The Similarity Graph: Analyzing Database Access Patterns Within A Company

Robert Searles
rsearles@udel.edu
Computer and Information Sciences
University of Delaware Newark, DE 19716

***Abstract*—A company's database can reveal a lot about its employees, and that information can be used to manage the workforce, assign tasks, and create collaborations more effectively. This paper proposes a framework to build a similarity graph between analysts within a company using the SQL queries they write. We show how we can represent SQL queries in graph form, and we propose a method that can be used for calculating pairwise similarity on these graphs. In order to guarantee that this method will scale to large systems, we accelerated our algorithm using OpenMP. The results we obtained revealed many behavioral similarities between employees from different business units within the company, and we achieved an almost linear speedup, with a minimum of 75% efficiency across all cores.**

I. Introduction

For a large company, effectively analyzing a database of SQL queries is a difficult task. There is much to gain from doing so successfully, but it is difficult to do in practice. The queries in a large company's database can reveal a lot about the company's workforce. One way to analyze a database is to perform similarity analysis on the queries that are written to access information in that database. It is challenging, but identifying similarities between queries within a database can help administrators better optimize the database and create new collaborations between workers.

There are many challenges associated with trying to perform this type of analysis on a large database. First, we have a lot of information to examine. We have a large number of SQL queries, which were each written by an analyst within the company. These analysts are broken up into departments, or business units, as we prefer to call them. It is necessary that we structure the method used to analyze this data in such a way that we can cluster these queries according to their corresponding analyst and/or business unit. More importantly, this framework needs to be scalable for use with corporate-level systems. This is extremely computationally intensive because we must calculate the pairwise similarity of every pair of queries in the database. We will propose a framework for achieving both of these goals, and we will test it on a dataset of SQL queries written by analysts within a large financial institution.

There are many ways to represent SQL queries. One elegant way is to represent them in the form of graphs. Instead of using a traditional string-based representation of queries, graphs allow us to represent the data in a more structured manner, which is much more computationally friendly. However, dealing with hundreds of thousands of these graphs is difficult. Calculating pairwise similarity between every combination of these graphs at that scale is not trivial. Most similarity measures for graphs have a computational cost of at least n^3 [1]. We will propose a method for overcoming these computational requirements, and we will use the results we obtain to construct a similarity matrix that we can then use to observe similarities between employees within the company. This will allow us to create a similarity graph, which shows a collection of workplace trends and relationships between analysts. This type of graph will use our similarity analysis as the metric for comparing analysts in the company.

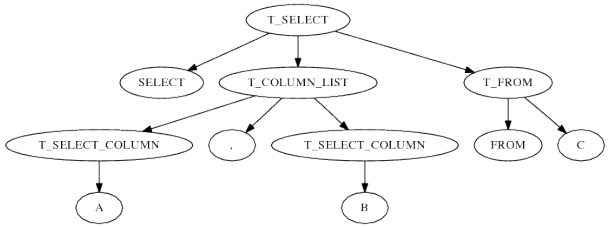


Fig. 1. An example parse tree showing a simple SQL query: SELECT A, B FROM C.

In order to form a good similarity graph, we first need to define a useful metric of comparison. In this paper, we decided to use a collection of SQL queries submitted by employees from different business units across a company to form our similarity graph. Specifically, we will use trees, which are special cases of graphs that contain no cycles. SQL statements can be decomposed and represented in a graph-based form called a parse tree. We first constructed parse trees for each SQL query using the open source software tool *ANTLR (ANother Tool for Language Recognition)* [2]. An example of these parse trees is shown in Figure 1. We then developed a tree similarity algorithm that efficiently calculates the similarity between SQL parse trees. The tree similarity algorithm is inspired by the idea of a fast subtree kernel that is used on graphs [3], [4]. The idea was to decompose graphs into a set of *fingerprints* that represent substructures. Once these *fingerprints* were calculated, the graph could be represented by a vector of *fingerprint counters*, where counters represent the occurrences of each fingerprint. The similarity between two graphs is calculated as the inner product of the corresponding two vectors.

We then use these pairwise similarity values to construct a similarity matrix that will contain the pairwise similarities of analysts within the company. We will examine each pair of users and collect all the queries written by those analysts. We

can then use our framework to compute the overall similarity of the two analysts based on their collections of queries. Finally, we can observe these results by using visualization tools to create figures that we can use to draw conclusions about the data [5], [6]. Figure 2 depicts our framework's pipeline.

The major contributions of this paper are the following:

1) We developed a framework to build a similarity graph between analysts (who generated SQL queries) from SQL queries, where each SQL query was represented by an SQL parse tree. To the best of our knowledge, our paper is the first to construct a similarity graph based on SQL queries people submit.

2) To calculate parse tree similarities, we designed a tree similarity algorithm that is based on a fast subtree kernel that uses discrete labels.

3) We also accelerated our algorithm using OpenMP in order to make it scalable for use on large systems with large datasets.

The rest of the paper is organized as follows: Section 2 will illustrate the practicality of representing data in the form of graphs. Section 3 describes the graph kernel we created to calculate pairwise similarity of our data that has been converted to discrete labels. Section 4 describes the processes we used to create our similarity matrices for each set of data. Section 5 reports the results. Section 6 describes related work, and we conclude in Section 7. The appendix following Section 7 explains the process that is used for relabeling the data. The result of this method is the input to our graph kernel.

II. Representing Data in the Form of Graphs

Representing data in a meaningful and useful manner is not trivial. A major challenge that one faces when trying to analyze large amounts of data is determining whether that data is in a form they can work with, and if it is not, how they can transform the data into a representation that is better suited for the analysis they are trying to perform. In our case, we examined SQL queries that were written by employees of a large financial institution. These queries were presented to us in text (string) form, which is not very computationally friendly. We require that the representation of the data be computationally friendly because our goal was to develop an automated system framework for analyzing this data.

Our solution to this challenge was to represent these SQL queries as graphs. Graphs are not only more aesthetically pleasing, but they are also much more computationally friendly than plain text is. Simply analyzing text would force us to do more one-to-one comparisons on the data. For example, checking two queries which are identical except for user data (such as a name or an address) would not yield optimal results because there is a difference in the characters contained in the strings. If both queries contained a name, we do not need to consider the exact characters in the name; we only need to consider the fact that a name exists in general. Unlike simple strings, representing queries in graph form allows us to observe the structural qualities of the query. More specifically, in this work we represented these SQL queries in the form of trees, which are undirected graphs in which each pair of vertices are connected by exactly one path. A tree is a special case of a graph where there exist no cycles, i.e., there is no path from any one node back to itself. We then relabeled, or encoded, these trees and represented them as a series of numbers corresponding to each sub-tree in the tree. We call this type of representation a feature vector. Feature vectors are particularly useful because there is a plethora of computational algorithms and functions in existence that operate on them.

