Graph Data Mining with Arabesque

Eslam Hussein[△], Abdurrahman Ghanem[△], Vinicius Vitor dos Santos Dias[♣],
Carlos H. C. Teixeira[♣], Ghadeer AbuOda[◇],
Marco Serafini[△], Georgos Siganos[△], Gianmarco De Francisci Morales[△],
Ashraf Aboulnaga[△], Mohammed Zaki[♠]
[△]Qatar Computing Research Institute - HBKU, [♣]Universidade Federal de Minas Gerais,
[◇]College of Science and Engineering - HBKU, [♠]Rensselaer Polytechnic Institute

ABSTRACT

Graph data mining is defined as searching in an input graph for all subgraphs that satisfy some property that makes them interesting to the user. Examples of graph data mining problems include frequent subgraph mining, counting motifs, and enumerating cliques. These problems differ from other graph processing problems such as PageRank or shortest path in that graph data mining requires searching through an exponential number of subgraphs. Most current parallel graph analytics systems do not provide good support for graph data mining. One notable exception is Arabesque, a system that was built specifically to support graph data mining. Arabesque provides a simple programming model to express graph data mining computations, and a highly scalable and efficient implementation of this model, scaling to billions of subgraphs on hundreds of cores. This demonstration will showcase the Arabesque system, focusing on the end-user experience and showing how Arabesque can be used to simply and efficiently solve practical graph data mining problems that would be difficult with other systems.

1. INTRODUCTION

Graph data is playing an increasingly important role in many fields such as biology, e-commerce, and social network analysis. Graph data appears in on-line operations, such as representing new "friend" relationships in a social network, and in analytics, such as predicting users who can become friends. The increase in the size of graph data and the complexity of workloads on this data have led to the development of parallel and distributed systems that support high throughput graph updates and retrieval, such as TAO [2], as well as systems that support large scale graph analytics, such as Pregel [10], GraphLab [9], and EmptyHeaded [1].

Most parallel graph analytics systems support computations that produce succinct properties of the graph or of individual vertices, or produce a small number of result subgraphs. Examples of these computations include PageRank, shortest path, and counting cliques. These systems do not provide good support for *graph data mining*, which we define as searching through the exponential number of subgraphs of an input graph to find subgraphs that satisfy some property that makes them interesting to the user. Examples of graph data mining problems include frequent subgraph mining and enumerating cliques or quasi-cliques (as opposed to only counting them). Some recent systems, such as Arabesque [15] and NScale [14], adopt a model where subgraphs (as opposed to vertices) are first class citizens in the computation, which enables better support for graph data mining.

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4197-4/17/05. . . $15.00

DOI: http://dx.doi.org/10.1145/3035918.3058742

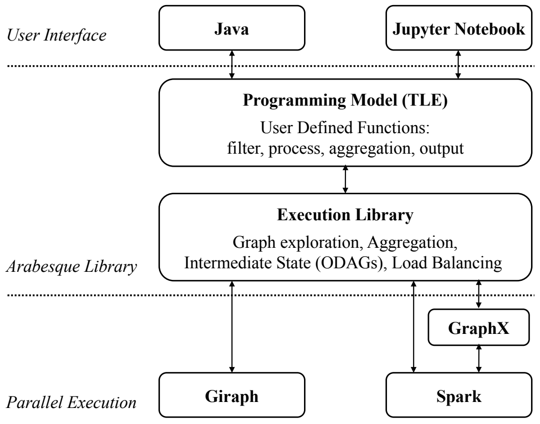


Figure 1: Overview of Arabesque.

This demonstration will showcase the Arabesque system, which was built with the specific goal of supporting efficient and scalable graph data mining. The technical details of Arabesque are presented elsewhere [15], and the code is available as open source[1]. This demonstration will focus on the end-user experience of Arabesque, and how it can be used to solve interesting and important graph data mining problems that would be difficult with other graph analytics systems. Participants in the demonstration will see how Arabesque fits within a typical data analytics toolchain, and will get a sense of the usability, programmability, and efficiency of Arabesque. The demonstration will be centered around three practical applications: finding frequent subgraphs in protein databases, analyzing cliques to detect communities of common interest among buyers on Amazon, and analyzing motifs to measure the reaction to various events on Twitter. The three applications are based on three different types of graph data mining problems supported by Arabesque: frequent subgraph mining, enumerating cliques, and counting motifs. In the next section, we present a brief overview of Arabesque, and in Section 3 we present the details of these applications.

