterations until all remote calls run out of work. The remaining graph is the k-core subgraph with all its vertices having core number no less than k.

Algorithm 2 Base distributed k-core construction algorithm - Node N_i Side

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

For high k values, one would expect to have fewer vertices qualifying for k-core subgraph. Thus the algorithm described above would incur a large number of edge deletions in its first iteration. This can be improved with an early pruning technique described next.

5.2 Early Pruning

The insight leads us to have nodes check for a given edge $\{u, v\}$, if $d_G(u)$ and $d_G(v)$ are both greater than or equal to k. In addition, the degrees of neighboring vertices must be greater than or equal to k, i.e., $N_G^k(u) \geq k$ and $N_G^k(v) \geq k$. A pruned edge list is populated by those edges passing this minimum requirement. The pruned graph is the same as the remaining graph after the first iteration of the base algorithm. For a large k, if the iteration reduces the graph size by 90 percent, applying the base algorithm will delete 90 percent of the edges, while applying the early pruning technique will insert 10 percent of the edges. In practice, we observed significant speedup due to the dramatic messaging and I/O reduction.

Algorithm 3 Distributed k-core construction algorithm with early pruning- Client Side

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

1: Create new table $T_L(C_{degree}, C_{qnc})$

2: **for** each partition P_i in $partitions(G)$ **do**
3: $RC_{Compute\ Degrees}(P_i, G, T_L, k)$

4: **for** each partition P_i in $partitions(G)$ **do**
5: $RC_{Compute\ Qnc}(P_i, G, T_L, k)$

6: Create new graph G_k
7: **for** each partition P_i in $partitions(G)$ **do**
8: $RC_{Filtered\ Export}(P_i, G, T_L, G_k, k)$

9: $G_k \leftarrow$ **Base** k**-Core**(G_k, k)
10: **return** G_k

The algorithm is described in Algorithms 3 and 4 for client and node part, respectively. It simply first computes degrees, then computes qnc values and then filters out qualified edges into a new table, and finally calls the basic algorithm over this new table.

Algorithm 4 Distributed k-core construction algorithm with early pruning- Node N_i Side

1: Upon receiving $RC_{Compute\ Degrees}(G, T_L, k)$
2: **for** each vertex $u \in P_i(G)$ **do**
3: compute the $d(u)$
4: **if** $d(u) \geq k$ **then**
5: put $d(u)$ into $T_L(C_{degree})$
6: **return**
7: Upon receiving $RC_{Compute\ Qnc}(G, T_L, k)$
8: **for** each vertex $u \in P_i(G)$ **do**
9: **if** $qnc_k(u) \geq k$ **then**
10: put $qnc_k(u)$ into $T_L(C_{qnc})$
11: **return**
12: Upon receiving $RC_{Filtered\ Export}(G, T_L, G_k, k)$
13: **for** each edge $\{u, v\} \in P_i(G)$ **do**
14: **if** $qnc_k(u) \geq k$ and $qnc_k(v) \geq k$ **then**
15: put $\{u, v\}$ into G_k
16: **return**

6 Incremental k-Core Maintenance

We formulate incremental k-core maintenance as a series of edge insertions and deletions to the graph. In case a vertex is deleted, the action to delete its edges is serialized and maintained as if the edges were deleted one at a time. We first describe edge insertion and then edge deletion logic.

6.1 Inserting an Edge

With graph $G = \{V, E\}$ and its materialized k-core subgraph $G_k = \{V_k, E_k\}$, we give the following edge insertion theorem.

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

- *If both $u, v \in V_k$, then G_k does not change.*
- *If u or v or both $\notin V_k$, then the subgraph consisting of vertices in $\{w | w \in V, d_G(w) \geq k, N_G^k(w) \geq k\}$, where every vertex is reachable from u or v, may need to be updated to include additional vertices into G_k.*

To prove the theorem, we first prove the following lemma.

Lemma 3. *If the vertex q is included in the k-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 every vertex also has a core number greater than or equal to k after the insertion of the new edge.*

Proof. Since q was not in V_k and is in $\tilde{V_k}$ after the edge insertion, its core number must have been increased from $k-1$ to k. The increase of q's core number is due to one or more of its neighboring vertices whose core numbers increased to k as well. The same logic applies to those neighbors and leads to 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.

Algorithm 5 Edge Insertion/Deletion- Client Side

Input: Graph $G = (V, E)$,
G_k: the k-core graph,
$\{u, v\}$: updated edge,
$Request$: Insertion or Deletion edge,
k: maintained core value
Output: the updated k-core graph

1: $P_i \leftarrow get\ partition\ of\ u\ in\ G$
2: **if** $Request == Insertion$ **then**
3: $RC_{Edge\ Insertion}(P_i, G, G_k, \{u, v\}, k)$
4: **if** $Request == Deletion$ **then**
5: $RC_{Edge\ Deletion}(P_i, G, G_k, \{u, v\}, k)$

Proof. *Case 1:* If $u, v \in V_k$, the new edge $\{u, v\}$ is inserted to E_k and there is no change to V_k.

Case 2: We prove by contradiction that a vertex q in G cannot be in the k-core, unless $q \in \{w | w \in V, d_G(w) \geq k, N_G^k(w) \geq k\}$ where all the vertices in the set are reachable by either u or v. Suppose q is in the k-core but $q \notin \{w | w \in V, d_G(w) \geq k, N_G^k(w) \geq k\}$ 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 every vertex also has a core number greater than or equal to k. By definition of k-core in Section 4, the vertices on the path must have $d_G \geq k$ and $N_G^k \geq k$. Therefore, the vertices on the path to q and including q must have $d_G \geq k$ and $N_G^k \geq k$ as well. Therefore, q must be in the subgraph expanded from u and v. □

Algorithms 5 and 6 implement the edge insertion theorem. The algorithm maintains two auxiliary information for every vertex in the graph, $\forall v \in V$, its degree $d(v)$ and its qnc, $qnc_k(v)$ for the given k.