One way we can compare a pair of our encoded trees is by using a graph kernel. The function of a graph kernel is to compute the similarity of two graphs. Graph kernels are a core component of our automated system and a major focus of this report. Being able to calculate similarity between each pair of our SQL queries enabled us to create a similarity matrix that can be used to draw conclusions about the data. A similarity matrix is simply a matrix containing all the pairwise similarity values in our dataset. We used this matrix to visualize our data and ultimately achieved our goal of observing potential collaborations within the company. Representing our data as graphs ultimately allowed us to construct a visual representation of the employees writing the SQL queries within a company by calculating the similarities between these graphs of SQL queries.

III. Calculating Tree Similarities

In this section, we show our method for calculating the similarity between a pair of trees. This method takes as input, two encoded trees. An encoded tree is one that has already been parsed from a query and relabeled using the relabeling process described in the appendix. Given a pair of trees in this format, we are able to compare their elements and structure and assign a normalized value, which denotes how similar the two trees are.

Using the encoding algorithm, each tree is represented as a set of numbers (a set of fingerprints). Since the similarity between two trees is defined as the inner product of two vectors of fingerprints, we can easily calculate this value from the set of numbers. Formally, let T, T' denote two ordered trees. The similarity value between T and T' is defined as follows:

$$k_{WL}(T,T') = \sum_{\substack{n\in T \\ n'\in T'}} \sum_{\substack{0\le i\le max\_h(n) \\ 0\le j\le max\_h(n')}} x \begin{cases} =1 & E_i(n) = E_j(n') \\ =0 & otherwise \end{cases} \tag{1}$$

Let n, n' denote a node in tree T, T'; $max\_h(n)$ denotes the maximum height of the subtrees that root at node n. $E_i(n)$ is the subtree rooted at node n with a height of i. The similarity algorithm counts the matching subtrees from height 0 to $min(H, H')$ between T and T', where H and H' are the heights of T and T'. Note that $H = max\_h(rootT)$ and $H' = max\_h(rootT')$. Height-0 subtrees refers to the singleton tree node from the tree, i.e. $E_0(n) = n$ and $E_0(n') = n'$.

Additionally, the definition of a weighted similarity value between two trees T and T' is as follows:

$$k_{WL}(T,T') = \sum_{\substack{n\in T \\ n'\in T'}} \sum_{\substack{0\le i\le max\_h(n) \\ 0\le j\le max\_h(n')}} x \begin{cases} =i & E_i(n) = E_j(n') \\ =0 & otherwise \end{cases} \tag{2}$$

The notation is the same as the notation in Equation (1). Note that the weighted value discards the common nodes

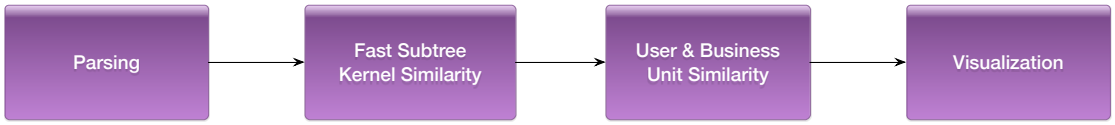


Fig. 2. This is a depiction of the pipeline of our framework. Each node represents a step in the framework that we use. The first portion (Parsing) is talked about in the appendix, and it provides the input to our framework. The fast subtree kernel then computes pairwise similarity between the SQL queries. The user and business unit similarity portion groups the SQL queries by their corresponding users, and it calls the fast subtree kernel to perform the similarity calculations on those groups of queries. The computationally intensive portion (Fast Subtree Kernel Similarity and User & Business Unit Similarity) is accelerated in order to exploit parallel architectures. Finally, we visualize the results of our similarity calculations using third party data visualization tools.

between two trees (the case $i = 0$) and boosts the similarity by multiplying the height of the subtree when there are common subtrees.

Although at this point we have successfully calculated a weighted similarity for a pair of trees, we still need to make an important consideration regarding normalization. All our trees will not be the same height, nor will they contain the same number of subtrees. We need a strategy for normalizing the similarities that we calculated because if we simply calculate weighted similarity, we will not have a uniform scale. We can do this by calculating the weighted similarity of each tree to itself and taking the square root of their dot product. We can use this in conjunction with the weighted similarity of the trees in question to produce a normalized similarity value. The similarity between two ordered tree T and T' is defined as the normalized kernel shown below:

$$s = k_N = \frac{k_{WL}(T, T')}{\sqrt{k_{WL}(T, T) \cdot k_{WL}(T', T')}}. \quad (3)$$

Figure 3 shows the encoding algorithm running on three trees. First, the trees are relabeled according to their structural qualities. A global alphabet is created from the dataset of SQL queries and each entry is assigned a number which that particular node in the tree is relabeled to. This leaves us with a set of fingerprints that can be used to compare the structures of different trees. The left trees's fingerprints are (0,1,2,3,4,5,6), the middle one's are (0,1,2,3,4,5,7), and the right one's are (0,1,2,3,4,5,8). We can get the corresponding fingerprint counter vector for each tree. A counter vector is simply a vector corresponding to the fingerprint vector of a tree that keeps track of how many times each fingerprint occurs in the given tree. The left, middle, and right trees can be viewed as the global fingerprints (0,1,2,3,4,5,6,7,8) with counter vectors (1,1,2,1,1,1,1,0,0), (1,1,2,1,1,1,0,1), and (1,1,2,2,1,2,0,0,1) respectively. The similarity between trees can be obtained by calculating the dot product of the trees' corresponding counter vectors.

1) Decoding encoded numbers back into subtrees: When comparing two trees using the encoding approach, it is necessary to decode the numbers back to the original subtrees for examination when we want to find out the structure (i.e., the sub-trees) the two trees have in common. For any number that encodes a subtree (say of height h), it is associated with a string that represents the root of the subtree and its encoded children (at least one with height of $h - 1$). The decoding is performed in a way that subtrees with a smaller height are decoded first. Later all subtrees with larger height can build upon these subtrees of smaller heights.