[1]http://arabesque.io

2. OVERVIEW OF ARABESQUE

As mentioned earlier, graph data mining is characterized by enumerating the exponential number of subgraphs of an input graph and searching for patterns in these subgraphs. The Arabesque system [15] (Figure 1) is designed to support parallel graph data mining on hundreds of CPU cores in multiple servers (also referred to as worker nodes). A fundamental assumption made by Arabesque is that the input graph fits in the main memory of a single worker node, and can be replicated on all worker nodes. Today, the main memory of servers is typically in the 256GB to 2TB range, so this assumption covers a large fraction of graph data sets. Arabesque still needs to address the challenge of managing the exponentially sized intermediate state, which does *not* fit in the memory of a single worker node. Another challenge faced by Arabesque is distributing the computation to the CPU cores in a scalable, efficient, and load balanced way.

To address these challenges, Arabesque uses a programming model that can express graph data mining problems in a simple and succinct way, and is amenable to easy distribution on multiple cores. Arabesque also provides a scalable and efficient implementation of this programming model that works on top of parallel dataflow platforms such as Giraph[2] and Spark [16]. We present the Arabesque programming model and implementation next.

2.1 Programming Model

The Arabesque programming model is designed to support the automatic graph exploration required for graph data mining. It is based on a paradigm that we call *think like an embedding*, or *TLE*. An *embedding* is a subgraph representing an instance of a *pattern* of interest in the graph data mining problem, and a key characteristic of graph data mining is that we are interested in producing all output embeddings. For example, consider frequent subgraph mining, in which we want to find all instances of frequently occurring subgraph patterns. If subgraphs with, say, a vertex labeled A connected to a vertex labeled B connected to a vertex labeled C occur frequently in the input graph, we are interested not only in finding that $A - B - C$ is frequent but also in producing all instances of $A-B-C$, say, $a_1-b_1-c_1, a_2-b_2-c_2, \ldots, a_n-b_n-c_n$. In this example, $A - B - C$ is the *pattern*, and the instances $a_i - b_i - c_i$ are the *embeddings* of this pattern, where lowercase letters indicate vertex ids in the input graph. A similar distinction between patterns and embeddings can be found in other graph data mining problems.

In the TLE programming model of Arabesque, the user provides two functions that accept one embedding as an argument: the *filter* function and the *process* function. The filter function is used to prune this search space: it takes an embedding as input and returns a boolean value indicating whether the embedding should be processed or not. The process function is used to analyze an embedding and generate the output required by the graph mining algorithm: it takes an embedding as input, processes the embedding as required by the graph data mining algorithm, and typically outputs a set of user-defined values to HDFS.

Arabesque explores the input graph in a series of bulk synchronous parallel (BSP) steps, and maintains a set of candidate embeddings at each step. In the first step, the individual vertices of the input graph are the candidate embeddings, and in each subsequent step, each candidate embedding is expanded by adding its neighbors to it one by one to create larger candidate embeddings. In each step, Arabesque calls the filter function on all candidate embeddings, and discards the embeddings for which filter returns false from the candidate set. Arabesque then calls the process functions on embeddings remaining in the candidate set, and further expands these embeddings in subsequent BSP steps.

In addition to the filter and process functions, Arabesque allows the user to specify other functions such as an aggregation filter function and an aggregation process function, which filter and process embeddings at the beginning of a BSP step based on aggregate information about all the embeddings found in the previous step.

2.2 Implementation

A key characteristic of the TLE model is that there are no dependencies among embeddings. Each embedding can be filtered, processed, and expanded independently of other embeddings within a BSP step. Embeddings may be aggregated by pattern at the end of a BSP step, which introduces dependencies, but there are no dependencies within a step. The lack of dependencies enables Arabesque to utilize a coordination free strategy to avoid redundant work while exploring the graph based on the concept of *embedding canonicality*. Each worker thread is assigned a set of embeddings to expand in each step, without coordinating with other worker threads. It is possible that two worker threads generate the same embedding independently. Without additional controls, both workers would call the filter and process functions on the same embedding. To avoid this situation, Arabesque defines a notion of canonicality for embeddings, and worker threads discard embeddings that they generate that are not canonical. To ensure coordination free exploration, canonicality in Arabesque is defined in a careful way that allows each thread to independently test the canonicality of embeddings that it generates.