Algorithm 6 Edge Insertion- Node N_i Side

Input: Graph $G = (V, E)$,
G_k: the k-core graph,
$\{u, v\}$: new edge,
k: maintained core value
Output: the updated k-core graph

1: **if** $u \in G_k$ and $v \in G_k$ **then**
2: insert edge $\{u, v\}$ to G_k
3: **return**
4: **if** $d(u) < k$ or $d(v) < k$ **then**
5: **return**
Sequential version:
6: $C \leftarrow \emptyset$
7: **if** ($d(u) \geq k$ and $qnc_k(u) \geq k$) or ($d(v) \geq k$ and $qnc_k(v) \geq k$) **then**
8: $C \leftarrow$**Find Possible Edges to Insert**(G, G_k,C,k, u)
9: **if** $C \neq \emptyset$ **then**
10: $G'_k \leftarrow$ **Partial KCore** (C, k)
11: $G_k \leftarrow G_k \cup G'_k$
Parallel version: ▷ calls Algorithm 14
12: **if** ($d(u) \geq k$ and $qnc_k(u) \geq k$) or ($d(v) \geq k$ and $qnc_k(v) \geq k$) **then**
13: $insertTraversals \leftarrow \{u\}$
14: **Perform Insert Traversals**(G, G_k, $insertTraversals$,k)

The algorithm starts by updating the auxiliary values of u and v and their direct neighbors, since a new edge is inserted. Next, if the vertices u and v are already part of the existing k-core subgraph, then the algorithm terminates after inserting this edge into k-core subgraph. When the degree of either u or v is less than k, then the algorithm terminates. Otherwise, *Find Possible Edges to Insert* subroutine, which is described in Algorithm 7, returns a set of candidate edges in C that may be part of the k-core. Then another subroutine *Partial KCore*, which is described in Algorithm 8, filters out the edges in C that are not part of the k-core. The remaining edges are returned to be inserted into the updated k-core subgraph. *Perform Insert Traversals* subroutine provides the same functionality utilizing parallel search in case parallel execution is preferred. For practical reasons sequential search can over-perform parallel search when available resources are limited (e.g., single core available to program), or parallelism cost, i.e., thread related overhead, is high with respect to paralleled operation cost. Hence, we provide both sequential and parallel algorithms.

6.2 Deleting an Edge

We first give the following edge delete theorem.

Theorem 2. *Given a graph $G = \{V, E\}$ and its k-core subgraph $G_k = \{V_k, E_k\}$, and an edge $\{u, v\}$ is deleted from G,*

- *If $\{u, v\} \notin E_k$, then G_k does not change.*
- *If $\{u, v\} \in E_k$, then the subgraph consisting of vertices in $\{w | w \in V, d_G(w) \geq k, N_G^k(w) \geq k\}$, where every vertex is reachable from u or v, may need to be updated to delete additional vertices from G_k.*

The logic of its proof is similar to that of the edge insertion theorem and thus needs not be repeated here. The k-core subgraph is only updated when one of its edges is deleted. One can easily construct an extreme example where a single edge delete removes the entire k-core.

Algorithm 7 Find Possible Edges to Insert

Input: Graph $G = (V, E)$,
G_k: the k-core graph,
C: set of candidate edges,
k: maintained core value,
u: start vertex
Output: C: set of candidate edges

```
1: Q ← new queue
2: Q.enqueue(u)
3: mark(u)
4: while Q ≠ ∅ do
5:     v ← Q.dequeue()
6:     for each vertex w adjacent to v do
7:         if {v, w} ∉ C then
8:             if d(w) ≥ k and qnc_k(w) ≥ k then
9:                 C ← C ∪ {v, w}
10:                if w ∉ G_k and w is not marked then
11:                    Q.enqueue(w)
12:                    mark(w)
13: return C
```

Algorithm 8 Partial KCore

Input: C: set of candidate edges,
k: maintained core value
Output: C: the updated set of edges qualifying for k-core

```
1: changed ← true
2: while changed do
3:     changed ← false
4:     for each {u, v} ∈ C do
5:         if d_C(u) < k then
6:             delete {u, v} from C
7:             changed ← true
8: return C
```

Algorithm 9 implements the theorem on the server side. After auxiliary data updates, if the deleted edge $\{u, v\}$ was not in the k-core subgraph G_k, per theorem, the k-core does not change. Otherwise, the edge is deleted from G_k and the updated $\tilde{G}_k$ is recomputed by the k-core construction algorithm. We further improve this basic version by checking the in-core degrees $d_{G_k}(u)$, and $d_{G_k}(v)$ of u and v, respectively. If their in-core degrees remain above k after the edge deletion, the current k-core subgraph does not change. Otherwise, the *Delete Edges Cascaded* subroutine is invoked to traverse G_k and update the portion of the k-core subgraph that needs update.

Delete Edges Cascaded algorithm described in Algorithm 10 first starts with a vertex with in-core-degree less than k, deletes all its edges, and then updates its neighbors' in-core-degree counts accordingly. Then it recursively traverses the neighbors whose in-core-degrees are now below k. The algorithm accelerates k-core re-computing by knowing, at each iteration, which vertices have changed their in-core degrees. Therefore, it can avoid recomputing all the in-core degrees for all the vertices in the k-core. For the average case where an edge deletion impacts a small fraction of vertices in the k-core, we have found this improved algorithm to be very effective. *Perform Delete Traversals* algorithm provides parallel version of *Delete Edges Cascaded* algorithm for the case parallel algorithm is preferred.

Algorithm 9 Edge Deletion- Node N_i Side

Input: Graph $G = (V, E)$,
G_k: the k-core graph,
$\{u, v\}$: the edge to be deleted,
k: maintained core value
Output: the updated k-core graph

```
1: if u ∉ G_k or v ∉ G_k then
2:     return
3: delete {u, v} from G_k
   Pruned sequential version:
4: if d_{G_k}(u) ≥ k and d_{G_k}(v) ≥ k then
5:     return
6: if d_{G_k}(u) < k then
7:     Delete Edges Cascaded(G_k,k,u)
8: if d_{G_k}(v) < k then
9:     Delete Edges Cascaded(G_k,k,v)
   Pruned parallel version:                    ▷ uses Algorithm 12
10: if d_{G_k}(u) < k then
11:     deleteList ← {u}
12:     Perform Delete Traversals(G_k,deleteList,k)
13: if d_{G_k}(v) < k then
14:     deleteList ← {v}
15:     Perform Delete Traversals(G_k,deleteList,k)
```

Algorithm 10 Delete Edges Cascaded

Input: G_k: the k-core graph,
k: maintained core value,
u: start vertex
Output: the updated G_k

```
1: Q ← new queue
2: Q.enqueue(u)
3: mark(u)
4: while Q ≠ ∅ do
5:     v ← Q.dequeue()
6:     for each vertex w adjacent to v do
7:         delete {v, w} from G_k
8:         if d_{G_k}(w) < k then
9:             if w is not marked then
10:                Q.enqueue(w)
11:                mark(w)
```

7 Batch k-Core Maintenance

In 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. Batch maintenance mitigate the cost of BFS overhead dramatically.