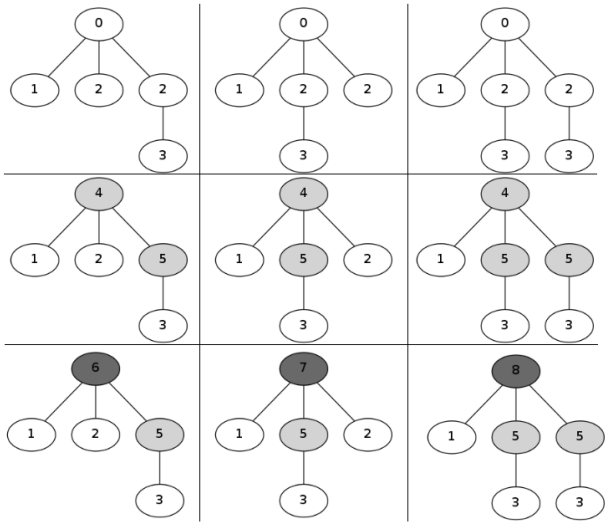
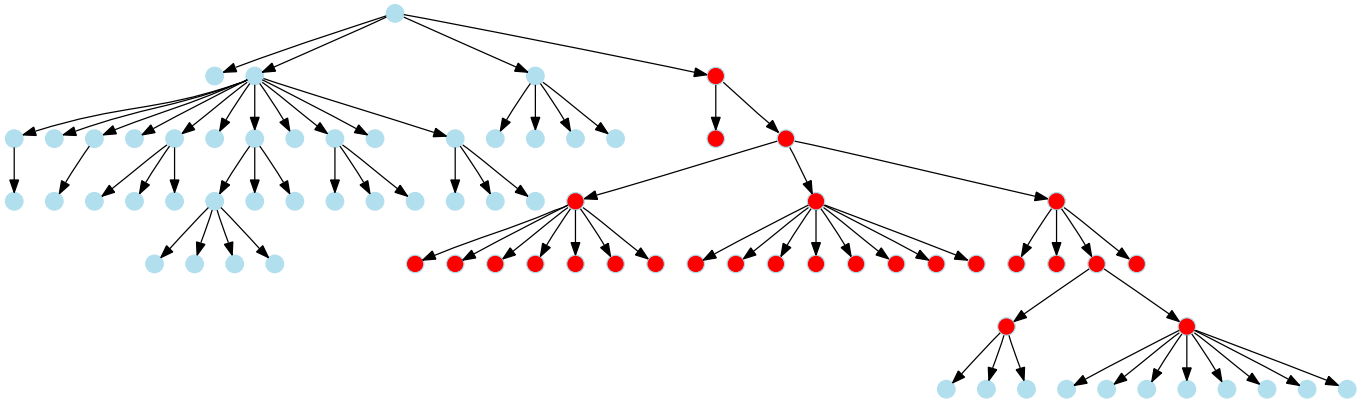


Fig. 3. This figure gives us an example of the encoding process for three basic parse trees. The first row represents three trees. The numbers assigned to the nodes by the encoding algorithm are the lookup identification numbers corresponding to that node's entry in the global alphabet. The second and third row show the state of each tree after the first two iterations of the subtree encoding process. Note that in the last iteration, the roots of the three tree are encoded to 6, 7, and 8 respectively. This means the algorithm considers the left two trees different, otherwise they would be assigned the same labels after the encoding procedure is complete. – 6, 7, and 8 represent the entirety of their respective trees.

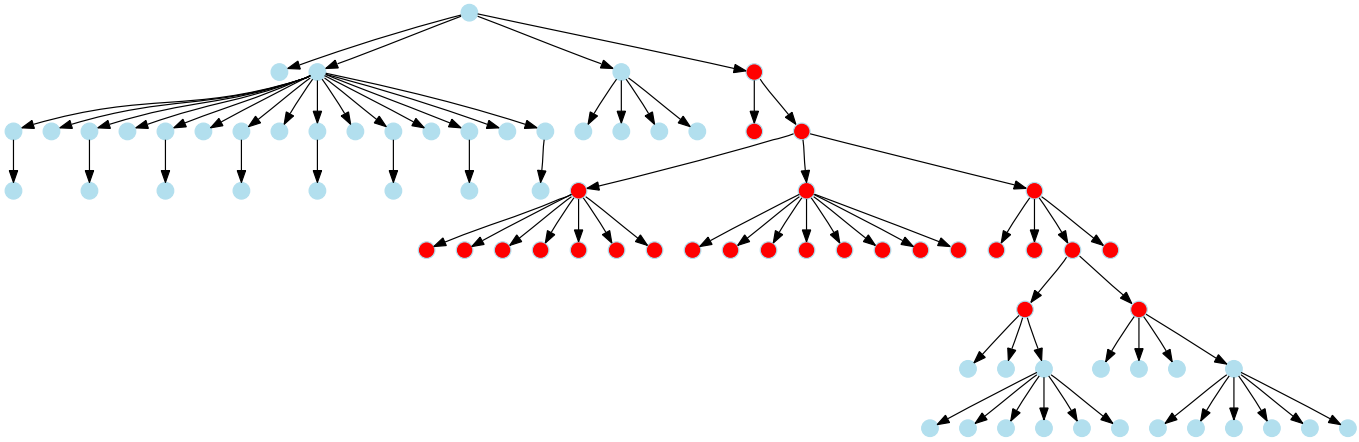
The visualization of the tree similarity algorithm on two small trees is shown in Figure 4(a) and Figure 4(b). Here, we see two full parse trees generated from two SQL statements. The nodes highlighted in red make up the largest common parts of the two trees, according to what our similarity algorithm deems similar. Note that the original two SQL queries have many common sub-strings.

IV. Constructing Similarity Matrices From Pairwise Similarity Values in a Dataset

In the previous section, we introduced an algorithm used to calculate the similarity between ordered trees. We were able to calculate the pairwise similarities between all SQL parse trees. Also, since each SQL query is associated with the analyst who submitted it, we were able to measure the overall similarity between analysts from the pairwise similarities we calculated Using our similarity analysis on SQL queries, we created a similarity matrix and ultimately a graph to show the similarities in terms of the SQL queries being submitted by analysts across the company. Suppose the set M contains all the queries analyst



(a) The parse tree of this SQL query: SELECT A, B, C AS c, function1(D) AS d, E AS e, F AS f FROM table1_name WHERE table1_name.c1 $\leq$ 'STR' and table1_name.c1 $<$ 'STR' AND (G='STR' AND C IN ('STR', 'STR')).



(b) The parse tree of this SQL query: SELECT A, H, E, C, F, G, I, D FROM table1_name WHERE table1_name.c1 $\leq$ 'STR' and table1_name.c1 $<$ 'STR' AND (I>function2(p1, 'STR') AND I$\leq$ function2(p1, 'STR')).

Fig. 4. Two SQL parse trees that the similarity algorithm deems similar. The largest common parts of the two trees are part of the WHERE clause of the SQL queries (nodes highlighted in red).