The ability to expand embeddings independently enables Arabesque to balance load very well among the worker threads. Contrast this, for example, to the traditional *think like a vertex (TLV)* paradigm used by graph analytics systems such as Pregel. In TLV, computation and state are expressed at the level of a vertex in the input graph. One could use TLV for graph exploration by storing at each vertex all embeddings that this vertex is part of. The vertex function can expand an embedding by adding neighbors of the vertex that are not already in the embedding. New embeddings would have to be sent to all vertices that these embeddings contain. This further multiplies the number of embeddings generated by the system, exacerbating the main bottleneck of graph mining algorithms. In addition, highly connected vertices generate a disproportionately large number of embeddings during expansion, leading to load imbalance. In our experiments, we have observed TLV to be up to two orders of magnitude slower than TLE.

As Arabesque explores the graph, it generates an exponential number of embeddings. To reduce the memory required for storing these embeddings, Arabesque uses a compact data structure called *Overapproximating Directed Acyclic Graph (ODAG)* that compresses the canonical embeddings generated in a BSP step. It also uses a *two-level aggregation* technique to speed up aggregation by pattern.

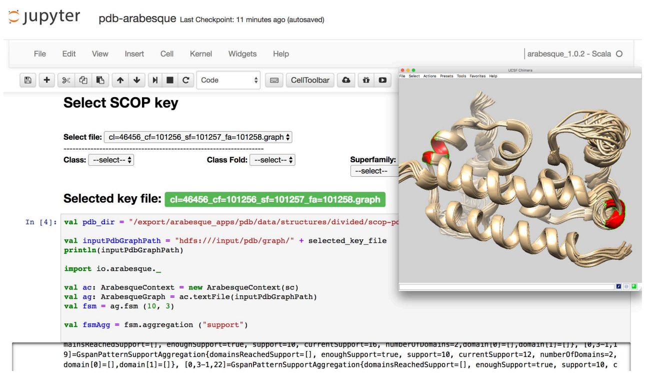
The full technical details of Arabesque are presented in [15]. In that paper, we show that Arabesque scales to billions of subgraphs on hundreds of cores on multiple worker nodes. From a software engineering perspective, Arabesque is implemented as a library that can easily be ported to any parallel dataflow execution engine.

We currently have versions of Arabesque that runs on top of Giraph[3] and Spark[4]. Note that the Giraph version does not use the TLV programming model that Giraph implements. Arabesque uses Giraph only to deploy a set of workers across multiple machines,

[2]http://giraph.apache.org

[3]https://github.com/Qatar-Computing-Research-Institute/Arabesque

[4]https://github.com/viniciusvdias/Arabesque



Select SCOP key

Select file: cl=46456_cf=101256_sf=101257_fa=101258.graph

Class: --select-- Class Fold: --select-- Superfamily: --select--

Selected key file: cl=46456_cf=101256_sf=101257_fa=101258.graph

```
In [4]: val pdb_dir = "/export/arabesque_apps/pdb/data/structures/divided/scop-pc

        val inputPdbGraphPath = "hdfs:///input/pdb/graph/" + selected_key_file
        println(inputPdbGraphPath)

        import io.arabesque._

        val ac: ArabesqueContext = new ArabesqueContext(sc)
        val ag: ArabesqueGraph = ac.textFile(inputPdbGraphPath)
        val fsm = ag.fsm (10, 3)

        val fsmAgg = fsm.aggregation ("support")
```

```
mainsReachedSupport=[], enoughSupport=true, support=10, currentSupport=16, numberOfDomains=2,domain[0]=[],domain[1]=[]}, [0,3-1,1
9]=GspanPatternSupportAggregation{domainsReachedSupport=[], enoughSupport=true, support=10, currentSupport=12, numberOfDomains=2,
domain[0]=[],domain[1]=[]}, [0,3-1,22]=GspanPatternSupportAggregation{domainsReachedSupport=[], enoughSupport=true, support=10, c
```

Figure 2: Mining protein structures with Arabesque.

and to have worker communicate to each other in an all-to-all fashion through message exchanges at the boundaries of the BSP steps. It also relies on Giraph for fault tolerance.