Algorithm 11 Batch Process- Client Side

Input: Graph $G = (V, E)$,
k: maintained core value,
G_k: k-core graph,
$batchOperations$: list of operations stored in batch part
Output: the updated k-core graph

```
1: deleteList ← choose delete operations from batchOperations
2: Perform Delete Traversals(G_k, deleteList, k)
3: insertTraversals ← choose insert operations from batchOperations
4: Perform Insert Traversals(G, G_k, insertTraversals, k)
```

In such batch maintenance scenario, edge insertion or 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

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
P_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
$P_i(G_A)$	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

updates and flushes them based on update count or clocked window. As described in Algorithm 11, when the list is flushed, updates that cancel each other are first removed from the list. Edge deletions, which typically incur shorter graph traversal, are then treated next followed by edge insertions, which may include longer traversal. Regardless of the processing order, the net effect is the same.

Algorithm 12, run at client side, presents the batch edge deletions in more detail. Edges in the deletion list $deleteList$ are grouped and sent to respective partition's node, where each remote call returns a list of cascaded deletion requests. The client then regroups the requests.

Algorithm 12 Perform Delete Traversals- Client Side

Input: G_k: k-core graph,
$deleteList$: list of edges to be deleted,
k: maintained core value
Output: the updated k-core graph

1: **while** $deleteList \neq \emptyset$ **do** ▷ at each iteration
2: **for** each partition P_i in $partitions(G_k)$ **do**
3: $bucket_i \leftarrow from\ deleteList\ filter\ edges\ stored\ in\ P_i$
4: $cascadedDeletes_i \leftarrow RC_{Delete\ Edges}(P_i, G_k, bucket_i, k)$
5: $deleteList \leftarrow \emptyset$
6: **for** each partition P_i in $partitions(G_k)$ **do**
7: **if** $cascadedDeletes_i \neq \emptyset$ **then**
8: $add\ cascadedDeletes_i\ to\ deleteList$

Algorithm 14 presents batch 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. It first takes the list of edges $insertTraversals$, and traverses them in parallel. Once the parallel traversal is done, candidate list C will be processed by *Partial KCore* algorithm to compute k-core over traversed graph.

Algorithm 13 Delete Edges- Node N_i Side

Input: G_k: k-core graph,
$deleteList$: list of edges to be deleted,
k: maintained k value
Output: $cascadedDeletes$ the cascaded delete list

1: $cascadedDeletes \leftarrow \emptyset$
2: **for** each edge $\{u, v\}$ in $deleteList$ **do**
3: delete $\{u, v\}$ from G_k
4: **if** $d_{G_k}(u) < k$ **then**
5: **for** each vertex w adjacent to u **do**
6: delete $\{u, w\}$ from G_k
7: $cascadedDeletes \leftarrow cascadedDeletes \cup \{w, u\}$
8: **return** $cascadedDeletes$

Algorithm 14 Perform Insert Traversals- Client Side

Input: Graph $G = (V, E)$,
G_k: k-core graph,
$insertTraversals$: list of vertices to be traversed,
k: maintained core value
Output: the updated k-core graph

1: $C \leftarrow \emptyset$
2: **while** $insertTraversals \neq \emptyset$ **do** ▷ at each iteration
3: **for** each partition P_i in $partitions(G)$ **do**
4: $bucket_i \leftarrow from\ insertTraversals\ filter\ edges\ stored\ in\ P_i$
5: $qualifyingList_i \leftarrow RC_{Pruned\ Traversal}(P_i, bucket_i, k)$
6: $insertTraversals \leftarrow \emptyset$
▷ Aggregate this turn results and compute next turn input
7: **for** each partition P_i in $partitions(G)$ **do**
8: **for** each edge $\{u, v\}$ in $qualifyingList_i$ **do**
9: **if** $\{u, v\} \notin C$ **then** ▷ Select a vertex only once
10: $C \leftarrow C \cup \{u, v\}$
11: **if** $v \notin G_k$ **then** ▷ do not go over vertices already in G_k
12: $insertTraversals \leftarrow insertTraversals \cup \{v\}$
13: $G'_k \leftarrow$ **Partial KCore** (C, k)
14: $G_k \leftarrow G_k \cup G'_k$

The $PrunedTraversal$ algorithm described in Algorithm 15 runs on the node side and performs a single BFS iteration for the vertices in the $insertTraversals$ list.

Algorithm 15 Pruned Traversal- Node N_i Side

Input: Graph $G = (V, E)$,
$insertTraversals$: list of vertices to be traversed,
k: maintained k value
Output: $qualifyingList$: list of edges to qualifying neighbors

1: $returnList \leftarrow \emptyset$
2: **for** each vertex u in $insertTraversals$ **do**
3: **for** each vertex w adjacent to u **do**
4: **if** $d(w) \geq k$ and $n(w) \geq k$ **then**
5: $qualifyingList \leftarrow qualifyingList \cup \{u, w\}$
6: **return** $qualifyingList$

8 Performance Evaluation

Our experiments consist of three parts. In the first part, we evaluate the performance of running distributed k-core construction algorithms. The experiments show that the k-core construction algorithm with early pruning provides significant speedup compared to the base algorithm. In the second part, we evaluate the performance of the incremental k-core maintenance algorithms on dynamic graphs. We show that recomputing the whole k-core subgraph is much costlier than incrementally maintaining it. In the third part, we show that maintaining the k-core subgraph with batch updates provides further speedup compared to applying the updates one by one. Thus we can keep the recency of the results with much lower cost compared to reconstruction of the k-core.

8.1 Implementation on HBase

The server side of the algorithms were implemented as HBase coprocessors to take advantage of distributed parallelism. Table 2 describes the mapping from the graph construct in Table 1 to physically materialized tables, table regions and coprocessors in HBase. While we instantiate and quantify the benefits of our algorithms through HBase, alternative implementation of the same algorithms may also be developed for other distributed processing platforms.