A submitted, and the set N contains all the queries analyst B submitted. The similarity of analyst A to analyst B is calculated using the following formula.

$$sim(A \rightarrow B) = \frac{\sum_{i \in M} \max(sim(i,j) | \forall j \in N)}{|M|} \quad (4)$$

In the above equation, sim(i,j) calculates the similarity between query i and query j. Additionally, we can calculate the similarity of analysts B to analyst A using the following formula.

$$sim(B \rightarrow A) = \frac{\sum_{j \in N} \max(sim(i,j) | \forall i \in M)}{|N|} \quad (5)$$

The similarity between analyst A and analyst B is calculated as the average of $sim(A \rightarrow B)$ and $sim(B \rightarrow A)$. Each SQL query is not only associated with an analyst, but also a business unit the analyst is from. We can measure the overall similarity between business units using the same approach as above by considering all the SQL queries submitted from within a business unit.

While we are using this method to calculate similarities between users based off of SQL queries they wrote, it is important to note that this algorithm can be used in other contexts as well. In this phase, we calculate a similarity between two users. These users are identified by a group of parse trees representing the queries they submitted. Those parse trees represent SQL queries in this case, but this same method can be used in other contexts as long as the data is represented in the form of trees.

A. Complexity Analysis

For a given dataset containing U users and Q total query subtrees, the worst-case complexity of the algorithm used to calculate the similarity between two given users (shown in Algorithm 1) is given by the following equation: $\mathbf{O}(U, Q) = U^2 \cdot Q^2$. This is because we have to compare each user with every other user in the company, and in order to compare two users, we must examine all the subtrees of queries they wrote.

Furthermore, the implementation indicates that every subtree must be compared to every other subtree. Since the number of users is most likely relatively small compared to the number of subtrees, the subtree comparison dominates the computation.

B. Hardware Acceleration

Since the number of subtrees dominates the computation, scalability is an important consideration to be made. The algorithm was modified from its original state to support OpenMP-based parallelization. Every user similarity is computed in parallel. To account for different quantities of work-size, the work items are sorted from largest size to least. This sorting allows for larger tasks to execute first while the smaller tasks will eventually fill in any available processing time with the ultimate goal of completing execution at the same time. An additional version of the OpenMP code was generated to further exploit parallel architectures. This implementation was able to exploit nested parallelism by not only parallelizing user similarity but also query similarity.

C. Required Modifications

Parallelizing this type of algorithm presents two challenges. The first is separating file processing from computation. If we are going to utilize an accelerator or a multi-core processor of some sort, we cannot read each subtree file as we require the data. This would cause a bottleneck during kernel execution because threads would have to wait for the data they require if it was not already present, and therefore, the data required by that accelerator should be put in the device's global memory before computation begins. This issue was solved by creating a lookup table for each file name and loading all of the files before computation. This is important because many large clusters and super computers that exist today utilize accelerators as their main computational tool.

The other challenge is presented when we start to analyze the query similarity algorithm. We notice that some values are used repeatedly for a given subset. The algorithm will pre-compute the self-similarity of every subtree and store the result in a lookup buffer. This results in less redundant computation, therefore reducing the total number of cycles required for user similarity evaluation.

Algorithm 1 This algorithm goes over the overall execution preparation for computing user similarity.

```
fileMap ← map_filename
queries ← read all query files
total ← userCount * (userCount - 1)
runs ← {}
for i in total do
    runs_i ← current run configuration
end for
sort runs
for r in runs do
    u_r ← compute user similarity
end for
```

When we begin execution, we simply read all the subtree graph files in our dataset instead of reading them individually on the fly based on which users we are currently looking at. We load all of these files into a hash table, and we index them by their file name (which is simply their encoded graph number). Then, when the algorithm is processing a pair of users, it can load the trees it needs for that pair from the hash table, rather than loading it from disk.

Business Unit Name	Unit Number	Queries Written
INF-IT	1	10178
OTH-BLK	2	35
OTH-IT	3	10190
FS:BEM	4	50
RSK	5	17500
CSD	6	1826
MKS	7	3292
OTH	8	750
FIN	9	351
MSS	10	335
ACT	11	15
PTR	12	228
BIZ	13	148
FRD	14	1083
ORI	15	1621
CMS	16	29
UNKN	17	1509
CEX	18	1
CPR	19	68
GLB	20	366
HNW	21	42
PIO	22	38

TABLE I. DATASET 1: 49,655 QUERIES. THIS IS A LIST OF BUSINESS UNITS AND HOW MANY QUERIES EACH UNIT WROTE. UNIT NUMBER SHOWS WHICH NODE EACH UNIT CORRESPONDS TO IN FIGURE 6.

Business Unit Name	Queries Written
CDW-IT	107112
FRD	16069
OTH	11641
INF-IT	4774
RSK	647490
OTH-IT	646

TABLE II. DATASET 2: 787,732 QUERIES. THIS IS A LIST OF BUSINESS UNITS AND HOW MANY QUERIES EACH UNIT WROTE.

V. RESULTS

A. Experimental Setup

Our experiments were run on a machine with 16 GB of memory and 2x AMD Opteron 6320 CPUs, which have 8 cores per CPU clocked at 1.4 GHz. We ran experiments on Dataset I multiple times, each time with a different number of cores in order to assess the scalability and efficiency of our algorithm. We ran our user similarity algorithm across a single core, 4 cores, 8 cores, and 16 cores, and we achieved speedups of 1.0x, 4.0x, 7.3x, and 12.3x (respectively), as shown in Figure 5. As can be seen by the graph, the algorithm is quite scalable (each added core yields a significant speedup), and it maintains over 75% efficiency in all cases as shown by Figure 5. Efficiency refers to the performance that was achieved relative to the theoretical maximum computational ability of the device. For example, theoretically if we had a device with 16 cores, we could achieve a 16x speedup over sequential code. This would be optimal efficiency, but it is impossible in practice because of different factors, such as data transfer and synchronization across cores, that create overhead. The more cores you introduce into a system, the more potential overhead you incur. In our case, we maintained just over 76% efficiency when running on 16 cores.

B. Analysis of Results

Figure 6 shows two graphs generated from calculating the similarity values between employees. Both graphs include nodes representing each of the 427 users that submitted queries in our sample set of 49,655 SQL queries (Dataset I). Nodes are colored and clustered according to which business unit they are a part of. Figure 6(a) shows edges between users who have a similarity value between them of 0.5 or greater. Figure 6(b) shows edges between users who have a similarity value between them of 0.8 or greater. By looking at the edges that we generated, we can see that there are a lot of similar SQL queries being written by users that exist in different business units, but it is hard to draw insight from a similarity graph when there are many edges between users (which seems to be the case in Figure 6(a)).