The Spark version of Arabesque, which we use in this demo, also implements graph exploration in BSP steps, taking advantage of Spark's ability to keep results in memory between iterations through the use of resilient distributed datasets (RDDs). It is a native implementation on top of Spark, not GraphX [4], although it interfaces with GraphX for tasks that combine graph mining and vertex-centric graph processing. Specifically, GraphX can be used to pre-process the Arabesque input graphs or post-process the output graphs. For additional usability, Arabesque on Spark has a Jupyter notebook interface[5] in addition to the Java interface. Next, we describe the applications that we use to showcase Arabesque.

3. APPLICATIONS DEMONSTRATED

In this demonstration we will present three applications that use Arabesque: finding frequent structures in a protein database, finding communities of common interest among buyers from Amazon, and modeling how users propagate information on Twitter in reaction to events. The graph data mining algorithms used by the applications are, respectively, frequent subgraph mining, enumerating cliques, and counting motifs.

3.1 Finding Frequent Structures in Proteins

The aim of this application is to identify and visualize frequently occurring patterns in the 3D structure of proteins. The input data for this application is data from the Protein Data Bank (PDB)[6], which is an online repository containing the 3D structure of over 120K proteins.

The structure of a protein can be converted to a graph as follows. Each protein structure comprises a set of, say n, 3D coordinates, namely (x_i, y_i, z_i), for $i = 1, \dots n$. Each position or element i, also called an amino acid i, has a label. Let us denote $a_i = (x_i, y_i, z_i)$, and let l_i be its amino acid label. We can construct a graph for each protein, with a vertex for each amino acid, labeled with l_i. An edge exists between any two amino acids, a_i and a_j, if the Euclidean distance between them is below a given threshold, that is, $||a_i - a_j||_2 \leq \theta$, where θ is the contact threshold (usually set to 7 angstroms, i.e., 7×10^{-10} meters). The graphs for a set of proteins can be considered as disconnected components in an input graph to Arabesque.

Frequent subgraphs in these protein graphs represent frequently occurring patterns among the different protein structures. Identifying such frequently occurring patterns is important for many bioinformatics applications (e.g., [5, 6, 7, 11]). As a matter of fact, one of the ways to classify proteins in the PDB database is to group proteins by structure in a hierarchical organization, as in the structural classification of proteins (SCOP) project[7] [3], which uses manually identified, human-curated structural groupings. A scalable frequent subgraph mining implementation would be extremely helpful for bioinformatics applications on the PDB database, and in the demonstration we show how Arabesque can play this role.

This application, like the other application in this demonstration, runs in the Jupyter notebook interface of Arabesque (Figure 2). The steps of the application are (1) extracting a relevant subset of the PDB database and constructing the input graph, (2) running frequent subgraph mining on the input graph, and (3) visualizing the frequent subgraphs found by Arabesque.

The goal of the application is to identify frequently occurring structures in a coherent subset of the PDB database. We use the SCOP classification to identify such a coherent subset. Thus, the first step of the application is for the user to choose one node of the SCOP hierarchy (referred to as a "SCOP key"), and to specify the threshold θ for adding an edge between two amino acids. The application uses these user inputs to extract the relevant PDB data and construct the input graph.

Next, the application runs frequent subgraph mining on Arabesque. The user controls this step by specifying the required support and the maximum subgraph size explored. The output of this step is a set of subgraphs representing frequently occurring structures in the input protein data.

During the demo, attendees will follow the steps above and visualize the frequently occurring structures identified by Arabesque. We use an external tool called UCSF Chimera[8] [13] for visualization. UCSF Chimera is a popular and powerful visualization tool that is widely used in the bioinformatics community, and users are able to take advantage of its full power for visualization.