TABLE 3
Key Characteristics of Data Sets in the Experiments

Name	Vertex Count	Bidirectional Edge Count	Ref
Orkut	3.1 M	234 M	[44]
LiveJournal	5.2 M	144 M	[44]
Flickr	1.8 M	44 M	[44]
Patents	3.8 M	33 M	[45]
Skitter	1.7 M	22.2 M	[45]
BerkStan	685 K	13.2 M	[45]
YouTube	1.1 M	9.8 M	[44]
WikiTalk	2.4 M	9.3 M	[45]
Dblp	317 K	2.10 M	[45]

8.2 System Setup

The experiment 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. The Hadoop File System (HDFS) replication factor is set at the default three replicas.

8.3 Data Sets

The data sets we used in the experiments were made available by Milove et al. [44] and the Stanford Network Analysis Project [45]. 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. Different from traditional graph processing approach, vertices and edges are stored in a distributed manner with large attribute data associated. Thus, total graph size is much larger than topology-only graphs in matrix or adjacency list form with possible edge weights.

8.4 k-Core Construction Experiments

In Section 5 we provided two algorithms for distributed k-core construction. The first algorithm we proposed, which is referred to as *Base k-core* algorithm, is described in Algorithms 1 and 2 in Section 5. The second algorithm, which is referred to as *Pruned k-core* algorithm, is an improved version of the first algorithm and is described in Algorithms 3 and 4. We implemented both of these distributed algorithms on HBase and compared their execution times of building a k-core subgraph for different k values. These k values are determined based on the degree distributions in our data sets. A vast majority of the vertices in these graphs have very low degrees as can be seen in the degree-distribution plot given in Fig. 1. As we want to identify the dense subgraphs with high cohesion in these real world data sets, we selected the k values based on the percentage of vertices with top degrees. We selected three different k values so that 4, 8 and 16 percent of the vertices in the data sets have a degree of at least k. See Appendix D.1.2, available in the online supplemental material, for more details on chosen k values.

Fig. 3 compares the execution times between the base and pruned algorithms for nine different data sets and three different k values. The execution time is shown in log-scale. The speedup factor of the pruned algorithm compared to the base algorithm is shown on the top of each bar corresponding to the pruned algorithm. As can be seen, pruned algorithm dramatically reduces the execution time, hence provides dramatic speedup. One key observation is that as the data set size gets bigger, the speedup also increases due to the significant reduction in messaging I/O among computing nodes. For the largest graphs such as Orkut, almost an order of magnitude improvement is observed. The parallel execution was well balanced as monitored through Ganglia [46] reported in Appendix D.1.4, available in the online supplemental material. Further experiments showed the k-core construction time decreases with an increasing number of servers as expected.

In Section 6, we presented distributed insertion and deletion algorithms to maintain k-core subgraph when edges are inserted into or deleted from the graph. Here, we evaluate the performance of these algorithms. We compare the maintenance time of each update with reconstruction time of the k-core subgraph every time an edge is inserted or deleted. Below are three update scenarios we consider on the given graph. For each scenario we measured the

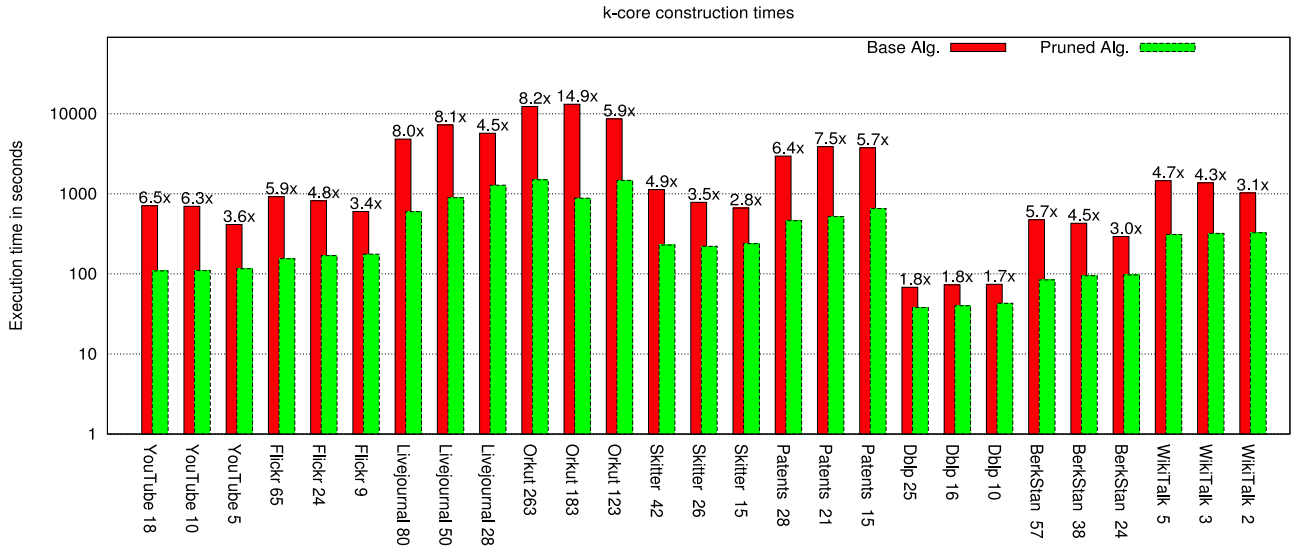

Fig. 3. k-core construction times for Base and Pruned k-core construction algorithms are shown for each data set with three chosen k values. Relative speedup achievement of Pruned algorithm over Base algorithm is provided above each bar.

