Platform for General-Purpose Distributed Data-Mining on Large Dynamic Graphs

Matthias Steinbauer
Johannes Kepler University Linz
Department of Telecooperation
Linz, Austria
Email: matthias.steinbauer@jku.at

Gabriele Kotsis
Johannes Kepler University Linz
Department of Telecooperation
Linz, Austria
Email: gabriele.kotsis@jku.at

***Abstract*—We present an approach to data mining on arbitrary graph data that uses a cloud-based distributed computing model for dynamic provisioning of computing resources as the graph model grows or shrinks. Further, we introduce the concept of logging graph changes as a basis for calculating properties of dynamic graphs. We briefly describe queries that leverage the dynamic graph model, for instance, by using a snapshot of the original graph while an algorithm executes or adapting query results as the graph changes. To demonstrate the feasibility of our approach, we conducted an initial evaluation, which shows that our parallel computing model can dramatically improve load times. Raw data imported into our system is processed faster on larger compute clusters.**

I. Introduction

Graphs and networks have a long-standing history as models of real-world problems. They have been applied in the context of communication networks (Internet, wireless, ad-hoc and sensor networks), social sciences (social networks), and biology. For instance, proteins [1], social networks, and connections on the Internet can be modeled as graphs. Research has so far concentrated mainly on static properties of graphs. However, the real-world reference of a graph model is often dynamic, which means that it changes over time in many aspects. This creates a demand for graph and network systems that reflect these changes in their models.

Real-world applications of graphs are not only dynamic, they often require very large models. For example, at the time of writing [1], the Facebook graph consisted of more than 1 billion active users [2], which clearly makes this a *big data* [3] problem.

The state of the art in computer science uses graph databases as the main storage and model for graphs. They usually support extensive query interfaces for querying and altering the graph. Since these graph databases have emerged from a long history of static graph processing, their query interfaces focus strongly on static graph properties. Dynamic graph properties such as the speed of growth of a sub-graph or the whole graph cannot be determined.

Another problem with dynamic graphs in static storage systems is that of graph changes during the execution of algorithms and queries. This can lead to unexpected results, for instance, if an edge or a vertex still needed for the execution of the algorithm is deleted from the graph.

Current implementations of graph databases tackle these problems with transactions and locking mechanisms ranging from ACID[2] to *eventual consistency*. Stricter locking could delay the execution of other queries or algorithms in other transactions.

In this paper, we propose novel ways of managing, querying, and computing on large dynamic graphs. We argue that by keeping track of the complete history of a graph, we can solve some of the problems described: (1) By storing the complete history of a graph model, we are able to compute dynamic properties of a graph. (2) Algorithms can be executed to completion on a snapshot of graph model. In effect a static view is created for the lifetime of each individual query or algorithm.

Our architecture is designed to leverage cloud computing resources as the graph and its version history grow larger. The graph model is distributed over several compute nodes of a cloud. Query and update mechanisms use the very same distributed architecture. This results in a platform that allows distributed data mining on large dynamic graphs.

We briefly present a basic implementation based on the distributed real-time computation system Storm [4], which provides mechanisms for reliable distributed computation; these are used to store a distributed model of a graph and to perform computations on that model.

Our prototype implementation allows the user to load large dynamic graphs and run simple queries. The framework ensures either (1) that the query is run on a static view of the graph or (2) that the query result is updated as the dynamic model of the graph changes.

We are currently experimenting with applications in the area of Reality Mining (RM) [5], more specifically, with gathering and analyzing data about human behavior and interaction in the real world. In the context of communication traces, RM allows a model of the communication network of individuals to be generated. We have previously shown that such models of communication networks provide interesting insight into groups [6].

[1]Data was retrieved in April 2013

[2]Atomicity, Consistency, Isolation, Durability

The remainder of this paper is structured as follows: In section II, we briefly describe work from relevant fields. Section III defines some terms that may be ambiguous in the context of this paper. In section IV, we explain our current prototype implementation. An evaluation and future work are presented in section V, and section VI concludes the paper.

II. Related Work

There are several fields that relate to this work, most importantly distributed graph databases and distributed graph processing engines.

A popular example of a graph database is the Neo4j [7] system. It also claims to run in a distributed manner, but only for fail-safe purposes. A master Neo4j database may be accompanied by several slaves that keep an exact copy of the graph. There are plans to implement sharding into Neo4j.

HyperGraphDB [8] is a general storage mechanism which supports graph-oriented storage. It implements several standards such as OWL 2.0 and Topic Maps 1.0, but can also be used to store general-purpose graphs because its query mechanisms support graph traversals and relational-style queries.

HyperGraphDB comes with a P2P framework that enables data distribution between many HyperGraphDB instances.

Pregel [9] (developed at Google)i and its open-source implementation Golden Orb [10] propose a massively distributed computational model in which graph algorithms are built from very small program fragments called *super-steps*. After completion of each super-step, the program may exchange data with other vertices or vote to halt the algorithm. Since Pregel is based on MapReduce [11], it is designed as a batch-processing system.

Trinity [12] is a distributed in-memory graph database that supports offline analysis and online query processing of graphs. Its distributed architecture, which is based on a memory cloud, allows low latency queries to be processed on "web-scale" billion-node graphs.

The InfiniteGraph system [13] uses a similar architecture. It provides a graph database system that can be distributed on a cluster of commodity hardware and is mainly used for online graph processing.

