EFFICIENT ANALYSIS OF LARGE-SCALE SOCIAL NETWORKS USING BIG-DATA PLATFORMS

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Hidayet AKSU
July, 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. İbrahim Körpeoğlu (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. Sinan Gezici

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Assist. Prof. Dr. Buğra Gedik

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Ahmet Coşar

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

EFFICIENT ANALYSIS OF LARGE-SCALE SOCIAL NETWORKS USING BIG-DATA PLATFORMS

Hidayet AKSU
Ph.D. in Computer Engineering
Supervisor: Assoc. Prof. Dr. İbrahim Körpeoğlu
July, 2014

In recent years, the rise of very large, rich content networks re-ignited interest to complex/social network analysis at the big data scale, which makes it possible to understand social interactions at large scale while it poses computation challenges to early works with algorithm complexity greater than O(n). This thesis analyzes social networks at very large-scales to derive important parameters and characteristics in an efficient and effective way using big-data platforms. With the popularization of mobile phone usage, telecommunication networks have turned into a socially binding medium and enables researches to analyze social interactions at very large scales. Degree distribution is one of the most important characteristics of social networks and to study degree characteristics and structural properties in large-scale social networks, in this thesis we first gathered a tera-scale dataset of telecommunication call detail records. Using this data we empirically evaluate some statistical models against the degree distribution of the country's call graph and determine that a Pareto log-normal distribution provides the best fit, despite claims in the literature that power-law distribution is the best model. We also question and derive answers for how network operator, size, density and location affect degree distribution to understand the parameters governing it in social networks.

Besides structural property analysis, community identification is of great interest in practice to learn high cohesive subnetworks about different subjects in a social network. 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. 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. Experimental evaluation results demonstrated orders of magnitude speedup and advantages of maintaining k-core incrementally and in batch windows over complete reconstruction approaches.

Moreover, the intensity of community engagement can be distinguished at multiple levels, resulting in a multiresolution community representation that has to be maintained over time. We also propose distributed algorithms to construct and maintain a multi-k-core graphs, 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. Furthermore, we propose a graph aware cache system designed for distributed graph processing. Experimental results demonstrate up to 15x speedup compared to traditional LRU based cache systems.

Keywords: social network analysis, Big Data analytics, degree distributions, k-core, graph theory, distributed computing, dynamic networks.

ÖZET

BÜYÜK ÖLÇEKLİ SOSYAL AĞLARIN BÜYÜK VERİ PLATFORMU KULLANARAK ETKİN ANALİZİ

Hidayet AKSU
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Doç. Dr. İbrahim Körpeoğlu
Temmuz, 2014

Son yıllarda zengin içerikli çok büyük ağlardaki artış kompleks/sosyal ağ analizine dönük ilgiyi yeniden arttırmıştır. Söz konusu analizler bir taraftan büyük çapta sosyal etkileşimleri anlamayı mümkün kılarken diğer taraftan O(n) üzeri kompleksitiye sahip algoritmalara dayalı önceki çalışmalarda sorun oluşturmaktadır. Bu tez önemli parametrelerini ve özelliklerini etkin ve verimli bir şekilde bulmak amacıyla büyük veri platformu kullanarak çok büyük ölçekli sosyal ağları analiz eder. Mobil telefon kullanımının popülerleşmesi ile birlikte telekomünikasyon ağları sosyal bağlayıcı ortamlara dönüşmüştür ve araştırmacıların sosyal etkileşimleri çok büyük ölçekte analiz etmesine olanak sağlamıştır. Derece dağılımları sosyal ağların en önemli karakteristikleri arasında yer alır ve büyük ölçekli sosyal ağlarda derece karakteristiği ile yapısal özellikleri araştırmak için biz bu tezde öncelikle tera-ölçekli bir telekomünikasyon arama detay kaydı veriseti derledik. Biz bu veriyi kullanarak bazı istatistik modelleri ülke çağrı çizgesi derece dağılımına karşı deneysel olarak değerlendirdik ve literatürdeki "power-law en iyi modeldir" iddalarına karşın, Pareto log-normal dağılımının en iyi uyumu sağladığına karar verdik. Ayrıca, sosyal ağlarda derece dağılımını yöneten parametreleri anlamak amacıyla, ağ operatörünün, büyüklüğünün, yoğunluğunun ve lokasyonunun derece dağılımını nasıl etkilediğini sorguladık ve cevap elde ettik.

Yapısal özellik analizi dışında, bir sosyal ağda farklı konularda çok bağlantılı alt ağları bulmak için yapılan topluluk tespiti çalışmaları pratikte büyük ilgi çekmektedir. Çizge teorisinde, k-core çizgenin 'yoğun' alanları olarakta bilinen çok bağlantılı alt çizgelerin tespiti için kullanılan anahtar bir ölçüttür. Sosyal ağ çizgeleri gibi gerçek dünya çizgeleri boyut yönünden büyüyüp, içerik yönünden zenginleşip ve topolojiler dinamik olarak değiştikçe, yalnız k-core altçizgesini bir defalığına hesaplama problemi ile değil ayrıca bunu dinamik değişikliklere göre güncel tutma problemi ile karşılaştık. Bu zorluklar bize

yatay ölçeklenebilir saklama ve hesaplama platformu üzerinde k-core görüntü hesaplama ve sürdürme amaçlı bir takım algoritmalar önerme konusunda esin vermiştir. Önerdiğimiz algoritmaların deneysel değerlendirme sonuçları bütün yeniden hesaplama yaklaşımına göre aşamalı ve yığın olarak k-core sürdürme avantajı ile birlikte birkaç basamak hızlandırma göstermiştir.

Bununla birlikte, topluluğa katılımın yoğunluğu birçok seviyede seçilebilir ki bu da zamanla sürdürülmesi gerekli çok-çözünürlüklü topluluk gösterimini sonuç doğurur. Bu nedenle biz ayrıca çoklu-k-core çizgesi hesaplayıp sürdürecek Apache HBase ölçeklenebilir büyük-veri platformunda uygulanmış dağıtık algoritmalar önerdik. Deneysel değerlendirme sonuçları aşamalı çoklu-k-core sürdürmenin bütün yeniden hesaplamaya göre birkaç basamak hızlandırma sağladığını göstermiştir. Diğer taraftan, dağıtık çizge işleme amaçlı tasarlanmış bir çizge-bilinçli önbellek sistemi önerdik. Deney sonuçları geleneksel LRU bazlı sistemlerle karşılaştırıldığında 15 kata kadar hızlanma göstermiştir.

Anahtar sözcükler: sosyal ağ analizi, Büyük Veri analitiği, derece dağılımları, k-core, çizge teorisi, dağıtık hesaplama, dinamik ağlar.

Acknowledgement

First of all, I am very grateful to my supervisor Assoc. Prof. Dr. İbrahim Körpeoğlu for his invaluable support, guidance and motivation during my graduate study, and for encouraging me a lot in my academic life. His vast experience and encouragement have been of great value during the entire study. It was a great pleasure for me to have a chance of working with him. I learned a lot from my supervisor, especially the endurance needed for this kind of study.

I would like to thank to the thesis committee members Prof. Dr. Özgür Ulusoy and Assoc. Prof. Dr. Sinan Gezici for their valuable comments for the past six years. I would also like to thank to the thesis jury members Assist. Prof. Dr. Buğra Gedik and Prof. Dr. Ahmet Coşar for kindly accepting to spend their valuable time and to evaluate this work.

I owe my warmest thanks to Dr. Mustafa Canım and Dr. Yuan-Chi Chang for their cooperation during this study. I would like to thank Mahmut Kutlukaya for his expert contributions on statistical tests. I also would like to express my appreciation to IBM Thomas J. Watson Research Center, Information and Communication Technologies Authority (ICTA), and my superiors for the understanding and support during my academic studies.

I would like to thank to my parents and grandparents for raising me with all their love. I would not be the person who I am without their never-ending support. I would also like to thank to my brothers and sisters. Despite the physical distance between us throughout our lives, they always cheer me up.

And most of all, my beloved wife Zeynep who has lived every stage of this long journey with me. Thank you for bearing with me for all this time. I cannot express how valuable your support has been to me, I love you. I apologize for the time I have stolen from you. I promise to be a better husband from now on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the rise of very large, rich content social and complex networks has made it possible to understand social interactions at large scale. Thus a new era for social network analysis field has emerged in which early study results and methods need to be re-visited. Previous results with limited empirical support require re-evaluation with large real data while early works with algorithm complexity greater than $O(n)$ are not feasible for big data scale studies.

Social networks were first analyzed by social scientists, who performed manual data collection and considered at most hundreds of individuals [1]. Later, social network analysis (SNA) became an interesting topic for many other sectors and research fields, including recommender systems [2, 3]; marketing [4]; web document clustering [5, 6]; intelligence analysis [7]; clustering and community detection [8, 9, 10, 11, 12, 13] and urban planning [14]. Massive use of electronic devices and online communication leaves traces of human interaction and relationships, such as phone call records, e-mail records, etc. Using these traces, collective human behavior and social interactions can be understood on a large scale, which was previously impossible [15]. Recently telecommunication datasets with location information have also been used to conduct research on human behavioral patterns [16, 17, 18].

Social network analysis tries to understand the characteristics a social network

exhibits. The first and most-cited characteristic among others is *degree distribution* of nodes constituting a social network. A bulk of studies in the literature on this topic reports that power-law with certain parameters fits best [19, 20, 21]. Other studies, however, propose different statistical fit models [22, 23, 24].

Since current studies are limited by the used datasets from which their proposals are derived/obtained, it is necessary to explore the influence of dataset specific parameters on discovered social network characteristics. This observation motivated us as part of this thesis to first conduct research on degree distribution on larger scales to discover the parameters governing degree distribution in social networks. Among many current research issues to be investigated, we preferred this less studied problem which requires a complete dataset. Therefore, in this thesis we first explore how parameters like network operator, network size, population density, and geographic location affect degree distribution in social networks.

On the other hand, 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. 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 [25]. 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, 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 of such measures include cliques, quasi-cliques [26], k-core, k-edge-connectivity [27], 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 [28], where other measures have complexity growing super-linearly or NP-hard.

In this thesis, we also propose scalable, distributed algorithms for k-core graph construction as well as its incremental and batch maintenance as dynamic changes are made to the graph. 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. 1.1 illustrates the degree distribution of nine published graph datasets, 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 [29, 30] use co-authorship as edge weight in the graph. In [31], Wei and Ram consider organization of social bookmarking tags using k-core with tag weight as a metric. Chun et al. in [32] consider friends and their bidirectional relations on a graph. The paper compares k-core of friendships and k-core of bidirectional activity relationships.

Moreover, the intensity of community engagement can be distinguished at

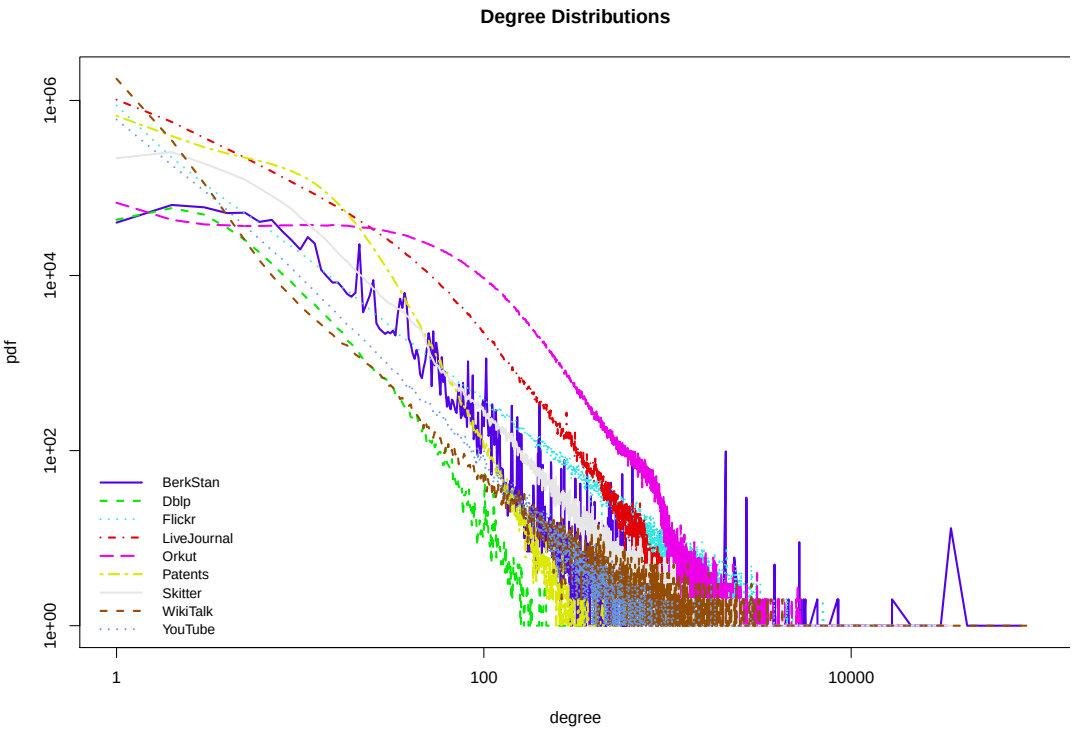


Figure 1.1: Degree distribution of vertices in nine social network datasets on the log scale.

multiple levels, resulting in a multi-resolution community representation that has to be maintained over time. A further distinction from the decade-old graph problem formulation is that multi-attributed content associated vertices and edges must be included in creating, managing, interpreting and maintaining results. Thus the problem of multi-resolution community analysis is a hybrid of content and graph analysis on various subjects of interest. The problem is made further complex with the observation that interactions with a community happen not just at one but multiple levels of intensity, which reflects in reality active to passive participants in a group. This results with multiple levels of depth in multi-resolution community identification. To make the solution practical, it is thus necessary to make community identification and continuing maintenance at multiple resolutions.

Our first study on the identification and maintenance of k-core subgraphs considers a fixed k value. We also propose algorithms to perform batch operations for maintenance purposes. The proposed approaches are quite effective when a constant k value is used. On the other hand, when subgraphs at multiple resolutions are needed, one has to run separate instances of the algorithms for each k value. In order to cope with this limitation, significant design changes are considered in our algorithms to efficiently handle k-core subgraphs at multiple, fixed k values. Integrated algorithms are proposed for k-core construction, maintenance and bulk processing of update operations. As we demonstrate with our experimental results, these algorithms yield orders of magnitude speed up compared to the base case k-core construction.

Consider the following scenario as an example of how the distributed multi k-core construction and maintenance algorithms we propose could be used in real life problems. Suppose that a data analytics company provides keyword based analytics services to its customers based on the Retweet graph of Twitter data. The customers subscribe to the service by providing certain keywords along with the queries and the company runs them whenever the customers want to get the results. The queries are periodically resubmitted by the customers as new Tweets get processed over time. Because of the incremental updates on the Retweet graph, the data grow rapidly. To keep up with the growing size of

the data and manage the query load on the system, the graph is horizontally partitioned and stored on distributed computing nodes. Suppose further that a customer is working on franchising Japanese restaurants and interested in finding communities potentially interested in Japanese food and their physical locations. To address this customer's needs, the analytics company runs k-core algorithm on the entire Retweet graph while filtering the tweets where Japanese food related keywords are used and returns the results to the customer. The customer periodically resubmits the query to get informed about the most recent trends to make healthier marketing decisions as the graph changes over time. The analytics company however has to reconstruct the entire k-core subgraph whenever the customer submits the same query repeatedly. As the company gets more popular over time and millions of customers subscribe to the system with different keywords, the load on the system becomes unmanageable. The company employees are now compelled to find a solution to reduce the computation load on the system and find a way of improving the response time. As a solution they decide to materialize the results of user queries and update them as the Retweet graph changes. They want to design a solution that supports both instant updates on the maintained results as well as batch updates depending on customer needs. The price charged to the customer increases with respect to the recency of the results and the speed with which these results are generated. For customers who care less about the recency of the results but more about the price of the service, the updates are accumulated and applied in batches to the materialized subgraphs.

The aforementioned scenario is quite realistic these days. As the popularity of social media sites increases, the demand for doing analytics on these large graphs grows dramatically. In the last few years many web companies, such as "Followerwonk" [33], "Tweetwall" [34], "SimplyMeasured" [35], emerged for helping customers make better marketing decisions based on the content of social media tools such as Twitter and Google+. These web companies have to deal with very large graphs to perform analytics. These graphs are considered large not only because they have many vertices and edges but also they maintain significantly large amounts of metadata associated with them. Many of these

social web companies tend to store these graphs in distributed datastores such as Google BigTable, MegaStore, Apache HBase or distributed parallel databases, with motivations behind Big Data trend, i.e., high availability, fault-tolerance, scalability, persistence. They have to provide high availability to their customers to maintain their popularity. Facebook, for instance, recently announced that 1.11 billion users connect to the site every month. Also the average number of users per day as of March 2013 is 665 million. The user related metadata such as messages, chats, emails, SMS messages and attachments are stored on thousands of HBase clusters. 6 Billion messages are sent between Facebook users daily. At peak times 1.5 million operations are executed per second on the metadata associated with graph vertices and edges. To keep up with the scale, store, and maintain these datasets efficiently, companies are compelled to use distributed data architectures. We believe that the distributed algorithms we present in this thesis can be leveraged on these large graph datasets to perform better analytics.

On the other hand, Big Data platforms utilize disk storage both to provide persistence and to handle the data that do not fit into the main memory. Because of this, distributed graph algorithm implementations display poor performance on big data platform when compared with traditional single server in memory implementations. Employing a caching layer is one of the most effective approaches to reduce performance bottlenecks due to slow disks. A high performance cache layer can hide most of the slow disk operations and improve the overall system performance. Most operating systems and applications implement disk buffering at some degree. However random access pattern, which is frequently observed in the graph algorithms, causes low performance at such disk buffers, i.e., buffer cache.

Thus, in this thesis we also study the caching problem in big data platforms. We focus on distributed graph processing use case and propose a graph-aware caching which is designed to exploit graph specific data access patterns. We revisit the principle of locality in the distributed graph algorithms context and figure out specific data reference patterns. Our proposed algorithms benefit from discovered locality of references and provide improved data access speed. Reducing data access overhead, graph algorithms perform faster on Big Data platforms and

allow working with larger data.

1.1 Contributions

Our contributions in this thesis can be summarized as follows:

- We first constructed a countrywide call graph utilizing a full call detail record (CDR) set of all mobile and fixed-line telco network operators. This comprehensive dataset allowed us to analyze a social network without wondering about possible bias from single-operator, size, location or density-limited datasets.
- We questioned the root cause of different conclusions in the literature about degree distribution in social networks, suggesting that they might be related to utilized datasets' density, location, size, and source operator.
- We performed controlled empirical analyses for various densities, sizes, locations and operators, and formed conclusions on density-degree, location-degree, size-degree and operator-degree distribution relations.
- We developed and accelerated distributed k-core construction algorithms through aggressive pruning of the graph that will not be in the final k-core subgraph.
- We developed new k-core *maintenance* algorithms 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 Co-processor computing framework [36]. Our system built on HBase stores graph data, including metadata and unstructured content, in the HBase tables.

- We proposed a novel cache design which is both graph access and distributed deployment aware.

1.2 Outline of the Dissertation

Organization of the thesis is as follows. In the next chapter we give the related studies in the literature together with some background information. In Chapter 3, we present an analysis of social networks based on tera-scale telecommunication datasets, mainly focusing on the degree distributions. We next introduce our distributed k-Core view materialization and maintenance algorithms for large dynamic graphs for social networks in Chapter 4. Chapter 5 describes our multiple resolution network community identification and maintenance algorithms. Our proposed graph-aware caching and its performance are presented in Chapter 6. Finally, in Chapter 7, we present our conclusions.

Chapter 2

Related Work and Background

In this chapter, we describe the previous work related to our study on efficient analysis of social networks on Big Data platform. We first give studies on social networks degree analysis using call graphs. Next, we present studies related to k-core decomposition, since our community identification studies focus on k-core algorithm. Then, we present other parallel graph algorithms. Finally, we discuss the studies related to caching of distributed graphs.

2.1 Call Graphs Analysis

Aiello et al. [19] study the statistics of phone call graphs for long-distance fixed-lines and report that in-degree distribution is fitted by power-law distribution with exponent $\gamma = 2.1$. In [20], Onnela et al. work on mobile phone data containing $N = 4.6 \times 10^6$ nodes and $L = 7.0 \times 10^6$ links and report a power-law distribution fit with exponent $\gamma = 8.4$. They describe the dataset as "all mobile phone call records of calls among $\approx$ 20% of the entire population of the country", which implies that they used a sub-network of a country's operator network. Dasgupta et al. [21] present another study on mobile phone data, with a reciprocal call graph containing $N = 2.1 \times 10^6$ nodes and $L = 9.3 \times 10^6$ directed edges. That dataset belongs to one of the world's largest mobile operators. The authors report that

degree distribution is fitted well by power-law distribution with exponent $\gamma = 2.91$. On the other hand, Bi et al. [22] propose the discrete Gaussian exponential (DGX) distribution and report that it provides a very good fit with many datasets, including telco data. Moreover, Seshadri et al. [23], using mobile phone data from an anonymous operator in the US, study modeling degree characteristics and report that degree distribution significantly deviates from what would be expected by power-law and log-normal distributions. Their findings suggest that double Pareto log-normal distribution (DPLN) provides better fits for degree distribution. In [24], Sala et al. analyze Facebook's social network data and report that Pareto log-normal (PLN) distributions are much better predictors of degree distributions in real graphs than power-law distributions are.

2.2 k-core Decomposition

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 [37, 38, 39, 40, 41]. k-core decomposition has been used in many applications such as network visualization [42, 43, 44, 45, 46, 47], Internet topology analysis [48, 49, 50], social networks [29, 51], and biological networks [52, 53, 54]. The notion of k-core is first introduced in [42] 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 [42] a well known algorithm for computing k-core decomposition is first proposed by Batagelj and Zaversnik (BZ) [28]. 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 [55] 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 [56] 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 [56], it is assumed that everything is held in the memories in computing nodes. The third important point is that in [56], only the number of iterations required to compute k-core decomposition is reported but not real execution times. In this thesis, 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 [57] and an improved alternative was introduced by Li et al. in [58]. In [57], Miorandi et al. 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 [58], 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. On the other hand, Sariyüce et al. [59] proposes state-of-the-art algorithms for incremental maintenance of k-core decomposition for streaming graph data which outperform the work by Li and Yu. They provide extensive theoretical and experimental analysis with various graph models and different graph sizes. Empirical evaluations show up to 6 orders of magnitude speedup for RMAT graph with 2^{24} vertices.

Also, in [60], Nguyen et al. focus on overlapping community detection and maintenance in mobile applications. However, the proposed approach is a centralized algorithm to maintain overlapping communities. It is neither distributed nor applicable to hierarchical community structure. In this thesis 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 in distributed dynamic graphs.

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 [61] [62] [63]. 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 $O(n)$.

2.3 Other Parallel Graph Algorithms

Parallel graph algorithms, on the other hand, have been studied extensively since the beginning of parallel computing era. Most of these studies, however, targeted static graphs [64] [25]. In the recent years the studies in this field gained momentum again due to the growing popularity of social media tools. To deal with the

scalability concerns graph algorithms were implemented on MapReduce framework [65] and its open source implementation Apache Hadoop [66] [67] [68]. By formulating common graph algorithms as iterations of matrix-vector multiplications, coupled with compression, [69] and [70] 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 [71] [68] [72]. Pregel [73] thus proposed a new parallel graph programming framework following the bulk synchronous parallel (BSP) model and message passing constructs. In Pregel, vertices are assigned to distributed machines and only messages about their states are passed back and forth. In our work, we achieved the same objective through coprocessors. Pregel did not elaborate, however, how to manage temporary data, if it is large, with a main memory implementation nor did it state if updates are allowed in its partitioned graph. Furthermore, by introducing a new framework, compatibility with MapReduce-based analytics is lost. Two Apache incubator projects Giraph [74] and Hama [75], inspired by Pregel, are looking to implement BSP with degrees of Hadoop compatibility. In addition to the above systems focusing mostly on global graph queries, plenty of needs exist for target queries and explorations, especially in intelligence and law enforcement communities. Systems such as InfiniteGraph [76] and Trinity [77] scale horizontally in memory and support parallel target queries well.

2.4 Graph-Aware Caching

Many major large scale applications rely on distributed key-value stores [78, 79, 80, 81]. Meanwhile, distributed graphs are used by many web-scale applications. An effective way to improve the system performance is to employ a cache system. Facebook utilizes memcached [82] as a cache layer over its distributed social graph. Memcached is a general-purpose distributed memory cache which employs LRU eviction policy [83] where it groups data into multiple slabs with different sizes. Neo4j [84] is a popular open-source graph database with the ability to shard

data across several machines. It provides two levels of caching [85]. The file buffer cache caches the Neo4j durable storage media data to improve both read and write performance. The object cache caches individual vertices and edges and metadata in a traversal optimized format. The object cache is not aware of graph topology and facilitates LRU as eviction policy. On the other hand, Facebooks distributed data store for its social graph [86], which is called TAO, is designed to serve as a cache layer for Facebook's social graph. It implements its own graph data model and uses a database for persistent storage. TAO is the closest work in the literature to our study. TAO keeps many copies of sharded graph regions in servers called Followers and provides consistency by using single Leader server per graph shard to coordinate write operations. TAO employs LRU eviction policy similar to memcached. Pregel [87] provides a system for large-scale graph processing, however, it does not provide a caching layer. It touches on poor locality in graph operations while we study on how to obtain high locality and achieve it through prefetching using graph topology information. Neither TAO nor other studies exploit graph characteristics but they handle graph data as ordinary objects. Thus, our study is novel in the sense that it exploits graph specific attributes.

Chapter 3

An Analysis of Social Networks based on Tera-scale Telecommunication Datasets

Human communication behavior is the root of the usage pattern in physical and virtual communication networks, including telecommunication (telco) networks and on-line social networks. While fixed-line phones and shared computers in homes and offices reflect family or colleague behavior, mobile phones and portable computers better reflect individual usage behavior. Technological developments in the last two decades have resulted in two significant trends in human behavior: 1) going frequently online and 2) owning personal mobile computing and communication devices. Thus, the end-user behavior of communication networks has changed from group behavior to individual behavior.