One way we can circumvent this analytical obstacle is to look at business units as a whole before examining individual users. We can do this using a graph like Figure 7, which is generated by examining the overall similarity between business units as a whole (comparing the collection of users within one business unit to the collection of users in another). We can first look for thick edges (which represent a high similarity value between units as a whole). For example, if we take a look at unit 14 in Figure 7, we can see that it has thick edges running to both unit 5 and unit 8. This tells us that those are the units most similar to unit 14, and we can then dive down to the lower level and examine similarities between the individual analysts in unit 14 and analysts in units 5 and 8. This information could prove quite useful in helping link up these people that may have never met before. Ideally, new collaborations within the company will be created because of this information.

We examined metrics other than just pure user similarity as well. Another metric we examined is betweenness centrality (or the measure of a node's centrality in a connected graph). Centrality is equal to the number of shortest paths from all vertices in the graph to all other vertices that pass through the node in question. As mentioned previously, Figure 7 uses one node to represent each business unit as a whole. Each node corresponds to a business unit that had analysts that submitted queries contained in Dataset I. Edges still represent similarity between nodes (calculated using Equation (4) and (5) on business units), so we can examine similarities across entire business units. The difference in this graph from the user similarity graphs mentioned previously is that node sizes in Figure 7 represent the betweenness centrality of that particular unit. The bigger the node is, the more central the business unit is in the company. This is interesting because we can see that certain units (such as unit 5) have a very high centrality value. This means that the work they are doing is somewhat similar to that of many other business units in the company.

Figure 8 also uses node size to represent the betweenness centrality of the given node. Nodes in this graph represent individual analysts that submitted queries in Dataset II. We can conclude from examining this graph that the higher an analyst's centrality value within the workforce is, the more similarities they have with other analysts. In Figure 8, for example, analysts 54, 78, and 92 have the highest centrality values, which means that all 3 of them are writing queries that are similar to the queries written by many other analysts in the company. This can be useful for workforce management

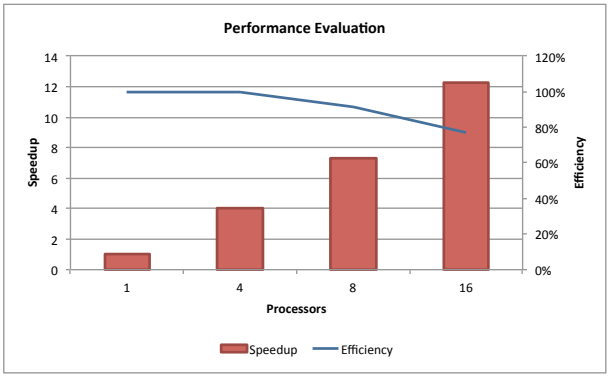


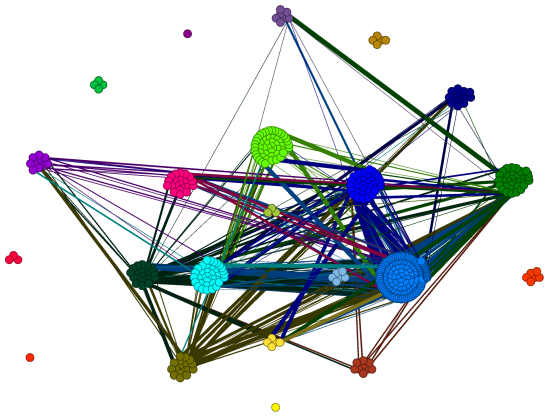
Fig. 5. This graph represents the parallel speedup and efficiency of our similarity algorithm. We can see that we continue to see a performance increase with each additional core. While there is some overhead because of data transfer and synchronization, we maintain good overall efficiency.

because they can view these types of employees as versatile employees that should be able to work productively with a large portion of the other analysts in the workforce. This graph doesn't dive as deep into examining how similar an analyst is to another individual analyst, but it does show us how similar the analyst in question is to his or her peers in a general sense. Combining this metric with our individual similarity metric gives a workforce's manager useful information that can help them create effective collaborations/teams within their workforce in order to obtain maximum overall efficiency.

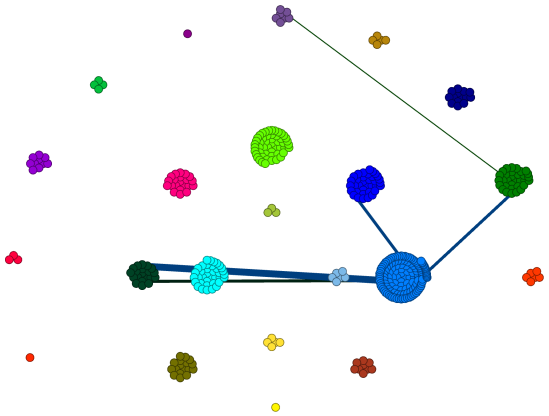
We can also use a heat map (as shown in Figure 9) to show us individual user similarity along side of business unit similarity. Analysts on the X and Y axes are sorted according to the business unit they are a member of. This means that points near the diagonal represent a similarity value between users that are in the same business unit, so we expect to see some similarities here. People in the same business unit typically work on similar things. However, we can see points representing high similarities far from the diagonal as well. These users are in different business units, but they seem to be accessing the database in a similar manner. This makes it easy for us to see that there are potential collaborations between users that currently are not contained within the same business unit. Because of the heat map being a visual representation of a matrix, we can easily pinpoint exactly which analysts are similar to each other.

VI. Related Work

a) Graph Similarity: There are many algorithms that measure the similarities between graphs [7]. Recently a subtree kernel on graphs was proposed based on the Weisfeiler-Lehman relabeling procedure to measure the similarity of two graphs [3], [4]. The Weisfeiler-Lehman (WL) algorithm was originally designed to test graph isomorphism [8], but it was extended in the fast subtree kernel to compute the *fingerprints*. Tabei and Tsuda [9] used the same idea combined with a new data structure to search similar graphs from massive graph datasets efficiently. In these applications of the WL algorithm, the inputs were undirected graphs. If the inputs were *rooted and ordered trees*, the fast subtree kernel would have some limitations, as described in Section 2. Also, the relabeled sub-structures of undirected graphs (the *fingerprints*)



(a) Sim >= 0.5



(b) Sim >= 0.8

Fig. 6. A sample similarity graph we created by analyzing the similarities between SQL queries written by analysts. The vertex represents an analyst from a certain business unit. An edge linking two analysts means they wrote similar queries. The thickness of the edge represents the similarity. Figure 6(a) filters out all edges between analysts whose designated similarity metric is less than 0.5, and Figure 6(b) does the same with values less than 0.8.

cannot be decoded back to the original structure because of graph isomorphism. In our work, we used the same idea of *fingerprints*, but we modified the Weisfeiler-Lehman algorithm to deal with *rooted and ordered trees*. There are also tree kernels that measure the similarity of two trees, such as the PT kernel [10], the ST (SubTree) kernel [11], and the SST(SubSetTree kernel) [12]. Matrino's work [10] extended the PT kernel to work on DAGs, and by decomposing graphs to DAGs, they came up with a new kernel based on the PT kernel for graphs. Moschitti [13] combined the ST Kernel and the SST kernel to experiment on predicate argument classification in natural language. We proposed a similarity measure based on the Weisfeiler-Lehman algorithm that allows us to decode the relabeled subtrees in addition to calculating similarity.