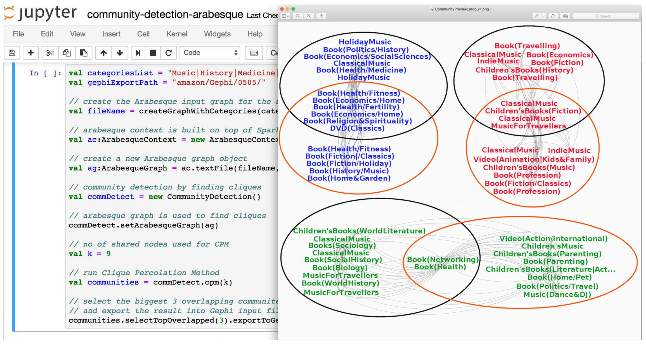
3.2 Finding Communities on Amazon

This application uses Arabesque to find communities in a co-purchase graph of the Amazon online shopping site. In particular, we use the k-clique percolation method [12], which is an established algorithm to find communities in a network. The method starts by identifying all maximal cliques. It then considers two cliques to be adjacent if they have $k-1$ common nodes. All cliques that are adjacent to each other, either directly or transitively, are considered to be part of the same community.

For this application, we use the Amazon co-purchase graph of [8]. In this graph, vertices correspond to items on sale, and each vertex is labeled with one or more categories that the item belongs to. An edge connects two items that are bought together in the same order. Therefore, communities represent items that are often bought together, and it is interesting to examine the categories of these items.

The application (Figure 3) consists of the following steps: (1) building the Arabesque input graph, (2) using Arabesque to find maximal cliques, (3) identifying adjacent cliques, (4) finding connected components/communities, and (5) visualizing some of the overlapping communities. Participants in the demonstration will be able to select a set of categories and only include vertices of these categories and their neighbors in the input graph. Participants will

[5]http://jupyter.org
[6]http://www.rcsb.org/pdb
[7]https://scop.berkeley.edu
[8]http://www.rbvi.ucsf.edu/chimera



```
In [ ]: val categoriesList = "Music|History|Medicine|
        val gephiExportPath = "amazon/Gephi/0505/"

        // create the Arabesque input graph for the s
        val fileName = createGraphWithCategories(cate

        // arabesque context is built on top of Spark
        val ac:ArabesqueContext = new ArabesqueContex

        // create a new Arabesque graph object
        val ag:ArabesqueGraph = ac.textFile(fileName,

        // community detection by finding cliques
        val commDetect = new CommunityDetection()

        // arabesque graph is used to find cliques
        commDetect.setArabesqueGraph(ag)

        // no of shared nodes used for CPM
        val k = 9

        // run Clique Percolation Method
        val communities = commDetect.cpm(k)

        // select the biggest 3 overlapping communite
        // and export the result into Gephi input fi
        communities.selectTopOverlapped(3).exportToGe
```

Figure 3: Finding communities on Amazon.

also be able to change the size k of the cliques used in the clique percolation method (e.g., Figure 3 uses $k = 10$).

The visualizations produced in this application show pairs of communities and the overlap between them. The user can choose the pairs of communities to show and the categories in these communities. We have observed that communities with a large overlap usually contain categories that are subjectively similar (e.g., personal wellness, spirituality, and home care). On the other hand, communities with a small overlap are more diverse (e.g., biology and parenting). Demo attendees will be able to explore how the degree of overlap between communities affects their coherence.

3.3 Reaction to Events on Twitter

This application shows how to use subgraph counting to understand the reaction of social networks to exogenous events. In particular, we look at Twitter, and analyze the retweet network structure before and after the occurrence of an event. We demonstrate how k-*profiles*, i.e., the counts of all the subgraphs of size k, can detect the occurrence of an event. The procedure is as follows.

Given a dataset consisting of tweets that span several weeks, for each week in the dataset where there are more than, say, 500 retweets. Construct a retweet network $G_i(V, E)$ such that the set of vertices V corresponds to the set of active users in the dataset, and there is an edge $(u, v) \in E$ iff user u retweets user v (for the purposes of this demo, we ignore the direction of the edge).

K-profiles are a useful generalization of triangle counting. For $k = 3$, there are 4 different possible subgraphs, as shown in Figure 4. For $k = 4$, there are 11 different ones. By counting the number of occurrences of each of these subgraph in the retweet network, we can extract a multi-dimensional vector that represents a *fingerprint* of the network. For simplicity, we only count connected subgraphs (e.g., only the last two graphs in Figure 4).

To detect the occurrence of an event, extract the 3- or 4-profile of each retweet network, corresponding to each week in the dataset, and then compute the Euclidean distance between consecutive vectors. A spike in the Euclidean distance indicates a significant change in retweet patterns, which is likely due to a real-world event. An example of such event would be the spreading of some important or controversial news in the media.