Human communication behavior is highly related to underlying social network relationships. Mobile phone communication patterns provide strong insights into human social relationships [88]. For instance, person A calls person B usually because of a social relationship, e.g., B is a friend of A or B does business with A. The more social interactions dominate communication networks and online media, the more user behavior on those networks is dominated by human social

relationships and networks. Hence, managing and planning today's communication networks require a deep understanding about user behavior on those networks and about their social structures.

Social network analysis tries to understand the characteristics a social network exhibits. The first and most-cited characteristic among others is *degree distribution* of nodes constituting a social network. A bulk of studies in the literature on this topic reports that power-law best fits with certain parameters [19, 20, 21]. Other studies, however, propose different statistical fit models [22, 23, 24]. Since current studies are limited by the used datasets from which their proposals are derived/obtained, it is necessary to explore the influence of dataset specific parameters on discovered social network characteristics. This observation motivates us to conduct research on degree distribution on larger scales to discover the parameters governing degree distribution in social networks. Among many current research issues to be investigated, we prefer this less studied problem which requires a complete dataset.

Therefore, we explore how

- network operator,
- network size,
- population density, and
- geographic location

affect degree distribution in social networks.

To investigate these issues, we perform degree analysis on different social networks derived from the telecommunication network call data of a country's[1] different mobile (GSM) and fixed-line (PSTN) telco operators. We obtain degree distribution results for these networks to understand how well existing distribution models fit reality.

[1]Data was provided on the condition of anonymization, including country anonymity.

The chapter proceeds as follows: In Section 3.1, we describe the dataset used in this study and highlight its unique features. In Section 3.2, we discuss the statistical modeling of degree distribution in social networks and report the results of our empirical analysis. We also provide an analysis and interpretation for each of the following factors, any or all of which may affect social network characteristics: network operator, network size, network density and network location. Then we provide structural properties of the communication network in Section 3.3. Finally, in Section 3.4, we conclude the chapter.

3.1 Dataset

Obtaining necessary and sufficient data is one of the most difficult steps in social network analysis. Until the current pervasive use of mobile phones, the lack of large-scale data has limited our knowledge regarding human relationships and social networks. Now, however, the situation has changed. Mobile phone companies can collect CDRs for all subscriber calls going through their networks, and this CDR database is the most exhaustive dataset to date on human mobility and social interactions. For billing purposes, GSM networks record the base station each mobile phone call is made from, and this data thus holds the details of individual user movements. Having almost 100% penetration of mobile phones, the GSM network can now function as the most comprehensive proxy of a large-scale social network available today [89].

The dataset used in this study covers all GSM (three networks) and PSTN (one network) CDRs for a whole country between 1 January 2010 and 31 January 2010[2]. Data is anonymized and used solely for this research. The structure of the data is presented in Table 3.1. Fig. 3.1 shows the list of data tables and the number of records in each table. Each table contains records belonging to one days CDRs for the three GSM networks for the one month. All the PSTN network CDRs for the month are stored in one data table. The dataset

[2] Unfortunately, we cannot make this dataset available due to a non-disclosure agreement signed.

contains $N \approx 5 \times 10^7$ nodes and $L \approx 3.6 \times 10^{10}$ links for the GSM networks, and $N \approx 1.4 \times 10^7$ nodes and $L \approx 1.9 \times 10^9$ links for the PSTN network. We modeled the network growth in our dataset and found out that the daily CDR volume grows linearly over time according to the following relationship: $cdr\_volume \sim 3.433e06 * day + 1.132e09$ form. This is a very slow growth-rate and it takes approximately 330 days to double the CDR volume. In this study we also refer to this dataset as the SNA (social network analysis) database.

Lack of large and comprehensive data was one of the main reasons for doubts behind social network claims like Milgram's six degrees of separation (his small-world experiment) [90]. Now, however, one can (with permission) access anonymized CDRs from all network carriers providing service in a country. Thus, we can extract information about social interactions and construct a social network of the whole country from data provided by all mobile and fixed-line operators. This situation has the following advantages over previous studies:

- To the best of our knowledge, the dataset we use is much larger than the largest dataset containing trajectories and social interactions analyzed to date [89].
- Our data represents all country communication interaction, which is free from bias for a particular operator, size, location or density.
- The data contains spatial positions so we can also analyze the effect of location on social networks.

We are aware of the following limitations of our dataset:

- It covers calls of a one-month period and therefore some infrequent links might be missing.
- It comprises data from only voice and SMS communications. People might be using many other communication channels including e-mails, instant messaging tools, smart phone apps, etc.

Consequently, our dataset does not contain whole social network but a projection of it. It also contains many non-social entities.

Table 3.1: Structure of the data used in this work

Field name	Value description
source	source party of communication: calling party
destination	destination party of communication: called party
operator	network operator ID
communication type	voice, SMS services, etc.
date time	time of communication in seconds resolution
duration	duration of communication in seconds resolution
cell ID	location of communication in connected base-station location resolution

3.2 Analysis

For a sound and complete understanding of degree distribution in a large-scale social network, we investigate the effects of the following factors: 1) network operator to which the dataset belongs; 2) size of the community network; 3) population density; 4) geographic location where the community live. For each factor, we perform an analysis to determine how it affects degree distribution.

3.2.0.1 Distribution Model Fitting

For each hypothesized distribution, we modeled datasets with the distribution and then solved least-squares estimates of the distribution parameters of the nonlinear model using Gauss-Newton algorithm [91]. We used the R language [92] for statistical computations and graphics. We used the internal statistical functions of R and wrote many R scripts to make the necessary computations, model fittings and

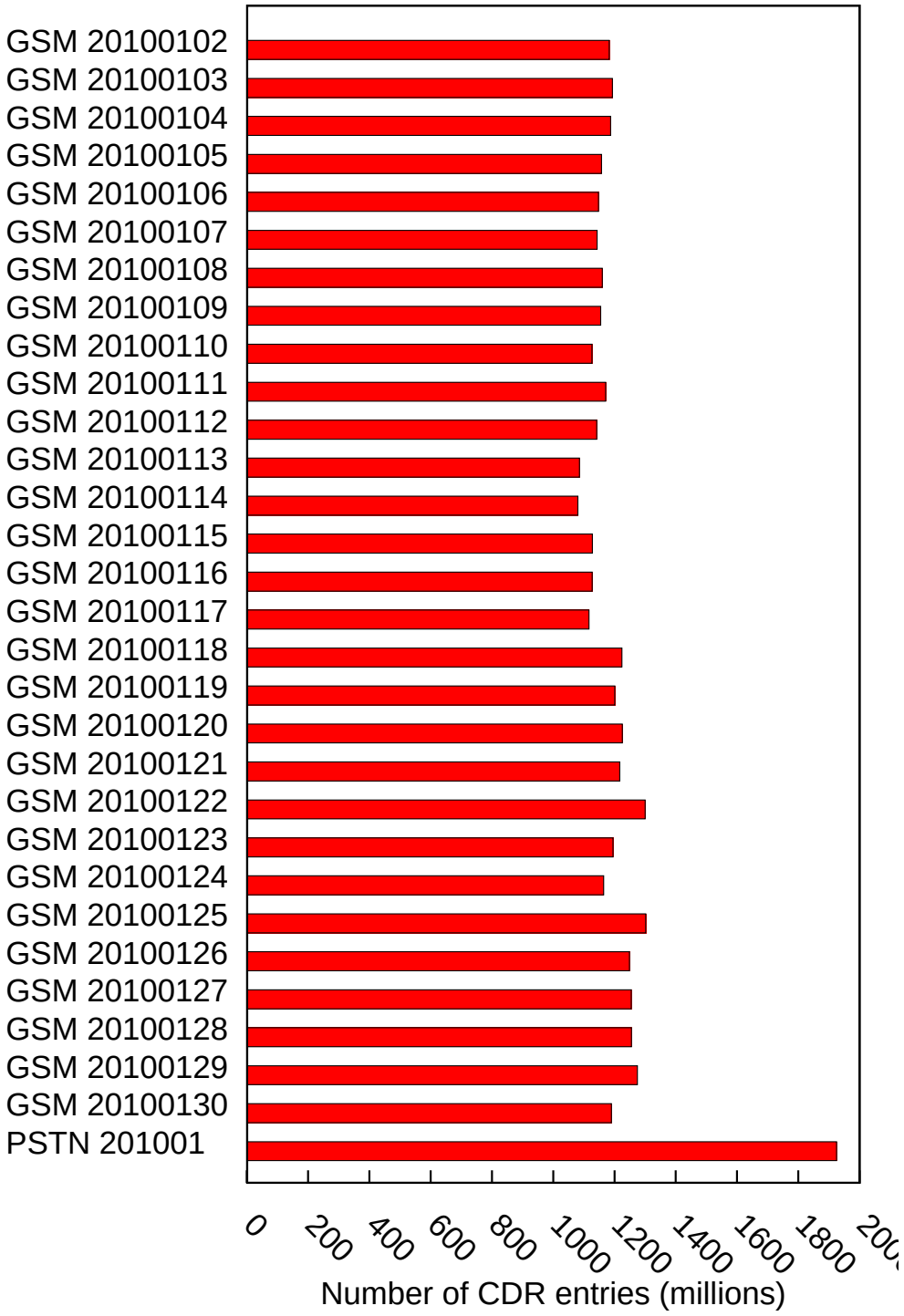


Figure 3.1: CDR data tables and number of entries in each table. There are approximately 1.19 billion records in each of daily GSM tables while there are 1.93 billion records in monthly PSTN table.

graphical plots. All analysis code including our fitness function implementation is available online[3]

For a set of n points $(x_i, y_i), i = 1 \ldots n$ where (x_i) is independent variable and (y_i) is known from dataset, let fitting model function be $m(x, parameters_p)$ which guesses y value for given x value for parameters $parameters_p$. Then modeling error (distance) is the residual sum-of-squares RSS_p as

$$RSS_p = \sum_{i=1}^{n} (y_i - m(x_i, parameters_p))^2$$

The employed Gauss-Newton algorithm computes model parameters which minimize the residual sum-of-squares measure.

3.2.0.2 Goodness-of-fit

In order to compare different distributions, we need a method to measure how good a hypothesized distribution fits to given dataset. The distance between the distribution of the empirical data and the hypothesized model is the base of *goodness-of-fit test* [93]. In this study we use "distance" to be the residual sum-of-squares.

To compute model fit success (p-value), we first compute normalized distance, then subtract it from 1. Thus we get a p-value which measures how tight the model fits the real dataset. A large p value indicates better fit to the empirical data.

$$\begin{aligned} TSS &= \sum_{i=1}^{n} (y_i)^2 \\ normalized_R SS &= RSS/TSS \\ p - value &= 1 - normalized_R SS \end{aligned}$$

[3]see `www.cs.bilkent.edu.tr/~haksu/callgraph/`.

3.2.0.3 Working With Large Datasets

We encountered some limitations while working with large datasets. Initially we started with a commercial relational database management system (RDBMS) on high-end hardware with $\sim$ 45 terabyte disk, 24 CPU cores and 96 GB memory. Extract, transform, and load processes take long time (i.e., days) and require careful performance tuning. Using this RDBMS solution, we are able to compute and export the degree distributions used in Sections 3.2.2, 3.2.3, 3.2.4, 3.2.5. 8 GB memory is sufficient for R programs to compute our fitting models, statistics and plots. On the other hand, relational databases perform poorly on graph traversal operations, i.e., multiple self-joins of large edges table become computationally infeasible. In order to be able to compute traversal-based network properties (e.g., clustering-coefficients) we setup a Hadoop/HBase cluster and loaded our dataset into HBase tables. We then implemented network analysis algorithms for graphs stored in HBase (see [94] for used platform details). Hadoop/HBase cluster solution enables us to compute the network properties reported in Section 3.3.

3.2.1 Social Network Modeling

A call graph is a projection of a social graph and reflects some properties of it (i.e., a call graph is considered to reveal citizens social interactions). Our dataset consists of call traces from the one PSTN and the three GSM operators in the country. Hence, we separately construct call graphs of the whole country for the three GSM operators and one PSTN operator. We also construct a call graph of the whole country for all GSM networks. Then we try to analyze degree distribution characteristics. We first compute the degree distribution of the call graph with no filtering. We call such a network *0-Core network*. Then we filter out automated one-way calls which may not imply a work-, family-, leisure- or service-based relationship [20]. To eliminate the automated calls, we use our so-called *1-Core network* (reciprocal network) to also characterize degree distribution. Each pair of nodes (A, B) in the 1-Core network has an edge if and only if A has called B and B has called A at least once in the observation duration. Please note that,

this filtering eliminates only non-social entities which make one-way calls. Still there may be many non-social entities in the dataset like customer support lines, business lines.

When we plot the degree distributions (i.e., degree versus frequency of appearance of that degree in the call graph) on linear x-y scales, all distributions resemble an L shape (the curve quickly declines and most of the x-axis is close-to-zero valued). Visually, it is hard to interpret behavior from these plots. If we plot the degree distributions in log-log scales, however, the plots are easier to follow. Thus, we use log-log plots in this study. Degree distributions in Fig. 3.2 are heavy tailed until a certain degree; then it takes an out-of-pattern fat-tail like shape. This means that the probability of having very high degree nodes is higher than what you would expect under a model fitting low-degree nodes. In Fig. 3.2 (a) we see a slope change around degree 5000 where $1/10^6$ of the nodes are covered. We can see similar situation in parts (b), (c), and (d). Nodes with large degree present a particular behavior, we think this is caused by non-social entities (e.g., business related phone numbers, customer support lines, etc.). Comparing 0-C GSM, 1-C GSM, 0-C PSTN and 1-C PSTN graphs, we see that out-of-pattern vertex ratio is higher in the PSTN network than the GSM network. Also in both PSTN and GSM networks, 1-C networks show lower out-of-pattern vertex ratio compared to 0-C networks. This observation support that out-of-pattern vertices are business phones or automated agents since 1-C networks cover less number of such non-social entities.

The literature related to degree distribution in call graphs and social networks includes various works on power-law distributions, power-law with cutoff distributions, log-normal distributions, exponential distributions, DPLN distributions and PLN distributions. All these distributions are possible candidates to statistically model degree distribution in a complex network with an L-shape-like degree-frequency distribution. Table 3.2 provides some general information about these distributions.

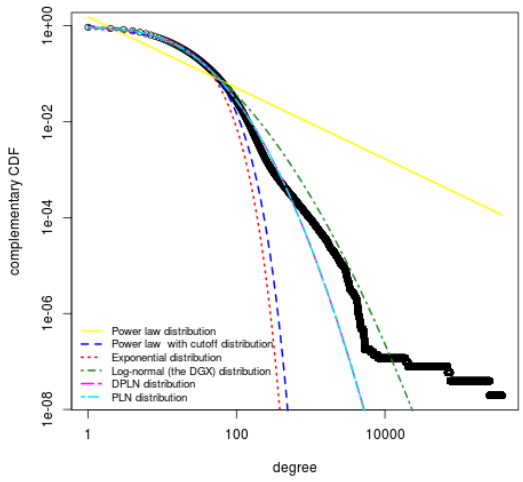
For each constructed social network (call graph) in our dataset, we try to fit all

Table 3.2: Definitions of several common statistical distributions referred to in SNA studies

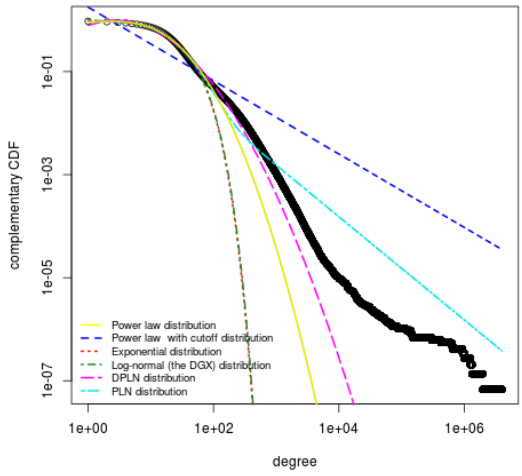
Distribution Name	Probability Density Function (pdf)	Parameters
Power-law	$pdf_{power-law}(x) = x^{-\gamma}$	γ
Power-law with cutoff	$pdf_{power-law\ with\ cutoff}(x) = x^{-\gamma}e^{-\lambda x}$	γ , λ
Log-normal	$pdf_{log-normal}(x) = \frac{1}{x\sqrt{2\pi\sigma^2}}exp[-\frac{(log(x)-\mu)^2}{2\sigma^2}]$	μ, σ
Exponential	$pdf_{exponential}(x) = \lambda e^{-\lambda x}$	λ
Double Pareto log-normal	$pdf_{DPLN}(x) = \frac{(\alpha\beta)}{(\alpha+\beta)}[e^{\alpha\nu+\frac{\alpha^2\tau^2}{2}}x^{-\alpha-1}\Phi(\frac{log(x)-\nu-\alpha\tau^2}{\tau}) + x^{\beta-1}e^{-\beta\tau+\frac{\beta^2\tau^2}{2}}(1-\Phi(\frac{log(x)-\nu+\beta\tau^2}{\tau}))]$	α, β, τ, ν
Pareto log-normal	$pdf_{PLN}(x) = \beta x^{\beta-1}e^{(-\beta\nu+\frac{\beta^2\tau^2}{2})}\left(1-\Phi\left(\frac{log(x)-\nu+\beta\tau^2}{\tau}\right)\right)$	β, τ, ν

candidate distributions and compute their goodness of fit. Fig. 3.2 shows GSM 0-Core, GSM 1-Core, PSTN 0-Core and PSTN 1-Core network fit results. In GSM 0-Core and 1-Core networks, power-law distribution provides the worst fit, while DPLN and PLN provide the best fit. When we look at each operator network shown in Fig. 3.4 and Fig. 3.3, DPLN and PLN continue to be the best-fitting models.

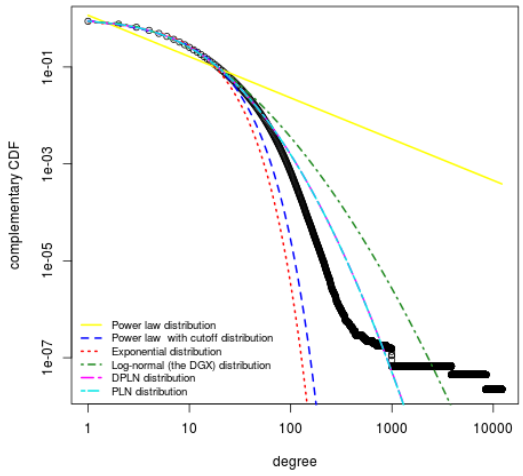
We also evaluate the fit success of these distribution models numerically. Table 3.3 summarizes the residual sum of squares (RSS)-based fit success values for each network-distribution pair. The best fits are shown in bold in the table. (See Section 3.2.0.2 on model fit success computation.)



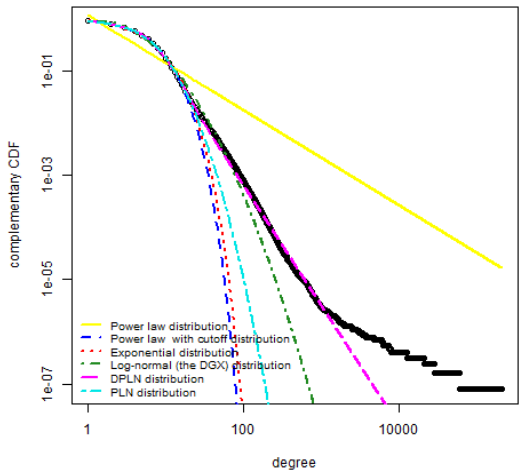
(a) 0-Core GSM ALL



(c) 0-Core PSTN ALL

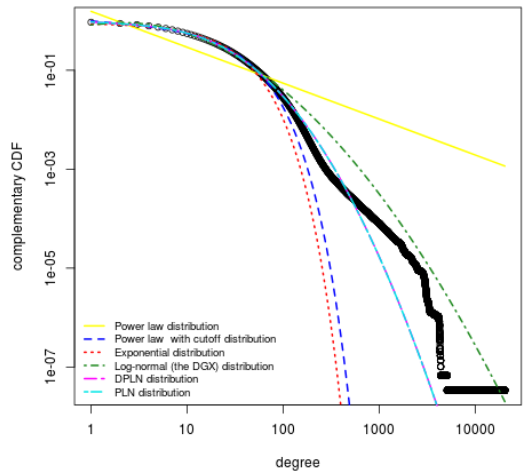


(b) 1-Core GSM ALL

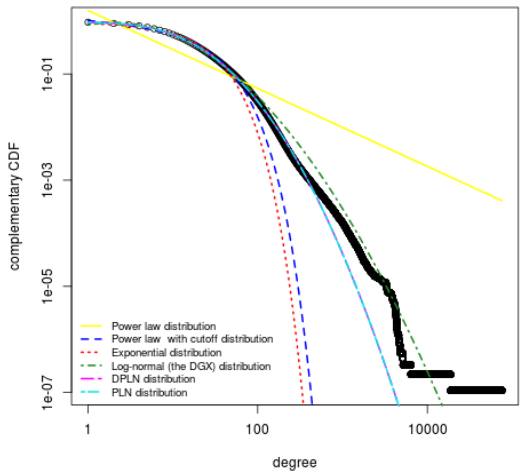


(d) 1-Core PSTN ALL

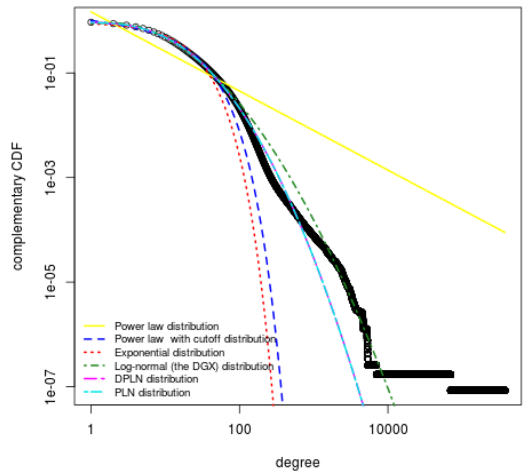
Figure 3.2: Network degree distributions and model fits for (a) 0-Core GSM ALL network (b) 1-Core GSM All network (c) 0-Core PSTN ALL network (d) 1-Core PSTN All network. Qualitative visual analysis suggest that PNL and DPLN distributions provides tightest fit while power-law distribution deviates most. See Table 3.3 for p-value based quantitative results.



(a) 0-Core GSM A

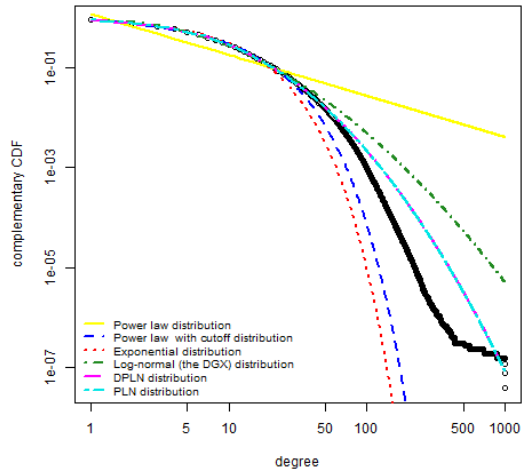


(b) 0-Core GSM B

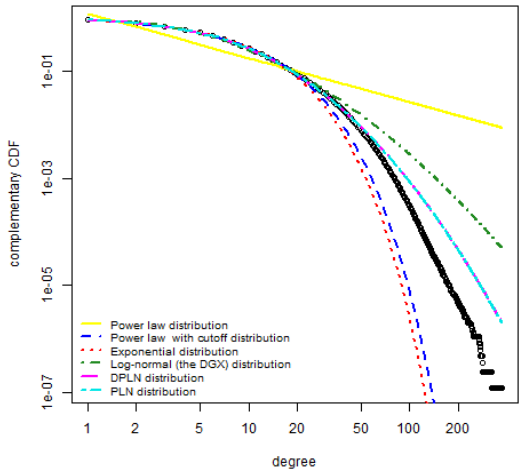


(c) 0-Core GSM C

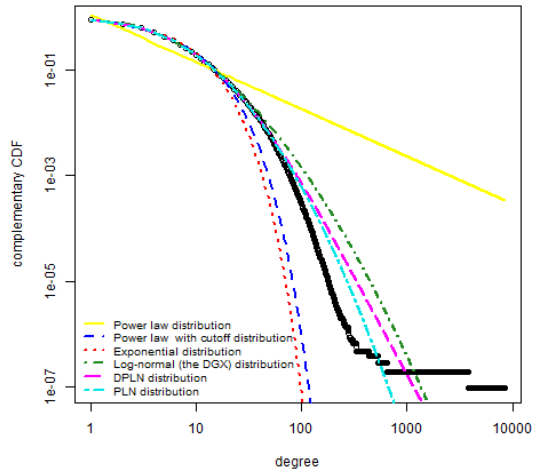
Figure 3.3: Model fits for 0-Core variations of GSM A, GSM B and GSM C networks are illustrated. In all networks DPLN and PLN models perform better then the rest of models. See Table 3.3 for p-value based quantitative results.



(d) 1-Core GSM A



(e) 1-Core GSM B



(f) 1-Core GSM C

Figure 3.4: Model fits for 1-Core variations of GSM A, GSM B and GSM C networks are illustrated. In all networks DPLN and PLN models perform better then the rest of models. See Table 3.3 for p-value based quantitative results.

Table 3.3: Numerical distribution fit success results for various networks

Network \ Distribution	Power-law	Power-law with cutoff	Exponential	Log-normal (DGX)	DPLN	PLN
1-Core GSM ALL	0.8597156	0.9980274	0.9983446	0.9954544	**0.9999636**	**0.9999639**
1-Core GSM B	0.8579531	0.9985913	0.9976061	0.9978552	0.9999707	**0.9999709**
1-Core GSM A	0.8579372	0.9981947	0.997876	0.9950699	0.9999429	**0.9999432**
1-Core GSM C	0.8799332	0.9977323	0.9991961	0.9961851	**0.9999637**	0.9999612
1-Core PSTN ALL	0.8473295	0.9991812	0.9955966	0.9976018	**0.9999069**	0.9996437
0-Core GSM ALL	0.7714906	0.9966974	0.9953066	0.991538	0.999826	**0.9998263**
0-Core GSM B	0.7733198	0.994963	0.9966673	0.9902132	**0.9999488**	**0.9999488**
0-Core GSM A	0.7642553	0.997863	0.9933416	0.993648	0.9997411	**0.9997416**
0-Core GSM C	0.7957198	0.9938651	0.997852	0.9879222	**0.9997517**	**0.9997517**
0-Core PSTN ALL	0.7228171	0.986819	0.9904483	0.9867846	**0.9969739**	0.9946071

The fit success results in Table 3.3 put forward two distributions: DPLN and PLN. The former provides the best fit for three social networks (0C PSTN, 1C PSTN and 1C GSM C), while the latter provides the best fit for four social networks (0C GSM A, 0C GSM ALL, 1C GSM Aand 1C GSM B). Both distributions provide equally good fits for three social networks (1C GSM ALL, 0C GSM B0C GSM C). There is no significant difference in their fit success; PLN is only slightly better than DPLN. Nevertheless, considering its lower number of parameters, we choose PLN distribution as the representative distribution (the best model) for our social network datasets. Hereafter, when we need to model a network, we use PLN.