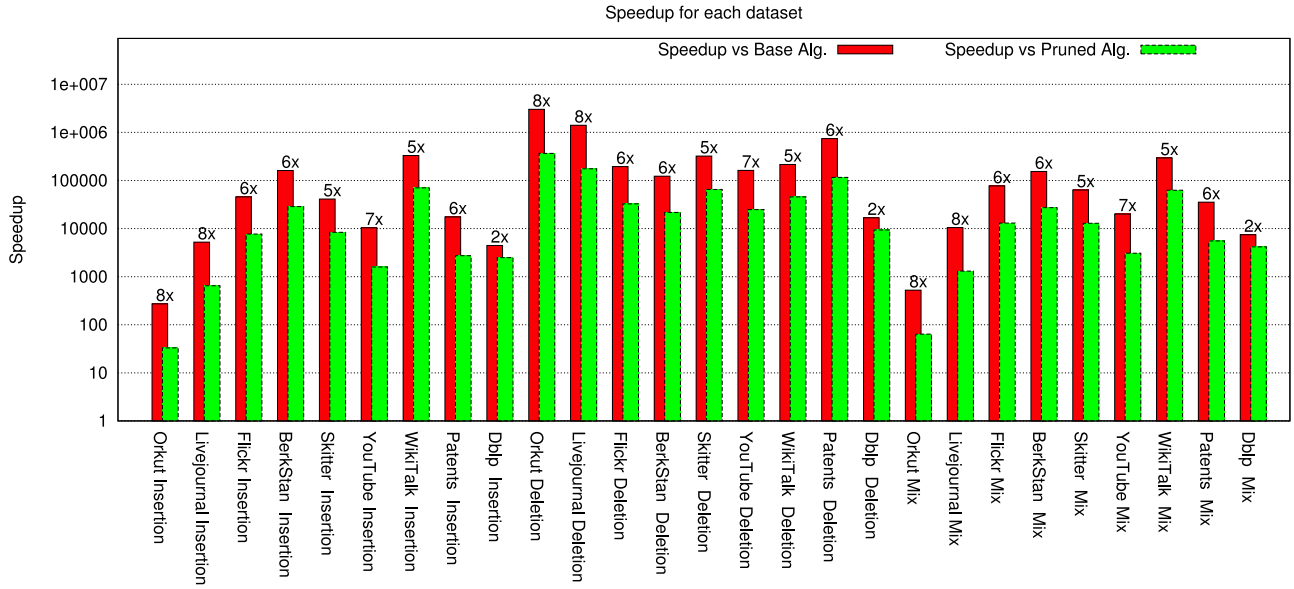

Fig. 4. k-core maintenance speedups for each data set with insertion, deletion, mix workload combinations. Maintenance algorithm speedup for both base and pruned construction algorithms is shown in the plot. Relative speedups are also provided above the bars.

performance of the system to maintain the previously materialized k-core subgraph:

1. In *Extending Window* scenario, a constant number of edges are continuously inserted into the original graph. We randomly choose 1,000 edges and insert them into the graph. Those random edges are selected from the graph and deleted before materialized k-core graph is constructed.
2. In *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 1,000 edges from the graph to delete them one by one while maintaining the k-core subgraph.
3. In *Moving Window* (Mix) scenario, Extending and Shrinking scenarios are run simultaneously where one insertion is followed by one deletion.

We repeat these three scenarios with each data set and measure their execution times. The largest k value chosen for each data set is used in the experiments. Fig. 4 plots the speedup through our incremental maintenance algorithms over recomputing k-core from scratch, for 9 different data sets. The y-axis shows the speedup in log-scale. For each data set and scenario, the figure gives the speedup of incremental update approach with respect to two versions of from-scratch construction, base construction algorithm and pruned construction algorithm. As the figure shows, three to four orders of magnitude speedup can be expected when the only updates are edge insertions (extending window scenario). Similar speedup factors can be observed for mixed edge insertions and deletions with one to one ratio (moving window-mix-scenario). Higher speedup, up to six orders of magnitude can be expected when the only updates are edge deletions (shrinking window scenario).

During our experiments, even though we measured the individual latencies for the three scenarios, we are not reporting them here due to space limitations. We observed that in the *Shrinking Window* workload, the average latencies are much smaller than those in the *Extending Window* workload. This is expected since in k-core maintenance, random edge deletes rarely incur the overhead of graph traversal and partial k-core recomputation. See Appendix D.1.3, available in the online supplemental material, for more details on update latency in milliseconds and its standard deviation.

Another notable observation is that few edge insertions take much longer time than average update latency. The main rationale behind this observation is the long traversals of the graph performed by calling the procedure described in Algorithm 7. Fig. 5 illustrates the latency measured over 1,000 random edge inserts. While the majority of inserts took 20 msec or less, a couple of inserts took longer than 10 seconds, which skewed the average and standard deviation. We observed as the graph gets larger, long graph traversals over the distributed servers are costly.

We further break down the update latency of k-core maintenance in Fig. 6 into three components: the first component is the normal update latency in HBase; the second

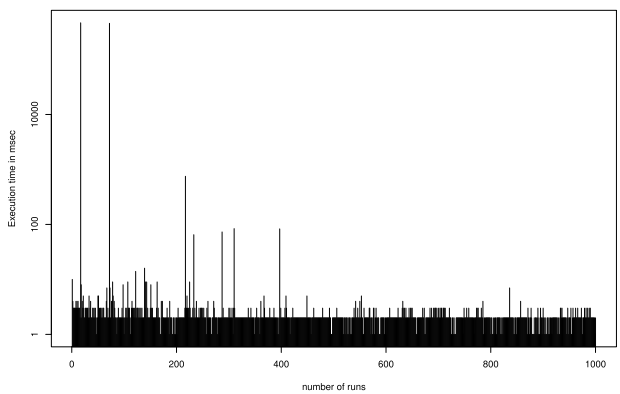

Fig. 5. Insert latency over 1,000 random edges to the LiveJournal data set.

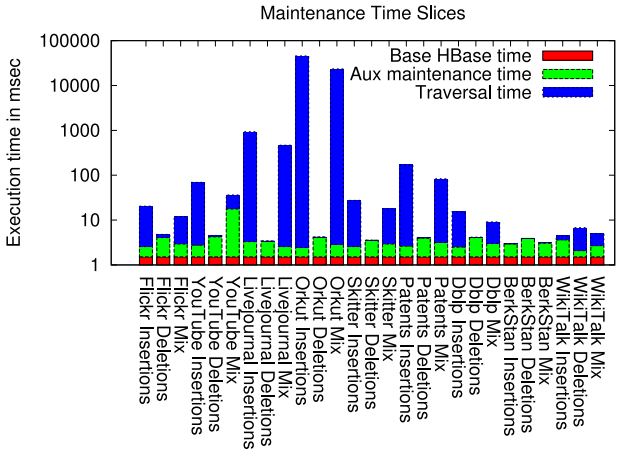

Fig. 6. k-core maintenance times for each data set-scenario where time slices for Base HBase insert/delete operation, auxiliary information maintenance and graph traversals are illustrated.

component is the time spent in updating the vertex degree and qnc; and the third component is the time in traversing the neighboring subgraph and partial k-core update. The first component stays mostly constant, while edge insertions contribute the most update latency due to graph traversal, when they occur. By performing batch updates our goal is to merge these costly traversals as much as possible as we discuss the batch maintenance experiments in the next section.