In the demo we will use a dataset consisting of several months of tweets related to an event in April 2016, when the Egyptian government announced that control of two islands in the Red Sea will be ceded to Saudi Arabia. This generated significant change in the network activity on Twitter, which is detected by our application as a clear peak in the inter-week Euclidean distance between 4-profiles (Figure 5).

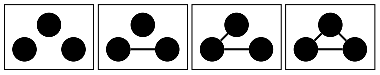
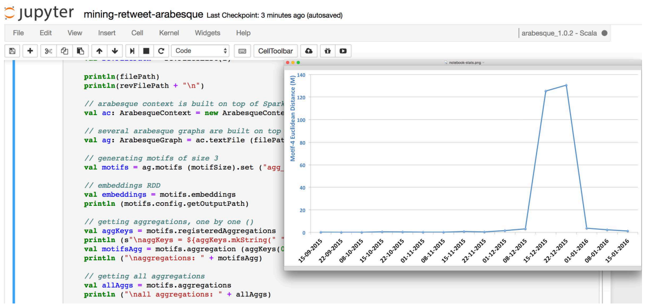


Figure 4: The 4 possible graphs on 3 vertices.



```
println(filePath)
println(revFilePath + "\n")

// arabesque context is built on top of Spark
val ac: ArabesqueContext = new ArabesqueConte

// several arabesque graphs are built on top
val ag: ArabesqueGraph = ac.textFile (filePat

// generating motifs of size 3
val motifs = ag.motifs (motifSize).set ("agg_

// embeddings RDD
val embeddings = motifs.embeddings
println (motifs.config.getOutputPath)

// getting aggregations, one by one ()
val aggKeys = motifs.registeredAggregations
println (s"\naggKeys = ${aggKeys.mkString(" "
val motifsAgg = motifs.aggregation (aggKeys(0
println ("\naggregations: " + motifsAgg)

// getting all aggregations
val allAggs = motifs.aggregations
println ("\nall aggregations: " + allAggs)
```

Figure 5: Detecting events on Twitter with Arabesque.

4. CONCLUSION

Arabesque is a scalable and efficient parallel system for graph data mining. It enables users to solve problems not easily solvable by other systems. This demonstration will present the user/programmer experience with Arabesque, focusing on complete applications that showcase different capabilities of the system.

References

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. EmptyHeaded: A relational engine for graph processing. In *SIGMOD*, 2016.
[2] N. Bronson et al. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, 2013.
[3] N. K. Fox, S. E. Brenner, and J.-M. Chandonia. SCOPe: Structural Classification of Proteins – extended, integrating SCOP and ASTRAL data and classification of new structures. *Nucleic Acids Research*, 42(D1), 2014.
[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
[5] J. Hu, X. Shen, Y. Shao, C. Bystroff, and M. J. Zaki. Mining protein contact maps. In *BIOKDD*, 2002.
[6] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining protein family specific residue packing patterns from protein structure graphs. In *Proc. Int. Conf. on Resaerch in Computational Molecular Biology*, 2004.
[7] M. Jambon, A. Imberty, G. Deléage, and C. Geourjon. A new bioinformatic approach to detect common 3D sites in protein structures. *Proteins: Structure, Function, and Bioinformatics*, 52(2), 2003.
[8] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5, 2007.
[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
[10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[11] P. Meysman, C. Zhou, B. Cule, B. Goethals, and K. Laukens. Mining the entire protein databank for frequent spatially cohesive amino acid patterns. *BioData Mining*, 8, 2015.
[12] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043), 2005.
[13] E. F. Pettersen, T. D. Goddard, C. C. Huang, G. S. Couch, D. M. Greenblatt, E. C. Meng, and T. E. Ferrin. UCSF Chimera – A visualization system for exploratory research and analysis. *J. Comp. Chemistry*, 25(13), 2004.
[14] A. Quamar, A. Deshpande, and J. J. Lin. NScale: Neighborhood-centric large-scale graph analytics in the cloud. *VLDB J.*, 25(2), 2016.
[15] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *SOSP*, 2015.
[16] M. Zaharia et al. Apache Spark: A unified engine for big data processing. *Comm. ACM*, 59(11), 2016.