3.2.2 Network Operator

By comparing the degree distribution characteristics of social networks derived from different operator data, we try to answer the question of whether characteristics are dependent on network operators or not. Doing so will clarify if investigating one operators social network of users is sufficient for social network analysis.

To analyze the effect of network operator, we again use the social networks constructed in Section 3.2.1, i.e., three GSM operators' social networks, one PSTN operator's social network and the GSM operators' joint social network. Fig. 3.5 illustrates and compares degree distribution in the GSM and PSTN networks. The former displays a higher density for lower degrees, while the latter displays a higher density for degrees larger than 122. We think that the high density for higher degrees in the PSTN network might be because fixed-line phones are used as household items rather than personal belongings, and are shared by many members in the house. Thus, PSTN node degrees can be considered as the sum of social degrees of multiple individuals. Fig. 3.6 shows the degree distributions of the various GSM operator networks. We can see that there is no significant difference between degree distributions of the three GSM operators networks and

the joint network derived from the three operators. We also apply the Kruskal-Wallis Test to compare the degree distribution of complex communication networks breakdown by network-operator. As the result of this test, the p-value turns out to be greater than the 0.05 significance level (p-value=0.84). Hence, we conclude that with 95% confidence the degree distributions of the analyzed social networks at network-operator breakdown are statistically identical.

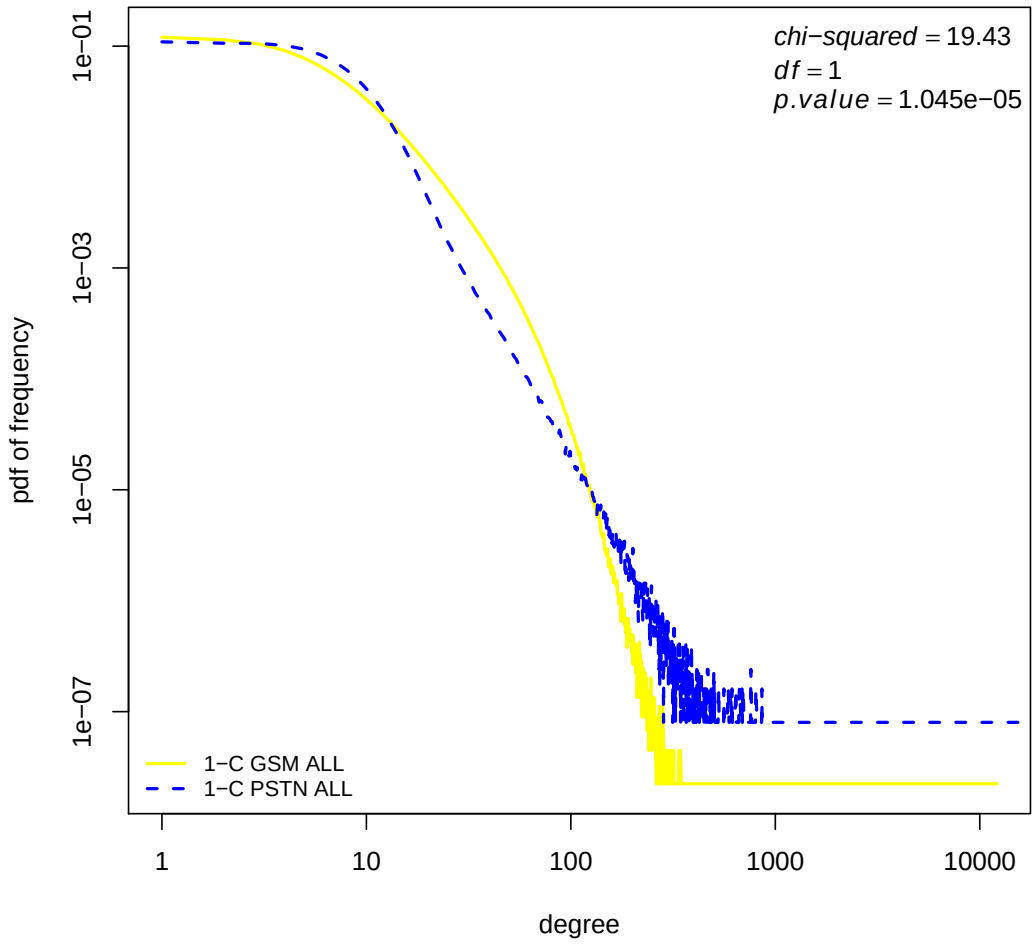


Figure 3.5: 1-Core GSM and PSTN network operators degree pdf distribution. Test shows that GSM and PSTN are not identical distribution at 0.05 significance.

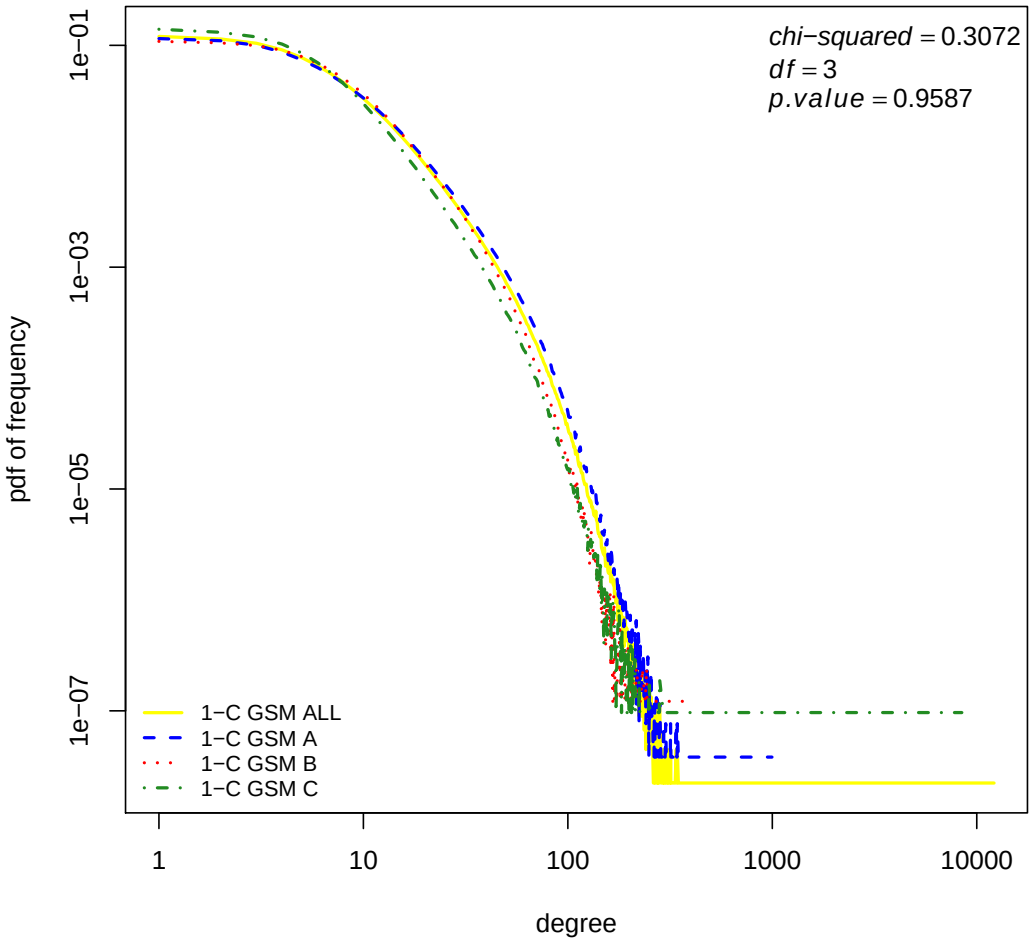


Figure 3.6: Degree distributions for different network operators are compared. Degree distributions are statistically identical for different network operators.

3.2.3 Network Size

To analyze the effect of network size on degree distribution, we start with a network around one base station and then expand it by including neighbor base station networks, just like snowball sampling (as described in Algorithm 3.1). Thus, we construct social networks of different sizes for a city.[4] Then for each social network of a different size, we compute and plot the corresponding degree distribution, resulting in a chart of network size versus degree distribution parameters.

Algorithm 3.1. *NetworkSizeEvaluation*

Input: **K**-List of base stations in neighborhood order.
N-Whole-city network.
Output:**ctable**-degree characteristic, network size, value table.

1: $sn \Leftarrow empty\ network$
2: **for** $i = 1 \rightarrow size(K)$ **do**
3: $s \leftarrow generateNetwork(N, K[i])$
4: $sn \leftarrow sn \cup s$
5: $v \leftarrow computeDegreeCharacteristic(sn)$
6: $ctable[size(sn)] \leftarrow v$
7: $analyze(ctable)$ **return** $ctable$

To obtain networks of various sizes, we use the SNA database, which has the cell IDs and geographic coordinates of the GSM base stations. We divide a dense urban part of city Xinto 1000 sub-parts, which host an approximately equal number of cell phones (users). Using Google Maps, we determine the coordinates of the urban part of city X. The dataset lists around 17000 base stations in this region, so each sub-part hosts about 17 base stations. Starting from a point in the city, we draw circles around the nearest 17 base stations and label the circles from 1 to 1000. Thus, in each iteration we draw a new circle around the nearest 17 base stations that are not yet covered by a circle as shown in Fig. 3.7.

Having 1000 circles determined, we start to filter the calls in these circles so that we have networks with an increasing number of nodes inside. We define a

[4] As part of anonymization, we refer to the chosen city as city X.

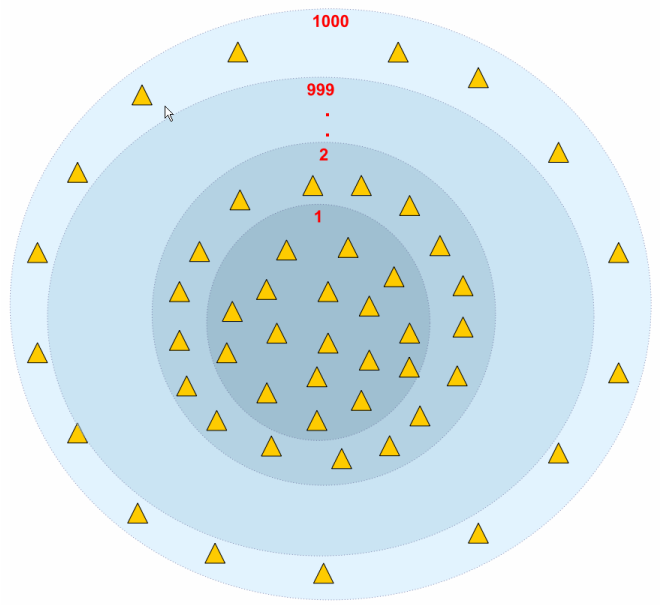


Figure 3.7: 1000 circles around base stations. Each circle is drawn to cover the nearest 17 base stations that are not yet covered by a circle.

ring as a circle containing all other circles with a label lower than its label. More precisely, $ring_N$ is the set of circles C_m, where $m \leq N$. In this manner, 1000 rings ($ring_1 \ldots ring_{1000}$) are defined. By filtering the calls established in each ring, we come up with 1000 networks that differ only in size (i.e., density, location, etc., are not considered).

To determine whether there is any effect of size on degree distribution we plot the pdf of degree versus network size. Since there are 1000 networks with increasing size, in order to make the plot easier to interpret we create a color list with a gradient of 1000 green-blue-red colors. As illustrated in Fig. 3.8, for increasing network size, the degree distribution curves in a specific direction: the pdf for low degrees decreases while the pdf for high degrees increases. We also apply the Kruskal-Wallis Test to compare the degree distribution of complex communication networks breakdown by network-size. As the result of this test, the p-value turns out to be less than the 0.05 significance level (p-value=5.122e-5). Hence, we conclude that the degree distributions of the analyzed social networks at network-size breakdown are statistically nonidentical.

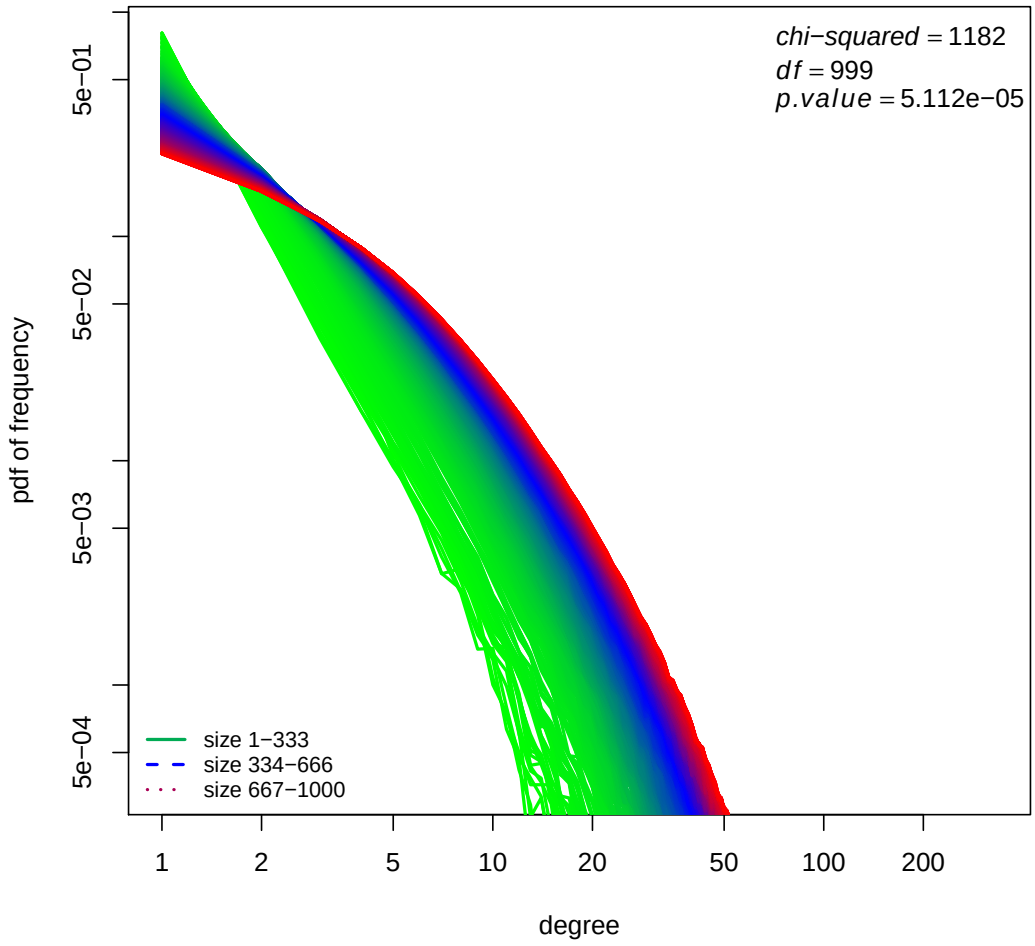


Figure 3.8: Degree distribution for increasing network size. Size unit is 17 base station, e.g., 100 means network size is 1700 base stations. Degree distribution for 1000 samples are plotted with gradient colors in green-blue-red range to visually follow network size v.s distribution shape change. Statistical test reject the hypothesis claiming that degree distributions for varied sized networks are identical.

To further investigate the effect of network size, we fit the PLN distribution to all 1000 networks with increasing size. Then we analyze each PLN distribution model parameter against the change in size. The PLN distribution has the following pdf function:

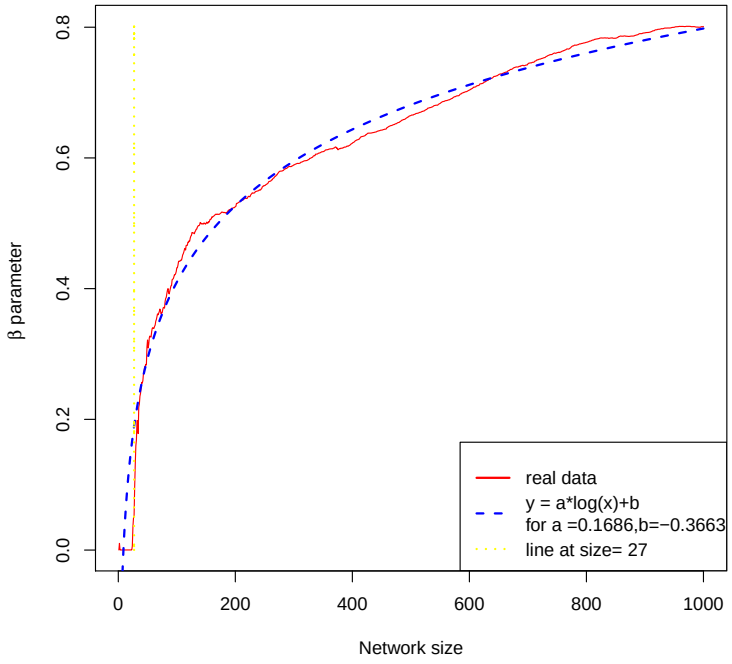
$$pdf_{PLN}(x) \quad = \beta x^{\beta-1} e^{(-\beta\nu+\frac{\beta^2\tau^2}{2})} \left(1 - \Phi\left(\frac{log(x)-\nu+\beta\tau^2}{\tau}\right)\right)$$

and $E[X] = \nu - \frac{1}{\beta}$.

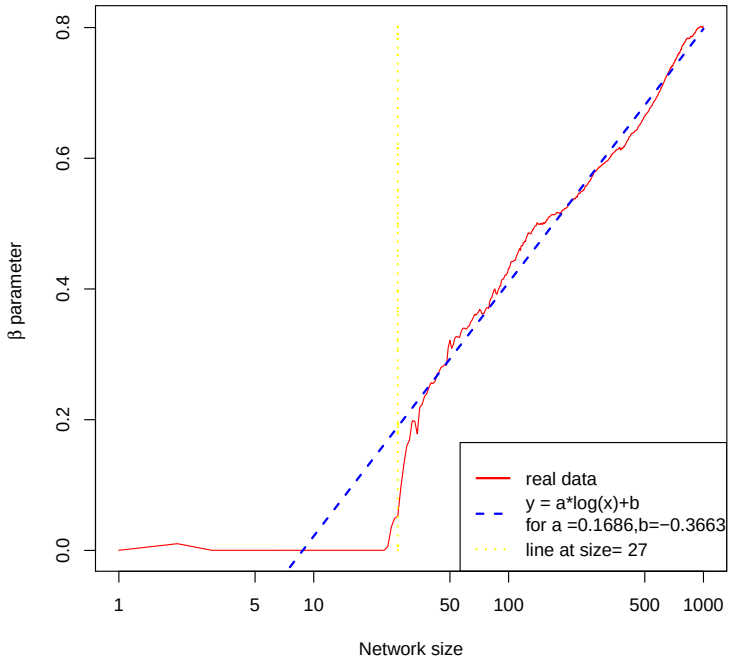
Fig. 3.9 shows the β parameter behavior of the PLN distribution as a function of network size. The linear-log scale figure indicates that $\beta \sim log(size)$. Thus, when we try to fit $\beta = a * log(size) + b$ to the results, we get a tight fit as illustrated by the blue dashed line.

We get similar results when we focus on PLN distribution ν parameter behavior versus network size. Fig. 3.10 demonstrates the $\nu \sim log(size)$ relationship, where $\nu = a * log(size) + b$ fit is represented by the blue dashed line. Since $E[X] = \nu - \frac{1}{\beta}$, considering the $\nu \sim log(size)$ and $\beta \sim log(size)$ observation, we conclude that the average degree of observed networks is proportional to the logarithm of the network size.

Following green-blue-red transition in Fig. 3.8 size v.s. degree distribution, we see that the distribution function shape changes from a line into a curve while the size of network increases. This empirical result does not follow power-law generating evolution models discussed in [95]. We know that our dataset is composed of both social and non-social (complex) entities. Considering the evolution of complex networks study, we think that while complex network entities follow preferential attachment, social entities do not, due to the natural upper-bound on a node degree. Therefore, small-size samples might result in overestimating the density of popular nodes where this natural upper bound is not hitted. For instance, the average number of received calls (in-degree) is less than 2 in the telephone call graph sample analysed in [95]. Thus, power-law fit for in-degree in this case may not remain valid for a larger sample. In fact, the study reports that it was impossible to fit out-degree by any power-law dependence.

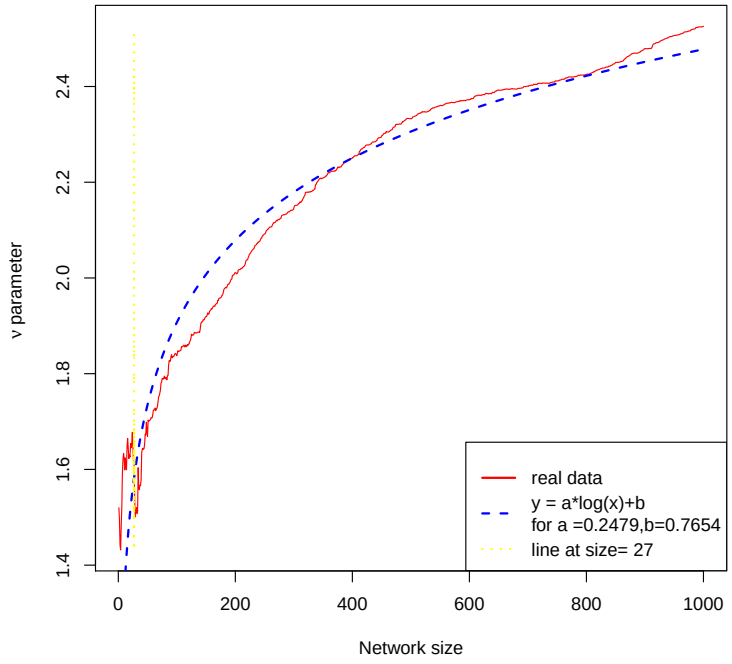


(a)

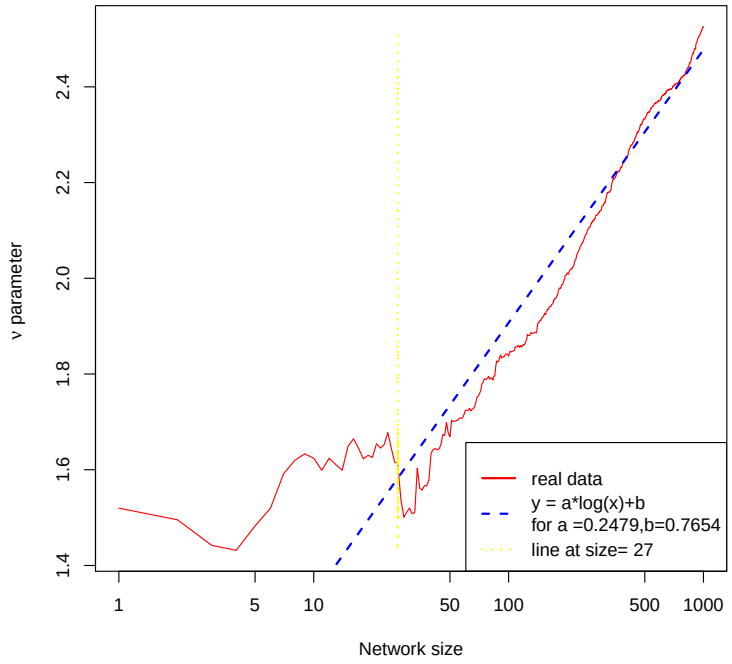


(b)

Figure 3.9: PLN β parameter versus network size in (a) linear-linear and (b) linear-log scale.



(a)



(b)

Figure 3.10: PLN ν parameter versus network size in (a) linear-linear and (b) linear-log scale.

3.2.4 Population Density

Algorithm 3.2. *NetworkDensityEvaluation*

Input: **B**-List of geographical locations in increasing density order.
N-Whole-city network.
Output ctable-degree characteristic, network density, value table.

1: $sn \Leftarrow empty\ network$
2: **for** $i = 1 \rightarrow size(B)$ **do**
3: $s \leftarrow generateNetwork(N, B[i])$
4: $sn \leftarrow sn \cup s$
5: $v \leftarrow computeDegreeCharacteristic(sn)$
6: $ctable[density(sn)] \leftarrow v$
7: $analyze(ctable)$ **return** $ctable$

Here we aim to understand the effect of population density (number of users in a geographic region) on degree distribution in social networks. We would like to see whether, for example, a denser region has a denser social network. For this analysis, we again use the SNA database with GSM base station cell IDs and geographic coordinates. We draw a rectangle that incorporates the dense urban area and naighbouring sparse rural areas. We divide the rectangle into 10 parts with approximately equal population sizes. The entire rectangle covers nearly 450 base stations, therefore, starting from the city center, each of 45 base station cells are grouped as a ring (see Algorithm 3.2). Then, by filtering the calls made in each ring, we get 10 social networks. For each ring, density is computed as the number of base stations per kilometer square.[5]

Fig. 3.11 shows the degree distributions for social networks of different densities. These distributions have no specific behavior regarding increasing network density. All distributions are close to each other and they cross many times. The highest-density line (dashed blue line) falls in the middle of all the density lines. We also apply the Kruskal-Wallis Test to compare the degree distribution of complex communication networks breakdown by network-density. As the result of this test, the p-value turns out to be greater than the 0.05 significance

[5]Because base stations are located with a density proportional to population density, we consider base station density to be a measure of population density.

level (p-value=0.98). Hence, we conclude that with 95% confidence the degree distributions of the analyzed social networks at network-density breakdown are statistically identical.