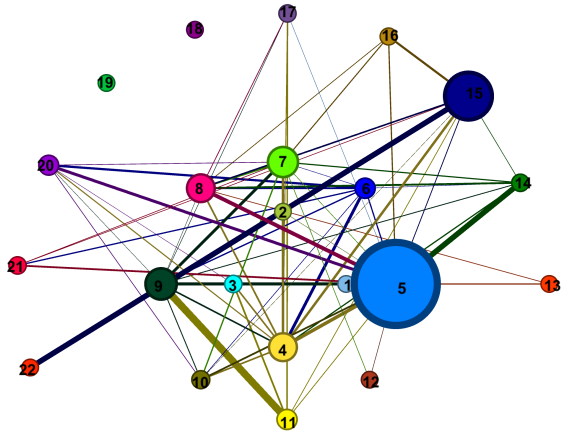


Fig. 7. A similarity graph we created that uses a single node to represent each business unit. The size of the node represents the betweenness centrality. Edge thickness represents the similarity between units. Edges representing a similarity value less than 0.175 were filtered out to reduce clutter.

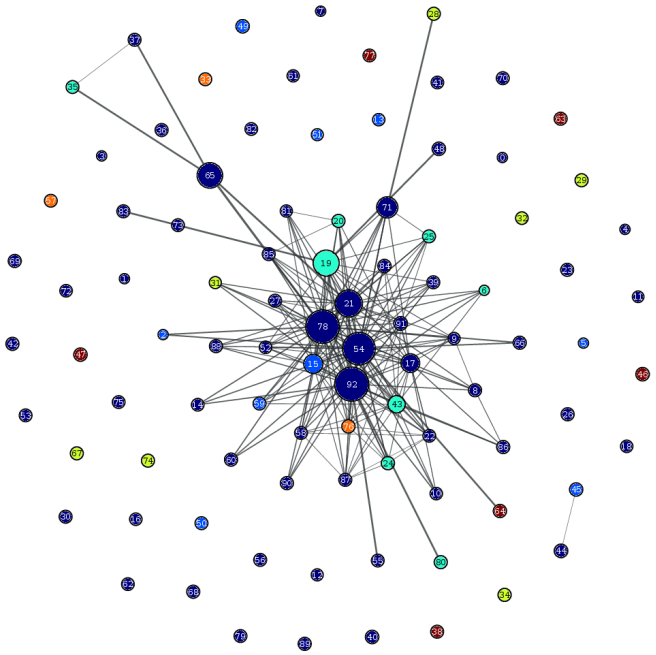


Fig. 8. This graph represents the betweenness centrality value for each user in Dataset II. Node size and edge weight correspond to the centrality of the analyst in question. The larger the node, the higher that analyst's centrality in the workforce.

b) Network Analysis: People are linked together in a variety of ways. There are many real-world examples of networks, such as telephone call graphs, coauthorship and citation networks of scientists, the exchange of email message within companies, etc [14], [15]. Magnusson [16] developed a model used to analyze telecommunication networks for the purpose of detecting influential subscribers within the network. Their tool, called 'Hadoop', was also compared with the more commercial Neo4j graph database. Our work also aims to detect influential people within a business, but we take it a step further and try to connect influential workers with similar workers by looking at how similar their queries are

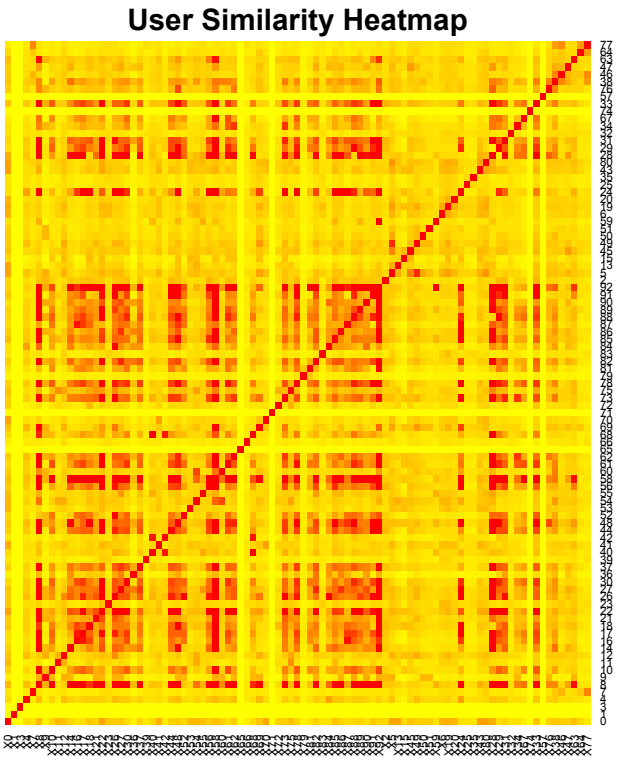


Fig. 9. This heat map shows us the similarity between each of our analysts in Dataset II. Analysts are sorted and grouped according to the business unit they belong to. Brighter colors (red) represent a high similarity between analysts, while neutral colors (yellow) represent a low similarity value between analysts.

overall. Centrality is also considered in our results which could later be extended by work focused on solving the problem of computing centrality measures in large social networks by using parallel and distributed algorithms [17].