8.5 Batch Maintenance Experiments

To evaluate our batch update approach for maintaining the k-core subgraph, we run experiments to investigate

- to what extent batch processing provides speedup for different data sets,
- how batch size affects the mean update time, and
- how large batch size can grow before batch update gets slower than constructing k-core subgraph from scratch.

To measure the performance improvement of batch processing approach compared to individual updates, we set up experiments for each data set for the update scenarios described in Section 8.5. For each scenario we use 10K batch size. Fig. 7 shows batch processing speedup versus individual processing in k-core maintenance for three different update scenarios and for nine different data sets. The speedup is shown in the y-axis in log-scale. For each speedup bar, we also indicate on top of the bar the speedup factor explicitly. Each subfigure also illustrates the size of the respective data set in the secondary y-axes using a curve of crosses.

For extending window scenario, we get greater performance improvement, up to three orders of magnitude speedup particularly for large graphs. Results indicate a strong correlation between data set size and speedup from batch approach. As the data sets get bigger we get better performance improvement which is quite promising in terms of scalability of the proposed algorithms. On the other hand, for the data sets with less than 10M edges we observed a performance loss as opposed to having faster maintenance. This is mainly caused by the fact that, batch

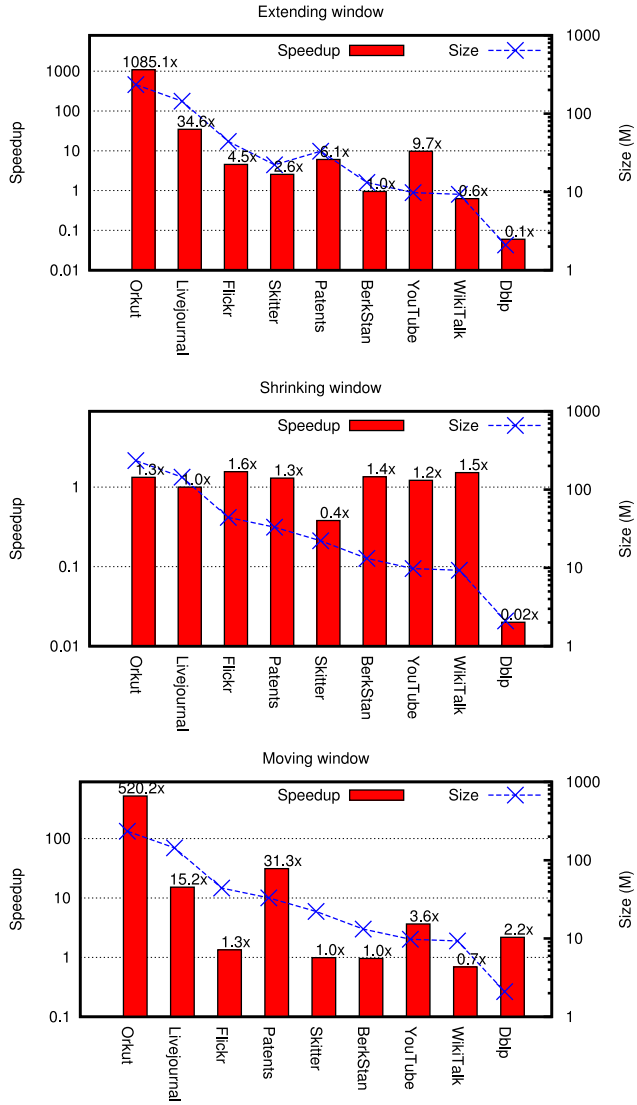

Fig. 7. 10K sized batch maintenance speedups for Extending window, Shrinking window and Moving window scenarios.

processing traversals are strictly parallelized and this results in many coprocessor calls. When a graph is small, the work performed by the coprocessors become quite negligible and therefore coprocessor call overhead outweighs the benefit it provides. 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 auxiliary maintenance cost plus base HBase update times. The moving window case provides speedups in between extending and shrinking window cases, which is as expected considering that it is a mixture of insertion and deletion operations.

We also conducted experiments to evaluate the relationship between batch size and mean update time of the k-core subgraph. For illustrative purposes, we reported results from the Flickr data set. By changing the batch size, we measured the average update time for each of the three update scenarios. Fig. 8 shows that the average update time gets smaller as the batch size increases. This is because a large batch size incurs more traversals to run in parallel and join together in case search space overlap. Thus, for larger batch

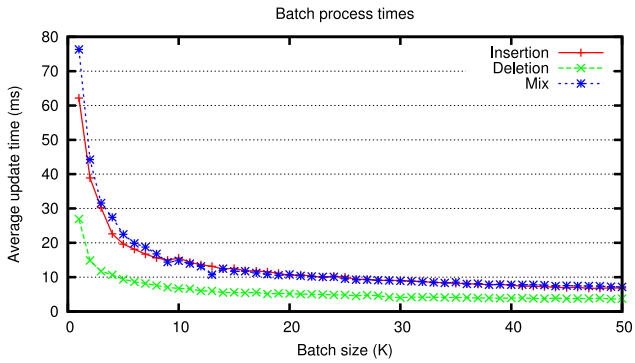

Fig. 8. Average edge update cost for increasing batch sizes from 1K up to 50K.

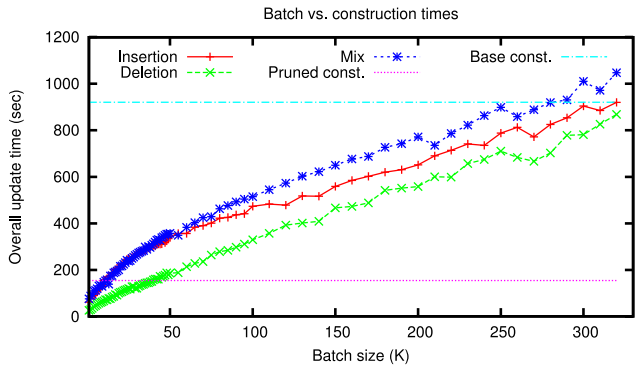

Fig. 9. Overall processing time of each batch of updates versus reconstruction time of k-core algorithm on Flickr data set.

sizes, average update time decreases and the traversal time no longer constitutes a significant cost for the updates. Instead auxiliary maintenance and base HBase updates become the main contributors of the overall cost. This is particularly valid for batch sizes greater than 10K-20K.