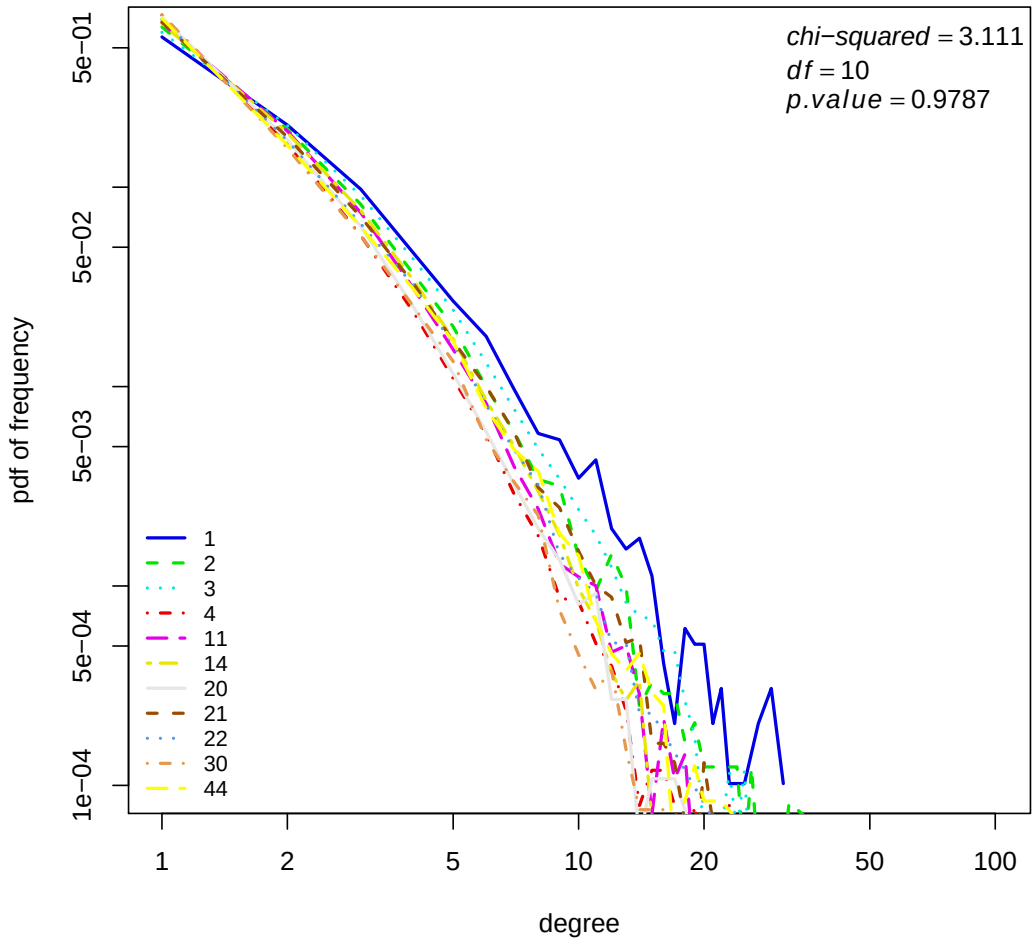


Figure 3.11: Network degree pdf versus network density plots.

3.2.5 Geographic Location

Next, we aim to understand the impact of geographic location on degree distribution characteristics. We investigate how degree distribution in social networks

changes when the networks are physically located in different places. As before, we use the SNA database, GSM base stations and geographic coordinates. For analysis, we construct some social networks for which the geographic locations are different, but network size, density, etc. are similar. To derive such networks, we choose cities with similar population sizes from different parts of the country. We sort all cities in the country by the number of base stations they have, and then we look for a consecutive sub-list of cities located as far apart as possible but with a similar number of base stations. As illustrated in Fig. 3.12, we choose 10 such cities, each having 1000 ± 100 base stations. We filter the calls made in each city and then construct 10 social networks.

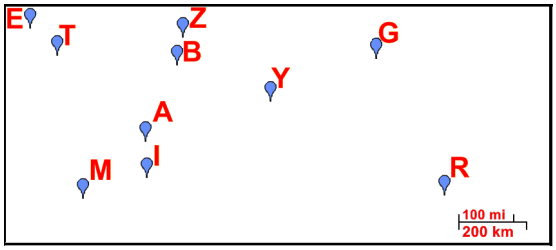


Figure 3.12: Locations of chosen cities in the country.

Fig. 3.13 shows degree distributions of the social networks of the selected cities. The anonymized list of cities north to south is: E, Z, G, T, B, Y, A, I, M, R; and west to east is: E, T, M, I, A, B, Z, Y, G, R. As can be observed from the figure, degree distribution curves are very close to each other and there is no specific curve behavior following city locations.

We also apply the Kruskal-Wallis Test to compare the degree distribution of complex communication networks breakdown by network-location. As the result of this test, the p-value turns out to be greater than the 0.05 significance level (p-value=0.99). Hence, we conclude that with 95% confidence the degree distributions of the analyzed social networks at network-location breakdown are statistically identical.

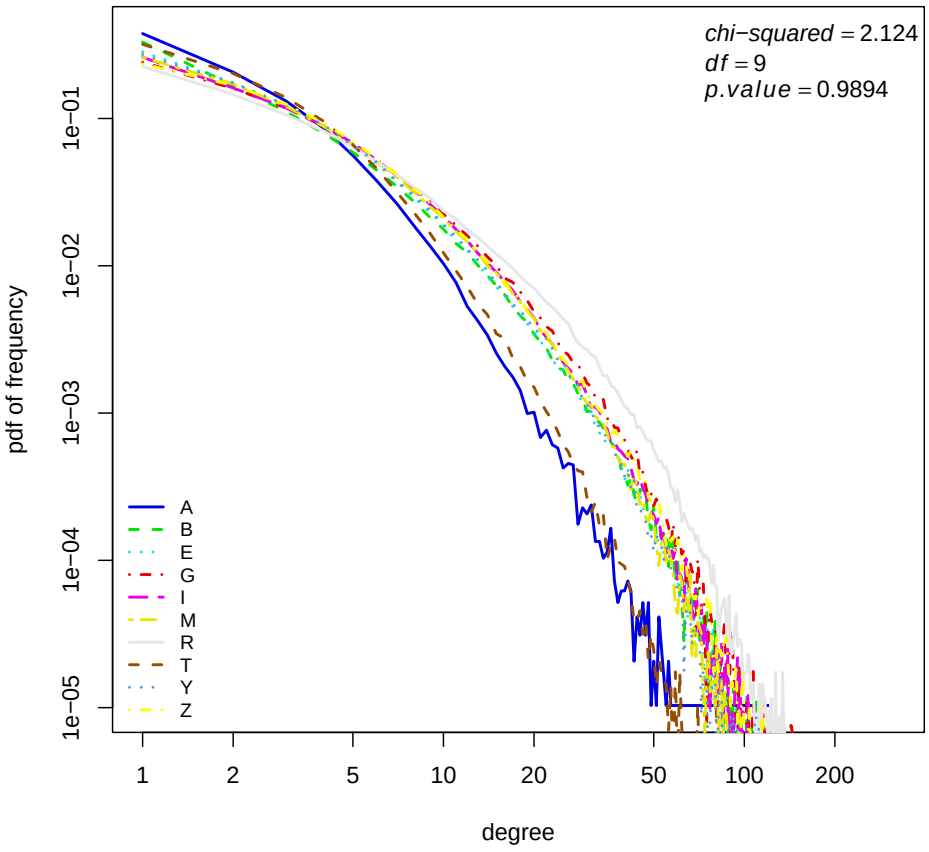
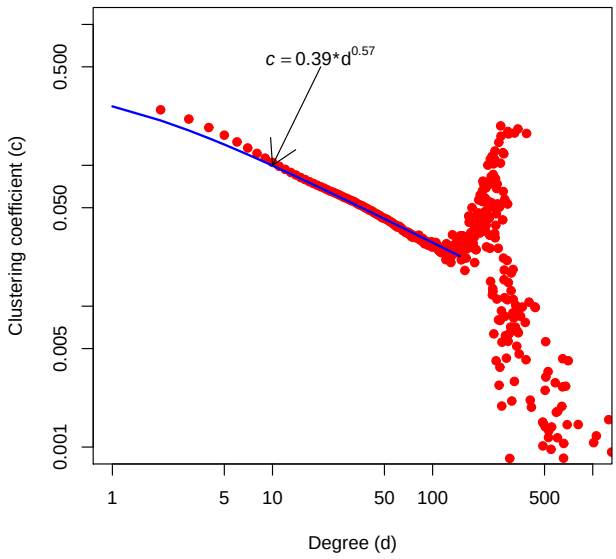


Figure 3.13: Network degree pdf versus network location.

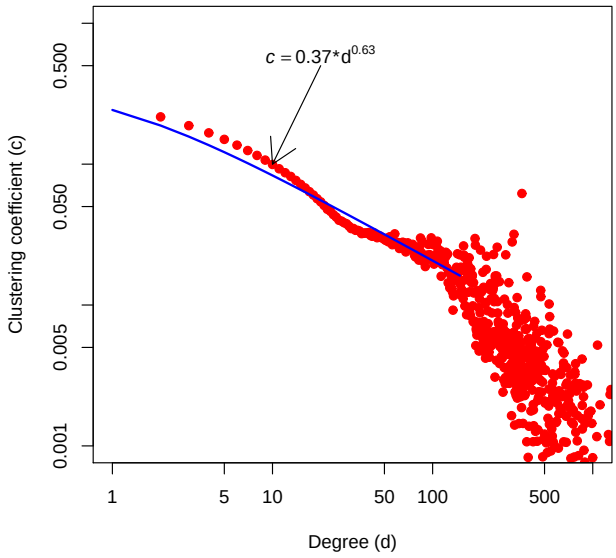
3.3 Structural Properties of the Communication Network

So far we have examined the effects of certain parameters on degree distribution. We now construct a general communication network from the dataset and analyze it for structural properties. *Clustering coefficient* is defined as the fraction of triangles around a node. This measure says how well a node's neighbors are connected. Social networks are known to have large clustering coefficients. Fig. 3.14 displays the clustering coefficient values as a function of the degree of a node for GSM and PSTN networks. The clustering coefficient decays slowly with exponent -0.37 ($c \sim d^{-0.57}$) with the degree of a node till degree d (~ 150), and then scatters around. Results on web graphs and theoretical analysis on hierarchical networks report decays with exponent -1 [96], while results on Messenger network report decays with exponent -0.37 [97]. Comparatively, our results suggest that clustering in phone call graphs is much higher than the theoretical expectation and web graph results, however, it is lower compared to the clustering in Messenger communication graph. In other words, phone users with common friends tend to be connected more probably than the theoretical expectation, and connected less probably than Messenger users with common friends. Scattering after a certain degree d (~ 150) implies that neighbors with high degree nodes know each other less, thus such nodes are non-social entities like customer support lines.
Fig. 3.15 displays size distribution of connected components in networks. Over 99% of the nodes belong to the largest connected component, and the remaining small components show a power-law like distribution. We further study *community structure* in the networks by computing k-core decomposition of the graph. k-core decomposition is a subgraph density measure and it identifies dense regions in the graph[6]. Fig. 3.16 displays the distribution of k-core sizes for (A) GSM and (b) PSTN networks. The decay in k-core sizes is stable up to a cutoff value ($k_{pstn\_cutoff} \approx 5$ in PSTN and $k_{gsm\_cutoff} \approx 12$ in GSM), then the k-core size drops rapidly which means that the nodes with degrees of less than the cutoff value are on the fringe of the network. This structure is similar to the Messenger

[6]The k-core of a graph is a subgraph K, where each vertex in K has at least k edges to other vertices in K.

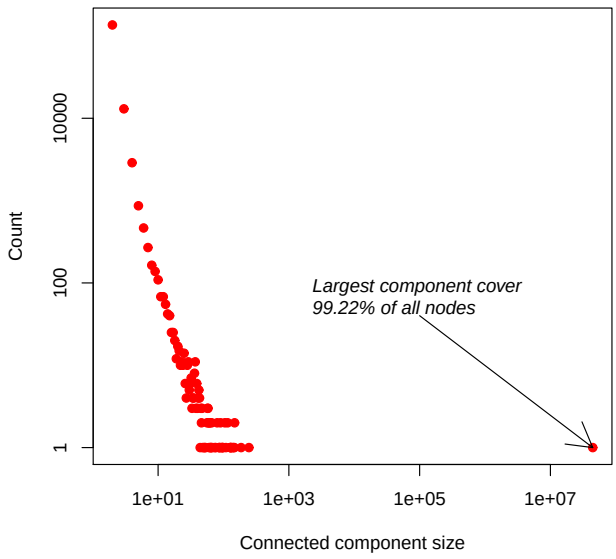


(a)

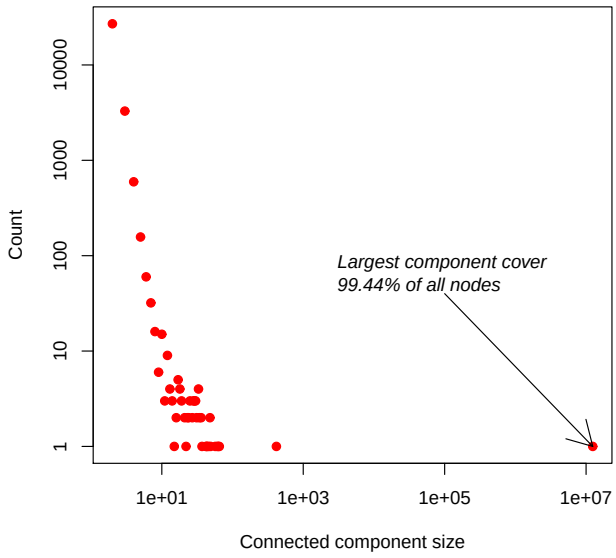


(b)

Figure 3.14: Average clustering coefficient distribution versus node degree for (a) 1-Core GSM and (b) 1-Core PSTN networks. Clustering coefficients decay with node degree with exponents (a) -0.57 and (b)-0.63, respectively. Variance increases after $d \sim 150$ where non-social entities appear more. Neighbors of non-social entities tend to know each other with high instability.

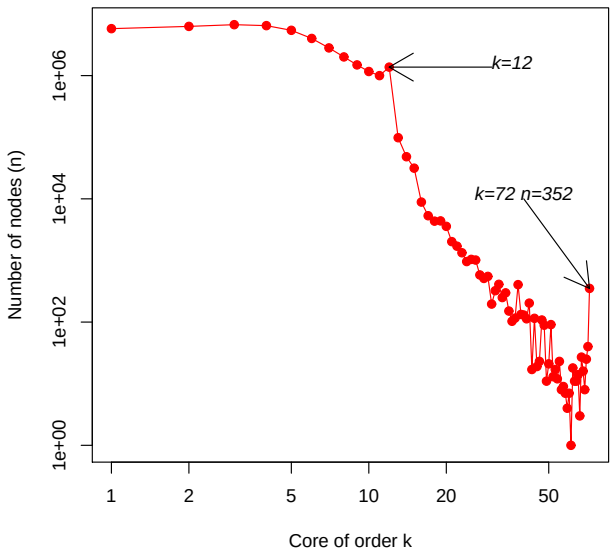


(a)

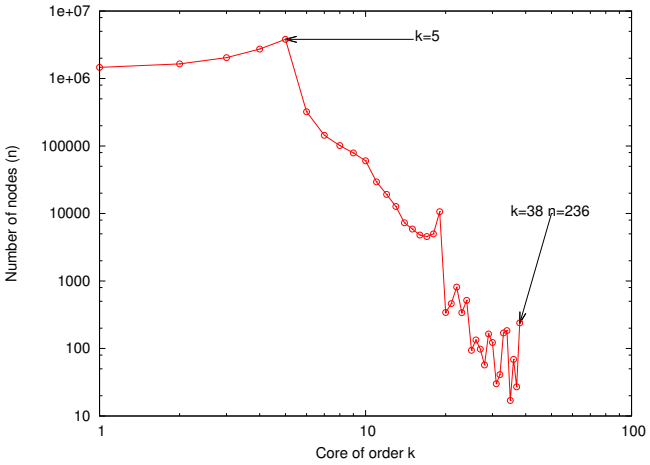


(b)

Figure 3.15: Distribution of connected components in (a) GSM (b) PSTN networks. Over 99% of the nodes belong to the largest connected component. Many small components exist against a few large components.



(a)



(b)

Figure 3.16: Size distribution of k-cores in (a) GSM (b) PSTN networks. The densest region in GSM network is composed of 352 nodes where each node has more than 72 edges inside the set, while the densest region in PSTN network is composed of 236 nodes where each node has more than 38 edges inside the set. The decay in k-core sizes is stable up to a cutoff value $k_{pstn\_cutoff} \approx 5$ in PSTN and $k_{gsm\_cutoff} \approx 12$ in GSM, and then the k-core size drops rapidly which means that the nodes with degrees of less than the cutoff value are on the fringe of the network.

communication network with $k_{msn\_cutoff} \approx 20$ [97], while it is quite different from the Internet graph in which k-core size decays as a power-law with k [98]. The densest region in GSM network is composed of 352 nodes where each of the nodes has more than 72 edges inside the set.

3.4 Conclusion

Collecting social network data is traditionally difficult, requiring extensive contact with the group of people being studied. Practically, research efforts are generally limited to between tens and hundreds of individuals [99, 100]. On the other hand, social interactions over telco infrastructures generate detailed traces of interactions and movements. Large-scale networks, even ones covering a whole society, can be generated from such traces. The ability to construct such rich and representative social networks makes it feasible to develop and evaluate social network models.

Different observations exist regarding degree distribution in social networks. For instance, some works (e.g., [20, 101]) claim that the degree distribution follows power-law distribution, while others (e.g., [23]) claim it follows double Pareto log-normal distribution. Using different datasets, different degree distributions have been obtained. In this study, we attempt to empirically test degree distribution versus different dataset scenarios to understand the parameters governing degree distribution in social networks. We observe that degree distribution in social networks does not show a significant correlation with population density, user telco operator, and user geographic location; however, population size directly affects the average degree of social network. Therefore, it is important to keep social network size as a parameter while interpreting degree distribution. It also seems acceptable to study a social network without considering its location, density and referred telco operator. For instance, a researcher could gather data from an urban part or a rural part of a country, or may choose a specific city or telco operator. However, any change in the size of the studied network would result in a considerable change in degree distribution characteristics and overall network topology. Hence, social network studies must indicate the size of the studied network and consider different size cases to come up with a sound and complete conclusion.

Chapter 4

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

In our study we mainly focus on large dynamic graphs maintained by social media companies and some government agencies. In the last few years many web companies, such as Followerwonk, SocialPing, SimplyMeasured and Gravity, emerged for helping customers to make better marketing decisions based on the content of social media tools such as Twitter, Google+, Orkut, Youtube. Government agencies also have a growing interest in performing analytics on these datasets for event detection and tracking. These web companies and agencies have to deal with massively large graphs to run various analytics queries.

These graphs are considered as large not only because they have many vertices and edges but also they maintain significantly large amount of metadata associated with them. These graphs are also considered dynamic because the interactions between the users such as tweets, chats, messages are all considered as part of the graph which changes over time. Since all of these interactions are recorded for further use for analytics, an unprecedented growth rate is observed in the data size. Twitter for instance reports that the average number of tweets

per day is 58 million and 2.1 billion user queries are processed every day as of May 2013 [102]. Because of the massive size of these datasets and user load, many of these social web companies tend to store these graphs in distributed datastores such as Google BigTable, MegaStore, Apache HBase or distributed parallel databases, with motivations behind Big Data trend, i.e., high availability, fault-tolerance, scalability, persistence. The underlying storage system should be able to process the queries instantly to maintain site popularity.

Unfortunately, centralized solutions do not scale well when it comes to answering billions of user queries expecting instant answers. Another social media company, Facebook, recently announced that 1.11 billion users connect to the site every month and the average number of users per day as of March 2013 is 665 million. The user related metadata such as messages, chats, emails, SMS messages and attachments are stored on hundreds of HBase clusters. 6 billion messages are sent between Facebook users daily. At peak times 1.5 million operations are executed per second on the metadata associated with graph vertices and edges. To keep up with the scale of these datasets the companies are compelled to use distributed data architectures for storage and maintenance. This chapter[1] 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 [26], k-core, k-edge-connectivity [27], 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 [28], where other measures have complexity growing super-linearly or NP-hard. For practical considerations, our focus is to identify and maintain

[1] 2014 IEEE. Reprinted, with permission, from H. Aksu, M. Canim, Y. Chang, I. Korpeoglu, and O. Ulusoy, "Distributed k-core view materialization and maintenance for large dynamic graphs," *Knowledge and Data Engineering, IEEE Transactions on*, 1/2014.

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. 1.1 illustrates the degree distribution of nine published graph datasets, 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. 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 chapter is structured as follows. We 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 4.1. We formally define and introduce key k-core properties in Section 4.2. Section 4.3 describes our distributed k-core construction algorithms in naïve implementation and pruning techniques. Section 4.4 details our incremental maintenance algorithms for edge insertions and deletions. Section 4.5 makes further improvement for maintenance over batch window updates. Experimental results are reported and discussed in Section 4.6. Finally, Section 4.7 concludes the chapter.

4.1 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 [36]. Apache HBase is a non-relational, distributed data management system modeled after Google's BigTable [103]. 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 [104] and many others [105]. Fig. 4.1(a) depicts a simplified architectural diagram of HBase with several key components relevant to this study. 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. 4.1(a), 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.

HBase's coprocessor feature was first introduced in version 0.92, released in January 2012 [106]. Like HBase itself, the idea of coprocessors was inspired by Google's BigTable coprocessors. It was recognized that pushing the computation to the server where user deployed code can operate on the data directly without communication overheads can give further 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.

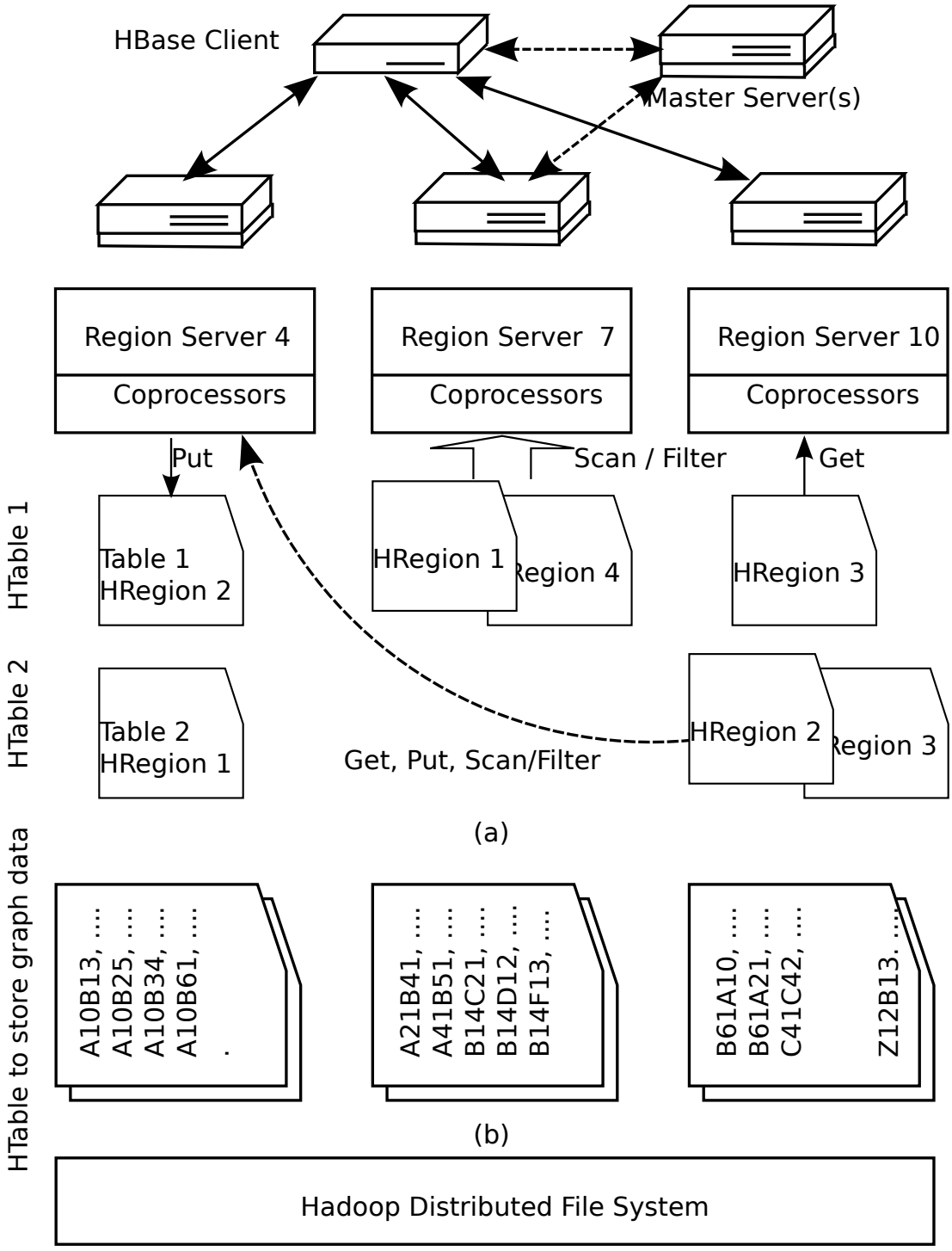


Figure 4.1: An HBase cluster consists of one or multiple master servers and region servers, each of which manages range partitioned regions of HBase tables. Coprocessors are user-deployed programs running in the region servers. They read and process data from local HRegion and can access remote data by remote calls to other region servers.

Fig. 4.1(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. For example, a CP can request to scan another table, at the expense of remote procedure calls (RPC). Similarly a CP can insert or delete rows into another table through RPC. The latter scenario usually applies when the results of CP processing are too large to be returned to the client.

The flexibility of CP introduced many possibilities to process local data in a targeted way. On the other hand, spreading the reads and writes across the cluster and incurring RPC penalty must be carefully worked into algorithm design as presented in the following sections.

We map the rich graph representation $G = \{V, E, M, C\}$ defined in Section 4.2 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. 4.1(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', 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 [107]. Note that, we use the terms partition and region interchangeably.

k-core algorithms in the study 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 below.