The connections between people on social networking sites like Facebook, LinkedIn, and Twitter are also something we can observe. To better understand the interaction between people, database analysis techniques are often used to study the characteristics of social networks. Link prediction and link recommendation are techniques that we see used all the time [18]. At MySpace [19], a friend recommendation engine was designed and deployed to connect more people by leveraging personalized information such as user profiles, favorite songs in common between users, etc. Other work took advantage of frequent appearances of people in uploaded personal photos to infer friendship between them [20]. Cha and Cho proposed the use of a topic model to differentiate the nodes by popularity so that more relevant friend recommendations could be made within a social network [21]. These types of studies are directly related to our strategy, but instead of focusing on who we think you should be friends with, we are looking at what people would be the most productive together in a work environment. Examining something such as which people like the same kind of music is actually quite similar to looking at what kind of queries two people write. A library of music is basically just a huge database of files. We are looking at which people are utilizing our database in a similar manner, which is almost exactly the same as what these social networking sites are doing when comparing things like favorite music, movies, and television shows in order to make friend recommendations.

Many research projects involved analyzing the structural properties of social networks as well. There are studies on people's culture and tradition being done by comparing their house layouts, represented by graphs [22]. They applied centrality measures to unveil the differences between which part of the house people view as important. In our paper, we study the differences of business units and analysts by measuring their betweenness centrality in addition to just simply looking at similarities between pairs of users. We used Gephi [5] and Graph-tool [6] to visualize the results. Gephi is an open-source tool that is capable of visualizing various kinds of graphs, including social network graphs. To calculate betweenness centrality, Gephi implements the betweenness centrality algorithm described in Brandes's work [23]. Graph-tool is an open source python module that is used for statistical analysis and visual representation of graphs. Graph-tool also implements the betweenness centrality algorithm [23].

VII. Conclusions and Future Work

We have constructed a series of similarity graphs of employees using our sample set of SQL queries, and we created an algorithm that was used to measure similarity between users in that workforce. Using this algorithm, we were able to create a visual representation of user similarity across the workforce, and we also were able to calculate and visualize similarity between business units as a whole, as well as the betweenness centrality of the users in the company. The similarity graphs of employees were derived from the similarities between the SQL queries the employees wrote. We designed a scalable, encoding-based algorithm to measure the similarity between SQL queries, which takes advantage of multi-core architectures to offset the cost of computation. With all this new knowledge, we hope that new collaborative efforts will emerge within the company in question.

In the future, we plan to extend this work by applying the framework we created to other domains. Specifically, we plan to adapt this technique in order to perform behavioral analysis on malicious code with the goal of developing a system to detect malware that is more effective than the current state of the art.

Acknowledgment

This work was funded in part by JP Morgan Chase & Co. as part of the Global Enterprise Technology (GET) Collaboration.

APPENDIX

The following work is part of a collaborative effort, and it is placed in an appendix for this reason. In this appendix, we will first introduce the original algorithm used to relabel graphs and calculate graph similarities, and we will describe the algorithm's undesired behavior when running on ordered trees afterward.

A. The Original Subtree Kernels on Graphs

The original subtree kernel iteratively constructs the *fingerprints* of a graph based on Weisfeiler-Lehman's procedure for isomorphism testing [3], [4], [8]. The algorithm is shown in Algorithm 2. In each iteration, each vertex and its neighbors are represented by a string consisting of their labels in the previous iteration (line 3). The string is then mapped to a unique value also known as a *fingerprint* (shown in lines 4-7). After H iterations, the graph is associated with a vector of $H|V|$ fingerprints because each vertex is relabeled once in each iteration, and, eventually, every vertex will be associated with H fingerprints. $|V|$ is the total number of nodes in the graph. With the vectors of *fingerprints counters*, a similarity value on two graphs can be obtained by calculating the inner product of the two vectors.

Algorithm 2 The Weisfeiler-Lehman Relabeling Process

```
1: for h = 1 → H do
2:     for each vertex v in the graph do
3:         cur_sub ← labelToString(v & its neighbors)
4:         if hashtable.find(cur_sub) then
5:             label(v) ← hashtable.get(cur_sub)
6:         else
7:             label(v) ← f(cur_sub)
8:             hashtable.insert(cur_sub, label(v))
9:         end if
10:    end for
11: end for
```

The subtree kernel is much faster than existing graphs kernels (like random walk graph kernels [24]), and it preserves information well using *fingerprints* [3]. However, for ordered trees, the construction of fingerprints renders the calculation of tree similarity biased toward the number of common fingerprints generated at small h. We found that the number of common *leaves* of two ordered trees dominated the kernel's output. This is due to the fact that without a specific adaptation of the algorithm for use with ordered trees, the common leaves will be taken into account during each iteration. Consider two ordered trees A and B. Suppose tree A contains M occurrences of node "NUM" and tree B contains N occurrences of node "NUM". In this case, the similarity metric would be at least $H*M*N$, where H is the number of iterations in Algorithm 2. So, the original subtree kernel algorithm considers two trees similar if they contain common leaves. In our data set of trees, it is very common for two dissimilar trees to contain a large number of common node labels, for example "(" and "," etc. Applying the subtree kernel directly on such a data set would result in a biased and inaccurate similarity metric.

B. The Adapted Algorithm for Ordered Trees

In order to address the above limitation, we developed an adapted algorithm that eliminates bias when calculating the similarity between ordered trees. The improved algorithm is still based on the idea of converting trees into sets of *fingerprints*, but it changes the way these *fingerprints* are calculated. To calculate the similarity between sets of trees, we convert each tree in the set to a set of numbers, where each number represents a unique subtree. Identical subtrees from the set would be represented by the same number. In fact, for a set of ordered trees, every *fingerprint* now corresponds to a unique subtree encoded by a number. This is different from the subtree kernel relabeling because in our adapted algorithm these numbers can be decoded back to the original subtree. The following describes how to calculate the *fingerprints* for a set of trees.

Consider the set F that consists of N trees. Let H be the maximum height of the tree(s) in the set. Further, we denote the total number of nodes in the set F as $|V_F|$. The algorithm that calculates the *fingerprints* of each tree in F is shown in Algorithm 3.

Algorithm 3 The Encoding Algorithm

```
1: for h = 1 → H do
2:     next_number ← 0
3:     for each node n of the trees in the forest do
4:         if n is not marked leaf then
5:             cur_subtree ← toStr(n & its children)
6:             if hashtable.find(cur_subtree) then
7:                 label(n) ← hashtable.get(cur_subtree)
8:             else
9:                 label(n) ← f(h, next_number)
10:                hashtable.insert(cur_subtree, label(n))
11:                next_number ← next_number + 1
12:            end if
13:            if every child of n is marked as leaf then
14:                mark n as leaf
15:            end if
16:        end if
17:    end for
18: end for
```

According to how the algorithm works, during iteration h, all subtrees of height h are encoded and added to the vector of *fingerprints*; this can be easily derived by the induction method (omitted).