Lastly, we studied the break even point before exhausting the benefit of batch maintenance compared to simply reconstructing k-core. Fig. 9 shows the total update time of each batch processing and reconstruction time of k-core. As expected, when the batch size increases, the total update time increases for all insertion, deletion and mix updates as more edges need to be processed for larger batches. For the Flickr graph, batch maintenance cost crosses the cost of the pruned construction algorithm around 12K-40K updates and crosses the cost of the base construction algorithm around 290K-320K updates. Application requirements dictate the tradeoff between data recency and maintenance cost.

9 Conclusions

To the best of our knowledge, this paper is the first to propose a horizontally scaling solution for the k-core view materialization and maintenance of large, dynamic graphs that do not fit into memory. Our proposed set of algorithms aggressively prune the search space to minimize messaging among computing nodes holding partitioned data. Our experimental results demonstrated 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.

We observed opportunities to further optimize for efficiency when two or more k-core views for the same raw graph share overlaps. This may be resulted from semantic hierarchies such as "technology", "computer", and "IBM" or simply different k values on the same topic, say "IBM". Our current algorithms serve as the foundation to pursue these optimization opportunities.

Acknowledgments

This research was sponsored by US Defense Advanced Research Projects Agency (DARPA) under agreements no. W911NF-11-C-0200 and W911NF-12-C-0028. The authors would like to thank the anonymous reviewers for their helpful comments.

References

[1] M.J. Quinn and N. Deo, "Parallel Graph Algorithms," *ACM Computing Surveys*, vol. 16, no. 3, pp. 319-348, Sept. 1984.

[2] 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 Systems*, vol. 32, no. 2, article 13, June 2007.

[3] R. Zhou, C. Liu, J.X. Yu, W. Liang, B. Chen, and J. Li, "Finding Maximal K-Edge-Connected Subgraphs from a Large Graph," *Proc. 15th Int'l Conf. Extending Database Technology (EDBT '12)*, pp. 480-491, 2012.

[4] V. Batagelj and M. Zaversnik, "An o(m) Algorithm for Cores Decomposition of Networks," *CoRR*, vol. cs.DS/0310049, 2003.

[5] C. Giatsidis, D.M. Thilikos, and M. Vazirgiannis, "Evaluating Cooperation in Communities with the k-Core Structure," *Proc. Int'l Conf. Advances in Social Networks Analysis and Mining (ASONAM '11)*, pp. 87-93, 2011.

[6] C. Giatsidis, K. Berberich, D.M. Thilikos, and M. Vazirgiannis, "Visual Exploration of Collaboration Networks Based on Graph Degeneracy," *Proc. 18th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '12)*, pp. 1512-1515, 2012.

[7] W. Wei and S. Ram, "Using a Network Analysis Approach for Organizing Social Bookmarking Tags and Enabling Web Content Discovery," *ACM Trans. Management Information Systems*, vol. 3, no. 3, pp. 15:1-15:16, Oct. 2012.

[8] H. Chun, H. Kwak, Y.-H. Eom, Y.-Y. Ahn, S. Moon, and H. Jeong, "Comparison of Online Social Relations in Volume vs Interaction: A Case Study of Cyworld," *Proc. Eighth ACM SIGCOMM Conf. Internet Measurement (IMC '08)*, pp. 57-70, 2008.

[9] hbase.apache.org, 2014.

[10] T. Luczak, "Size and Connectivity of the k-Core of a Random Graph," *Discrete Math.*, vol. 91, no. 1, pp. 61-68, July 1991.

[11] B. Pittel, J. Spencer, and N. Wormald, "Sudden Emergence of a Giant k-Core in a Random Graph," *J. Combinatorial Theory, Series B*, vol. 67, no. 1, pp. 111-151, May 1996.