4.1.1 A Concrete Example of a Distributed Social Graph With Metadata

Lets consider a facebook like social networking application called XBook. XBook keeps users as vertices and theirs profiles as vertex metadata. Also, user messages and published files are stored as vertex content. Users social acts are stored as edges and act details are stored as edge metadata, i.e., user A is a friend of user B, user B likes company C, or user D likes photo P posted by user A. XBook uses HBase to store its rich graph on a Hadoop cluster using the schema described in

this study.

4.1.1.1 The Role of k-core Subgraph in Presence of the Metadata

k-core analytics simply provides the dense regions of a graph and the meaning of a k-core sub-graph depends on the initial graph from which it is derived. Rich graph storing all XBook data needs to be interpreted according to analytics targeted to the raw graph. Real world analytics inherently targets some interest domain graphs which emerge as projections in social network graph. For instance, a camera manufacturing company asks for a query like display my advertisement to ones in the k-core of users who live in the US, older than the age of 18 and have more than 300 photos which requires a projection of raw XBook graph based on which k-core will be constructed and maintained. Meanwhile a security company requests k-core of users who like video V and live in NY to monitor a crime-net evaluation via the particularly uploaded video V.

4.1.1.2 Advantage of Storing k-core in Distributed Sites

XBook stores all its data in an HBase cluster in distributed sites. All of the data related to a user is stored in the associated vertex metadata and content. Similarly, for each k-core materialized view maintained by the XBook, a field in vertex metadata holds vertex/k-core status. Whenever XBook application receives query related to a vertex, it is dispatched to the associated distributed site. Similarly, queries related to a vertex participating at a certain k-core subgraph are responded by the associated distributed site. Thus, XBook application handles the user related processing locally, i.e., constructs user customized pages holding k-core participation specific advertisements using minimal out-of-site communication. On the other hand, a k-core graph search is handled just like another graph search with metadata filtering. By keeping k-core data with vertex together, XBook accomplishes working with inherently distributed rich social graph without introducing a centralized single point of failure to the architecture.

4.2 Preliminaries

We define a rich graph representation G

$$G = \{V, E, M[V, E], C[V, E]\} \tag{4.1}$$

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 study 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. $N_G^k(v) \geq k$. □

We illustrate the relationship between a vertex's core number, its degree in the entire graph, its degree and neighbor count in the 2-core and 3-core subgraphs respectively in Fig. 4.2 and Table 4.1. From this simple example, it is easy to observe that a vertex with high degree, such as T, can have a low core number. For commonly known social graph structures that follow power law distribution, this suggests those vertices with very high degrees are likely to have a relatively small core number. This example also illustrates that as k increases, both d_{G_k} and N_G^k stay the same or decrease, which is a key property applied to our distributed construction and maintenance algorithms.

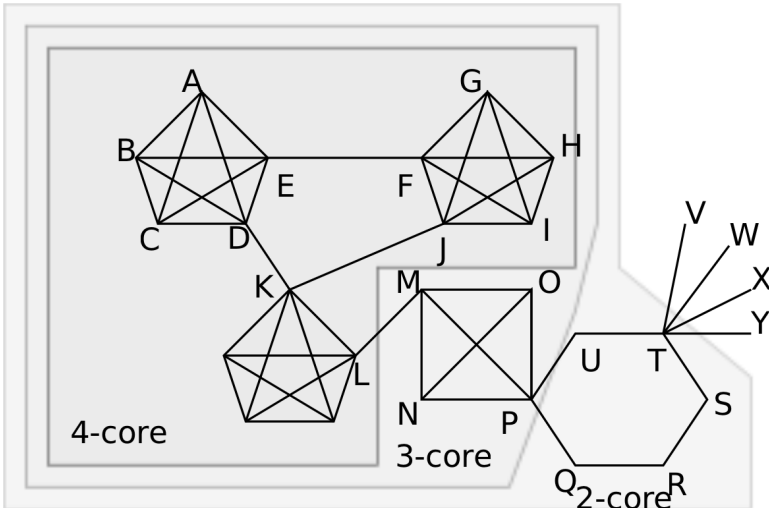


Figure 4.2: An example graph to illustrate the relationship between a vertex's core number, d_{G_k} and N_G^k.

4.3 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

Table 4.1: Vertices in Fig. 4.2 and their 2-core and 3-core properties

Vertices	Core	d_G	d_{G_2}	N_G^2	d_{G_3}	N_G^3
A - C	4	4	4	4	4	4
D - F	4	5	5	5	5	5
G - I	4	4	4	4	4	4
J	4	5	5	5	5	5
K	4	6	6	6	6	6
L	4	5	5	5	5	5
M	3	4	4	4	4	4
N - O	3	3	3	3	3	3
P	3	5	5	5	3	3
Q - S	2	2	2	2	0	0
T	2	6	2	2	0	0
U	2	2	2	2	0	0
V - Y	1	1	0	0	0	0

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 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 [28] 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 4.1, 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 4.2 summarizes notations used in our pseudocodes.

4.3.1 Base Algorithm

The base algorithm, as described in Algorithms 4.1 and 4.2, runs at server nodes and the client coordinates the execution of these remote servers. Each node scans

Table 4.2: 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

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.

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.

4.3.2 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.

Algorithm 4.1. 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

Algorithm 4.2. 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** $anyEdgeDeleted$

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%, applying the base algorithm will delete 90% of the edges, while applying the early pruning technique will insert 10% of the edges. In practice, we observed significant speedup due to the dramatic messaging and I/O reduction.

The algorithm is described in Algorithms 4.3 and 4.4 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 4.3. 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

4.4 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

Algorithm 4.4. 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 ∈ P_i(G) do
3:     compute the d(u)
4:     if d(u) ≥ 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 ∈ P_i(G) do
9:     if qnc_k(u) ≥ 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} ∈ P_i(G) do
14:    if qnc_k(u) ≥ k and qnc_k(v) ≥ k then
15:        put {u, v} into G_k
16: return
```

is serialized and maintained as if the edges were deleted one at a time. We first describe edge insertion and then edge deletion logic.

4.4.1 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.

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 4.2, 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 4.5 and 4.6 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.

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

Algorithm 4.5. 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: $RC_{Edge\ Deletion}(P_i, G, G_k, \{u, v\}, k)$

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 4.7, 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 4.8, 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.

4.4.2 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*,*

Algorithm 4.6. 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 4.14
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)

- *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 4.9 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

Algorithm 4.7. 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
```

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 4.10 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 4.8. 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 to be very effective. *Perform Delete Traversals* algorithm provides parallel version of *Delete Edges Cascaded* algorithm for the case parallel algorithm is preferred.

4.5 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. 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 updates and flushes them based on update count or clocked window. As described in Algorithm 4.11, 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 4.12, run at client side, presents the batch edge

Algorithm 4.9. 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** $u \notin G_k$ or $v \notin G_k$ **then**
2: **return**
3: delete $\{u, v\}$ from G_k
Pruned sequential version:
4: **if** $d_{G_k}(u) \geq k$ and $d_{G_k}(v) \geq k$ **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 4.12
10: **if** $d_{G_k}(u) < k$ **then**
11: $deleteList \leftarrow \{u\}$
12: **Perform Delete Traversals**(G_k,$deleteList$,k)
13: **if** $d_{G_k}(v) < k$ **then**
14: $deleteList \leftarrow \{v\}$
15: **Perform Delete Traversals**(G_k,$deleteList$,k)

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 4.14 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 4.10. 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)
```

The *Pruned Traversal* algorithm described in Algorithm 4.15 runs on the node side and performs a single BFS iteration for the vertices in the $insertTraversals$ list.

4.6 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.

Algorithm 4.11. 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: $deleteList \leftarrow$ choose delete operations from *batchOperations*
2: **Perform Delete Traversals**$(G_k, deleteList, k)$
3: $insertTraversals \leftarrow$ choose insert operations from *batchOperations*
4: **Perform Insert Traversals**$(G, G_k, insertTraversals, k)$

Algorithm 4.12. 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: $add\ cascadedDeletes_i\ to\ deleteList$

4.6.1 Implementation on HBase

The server side of the algorithms were implemented as HBase coprocessors to take advantage of distributed parallelism. Table 5.2 describes the mapping from the graph construct in Table 4.2 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.

Algorithm 4.13. 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*

Table 4.3: Mapping of graph notations in Table 4.2 to the HBase implementation

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

4.6.2 System Setup

The experiment cluster consists of one master server and thirteen 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 HDFS (Hadoop File System) replication factor is set at the default three replicas.

Algorithm 4.14. 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: $G_k \leftarrow G_k \cup G'_k$

4.6.3 Datasets

The datasets we used in the experiments were made available by Milove et al. [108] and the Stanford Network Analysis Project [109]. 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 4.4. 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.

Algorithm 4.15. 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 ← ∅
2: for each vertex u in insertTraversals do
3:     for each vertex w adjacent to u do
4:         if d(w) ≥ k and n(w) ≥ k then
5:             qualifyingList ← qualifyingList ∪ {u, w}
6: return qualifyingList
```

Table 4.4: Key characteristics of the datasets used in the experiments

Name	Vertex Count	Bidirectional Edge Count	Ref
Orkut	3.1 M	234 M	[108]
LiveJournal	5.2 M	144 M	[108]
Flickr	1.8 M	44 M	[108]
Patents	3.8 M	33 M	[109]
Skitter	1.7 M	22.2 M	[109]
BerkStan	685 K	13.2 M	[109]
YouTube	1.1 M	9.8 M	[108]
WikiTalk	2.4 M	9.3 M	[109]
Dblp	317 K	2.10 M	[109]

4.6.4 k-core Construction Experiments

In Section 4.3 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 4.1 and 4.2 in Section 4.3. 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 4.3 and 4.4. 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 datasets. 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. 1.1. As we want to identify the dense subgraphs with high cohesion in these real world datasets, 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 datasets have a degree of at least k. Table 4.5 lists the chosen k values along with the percentage and the number of vertices with degree greater or equal to chosen k value for each dataset.

Table 4.5: k values used in the experiments and the ratio of vertices with degree at least k in the corresponding graphs

Name	degree percentage	k value	Number of vertices
Orkut	4	263	123,241
Orkut	8	183	247,134
Orkut	16	123	493,426
LiveJournal	4	80	194,417
LiveJournal	8	50	391,560
LiveJournal	16	28	787,161
Flickr	4	65	69,095
Flickr	8	24	140,283
Flickr	16	9	288,479
Patents	4	28	162,270
Patents	8	21	332,484
Patents	16	15	670,183
Skitter	4	42	68,612
Skitter	8	26	139,518
Skitter	16	15	277,190
BerkStan	4	57	27,993
BerkStan	8	38	56,982
BerkStan	16	24	110,185
WikiTalk	4	5	115,846
WikiTalk	8	3	277,500
WikiTalk	16	2	626,474
YouTube	4	18	46,471
YouTube	8	10	91,787
YouTube	16	5	201,529
Dblp	4	25	13,031
Dblp	8	16	27,065
Dblp	16	10	53,801

Fig. 4.3 compares the execution times between the base and pruned algorithms for 9 different datasets and 3 different k values. The execution time is shown

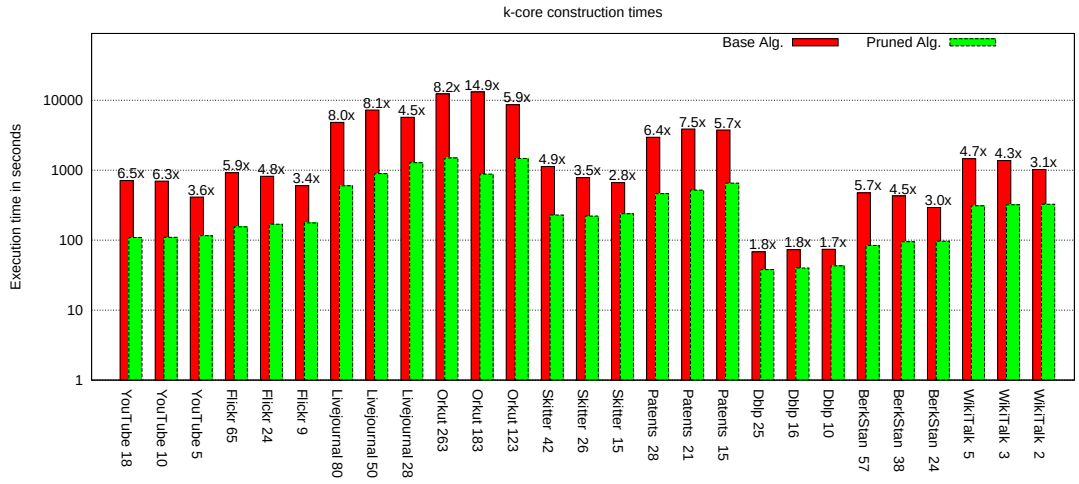


Figure 4.3: k-core construction times for Base and Pruned k-core construction algorithms are shown for each dataset with three chosen k values. Relative speedup achievement of Pruned algorithm over Base algorithm is provided above each bar.

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 dataset 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. Further experiments showed the k-core construction time decreases with an increasing number of servers as expected. We monitored the cluster using Ganglia [110] software while running our experiments. To give an example of how the workload is distributed among the cluster nodes, a snapshot of the network traffic is provided in Fig. 4.4. The plot shows the network traffic during one experiment in k-core construction on the Flickr dataset. What is notable is that besides the master node on the server sg01, all other HBase region server nodes exhibits similar network traffic patterns. This means our implementation on HBase takes full advantage of the distributed parallelism and balances the workload during algorithm execution.

In Section 4.4, 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

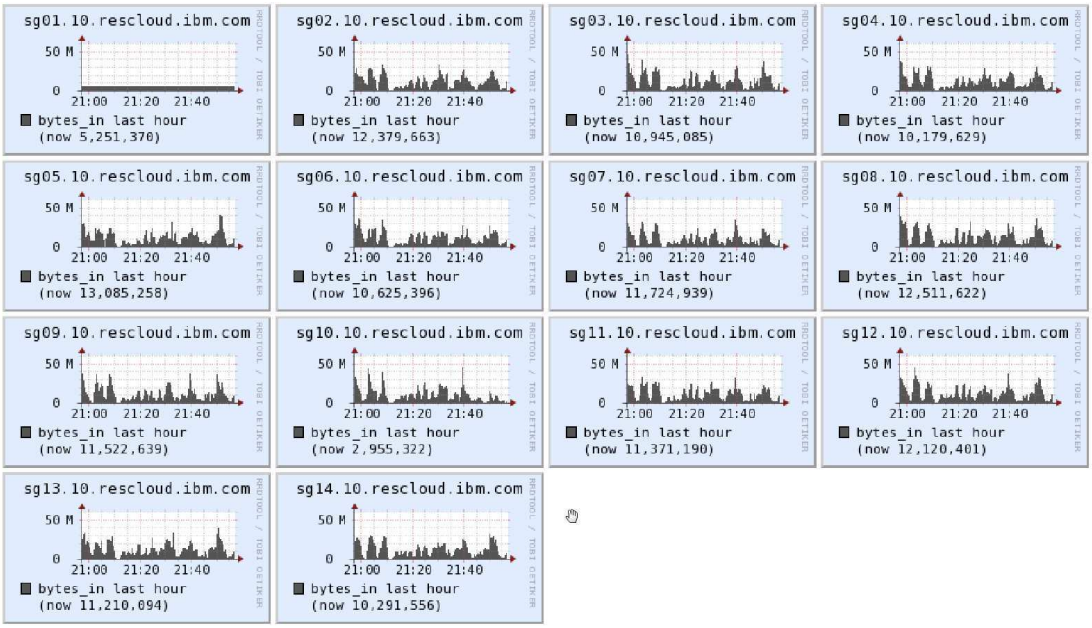


Figure 4.4: Network activities on 14 physical nodes while constructing k-core on Flickr dataset.

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 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 1000 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 1000 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 dataset and measure their execution

times. The largest k value chosen for each dataset is used in the experiments. Fig. 4.5 plots the speedup through our incremental maintenance algorithms over recomputing k-core from scratch, for 9 different datasets. The y-axis shows the speedup in log-scale. For each dataset 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).

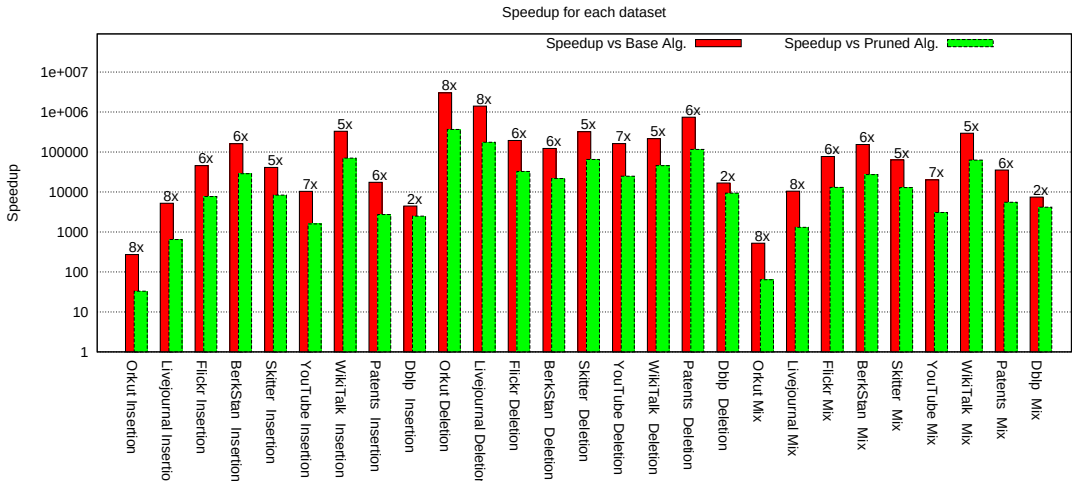


Figure 4.5: k-core maintenance speedups for each dataset with insertion, deletion, mix workload combinations. Maintenance algorithm speedup for both base and pruned construction algorithms is shown in the plot. Relative speedups are also provided above the bars.

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. Table 4.6 lists the mean time and standard deviation for each dataset and experiment scenario.

Table 4.6: Graph update latency in msec to maintain k-core. For each dataset and experiment scenario mean and standard deviation of update time is provided. For large graphs, scenarios with insertions show high standard deviation. Smaller dataset scenarios and ShrinkingWindow scenarios show low update times.

Name	scenario	mean	std
Orkut	ExtendingWindow	45,266.27	1,017,132
Orkut	ShrinkingWindow	4.09	6.05
Orkut	MovingWindow	23,520.74	743,701
LiveJournal	ExtendingWindow	923.46	20566
LiveJournal	ShrinkingWindow	3.41	3.51
LiveJournal	MovingWindow	458.78	14,423
Flickr	ExtendingWindow	20.14	392.85
Flickr	ShrinkingWindow	4.74	16.39
Flickr	MovingWindow	11.89	275.23
Patents	ExtendingWindow	168.83	2,624
Patents	ShrinkingWindow	3.99	2.47
Patents	MovingWindow	83.55	1,799
Skitter	ExtendingWindow	27.39	785.4
Skitter	ShrinkingWindow	3.51	1.66
Skitter	MovingWindow	17.8	663.94
BerkStan	ExtendingWindow	2.94	4.38
BerkStan	ShrinkingWindow	3.87	4.47
BerkStan	MovingWindow	3.09	3.26
WikiTalk	ExtendingWindow	4.41	3.62
WikiTalk	ShrinkingWindow	6.78	7.24
WikiTalk	MovingWindow	4.94	4.45
YouTube	ExtendingWindow	68.63	690.4
YouTube	ShrinkingWindow	4.39	9.17
YouTube	MovingWindow	35.46	485.78
Dblp	ExtendingWindow	15.25	180.38
Dblp	ShrinkingWindow	4.05	2
Dblp	MovingWindow	9.1	122.23

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 4.7. Fig. 4.6 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.

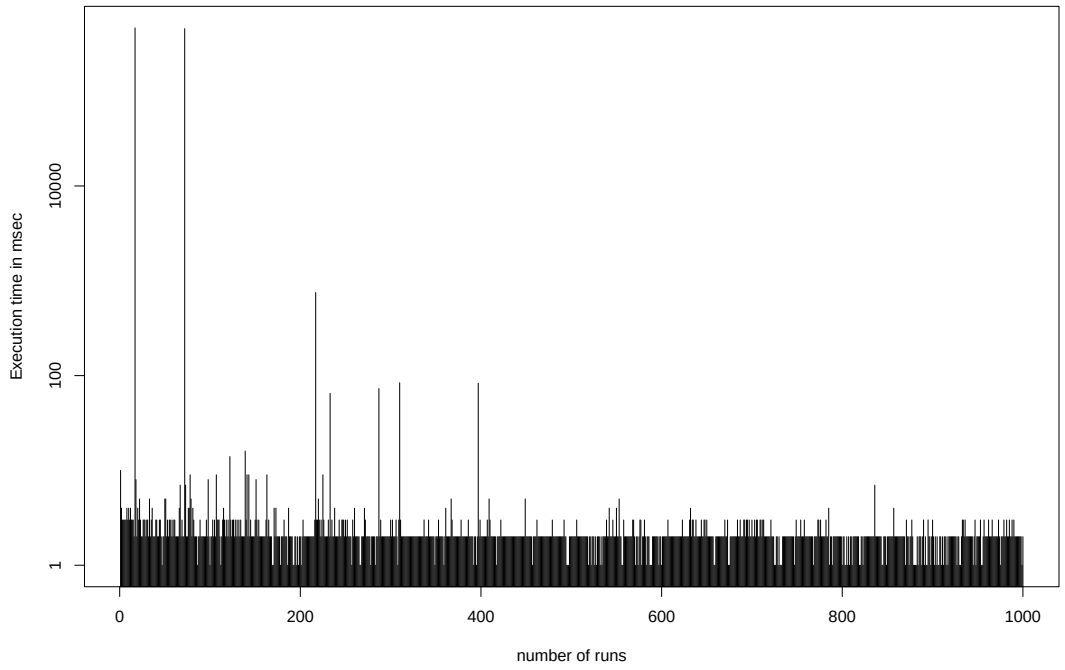


Figure 4.6: Insert latency over 1,000 random edges to the LiveJournal dataset.

We further break down the update latency of k-core maintenance in Fig. 4.7 into three components: the first component is the normal update latency in HBase; the second 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.

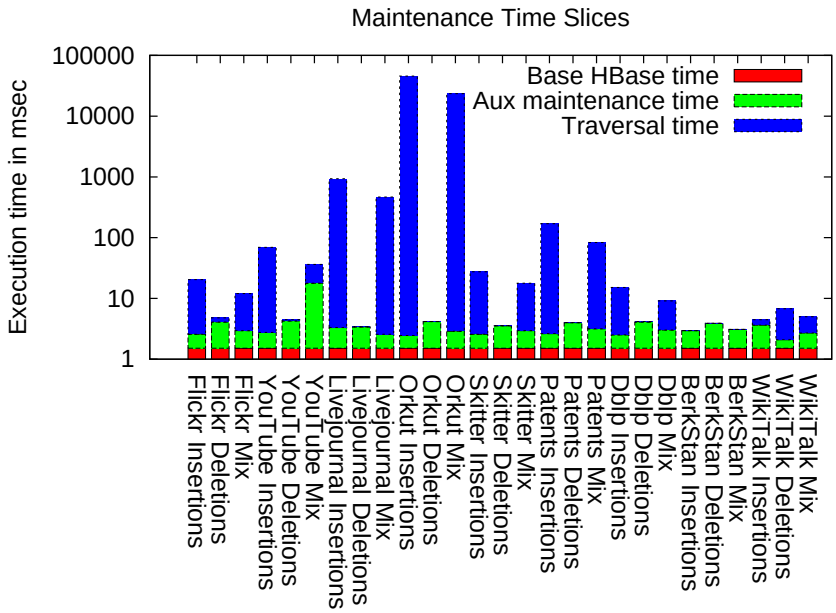


Figure 4.7: k-core maintenance times for each dataset-scenario where time slices for Base HBase insert/delete operation, auxiliary information maintenance and graph traversals are illustrated.

4.6.5 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 datasets,
- 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 dataset for the update scenarios described in Section 4.6.4. For each scenario we use 10K batch size.

Fig. 4.8 shows batch processing speedup versus individual processing in k-core maintenance for three different update scenarios and for 9 different datasets. 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 dataset 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 dataset size and speedup from batch approach. As the datasets 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 datasets 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 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 dataset. By changing the batch size, we measured the average update time for each of the three update scenarios. Fig. 4.9 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 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.

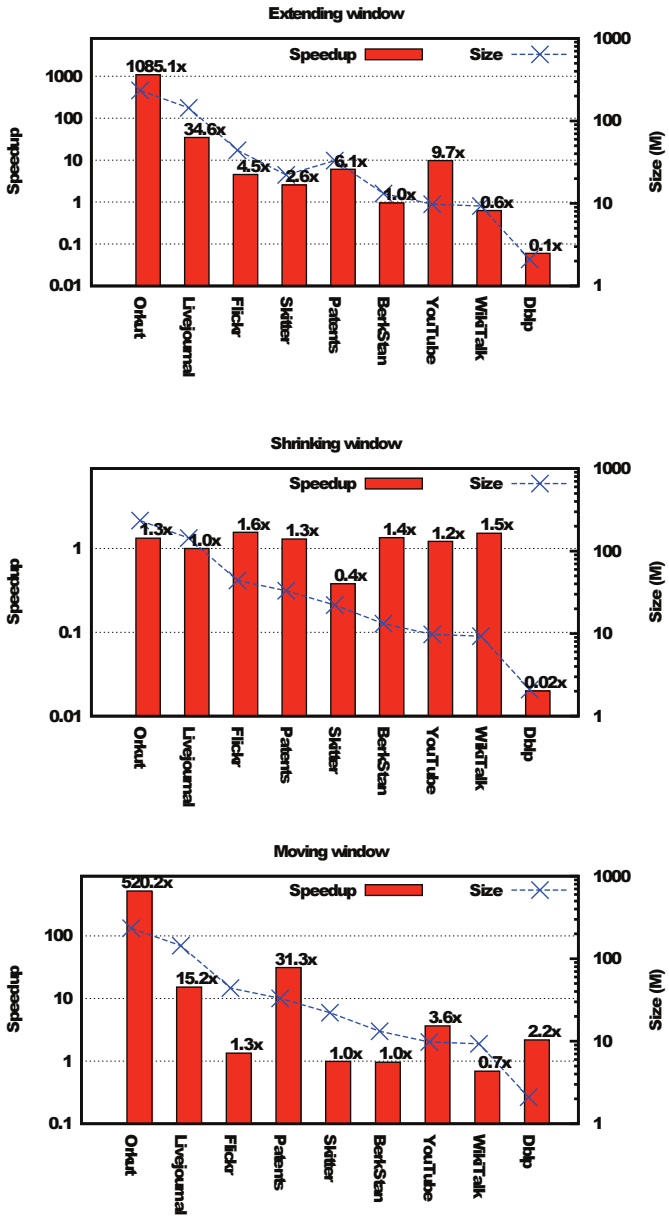


Figure 4.8: 10K sized batch maintenance speedups for Extending window, Shrinking window and Moving window scenarios.