In each iteration after the encoding is performed, the algorithm collapses the tree by marking the nodes as leaves if these nodes are coming to the front-edge of the tree (all the children have been encoded in the previous iteration(s), i.e. all the children have been marked as leaves). Note that the height of a tree is limited, so the algorithm will terminate when h becomes the same as H, in which case the whole tree will be encoded to a number. In the orginal subtree kernel, each node corresponds to H fingerprints. However, in the adapted algorithm for ordered trees, the number of fingerprints associated with a node is dependent on the height of the tree and which level a node is at in that tree. An example of encoding a simple tree is shown in Figure 10.

Figure 10 shows the process of encoding a set consisting of only a single tree, but the process for a large set works the same way. One important issue to address when encoding subtrees is to ensure that the same subtrees are encoded to the same number and each number corresponds to only one tree (one-to-one mapping between a subtree and a number). To achieve this, the algorithm maintains a counter (*next_number*) in each iteration h to indicate how many subtrees (of height h) have been encoded. When assigning new label numbers, the number can be obtained by calculating: $f(h, next\_number) = h * C + next\_number$. Here, constant C is chosen such that

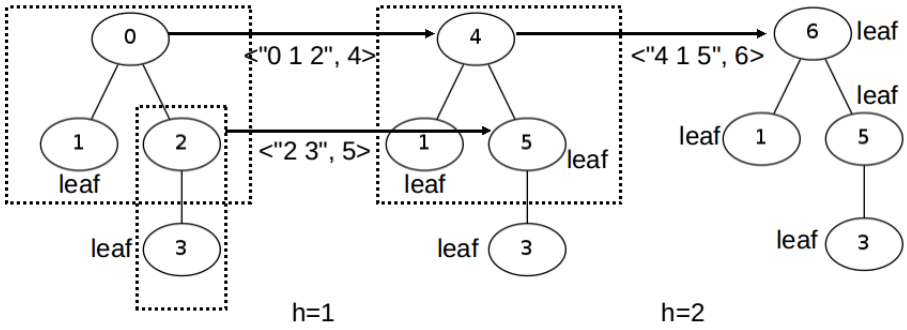


Fig. 10. An example showing the procedure of encoding subtrees of a tree into numbers. In each iteration ($h = 1$, $h = 2$), all subtrees of height 1 (shown in dotted rectangle) will be encoded to a number. In fact, each number generated in iteration h encodes a subtree of height h. For example, when $h = 1$, the subtree consisting of nodes 2 and 3 is represented as *"2 3"* and is encoded as 5. When $h = 2$, the subtree *"4 1 5"* is encoded as 6. Since 5 encodes a subtree of height 1, 6 actually represent the subtree of height of 2 (in this case the whole tree). Note that in each iteration only the nodes not marked as leaves are eligible to be relabeled.

in any iteration, $next\_number$ would not become larger than C. This way, we are certain that there will be no case where the encodings of one iteration will collide with those in other iterations. Therefore, C can equal $|V_F|$ as it is obvious that the number of subtrees in any iteration is less than $|V_F|$. In this case, $next\_number$ will be less than C in any iteration. Note that within each iteration, the uniqueness is maintained by a hash table. This concludes the appendix.

REFERENCES

[1] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," in *Journal of Machine Learning Research 11*, 2010, pp. 1201–1242. [Online]. Available: http://www.jmlr.org/papers/volume11/vishwanathan10a/vishwanathan10a.pdf

[2] "Antlr parser generator," 2012, http://www.antlr.org/.

[3] N. Shervashidze and K. M. Borgwardt, "Fast subtree kernels on graphs," in *NIPS*, 2009, pp. 1660–1668.

[4] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *J. Mach. Learn. Res.*, vol. 999888, pp. 2539–2561, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2078183.2078187

[5] "Gephi, an open source graph visualization and manipulation software," 2013, https://gephi.org/.

[6] "Graph-tool is an efficient python module for manipulation and statistical analysis of graphs (a.k.a. networks)." 2014, http://graph-tool.skewed.de/.

[7] K. M. Borgwardt, "Graph kernels," Ph.D. dissertation, Ludwig-Maximilians-Universität München, 2007.

[8] B. L. Douglas, "The Weisfeiler-Lehman Method and Graph Isomorphism Testing," *ArXiv e-prints*, Jan. 2011.

[9] Y. Tabei and K. Tsuda, "Kernel-based similarity search in massive graph databases with wavelet trees," in *SDM*. SIAM / Omnipress, 2011, pp. 154–163.

[10] G. D. S. Martino, N. Navarin, and A. Sperduti, "A tree-based kernel for graphs," in *Proceedings of the Twelfth SIAM International Conference on Data Mining*, 2012, pp. 975–986.

[11] S. V. N. Vishwanathan and A. J. Smola, "Fast kernels for string and tree matching," in *NIPS*, 2002, pp. 569–576.

[12] M. Collins and N. Duffy, "New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 263–270.

[13] A. Moschitti, "Making tree kernels practical for natural language learning," in *EACL*, 2006.

[14] J. Han and M. Kamber, *Data mining: Concepts and techniques*. Amsterdam: Elsevier, 2006.

[15] J. Scott, *Social Network Analysis*. London:SAGE, 2012.

[16] J. Magnusson, "Social network analysis utilizing big data technology," Jan. 2012. [Online]. Available: http://uu.diva-portal.org/smash/record.jsf?pid=diva2:509757

[17] M. Lambertini, M. Magnani, M. Marzolla, D. Montesi, and C. Paolino, "Large-scale social network analysis," in *Technical Report UBLCS-2011-05*, 2011, pp. 1–25. [Online]. Available: http://www.informatica.unibo.it/it/ricerca/technical-report/2011/pdfs/2011-05.pdf

[18] L. Backstrom and J. Leskovec, "Supervised random walks: predicting and recommending links in social networks," in *Proceedings of the fourth ACM international conference on Web search and data mining*, ser. WSDM '11, 2011, pp. 635–644.

[19] M. Moricz, Y. Dosbayev, and M. Berlyant, "Pymk: friend recommendation at myspace," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10, 2010, pp. 999–1002.

[20] H.-N. Kim, A. E. Saddik, and J.-G. Jung, "Leveraging personal photos to inferring friendships in social network services," *Expert Systems with Applications*, vol. 39, no. 8, pp. 6955 – 6966, 2012.

[21] Y. Cha and J. Cho, "Social-network analysis using topic models," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '12, 2012, pp. 565–574.

[22] Y.-S. Chiou and Y.-T. Huang, "Applying sna for the characterization of the spatial organization of xiaolin village," in *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, 2012, pp. 51–57.

[23] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[24] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *ICML*, 2003, pp. 321–328.