[12] C. Cooper, "The Cores of Random Hypergraphs with a Given Degree Sequence," *Random Structures & Algorithms*, vol. 25, pp. 353-375, 2004.
[13] M. Molloy, "Cores in Random Hypergraphs and Boolean Formulas," *Random Structures & Algorithms*, vol. 27, no. 1, pp. 124-135, Aug. 2005.
[14] S. Janson and M.J. Luczak, "A Simple Solution to the k-core Problem," *Random Structures & Algorithms*, vol. 30, no. 1/2, pp. 50-62, Jan. 2007.
[15] S.B. Seidman, "Network Structure and Minimum Degree," *Social Networks*, vol. 5, no. 3, pp. 269-287, 1983.
[16] J.I.A. Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-Core Decomposition: A Tool for the visualization of Large Scale Networks," *CoRR*, vol. abs/cs/0504107, 2005.
[17] J.I.A. Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large Scale Networks Fingerprinting and Visualization using the k-core Decomposition," *Proc. Advances in Neural Information Processing Systems*, pp. 41-50, 2005.
[18] Batagelj, Mrvar, and Zaversnik, "Partitioning Approach to Visualization of Large Graphs," *Proc. GDRAWING: Seventh Int'l Symp. Graph Drawing (GD)*, 1999.
[19] Baur, Brandes, Gaertler, and Wagner, "Drawing the AS Graph in 2.5 Dimensions," *Proc. GDRAWING: Conf. Graph Drawing (GD)*, 2004.
[20] Y. Zhang and S. Parthasarathy, "Extracting Analyzing and Visualizing Triangle K-core Motifs within Networks," *Proc. IEEE 28th Int'l Conf. Data Eng. (ICDE)*, pp. 1049-1060, 2012.
[21] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "A Model of Internet Topology using k-shell Decomposition," *Proc. Nat'l Academy of Sciences USA*, vol. 104, no. 27, pp. 11 150-11 154, 2007.
[22] J.I.A. Hamelin, M.G. Beiró, and J.R. Busch, "Understanding Edge Connectivity in the Internet through Core Decomposition," *Internet Math.*, vol. 7, no. 1, pp. 45-66, 2011.
[23] J.I.A. Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-Core Decomposition of Internet Graphs: Hierarchies, Self-Similarity and Measurement Biases," *Networks and Heterogeneous Media*, vol. 3, no. 2, pp. 371-393, 2008.
[24] C. Giatsidis, D. Thilikos, and M. Vazirgiannis, "Evaluating Cooperation in Communities with the k-Core Structure," *Proc. Int'l Conf. Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 87-93, 2011.
[25] P. Sanders and C. Schulz, "10th Dimacs Implementation Challenge-Graph Partitioning and Graph Clustering," 2012.
[26] M. Altaf-Ul-Amin et al., "Prediction of Protein Functions Based on k-Cores of Protein-Protein Interaction Networks and Amino Acid Sequences," *Genome Informatics Series*, vol. 14, pp. 498-499, 2003.
[27] G.D. Bader and C.W.V. Hogue, "An Automated Method for Finding Molecular Complexes in Large Protein Interaction Networks," *BMC Bioinformatics*, vol. 4, article 2, 2003.
[28] S. Wuchty and E. Almaas, "Peeling the Yeast Protein Network," *Proteomics*, vol. 5, no. 2, pp. 444-449, 2005.
[29] J. Cheng, Y. Ke, S. Chu, and M.T. Özsu, "Efficient Core Decomposition in Massive Networks," *Proc. IEEE 27th Int'l Conf. Data Eng. (ICDE)*, pp. 51-62, 2011.
[30] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed K-Core Decomposition," *IEEE Trans. Parallel and Distributed Systems*, vol. 24, no. 2, pp. 288-300, Feb. 2013.
[31] D. Miorandi and F. De Pellegrini, "K-Shell Decomposition for Dynamic Complex Networks," *Proc. Eighth Int'l Symp. Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*, pp. 488-496, 2010.
[32] R. Li and J. Yu, "Efficient Core Maintenance in Large Dynamic Graphs," arXiv preprint arXiv:1207.4567, 2012.
[33] J. Greiner, "A Comparison of Parallel Algorithms for Connected Components," *Proc. Sixth Ann. ACM Symp. Parallel Algorithms and Architectures (SPAA '94)*, pp. 16-25, 1994.
[34] U. Kang, C.E. Tsourakakis, and C. Faloutsos, "Pegasus: Mining Peta-Scale Graphs," *Knowledge and Information Systems*, vol. 27, no. 2, pp. 303-325, May 2011.
[35] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: A Scalable and General Graph Management System," *Proc. 17th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '11)*, pp. 1091-1099, 2011.
[36] Y. Bu, B. Howe, M. Balazinska, and M.D. Ernst, "Haloop: Efficient Iterative Data Processing on Large Clusters," *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 285-296, Sept. 2010.
[37] J. Lin and M. Schatz, "Design Patterns for Efficient Graph Algorithms in Mapreduce," *Proc. Eighth Workshop Mining and Learning with Graphs (MLG '10)*, pp. 78-85, 2010.
[38] 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," *Proc. Int'l Conf. Management of Data (SIGMOD '10)*, pp. 135-146, 2010.
[39] B. Shao, H. Wang, and Y. Li, "The Trinity Graph Engine," Technical Report 161291, Microsoft Research, 2012.
[40] 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. Computer Systems*, vol. 26, no. 2, pp. 4:1-4:26, June 2008.
[41] D. Borthakur, J. Gray, J.S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache Hadoop Goes Realtime at Facebook," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '11)*, pp. 1071-1080, 2011.
[42] wiki.apache.org/hadoop/Hbase/PoweredBy, 2014.
[43] J. Mondal and A. Deshpande, "Managing Large Dynamic Graphs Efficiently," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '12)*, 2012.
[44] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," *Proc. Fifth ACM/Usenix Internet Measurement Conf. (IMC '07)*, Oct. 2007.
[45] snap.stanford.edu/, 2014.
[46] ganglia.info, 2014.
[47] blogs.apache.org/hbase/entry/coprocessor_introduction, 2014.

Hidayet Aksu received the BS and MS degrees from the Department of Computer Engineering of Bilkent University, Turkey, where he is currently working toward the PhD degree in the Department of Computer Engineering. Currently, he is a visiting scholar at IBM T.J. Watson Research Center. His research interests include social networks, big data analytics, distributed computing, wireless networks, wireless ad hoc and sensor networks, localization, and p2p networks. He is a member of the IEEE.

Mustafa Canim received the BS degree from Bilkent University, Ankara, Turkey, and the PhD degree in computer science from the University of Texas at Dallas, Richardson. He is currently a research staff member at IBM T.J. Watson Research Center, Yorktown Heights, New York. Before joining the PhD program, he was a senior researcher with the Database Research Group, Scientific and Technological Research Council of Turkey, a research institute in Turkey. He interned at the Database Research Group, IBM T.J. Watson Research Center in the Solid State Sensitive Systems project in 2008 and 2009. His research at IBM has led to several patent applications and top-level conference papers. He was also at the Engineering Tools Department, Google, as an Intern where he gained significant experience on cloud computing platforms. His research interests include the area of architecture conscious data management with a focus on privacy and performance issues and large-scale social network analysis.

Yuan-Chi Chang received the BS degree from National Taiwan University, and the MS and PhD degrees from the University of California, Berkeley. He is a research staff member and a manager in software research at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. His research interest is in the areas of data management, including data server, integration, analytics and big data. He is a senior member of the IEEE and a member of the ACM.

Ibrahim Körpeoğlu received the BS degree in computer engineering from Bilkent University in 1994. He received the MS and PhD degrees from the University of Maryland at College Park, in computer science, in 1996 and 2000, respectively. He joined Bilkent University in 2002, and he is an associate professor in the Department of Computer Engineering. Before that, he was in several research and development companies in US including Ericsson, IBM T.J. Watson Research Center, Bell Laboratories, and Bell Communications Research (Bellcore). He received Bilkent University Distinguished Teaching Award in 2006 and IBM Faculty Award in 2009. He is a member of the ACM and a senior member of the IEEE.

Özgür Ulusoy received the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He is currently a professor in the Computer Engineering Department of Bilkent University in Ankara, Turkey. His current research interests include web databases and web information retrieval, multimedia database systems, social network analysis, and peer-to-peer systems. He has published more than 120 articles in archived journals and conference proceedings. He is a member of the IEEE and ACM.