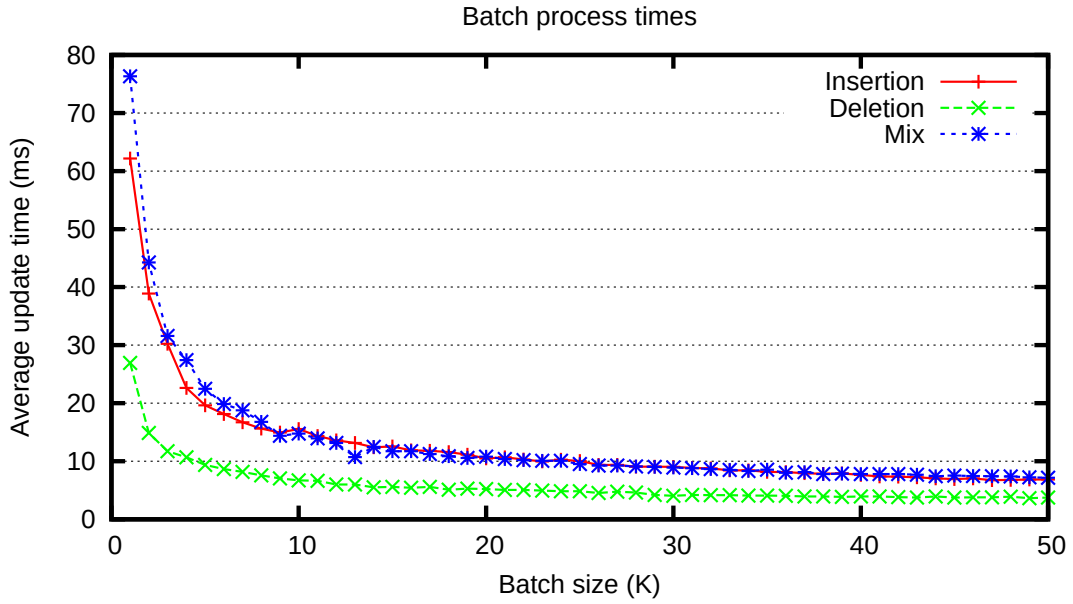


Figure 4.9: Average edge update cost for increasing batch sizes from 1K up to 50K.

Lastly, we studied the break even point before exhausting the benefit of batch maintenance compared to simply reconstructing k-core. Fig. 4.10 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.

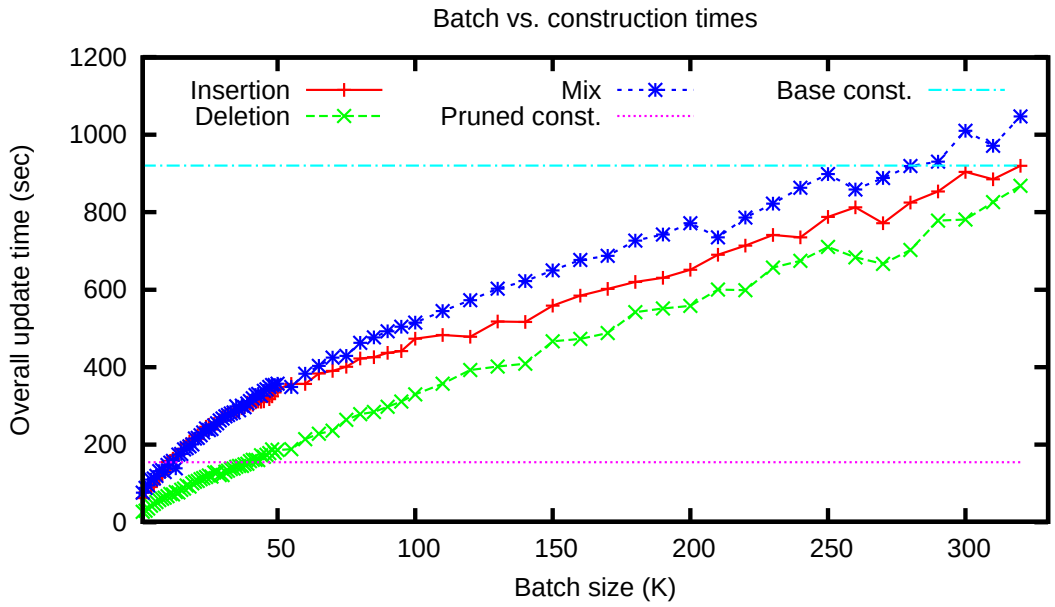


Figure 4.10: Overall processing time of each batch of updates versus reconstruction time of k-core algorithm on Flickr dataset.

4.7 Conclusion

To the best of our knowledge, our study 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.

Chapter 5

Network Community Identification and Maintenance at Multiple Resolutions

Multi-resolution 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 content networks re-ignited interests to the problem at the big data scale that poses computation challenges to early work with algorithm complexity greater than $O(n)$. A further distinction from the decade-old graph problem formulation is that multi-attributed content associated vertices and edges must be included in creating, managing, interpreting and maintaining results. Thus the problem of multi-resolution community analysis is a hybrid of content and graph analysis on various subjects of interest. The problem is made further complex with the observation that interactions with a community happen not just at one but multiple levels of intensity, which reflects in reality active to passive participants in a group. This results with multiple levels of depth in multi-resolution community identification. To make the solution practical, it is thus necessary to make community identification and continuing maintenance at multiple resolutions.

In this chapter[1], 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 [26], k-core, k-edge-connectivity [27], 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 [28], where other measures have complexity growing super-linear or NP-hard.

Our first study on the identification and maintenance of k-core subgraphs considers a fixed k value. We also propose algorithms to perform batch operations for maintenance purposes. The proposed approaches are quite effective when a constant k value is used. On the other hand, when subgraphs at multiple resolutions are needed, one has to run separate instances of the algorithms for each k value. In order to cope with this limitation, significant design changes are considered in our algorithms to efficiently handle k-core subgraphs at multiple, fixed k values. Integrated algorithms are proposed for k-core construction, maintenance and bulk processing of update operations. As we demonstrate with our experimental results, these algorithms yield orders of magnitude speed up compared to the base case k-core construction.

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

[1]2013 IEEE. Reprinted, with permission, from H. Aksu, M. Canim, Y. Chang, I. Korpeoglu, and O. Ulusoy, "Multi-resolution Social Network Community Identification and Maintenance on Big Data Platform," *Big Data (BigData Congress), 2013 IEEE International Congress on*, 7/2013.

computational expense of a full decomposition is not justified. In addition to reduced cost in constructing k-core, it is also computationally less expense to maintain it, compared to maintaining core numbers for large numbers of low degree vertices.

5.1 Preliminaries

We define a rich graph representation G

$$G = \{V, E, M[V, E], C[V, E]\} \tag{5.1}$$

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. This study 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 3. *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 4. *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 4. $\forall v \in V_k$, $D_G(v) \geq k$

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

5.2 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 5.1 summarizes notations used in our pseudocode.

Table 5.1: Notations used in algorithms

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

5.2.1 Base Algorithm

The base algorithm is an adaptation of the BZ algorithm to distributed processing for a fixed k value. As described in Algorithms 5.1 and 5.2, 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.

Algorithm 5.1. 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 5.2. 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 $anyEdgeDeleted$

5.2.2 Multi k-core Construction

Our proposed algorithm computes k-core subgraphs for a list of distinct k values. As stated in the notation, k values are ordered and k_i is the i'th k value, e.g. $k_{1...3} = \{15, 20, 30\}$. 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.

The algorithms are described in Algorithms 5.3 and 5.4 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 5.3. 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: $\quad RC_{Compute\ Degrees}(R_i, G_k, T_L)$
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: $\quad$ **while** $next \geq k_i$ and $next < k_{i+1}$ **do**
10: $\quad\quad next \leftarrow infinity$
11: $\quad\quad$ **for** each region j in $regions(G_k)$ **do**
12: $\quad\quad\quad next_j \leftarrow RC_{Compute\ Core}(R_j, k_i, k_{i+1})$
13: $\quad\quad$ Wait RCs to complete
14: $\quad\quad$ **for** each region j in $regions(G_k)$ **do**
15: $\quad\quad\quad next \leftarrow min(next, next_j)$

Algorithm 5.4. Multi k-core construction- Node N_i Side

```
1: Upon receiving RC_{Compute Degrees}(G_k, T_L)
2: for each vertex u ∈ 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 ← infinity
7: for each vertex u ∈ R_i do
8:     if d_{G_k}(u) ≥ k_i and d_{G_k}(u) < k_{i+1} then
9:         core[{u}] ← k_i
10:        for each vertex v adjacent to u do
11:            if d_{G_k}(v) ≥ k_{i+1} then
12:                d_{G_k}(v) ← d_{G_k}(v) − 1
13:                if d_{G_k}(v) < k_{i+1} then
14:                    next ← d_{G_k}(v)
15:    if d_{G_k}(u) ≥ k_{i+1} then
16:        next ← min(next, d_{G_k}(u))
17: return next
```

5.3 Incremental Multi k-core Maintenance

5.3.1 Edge Insertion

With graph $G = \{V, E\}$ and its materialized multi k-core subgraph $G_k = \cup_{i=1..n} G_{k_i}$ where $G_{k_i} = \{V_{k_i}, E_{k_i}\}$, we give the following edge insertion theorem without proof due to space limitation.

Theorem 3. *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 5.1 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.

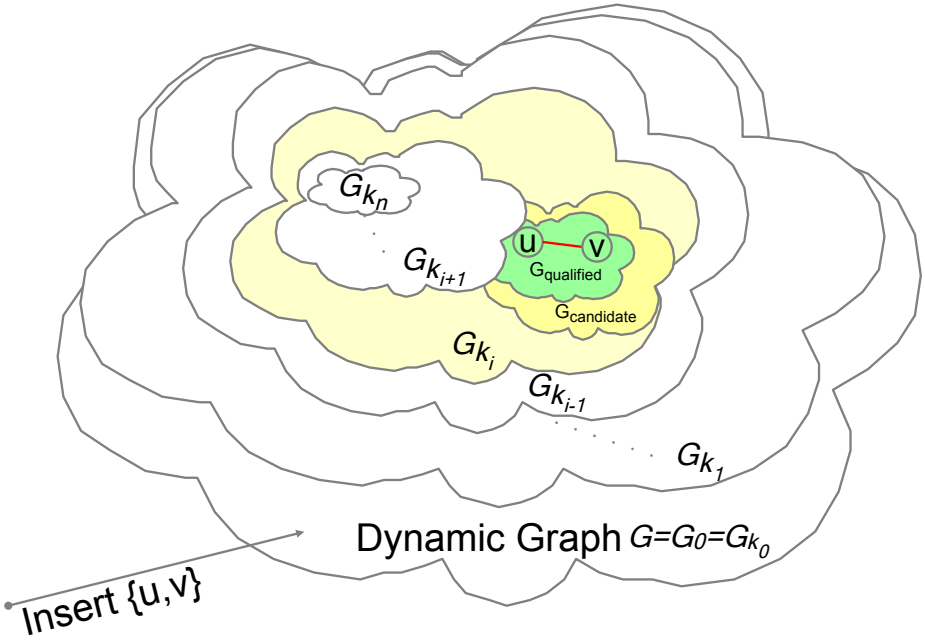


Figure 5.1: Upon an edge $\{u, v\}$ insertion where u or v resides in k_i-core G_{k_i}, first tightly bounded $G_{candidate}$ graph is discovered exploiting maintained auxiliary information, then it is processed to compute $G_{qualified}$ subgraph qualifying for k_{i+1}-core.

Algorithms 5.5, 5.6 and 5.7 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 5.6 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 candidate edges for $G_{k_{i+1}}$. *Partial KCore* in Algorithm 5.7 then processes $G_{candidate}$ subgraph and returns the graph qualified for k_{i+1} core into $G_{qualified}$.

Algorithm 5.5. 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: $G_{k_{i+1}} \leftarrow G_{k_{i+1}} \cup G_{qualified}$

Algorithm 5.6. 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 ← new queue
2: Q.enqueue(u)
3: mark(u)
4: while Q ≠ ∅ do
5:     v ← 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} ∉ C then
9:             if d(w) ≥ k_j and qnc_{k_j}(w) ≥ k_j then
10:                C ← C ∪ {v, w}
11:                if w ∉ G_{k_{i+1}} then
12:                    C ← C ∪ {w, v}
13:                    if w is not marked then
14:                        Q.enqueue(w)
15:                        mark(w)
16: return C
```

5.3.2 Edge Deletion

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

Theorem 4. *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 5.7. 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 ← true
2: while changed do
3:     changed ← false
4:     for each {u, v} ∈ C do
5:         if d_C(u) < k_j then
6:             delete {u, v} and {v, u} from C
7:             changed ← 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 5.8 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 5.9 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.

Algorithm 5.8. 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)

5.4 Batch Multi 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.

In such a batch maintenance scenario, edge insertion and deletion incurs immediate updates to the auxiliary information, degree and QNC, while updates to the k-core subgraph are deferred. The system maintains a list of updates and flushes them based on update count or clocked window. As described in Algorithm 5.10, 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 5.9. Update Coreness Cascaded

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

```
1: Q ← new queue
2: Q.enqueue(u)
3: mark(u)
4: while Q ≠ ∅ do
5:     v ← Q.dequeue()
6:     core[v] ← 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)
```

Algorithm 5.11 presents the batch edge deletions in more detail. Edges in the deletion list *deleteList* are grouped and sent to the respective region's node, where each remote call returns a list of cascaded deletion requests. The client then regroups the requests.

Algorithm 5.12 describes the node side of edge deletions. The algorithm first receives the list *deleteList*, the list of edges to be deleted, and the list *downgradeList*, the list of vertices to be updated into one lower k values in maintained k-core list $k_{1\ldots n}$. For each edge $\{u, v\}$ in *deleteList* the edge is first deleted. To do that, all outgoing edges of u are deleted and incoming edges are returned to the client as cascaded delete. We do not delete incoming edges in this remote call since those edges are not necessarily reside in the current region (edges are stored according to source vertex). If the degree of u gets smaller than the k value of core in which it currently resides, this vertex should be downgraded. Such vertices are added to *downgradeList*. Each vertex in *downgradeList* is moved from its current core value, lets say k_i, to one lower core value k_{i-1}. When the vertex gets lower than minimum maintained k value, it is deleted from the

Algorithm 5.10. Batch Process- Client Side

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

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

materialized view and any cascaded delete is added to *cascadedDeletes* list. After each core change, if any direct neighbor also needs to be updated, it is added to *cascadedDowngrades* list to be processed in the next iteration.

Algorithm 5.13 presents batched edge insertion maintenance in detail. In essence, the independently launched graph traversal in each incremental maintenance is now aggregated into a single parallel graph traversal launched simultaneously from all the new edges. The Algorithm first takes the list of edges $insertList$, and traverses them in parallel. All candidate edges discovered during BFS traversals are kept in the client side in the sets called $G_{candidate_{1...n}}$ for post traversal computation. Firstly, the client groups all vertices in $insertList$ according to their regions. A BFS operation is performed for each region by calling $qualifyingList_i \leftarrow RC_{Pruned\ MultiTraversal}(R_i, bucket_i, k_{1...n})$ remote call function. Each node handles the BFS iterations over its associated region and then returns the selected edge list to the client. The list $insertList$ is cleared after all remote calls are made. An edge $\{u, v\}$ returned from the remote call is skipped if it already exists in the set $G_{candidate_{1...n}}$, which means it was already traversed previously. When the new edge $\{u, v\}$ is not connected to the next inner k-core subgraph, then v is inserted into the list $insertList$ to be traversed in the next iteration.

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

Algorithm 5.11. Perform Delete Traversals- Client Side

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

1: $downgradeCoreList \leftarrow \emptyset$
2: **while** $deleteList \neq \emptyset$ or $downgradeCoreList \neq \emptyset$ **do**
3: **for** each region i in $regions(G_k)$ **do**
4: $del_i \leftarrow from\ deleteList\ filter\ edges\ stored\ in\ R_i$
5: $down_i \leftarrow from\ downgradeCoreList\ filter\ vertices\ stored\ in\ R_i$
6: $\{cDel_i, cDown_i\} \leftarrow RC_{Handle\ Delete}(R_i, G_k, del_i, down_i, k_{1...n})$
7: Wait RCs to complete
8: $deleteList \leftarrow \emptyset$
9: $downgradeCoreList \leftarrow \emptyset$
10: **for** each region i in $regions(G_k)$ **do**
11: **if** $cDel_i \neq \emptyset$ **then**
12: $add\ cDel_i\ to\ deleteList$
13: **if** $cDown_i \neq \emptyset$ **then**
14: $add\ cDown_i\ to\ downgradeCoreList$

traversed graph.

The *Pruned MultiTraversal* algorithm described in Algorithm 5.14 runs on the node side and performs a single BFS iteration for the vertices in the *insertList* list. It selects the edges to the vertices with QNC value greater than the next maintained core value.

5.5 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 5.12. Handle Delete- Node N_i Side

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

1: **for** each edge $\{u, v\}$ in $deleteList$ **do**
2: delete $\{u, v\}$ from G_k
3: $i \leftarrow$ core index for $core[u]$
4: **if** $d_{G_{k_i}}(u) < k_i$ **then**
5: $downgradeList \leftarrow downgradeList \cup \{u\}$
6: $cascadedDeletes \leftarrow \emptyset$
7: $cascadedDowngrades \leftarrow \emptyset$
8: **for** each vertex $\{u\}$ in $downgradeList$ **do**
9: $i \leftarrow$ core index for $core[u]$
10: $core[u] \leftarrow k_{i-1}$
11: **for** each vertex w adjacent to u in G_{k_i} **do**
12: **if** $k_{i-1} == 0$ **then**
13: delete $\{u, w\}$ from G_{k_i}
14: $cascadedDeletes \leftarrow cascadedDeletes \cup \{w, u\}$
15: **if** $d_{G_{k_i}}(w) < k_i$ **then**
16: $cascadedDowngrades \leftarrow cascadedDowngrades \cup \{w\}$
17: **return** $\{cascadedDeletes, cascadedDowngrades\}$

5.5.1 System Setup and Datasets

Graph data is stored in HBase and the algorithms are implemented as HBase Coprocessors where distributed parallelism is applicable. Table 5.2 shows how notations in algorithms are interpreted in HBase implementation. Our cluster consists of one master server and 13 slave servers, each of which is an Intel CPU based blade running Linux connected by a 10-gigabit Ethernet. We use 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)

Algorithm 5.13. Perform Insert Traversals- Client Side

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

1: $G_{candidate_{1...n}} \leftarrow \emptyset$
2: **while** $insertList \neq \emptyset$ **do** ▷ at each iteration
3: **for** each region i in $regions(G)$ **do**
4: $bucket_i \leftarrow from\ insertList\ filter\ vertices\ stored\ in\ R_i$
5: $qualifyingList_i \leftarrow RC_{Pruned\ MultiTraversal}(R_i, bucket_i, k_{1...n})$
6: Wait RCs to complete
7: $insertList \leftarrow \emptyset$
▷ Aggregate this turn results and compute next turn input
8: **for** each region j in $regions(G)$ **do**
9: **for** each edge $\{u, v\}$ in $qualifyingList_j$ **do**
10: $i \leftarrow$ core index for $core[u]$
11: **if** $\{u, v\} \notin G_{candidate_i}$ **then** ▷ Select a vertex only once
12: $G_{candidate_i} \leftarrow G_{candidate_i} \cup \{u, v\}$
13: **if** $v \notin G_{k_{i+1}}$ **then** ▷ do not go over vertices already in $G_{k_{i+1}}$
14: $G_{candidate_i} \leftarrow G_{candidate_i} \cup \{v, u\}$
15: $insertList \leftarrow insertList \cup \{v\}$ ▷ continue traverse
16: **for** i in $1 \ldots n$ **do**
17: **if** $G_{candidate_i} \neq \emptyset$ **then**
18: $G_{qualified_i} \leftarrow$ **Partial KCore** $(G_{candidate_i}, k_{i+1})$
19: **for** each vertex $\{u\}$ in $G_{qualified_i}$ **do**
20: $core[u] \leftarrow k_{i+1}$

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. [108] and the Stanford Network Analysis Project [109]. 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 5.3.

Algorithm 5.14. Pruned MultiTraversal- Node N_i Side

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

1: $returnList \leftarrow \emptyset$
2: **for** each vertex u in $insertList$ **do**
3: $i \leftarrow$ core index for $core[u]$
4: **for** each vertex w adjacent to u in G_{k_i} **do**
5: **if** $d_{G_{k_i}}(w) \geq k_{i+1}$ and $qnc_{G_{k_i}}(w) \geq k_{i+1}$ **then**
6: $qualifyingList \leftarrow qualifyingList \cup \{u, w\}$
7: **return** $qualifyingList$

5.5.2 Experiments

We use multiple k values to represent a community at multiple resolutions. For each social network dataset, we select three distinct k values so that 4, 8 and 16 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 5.4 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 5.3 and 5.4, for each dataset with all chosen k values at once to measure k-core construction for multiple k values. Figure 5.2 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 5.5 and 5.6, we first construct and materialize k-core graph for selected multiple k values and under three scenarios explained below we measure average maintenance times.

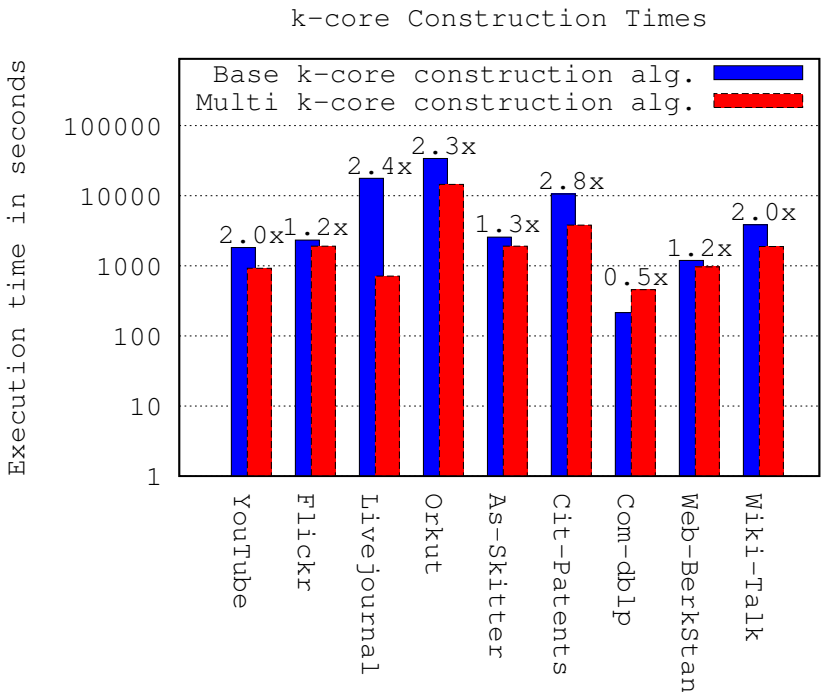


Figure 5.2: k-core construction times for Base and Multi k-core construction algorithms are shown for each dataset with three chosen k values. Relative speedup achievement of Multi algorithm over Base algorithm is provided above each bar.

Table 5.2: Mapping of graph notations in Table 5.1 to implementation in HBase

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

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

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 1000 vertices from the graph to delete them one by one while we maintain the k-core subgraph.

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

Table 5.3: Key characteristics of the datasets used in the experiments

Name	Vertex Count	Bidirectional Edge Count	Ref
Orkut	3.1 M	234 M	[108]
LiveJournal	5.2 M	144 M	[108]
Flickr	1.8 M	44 M	[108]
Patents	3.8 M	33 M	[109]
Skitter	1.7 M	22.2 M	[109]
BerkStan	685 K	13.2 M	[109]
YouTube	1.1 M	9.8 M	[108]
WikiTalk	2.4 M	9.3 M	[109]
Dblp	317 K	2.10 M	[109]

random deletion from the graph. By doing so we insert 1000 edges to the graph and delete 1000 edges from the graph while maintaining the k-core subgraph.

Table 5.4: k values used in the experiments and the ratio of vertices with degree at least k in the corresponding graphs

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

We repeated these three scenarios with each dataset and measured their execution times. Figs. 5.3 plot the speedup through our incremental maintenance algorithms over recomputing k-core from scratch, for 9 different datasets. The y-axis shows the speedup in log-scale. For Extending, Shrinking, Moving window scenarios and each dataset, the figures give the speedup of incremental update approach with respect to from-scratch construction using the multi k-core construction algorithm. As the figures show, three to five orders of magnitude speedup

can be expected for 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.

5.5.3 Batch Maintenance Experiments

In order to investigate the speedup provided by our batch update approach for maintaining the multi-k-core subgraph, we run experiments on different datasets. To measure the performance improvement of batch processing approach compared to individual updates and full reconstruction, we set up experiments for each dataset and for each update scenario described in Section 5.5. For each experiment, we use 10K batch size. Figs. 5.4- 5.6 show batch processing speedup versus individual processing in k-core individual maintenance and reconstruction for three different update scenarios and 9 different datasets. The speedup is shown in the y-axis in log-scale. Each figure illustrates batch maintenance speedup versus both individual maintenance and reconstruction.

For extending window scenario, in Fig. 5.4 we get greater performance improvement, four to five orders of magnitude speedup when compared with reconstruction case. We get upto 2 orders of magnitude speedup compared to individual maintenance. When compared with Fig. 5.3, we figure out that batch window extending approach provides a more stable speedup ratio for different datasets. For instance, Cit-Patent dataset shows the largest speedup in batch maintenance case compared to individual maintenance while it provides the smallest speedup in individual maintenance case. Totally, its speedup is close to the average speedup of other datasets. For the shrinking window scenario, the batch processing approach does not provide significant speedup, as the deletion cost in individual processing is already minimal, i.e., close to auxiliary maintenance cost plus the 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. Experiment results show that batch processing provides stable speedup for all scenarios and datasets with

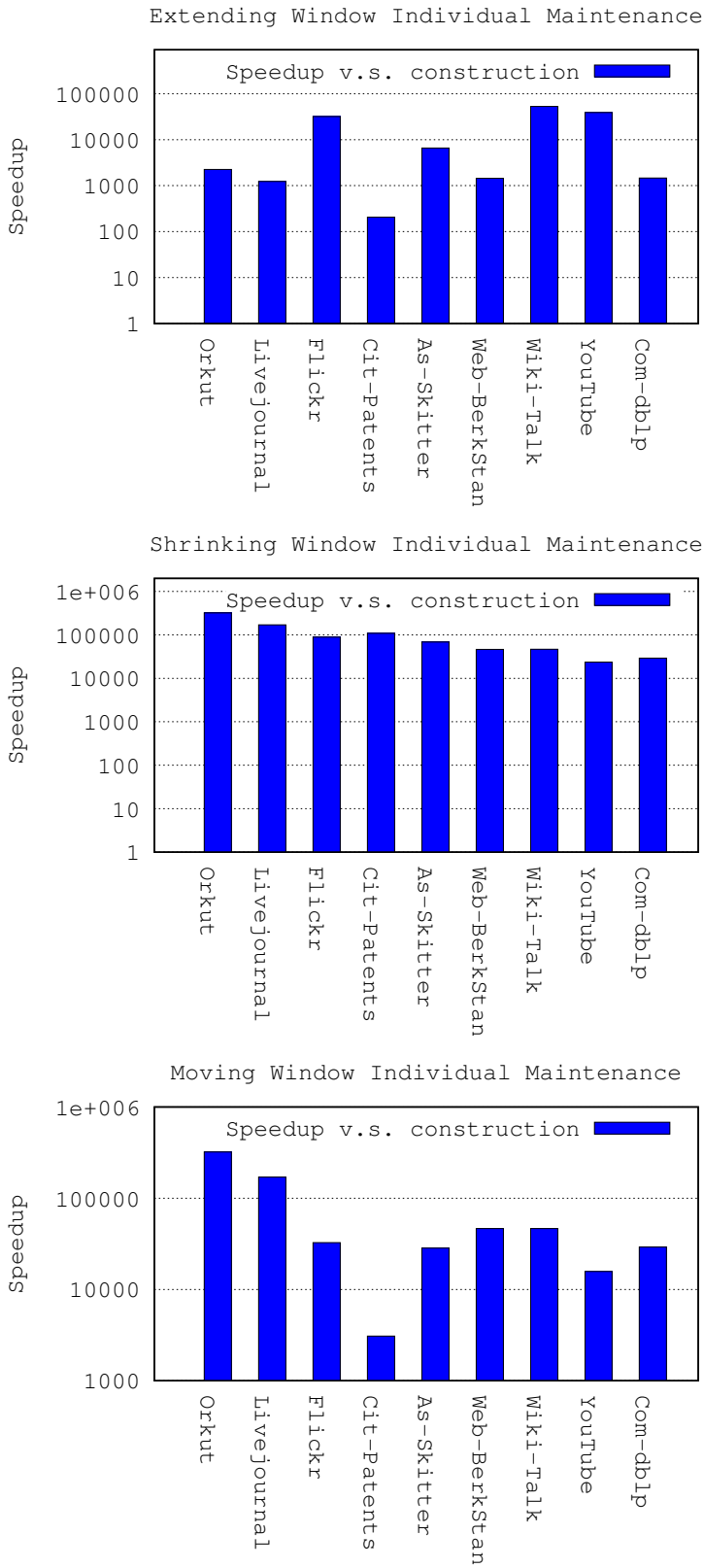


Figure 5.3: k-core maintenance algorithm speedup over construction algorithms for Extending, Shrinking, and Moving window scenario.

different sizes and topologies.

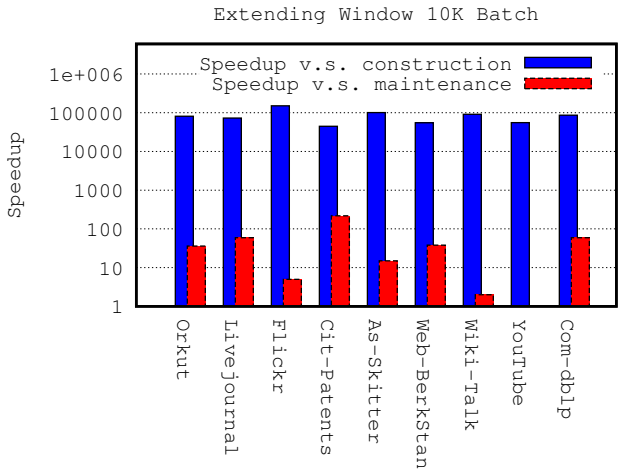


Figure 5.4: 10K sized batch maintenance speedups for Extending window scenario.

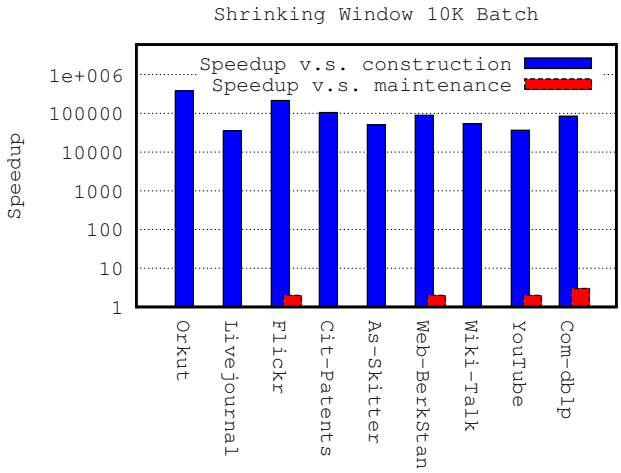


Figure 5.5: 10K sized batch maintenance speedups for Shrinking window scenario.

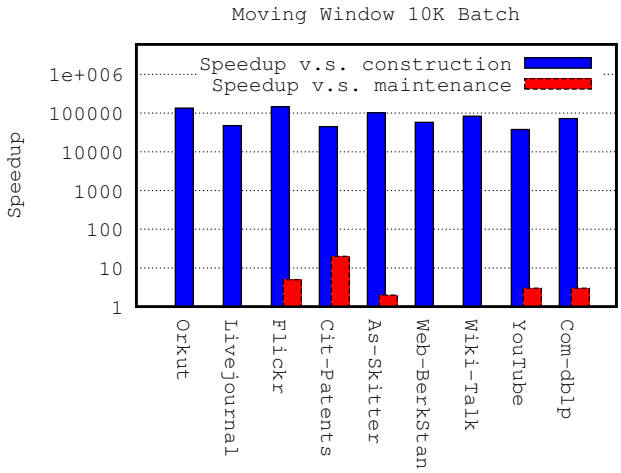


Figure 5.6: 10K sized batch maintenance speedups for Moving window scenario.

5.6 Conclusion

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

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

We provided a distributed implementation of the algorithms on top of Apache HBase, leveraging its horizontal scaling, range-based data partitioning, and the newly introduced Coprocessor framework. Our implementation 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.

Chapter 6

Graph Aware Caching

6.1 Introduction

The technique of data caching is well known and widely applied across tiers of computing and storage systems. With the emergence of a new generation of social and mobile applications built on graph data stores or graph data model implemented on legacy database technology, the knowledge about graph traversal based queries can be exploited to devise efficient caching policies that are graph topology aware. Simultaneously, the policy must address metadata properties that come with nodes and edges in the graph, since query predicates are often imposed on those properties to select next steps in the traversal.

Among the use cases of graph data store such as social networks, knowledge representations, and Internet of things, while their respective graph topology may be small and fit on a single server, adding all the metadata properties easily drives up computing and storage requirements beyond the capacity of one server. The context of our investigation thus is anchored on scale out, big data clusters in which the graph and its data is partitioned horizontally across servers in the cluster. We assume topology and metadata about a node or edge are co-located since they are most often accessed together. In addition, as reflected in real-world workload, updates to change graph topology are allowed, which makes one-time

static graph clustering less beneficial.

Figure 6.1 illustrates the context in which our cache solutions fit. The graph data is partitioned and distributed over a cluster of servers with low communication latency. Each distributed node hosts its own data cache and manages data stored in the cache with knowledge of local vs. remote graph data. A client issuing query to the server hosting the queried root node and the server communicates with its peers to process the client's query.

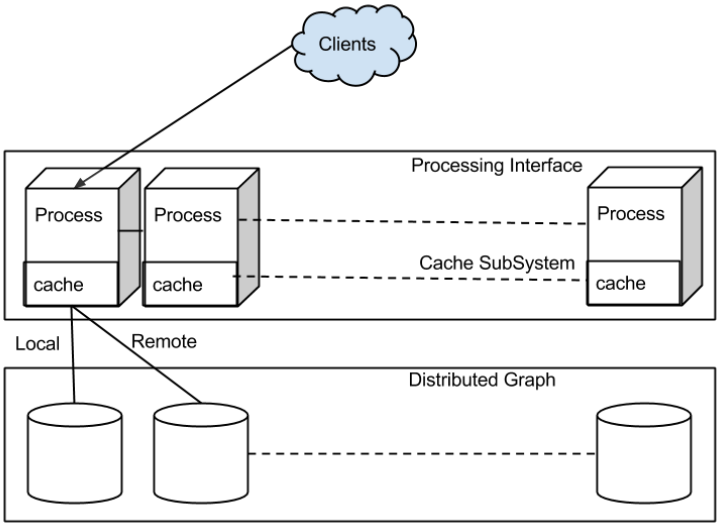


Figure 6.1: Cache layer is located between graph storage and distributed processing node. Cache layer knows if a graph file is local or remote and designed to fetch and evict items with graph-aware optimizations.

The rest of the chapter is organized as follows. We discuss the graph implementation on big data platform in Section 6.2. Our proposed graph-aware cache is presented in Section 6.3 and we evaluate its performance on real social network datasets in Section 6.4. Finally, Section 6.5 concludes the chapter.

6.2 Distributed Graph Handling with 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 [36]. 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.

6.2.1 HBase and Coprocessors

Apache HBase is a non-relational, distributed data management system modeled after Google's BigTable [103]. 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. 6.2 depicts a simplified diagram of HBase with several key components relevant to this chapter. 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. 6.2, HTable 1 has four regions managed by region servers 1, 2 and 10 respectively, while HTable 2 has three regions stored in region servers 1 and 2. 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

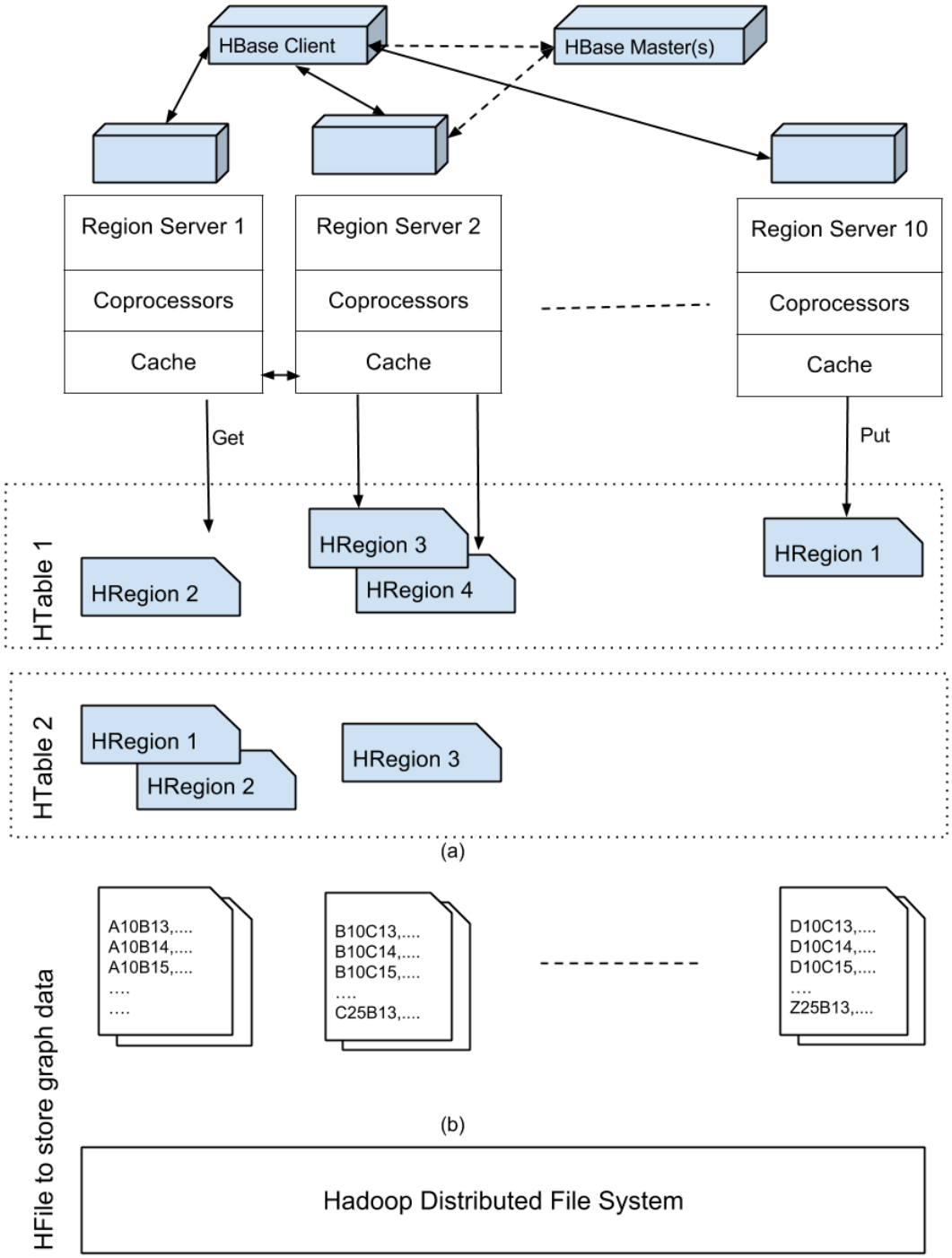


Figure 6.2: Coprocessors are user-deployed programs running in the region servers. Cache is distributed with graph regions and used by coprocessors. It is located between Coprocessor and HRegions where HRegion accesses are first handled by the cache layer.

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. 6.2 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.

6.2.2 Graph Processing on HBase

We map the rich graph representation $G = \{V, E, M, C\}$ 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 that a vertex's row always immediately proceeds the rows of its outbound edges in an HTable. Fig. 6.2, 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-hop neighbors queries in Section 6.4 are implemented in several HBase Coprocessors to achieve maximal parallelism. When a non-local vertex neighbors are to be read, a Coprocessor instance issues a "neighbors read message" to the remote HBase region server, which reads and returns the neighbors.

6.3 Cache Systems

Big-Data philosophy suggests working with commodity servers. While working on Hadoop/HBase systems, we realized that a significant amount of memory on commodity servers is available with optimized configurations. Meanwhile, such databases suffer from low I/O speed of distributed file systems. Therefore, implementing a caching mechanism would improve overall system performance. A straightforward approach is to use existing caching techniques. On the other hand, we are working on a graph specific system. Thus, we know access patterns and insight about the data characteristics which allow us to design domain specific cache with improved performance. Cache systems try to predict future data access requests and optimize its resources accordingly. Future data access prediction in generic cache algorithms is based on two data access patterns:

1. **spatial locality**, which indicates that future data accesses will target spatially close data to current accesses.
2. **temporal locality**, which means that future data accesses will target the same data currently accessed.

Spatial locality feature is exploited to prefetch data into the cache even before it is accessed for the first time. Temporal locality feature is exploited to keep already accessed data in the cache to serve possible future requests received under limited cache size challenge. We discuss spatial locality exploiting features in *Fetch algorithms* section and temporal locality exploiting features in *Eviction algorithms* section below.

6.3.1 Fetch Algorithms

6.3.1.1 Traditional Fetching

Generic cache algorithms assume that iteration over logical data order is correlated to physical data order in lower layer in cache hierarchy. For instance, let's consider an iteration over array elements as illustrated in Algorithm 6.1. Access to $a[i]$ proceeds with an access to $a[i+1]$ where $a[i]$ and $a[i+1]$ are stored next to each other. Thus, prefetching $a[i+1]$ upon fetch operation on $a[i]$ achieves a hit due to right prediction of future access.

Algorithm 6.1. *ArrayAccessPattern*

```
1: for i = 1 → size(a) do
2:     s ← a[i]
```

6.3.1.2 Graph Fetching

Comparing iterations over arrays with those over graphs, graph accesses are not following physical data order in lower layer in cache hierarchy. Algorithm 6.2 illustrates a simple traversal in a graph. Graph traversal is correlated with graph topology rather then its storage pattern. It is obvious that graph access prediction as suggesting next element in storage layout will be a poor prediction although it is acceptable for non-graph access patterns.

Algorithm 6.2. *GraphAccessPattern*

```
1: for vertex u in v.neighbors() do
2:     enqueue(u)
```

6.3.2 Eviction Algorithms

Temporal locality concept states that currently requested data will be requested again in near future. Keeping every requested item in the cache would exploit

temporal locality at maximum benefit. However, cache systems come with limited memory space, thus when the cache area gets full, some items should be evicted from it. When such evicted items are requested from the cache, a cache miss will happen. Therefore, eviction algorithms are designed to find optimal suggestion in item selection for eviction such that overall miss penalty is minimized.

Least Recently Used algorithm (LRU) [84, 86, 83] is the most popular eviction policy in the literature. LRU keeps track of access order and selects the least recently used item for eviction.

Largest Item First algorithm is item size sensitive where it evicts largest item in the cache. Evicting largest item allows cache to store several small items. This algorithm does not assume any correlation with item size and its access frequency.

Smallest Item First algorithm is also item size sensitive and it evicts the smallest item in the cache. Thus, the algorithm tries to minimize miss penalty where small items are fetched faster then large items.

Clock Based Graph Aware algorithm is a hybrid algorithm we propose, which assigns a time-to-live value to each item and evicts the one expired first. Next we discuss this algorithm.

6.3.3 Clock Based Graph Aware Cache (CBGA)

Distributed graph context has its specific characteristics which can be exploited by the cache system. Different from generic applications, distributed graphs display the following characteristics:

1. *local/remote placement*: some items are stored locally while some others are stored in remote server which has an additional network call overhead.
2. *different sizes*: graph vertices have different number of neighbors and its metadata might have variable size. Thus some items require more space in

cache.

3. *uneven access probabilities*: graph vertices have different centrality/popularity in networks. Thus some central items are requested more frequently than others.

4. *iteration on topology*: graph traversal algorithms use vertex-neighbor jumps which are stored randomly in storage layout.

We introduce clock-based graph aware caching to exploit all graph specific access patterns. First, interpreting spatial locality to handle topological closeness instead of storage-level closeness, we propose the graph-aware fetching algorithm in Algorithm 6.3 which prefetchs topological neighbors instead of neighbors in storage layout.

Algorithm 6.3. $Graph - AwareFetching$

```
1: upon get(Vertex v)
2: put_cache(v, v.neighbors())
3: for vertex u in v.neighbors() do
4:     put_cache(u, u.neighbors)
```

Furthermore, we assign a time-to-live (TTL) value to each cached item so that the graph-aware eviction algorithm minimizes overall miss penalty in distributed graph context. The TTL value is computed using the following equation:

$$TTL = \frac{l}{s * d} \tag{6.1}$$

where l is the average duration (latency) to fetch an item into the cache, d is the fetched distance to the source vertex which triggers this fetch operation, and s is the size of the cached item. The latency (l) parameter makes TTL value to become local/remote placement sensitive while the distance parameter (d) assists the vertices close to query source stay longer in the cache. Thus, the large local vertices on the fringe of the graph are preferred for eviction. In other words, small, remotely served and central vertices are given priority to be kept in the cache. Important procedures of the graph aware eviction algorithm are presented

in Algorithm 6.4. The *normalize* procedure scales up the computed TTL value to make sure it is larger than or equal to 1.

Algorithm 6.4. *Graph − AwareEviction*

```
1: upon put_cache(v,...) call
2: if v ∈ local_partition then
3:     latency ← LOCAL_ACCESS_LATENCY
4: else
5:     latency ← REMOTE_ACCESS_LATENCY
6: size ← get_size(v)
7: distance ← get_distance(v, s)
8: TTL_v ← latency / (size*distance)
9: TTL_v ← normalize(TTL_v) return
10:
11: upon CBGA_evict() call
12: while TRUE do
13:     for item u in cache.items() starting from last index do
14:         TTL_u ← TTL_u − 1
15:         if TTL_u ≤ 0 then
16:             evict(u) return
```

CBGA uses eventual consistency model for cache coherency, a relaxed consistency model that is described by Terry et al. [111] and discussed by Werner [112]. Any item in the cache is associated with a TTL value which eventually decreases to zero and causes the item to be evicted. Essentially, any change on items is reflected to the cache after a sufficient period of time which is acceptable for many social network applications, e.g., Facebook [86]. Thus, all copies of an item in the cache will be consistent and reflect all updates to the item.

6.4 Performance Evaluation

We ran experiments to evaluate the performance of the discussed caching policies.

Table 6.1: Key characteristics of the datasets used in the experiments

Name	Vertex Count	Bidirectional Edge Count	Ref
Twitter	1.1 M	170 M	[113]
Orkut	3.1 M	234 M	[108]
LiveJournal	5.2 M	144 M	[108]
Flickr	1.8 M	44 M	[108]
Patents	3.8 M	33 M	[109]
Skitter	1.7 M	22.2 M	[109]
BerkStan	685 K	13.2 M	[109]
YouTube	1.1 M	9.8 M	[108]
WikiTalk	2.4 M	9.3 M	[109]
Dblp	317 K	2.10 M	[109]

6.4.1 System Setup and Datasets

Our graph data are stored in HBase and the algorithms are implemented as HBase Coprocessors where distributed parallelism is applicable. HBase Coprocessors can access to local and remote cache areas. Our cluster consists of 1 master server and 5 slave servers, each of which is a c3.large instance running Linux on Amazon EC2. 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 HDFS (Hadoop File System) replication factor is set at one replica. 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. [108], Social Computing Data Repository at ASU [113], and the Stanford Network Analysis Project [109]. 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 6.1.

6.4.2 Experiments

We implemented all cache approaches discussed in Section 6.3 in HBase Coprocessors. Each Coprocessor reserves 10MB cache area in memory. Whenever a graph read operation is received, we first try to serve the operation from the cache. When it fails to serve from the cache, it reads the graph data from local or remote storage depending on where the requested data resides. Each miss is handled according to Algorithm 6.3, and when the cache becomes full, appropriate eviction policy is executed. However, in LRU case only the requested item is read from HBase, i.e., no prefetching is done.

In order to evaluate cache policy performance, we set up our HBase cluster to use the chosen policy, and then we run 10000 random k-hop neighbors queries on each social network dataset. We limit a k-hop neighbors query result to top 10K vertices, since this is a large enough size for most of the graph applications, i.e, social network applications.

We measure both hit-ratio and execution-time for each test case. Figure 6.3 shows the hit-ratio at left vertical axis and the execution-time at right vertical axis for each evaluated policy on Twitter dataset. We observe similar results for the other datasets. Our CBGA policy achieves the highest hit-ratio and the lowest execution-time, while the traditional LRU policy shows the poorest performance. The LRU-SP policy (which is LRU with graph-aware spatial locality caching extension) displays sharp increase in hit-ratio while its execution-time performance is still behind the other policies with size and path awareness. CBGA versus LRU speedup for all datasets is shown in Figure 6.4 where the speedup is computed by dividing the execution time of LRU by the execution time of CBGA. Different datasets provide different speed up values. The datasets used in the experiments are real graphs from different domains, for instance Twitter is a social network, BerkStan is a web graph while Patents is a citation network among US Patents. Each network has different network properties which result in different performance results when the proposed cache system is employed. Note that, for all datasets CBGA cache outperforms LRU based cache.

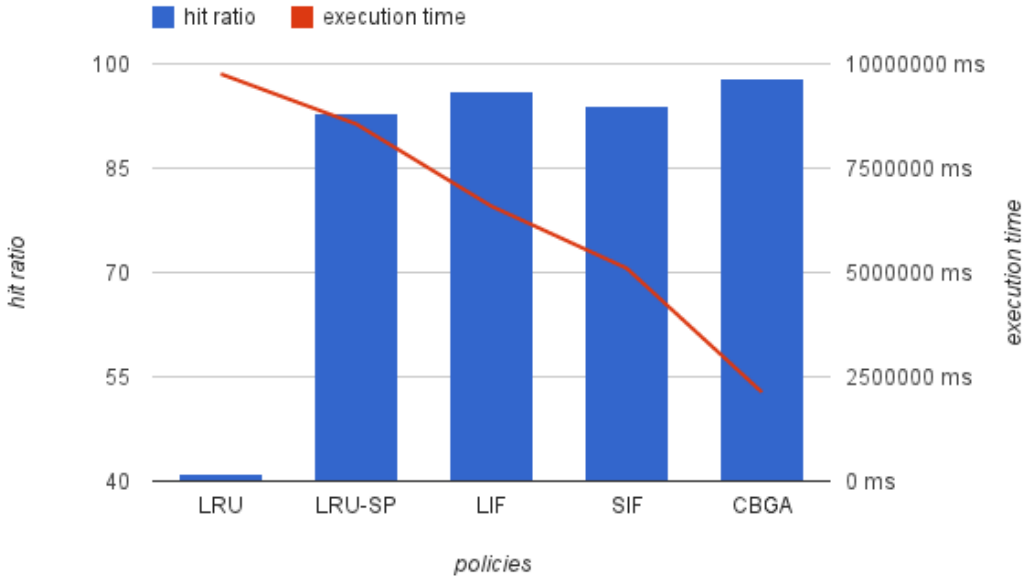


Figure 6.3: Performance for Twitter dataset under 10M cache and 10K queries in which the first 500 are warmup queries. Left y axis shows hit ratio while right y axis shows execution times in msec.

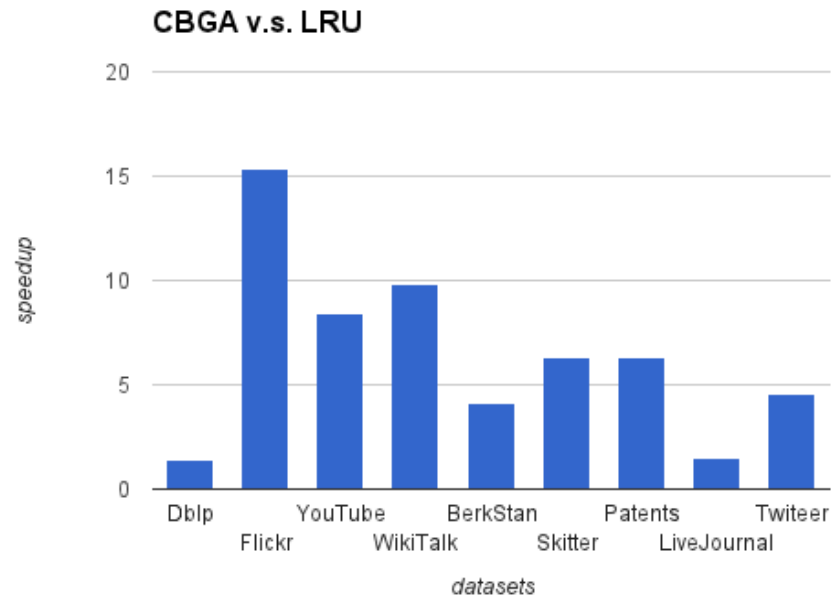


Figure 6.4: Speedup achieved for each dataset when CBGA and LRU are compared.

Figure 6.5 presents the performance of the policies under long runs for Flickr dataset. For increasing number of queries from 10K to 100K, our CBGA policy provides stable lowest execution time.

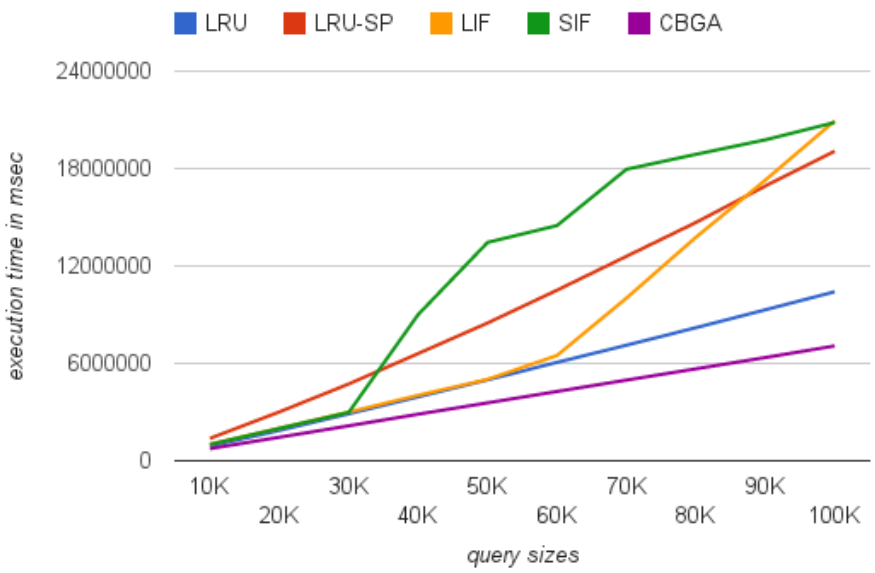


Figure 6.5: Performance of various policies under long runs of Flickr dataset.

We trace the query execution times and Figure 6.6 shows that the average query execution time initially decreases and then stabilizes as the caches warm up. Since queries are random and their overhead is not equal (e.g., a vertex might have 10 neighbors in two hops while another vertex have 10000 neighbors in two hops) individual query times display some fluctuation. Thus, we also compute bintime line shown in red which displays the average execution time for the last 10 queries. Similarly, the average number of queries executed per minute increases and then stabilizes while cache warms up as displayed in Figure 6.7. Here we provide two representatives from all datasets to save space.

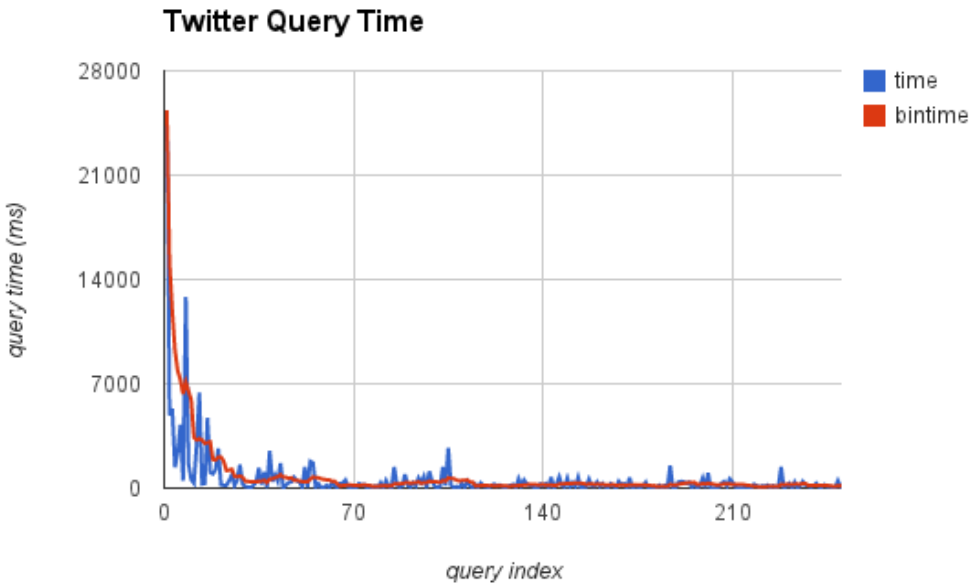


Figure 6.6: Average query time is decreased while cache warms up for Twitter dataset. Red bintime line displays the average execution time for the last 10 queries instead of individual queries.

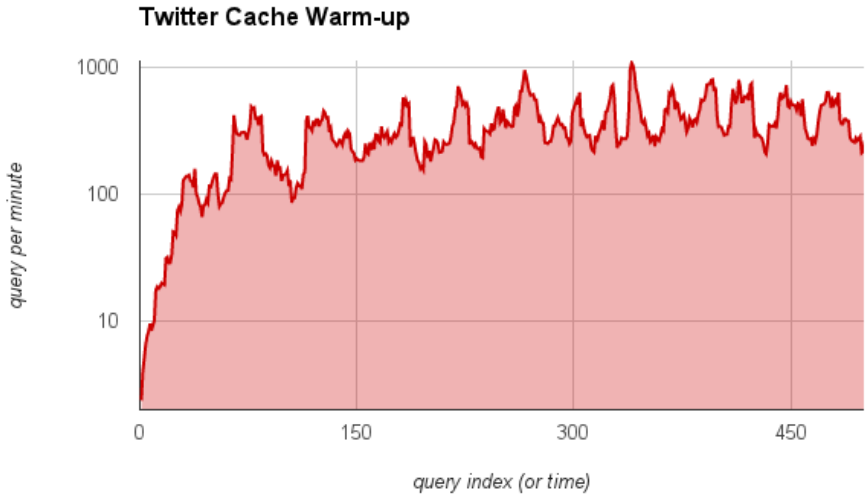


Figure 6.7: The number of queries processed per minute increase while cache warms up for Twitter dataset. A stable high query-per-minute performance is observed when the cache is warm.

6.5 Conclusion

To the best of our knowledge, this study is the first to propose a graph aware caching scheme for efficient graph processing in horizontally scaling solutions on big data platforms. We proposed a clock based graph aware cache (CBGA) system with cache and eviction algorithms designed with distributed graph processing context in mind. We ran experiments on our HBASE cluster, which demonstrate up to 15x speedup compared to traditional LRU based cache systems.

We provided a distributed implementation of the caching 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.

Chapter 7

Conclusions and Future Work

Collecting social network data is traditionally difficult, requiring extensive contact with the group of people being studied. Practically, research efforts are generally limited to between tens and hundreds of individuals [99, 100]. On the other hand, social interactions over telco infrastructures generate detailed traces of interactions and movements. Large-scale networks, even ones covering a whole society, can be generated from such traces. The ability to construct such rich and representative social networks makes it feasible to develop and evaluate social network models.

One of the most interesting properties summarizing the structure of a social network is the degree distribution of nodes. The degree of a social node is the number of other nodes the node has a social interaction. Different observations exist regarding degree distribution in social networks. For instance, some works (e.g., [20, 101]) claim that the degree distribution follows power-law distribution, while others (e.g., [23]) claim it follows double Pareto log-normal distribution. Using different datasets, different degree distributions have been obtained in the literature.

In this thesis, we first attempt to empirically test degree distribution versus different dataset scenarios to understand the parameters governing degree distribution in social networks. We observe that degree distribution in social networks

does not show a significant correlation with population density, user telco operator, and user geographic location; however, population size directly affects the average degree of social network. Therefore, it is important to keep social network size as a parameter while interpreting degree distribution. It also seems acceptable to study a social network without considering its location, density and referred telco operator. For instance, a researcher could gather data from an urban part or a rural part of a country, or may choose a specific city or telco operator. However, any change in the size of the studied network would result in a considerable change in degree distribution characteristics and overall network topology. Hence, social network studies must indicate the size of the studied network and consider different size cases to come up with a sound and complete conclusion.

We also study community identification problem in social networks which is reduced to k-core metric in graph theory. To the best of our knowledge, our study is the first to propose a horizontally scaling solution on the big data platform for multiresolution social network community identification and maintenance. By using k-core as the measure of community intensity, we propose multi-k-core construction and incremental maintenance algorithms and run experiments to demonstrate orders of magnitude speedup with the aggressive pruning and fairly low maintenance overhead in the majority of graph updates at relatively high k-valued cores. We further extend algorithms to handle batch maintenance of a window of updates, which provides larger and more stable speedup for multi-k-core maintenance.

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

Finally, we propose a clock based graph aware cache system with cache and eviction algorithms designed with distributed graph processing context in mind. We provide experimental results on our HBase cluster, which demonstrate up to

15x speedup compared to traditional LRU based cache systems.

We provide 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 takes 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.

The studies presented in this thesis can be extended in various dimensions. The analysis that is presented in Chapter 3 on degree distribution can be repeated for other social network characteristics. For instance, the effect of network size, density, operator, and location on clustering co-efficient or k-core distribution can be discovered. Also, temporal evaluation of social network characteristics can be studies using datasets with time attributes.

On the other hand, coprocessor based distributed processing framework described in the Chapter 4 can be extended for other social network algorithms, e.g., centrality computation algorithms, link recommendation algorithms. In this way, a generic social network analysis framework on Big Data platform would be obtained.

Bibliography

[1] S. Wasserman and K. Faust, *Social network analysis: methods and applications (structural analysis in the social sciences)*. Cambridge University Press, January 1995.

[2] J. Palau, M. Montaner, B. Lpez, and J. de la Rosa, "Collaboration analysis in recommender systems using social networks," in *Cooperative Information Agents VIII* (M. Klusch, S. Ossowski, V. Kashyap, and R. Unland, eds.), vol. 3191 of *Lecture Notes in Computer Science*, pp. 137–151, Springer Berlin Heidelberg, 2004.

[3] P. Kazienko, K. Musial, and T. Kajdanowicz, "Multidimensional social network in the social recommender system," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 41, pp. 746–759, July 2011.

[4] J. Carrasco, D. Fain, K. Lang, and L. Zhukov, "Clustering of bipartite advertiser-keyword graph," in *Proceedings of international Conference on Data Mining*, (Melbourne, Florida), November 2003.

[5] C. Yang and T. Dorbin Ng, "Analyzing and visualizing web opinion development and social interactions with density-based clustering," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 41, pp. 1144–1155, November 2011.

[6] C. Lu, X. Hu, and J. R. Park, "Exploiting the social tagging network for web clustering," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 41, no. 5, pp. 840–852, 2011.

[7] J. Schroeder, J. Xu, and H. Chen, "Crimelink explorer: using domain knowledge to facilitate automated crime association analysis," in *Intelligence and Security Informatics* (H. Chen, R. Miranda, D. Zeng, C. Demchak, J. Schroeder, and T. Madhusudan, eds.), vol. 2665 of *Lecture Notes in Computer Science*, pp. 168–180, Springer Berlin Heidelberg, 2003.

[8] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," in *Proceedings of the National Academy of Sciences*, vol. 99, pp. 7821–7826, June 2002.

[9] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral, "Modularity from fluctuations in random graphs and complex networks," *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, vol. 70, no. 2, 2004.

[10] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," in *Proceedings of the National Academy of Sciences*, vol. 104, pp. 36–41, National Academy of Sciences, January 2007.

[11] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, pp. 066111+, December 2004.

[12] B. Karrer, E. Levina, and M. E. J. Newman, "Robustness of community structure in networks," *Physical Review E*, vol. 77, pp. 046119+, September 2007.

[13] M. Cha, F. Benevenuto, H. Haddadi, and P. K. Gummadi, "The world of connections and information flow in twitter," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 42, no. 4, pp. 991–998, 2012.

[14] C. Ratti, Pulselli, S. Williams, and D. Frenchman, "Mobile landscapes: using location data from cell-phones for urban analysis," *Environment and Planning B: Planning and Design*, vol. 33, no. 5, pp. 727–748, 2006.

[15] D. J. Watts, "A twenty-first century science," *Nature*, vol. 445, p. 489, January 2007.

[16] A. Wesolowski, N. Eagle, A. J. Tatem, D. L. Smith, A. M. Noor, R. W. Snow, and C. O. Buckee, "Quantifying the impact of human mobility on malaria," *Science*, vol. 338, no. 6104, pp. 267–270, 2012.

[17] N. Eagle, A. S. Pentland, and D. Lazer, "Inferring friendship network structure by using mobile phone data," in *Proceedings of the National Academy of Sciences*, vol. 106, pp. 15274–15278, National Acad Sciences, 2009.

[18] A. Le Menach, A. J. Tatem, J. M. Cohen, S. I. Hay, H. Randell, A. P. Patil, and D. L. Smith, "Travel risk, malaria importation and malaria transmission in zanzibar," *Scientific Reports*, vol. 1, 2011.

[19] W. Aiello, F. Chung, and L. Lu, "A random graph model for massive graphs," in *Proceedings of the 32nd annual ACM symposium on Theory of computing*, STOC '00, (New York, NY, USA), pp. 171–180, ACM, 2000.

[20] J. P. Onnela, J. Saramäki, J. Hyvönen, G. Szabó, D. Lazer, K. Kaski, J. Kertész, and A. L. Barabási, "Structure and tie strengths in mobile communication networks," in *Proceedings of the National Academy of Sciences*, vol. 104, pp. 7332–7336, May 2007.

[21] K. Dasgupta, R. Singh, B. Viswanathan, D. Chakraborty, S. Mukherjea, A. A. Nanavati, and A. Joshi, "Social ties and their relevance to churn in mobile telecom networks," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, (New York, NY, USA), pp. 668–677, ACM, 2008.

[22] Z. Bi, C. Faloutsos, and F. Korn, "The "dgx" distribution for mining massive, skewed data," in *Proceedings of the 7th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, (New York, NY, USA), pp. 17–26, ACM, 2001.

[23] M. Seshadri, S. Machiraju, A. Sridharan, J. Bolot, C. Faloutsos, and J. Leskove, "Mobile call graphs: beyond power-law and lognormal distributions," in *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, (New York, NY, USA), pp. 596–604, ACM, 2008.

[24] A. Sala, S. Gaito, G. P. Rossi, H. Zheng, and B. Y. Zhao, "Revisiting degree distribution models for social graph analysis," *arXiv preprint arXiv:1108.0027*, vol. abs/1108.0027, 2011.

[25] M. J. Quinn and N. Deo, "Parallel graph algorithms," *ACM Computing Surveys*, vol. 16, pp. 319–348, September 1984.

[26] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Out-of-core coherent closed quasi-clique mining from large dense graph databases," *ACM Transactions on Database Systems*, vol. 32, June 2007.

[27] 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*, EDBT '12, (New York, NY, USA), pp. 480–491, ACM, 2012.

[28] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *arXiv preprint arXiv:1207.4567*, vol. cs.DS/0310049, 2003.

[29] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "Evaluating cooperation in communities with the k-core structure," in *Proceedings of the 2011 International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '11, (Washington, DC, USA), pp. 87–93, IEEE Computer Society, 2011.

[30] C. Giatsidis, K. Berberich, D. M. Thilikos, and M. Vazirgiannis, "Visual exploration of collaboration networks based on graph degeneracy," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, (New York, NY, USA), pp. 1512–1515, ACM, 2012.

[31] W. Wei and S. Ram, "Using a network analysis approach for organizing social bookmarking tags and enabling web content discovery," *ACM Transactions on Management Information Systems*, vol. 3, pp. 15:1–15:16, October 2012.

[32] 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," in *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, (New York, NY, USA), pp. 57–70, ACM, 2008.

[33] "Followerwonk: twitter analytics, follower segmentation, social graph tracking." Available from: `http://followerwonk.com/`, Last Access: July, 2014.

[34] "The original twitter wall." Available from: `http://tweetwall.com/`, Last Access: July, 2014.

[35] "Easy social media analytics & measurement — simply measured." Available from: `http://simplymeasured.com/`, Last Access: July, 2014.

[36] "Hbase - apache hbase home." Available from: `http://hbase.apache.org/`, Last Access: July, 2014.

[37] T. Luczak, "Size and connectivity of the k-core of a random graph," *Discrete Mathematics*, vol. 91, pp. 61–68, July 1991.

[38] B. Pittel, J. Spencer, and N. Wormald, "Sudden emergence of a giant k-core in a random graph," *Journal of Combinatorial Theory Series B*, vol. 67, pp. 111–151, May 1996.

[39] C. Cooper, "The cores of random hypergraphs with a given degree sequence," *Random Structures & Algorithms*, vol. 25, pp. 353–375, December 2004.

[40] M. Molloy, "Cores in random hypergraphs and boolean formulas," *Random Structures & Algorithms*, vol. 27, pp. 124–135, August 2005.

[41] S. Janson and M. J. Luczak, "A simple solution to the k-core problem," *Random Structures & Algorithms*, vol. 30, pp. 50–62, January 2007.

[42] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269 – 287, 1983.

[43] J. I. A. Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the visualization of large scale networks," *arXiv preprint arXiv:1207.4567*, vol. abs/cs/0504107, 2005.

[44] J. I. A. Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Advances in neural information processing systems*, pp. 41–50, 2005.

[45] Batagelj, Mrvar, and Zaversnik, "Partitioning approach to visualization of large graphs," in *GDRAWING: Conference on Graph Drawing (GD)*, 1999.

[46] Baur, Brandes, Gaertler, and Wagner, "Drawing the as graph in 2.5 dimensions," in *GDRAWING: Conference on Graph Drawing (GD)*, 2004.

[47] Y. Zhang and S. Parthasarathy, "Extracting analyzing and visualizing triangle k-core motifs within networks," in *Data Engineering, 2012 IEEE 28th International Conference on*, pp. 1049–1060, April 2012.

[48] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "A model of internet topology using k-shell decomposition," in *Proceedings of the National Academy of Sciences*, vol. 104, pp. 11150–11154, National Acad Sciences, 2007.

[49] J. I. A. Hamelin, M. G. Beiró, and J. R. Busch, "Understanding edge connectivity in the internet through core decomposition," *Internet Mathematics*, vol. 7, no. 1, pp. 45–66, 2011.

[50] 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.

[51] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, vol. 588 of *Contemporary Mathematics*, American Mathematical Society, 2013.

[52] M. Altaf-Ul-Amin, K. Nishikata, T. Koma, T. Miyasato, Y. Shinbo, M. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori, *et al.*, "Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences," *Genome Informatics Series*, pp. 498–499, 2003.

[53] G. D. Bader and C. W. V. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC Bioinformatics*, vol. 4, p. 2, 2003.

[54] S. Wuchty and E. Almaas, "Peeling the yeast protein network," *Proteomics*, vol. 5, no. 2, pp. 444–449, 2005.

[55] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *ICDE* (S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, eds.), pp. 51–62, IEEE Computer Society, 2011.

[56] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 2, pp. 288–300, 2013.

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

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

[59] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," in *Proceedings of the VLDB Endowment*, vol. 6, pp. 433–444, 2013.

[60] N. P. Nguyen, T. N. Dinh, S. Tokala, and M. T. Thai, "Overlapping communities in dynamic networks: their detection and mobile applications," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, (New York, NY, USA), pp. 85–96, ACM, 2011.

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

[62] 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.

[63] 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*, KDD '07, (New York, NY, USA), pp. 717–726, ACM, 2007.

[64] J. Greiner, "A comparison of parallel algorithms for connected components," in *Proceedings of the 6th annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, (New York, NY, USA), pp. 16–25, ACM, 1994.

[65] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, January 2008.

[66] "Welcome to apache hadoop!." Available from: `http://hadoop.apache.org/`, Last Access: July, 2014.

[67] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science Engineering*, vol. 11, pp. 29 –41, july-aug. 2009.

[68] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the 8th Workshop on Mining and Learning with Graphs*, MLG '10, (New York, NY, USA), pp. 78–85, ACM, 2010.

[69] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: mining peta-scale graphs," *Knowledge and Information Systems*, vol. 27, pp. 303–325, May 2011.

[70] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: a scalable and general graph management system," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, (New York, NY, USA), pp. 1091–1099, ACM, 2011.

[71] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," in *Proceedings of the VLDB Endowment*, vol. 3, pp. 285–296, VLDB Endowment, September 2010.

[72] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," in *Proceedings of the VLDB Endowment*, vol. 4, VLDB Endowment, September 2011.

[73] 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*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[74] "Giraph - welcome to apache giraph!." Available from: `http://giraph.apache.org/`, Last Access: July, 2014.

[75] "Hama - a general bsp framework on top of hadoop." Available from: `http://hama.apache.org/`, Last Access: July, 2014.

[76] "Objectivity infinitegraph." Available from: `http://infinitegraph.com/`, Last Access: July, 2014.

[77] "Trinity - microsoft research." Available from: `http://research.microsoft.com/en-us/projects/trinity/`, Last Access: July, 2014.

[78] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[79] "Project voldemort: a distributed database." Online, March 2012.

[80] "Redis." Available from: `http://redis.io`, Last Access: July, 2014.

[81] "The apache cassandra project." Available from: `http://cassandra.apache.org`, Last Access: July, 2014.

[82] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at

facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pp. 385–398, 2013.

[83] C. Xu, X. Huang, N. Wu, P. Xu, and G. Yang, "Using memcached to promote read throughput in massive small-file storage system," in *Grid and Cooperative Computing, 2010 9th International Conference on*, pp. 24–29, November 2010.

[84] "Neo4j - the world's leading graph database." Available from: `http://neo4j.org/`, Last Access: July, 2014.

[85] "Caches in neo4j - the neo4j manual v2.1.2." Available from: `http://docs.neo4j.org/chunked/milestone/configuration-caches.html`, Last Access: July, 2014.

[86] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, *et al.*, "Tao: Facebook's distributed data store for the social graph," in *USENIX Annual Technical Conference*, pp. 49–60, 2013.

[87] 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 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[88] J.-K. Min and S.-B. Cho, "Mobile human network management and recommendation by probabilistic social mining," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 41, no. 3, pp. 761–771, 2011.

[89] D. Wang, D. Pedreschi, C. Song, F. Giannotti, and A. L. Barabasi, "Human mobility, social ties, and link prediction," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, (New York, NY, USA), pp. 1100–1108, ACM, 2011.

[90] S. Milgram, "The small world problem," *Psychology Today*, vol. 2, pp. 60–67, 1967.

[91] Å. Björck, *Numerical methods for least squares problems*. Philadelphia, PA: SIAM, 1996.

[92] R Development Core Team, *R: a language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[93] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Reviews*, June 2007.

[94] H. Aksu, M. Canim, Y. Chang, I. Korpeoglu, and O. Ulusoy, "Distributed k-core view materialization and maintenance for large dynamic graphs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[95] S. N. Dorogovtsev and J. F. Mendes, "Evolution of networks," *Advances in Physics*, vol. 51, no. 4, pp. 1079–1187, 2002.

[96] E. Ravasz and A. L. Barabási, "Hierarchical organization in complex networks," *Physical Review E*, vol. 67, pp. 026112+, February 2003.

[97] J. Leskovec and E. Horvitz, "Planetary-scale views on a large instant-messaging network," in *Proceeding of the 17th international conference on World Wide Web*, WWW '08, (New York, NY, USA), pp. 915–924, ACM, 2008.

[98] J. I. Alvarez-Hamelin, L. DallAsta, A. Barrat, and A. Vespignani, "Analysis and visualization of large scale networks using the k-core decomposition," in *European Conference on Complex Systems*, 2005.

[99] J. Kleinberg, "The convergence of social and technological networks," *Communications of the ACM*, vol. 51, pp. 66–72, November 2008.

[100] H. Zhang, R. Dantu, and J. W. Cangussu, "Socioscope: human relationship and behavior analysis in social networks," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 41, pp. 1122–1143, November 2011.

[101] A. A. Nanavati, S. Gurumurthy, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi, "On the structural properties of massive telecom call graphs: findings and implications," in *Proceedings of the 15th ACM international conference on Information and knowledge management*, (New York, NY, USA), pp. 435–444, ACM, 2006.

[102] "Twitter statistics." Available from: `http://www.statisticbrain.com/twitter-statistics`, Last Access: July, 2014.

[103] 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 Transactions on Computer Systems*, vol. 26, pp. 4:1–4:26, June 2008.

[104] 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," in *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 1071–1080, ACM, 2011.

[105] "Hbase/poweredby - hadoop wiki." Available from: `http://wiki.apache.org/hadoop/Hbase/PoweredBy/`, Last Access: July, 2014.

[106] "Coprocessor introduction : apache hbase." Available from: `http://blogs.apache.org/hbase/entry/coprocessor_introduction/`, Last Access: July, 2014.

[107] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 international conference on Management of data*, SIGMOD '12, ACM, 2012.

[108] 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.

[109] "Snap: Stanford network analysis project." Available from: `http://snap.stanford.edu/`, Last Access: July, 2014.

[110] "Ganglia monitoring system." Available from: `http://ganglia.info/`, Last Access: July, 2014.

[111] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review*, (Copper Mountain, CO), pp. 172–183, ACM SIGOPS, December 1995.

[112] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, pp. 40–44, January 2009.

[113] R. Zafarani and H. Liu, "Social computing data repository at ASU." Available from: `http://socialcomputing.asu.edu/`. Last Access: July, 2014.