The graph database system presented in [14] is accompanied by a graph-processing system which allows complex analysis of social graphs stored in the database. The authors use several Neo4j databases as their storage back-end.

Our system differs from others in the following points.

(1) Many systems lack the ability to address dynamic properties of graphs and networks. Most of them are implemented as distributed graph storage systems, distributed graph processing systems or as a combination of both, but all of them focus on static properties of graphs.

(2) The most common application area is social network analysis. We have also identified this as an important field, but our concept is targeted towards general-purpose graph processing, covering several application areas, such as network analysis, social network analysis, and analysis of biological processes.

III. Preliminaries

In this section we define key terms used throughout this paper to avoid ambiguities. For example, the term *node* might be ambiguous as it could refer to both a computing node within a compute cluster and a vertex in a graph. In this work, the term node is used only for computing nodes.

We call a graph a set of *vertices* and *edges*. The edges connect the vertices and have a direction. If not specified otherwise, our system operates on directed graphs.

We allow further information, which we call *profile* to be stored with each vertex. The profile may contain arbitrary information describing the vertex in more detail.

A *dynamic graph* is a set of graphs, each of which is a model of the same sample at different points in time [15], [16].

Next, we define some terms in the context of *cloud computing*. Cloud computing itself is defined as software-based on-demand access to a shared pool of configurable resources. These resources can be computers, storage, or networking resources and can be provisioned and released entirely through software-based interaction. Usually, this model of computing requires no management effort or service provider interaction to acquire new computing resources.

NIST [17] defines three different cloud computing service models: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

In this work, we make heavy use of the IaaS model. It allows us to provision computing resources as and when needed by the system. An infrastructure cloud provides mechanisms for creating and managing virtual computing instances with software. The user is responsible for all the software that runs on the virtual computing resources and thus gains the flexibility to install any operating system needed. In this paper, the term cloud refers to an IaaS cloud.

Our system, however, is designed to operate as a PaaS cloud, with applications enabled to use our system as their platform for distributed graph storage and processing.

In this paper, we call a virtual machine provided by an infrastructure cloud a *compute node*. A compute node is an abstract concept of a computer; it shares many properties with a computer, but some of them are poorly defined. Only abstract specifications are stated, for instance, "Large Machine Template" instead of exact processor, memory, and storage specifications.

Since our architecture is built around a distributed computing model, we mainly work with sets of compute nodes that work in concert towards a common goal. In this work, we call such a set of compute nodes a *cluster*.

IV. Data Mining on Dynamic Graphs

To allow analysis of dynamic graph properties, we propose a system that leverages loud computing resources in order to run a distributed computing cluster.

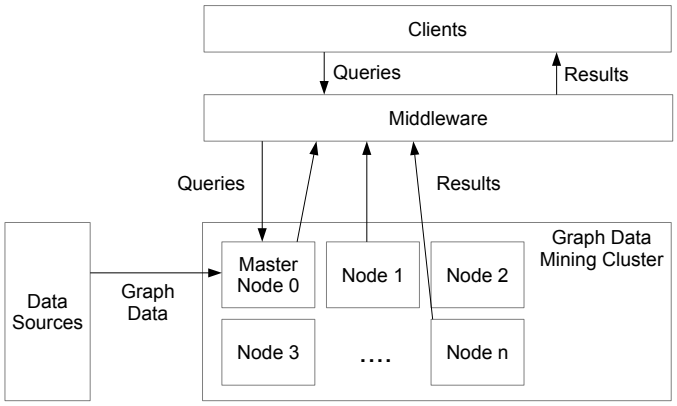

Fig. 1. Schematic diagram of our architecture.

A. Concept

As previously stated, our system is modeled around a distributed computing model. One of its main components is a cluster of compute nodes which work together to store the global model of the graph. Each compute node is stores and processes part of the global graph.

The cluster is interfaced by a middle-ware layer, which manages all aspects of the software system that are not related to the graph model for instance, authentication, user session management, and keeping track of general system health.

Client applications access the system via the middle-ware layer. They can be diverse depending on the field of application. Our prototype contains an example application which allows simple queries. The results are presented as visualizations.

Figure 1 shows a holistic view of the architecture and how the principal components of our system and illustrates how the components are linked.

The middle-ware interfaces with the master node of the cluster and is used by the clients to query data from the system. Individual compute nodes report their results directly back to the middle-ware. Data sources interface with the master node in order to change the global graph model. However, it is also possible that data sources are run the same cluster system.

Below we describe each component in more detail.

Graph Data Mining Cluster: This component is at the heart of the system and comprises many different compute nodes. The first compute node is the master node of the system and plays a special role: it tracks a list of sub-graphs that are available in the system. This list contains pointers to individual compute nodes that own a local copy of that specific sub-graph.

All update commands are sent to the master node, which ensures that the update commands are forwarded to the correct compute node.

Each compute node manages its own part of the graph. This is achieved by keeping a list of all vertices that are stored on a node. The list points to description files - termed profiles - that contain more details about these vertices.

A profile may contain arbitrary data as long as it is encoded in a structured format. In our implementation, we use JSON [18] to encode profile data. Only two fields in each profile are critical. The first is a vertex id, which identifies a vertex globally within the system, and the second stores the incoming and outgoing edges of the vertex.

As previously mentioned, the rest of the profile may contain any kind of data and be specific to a particular application. For instance, in the context of social network analysis, the profile could contain data that describes a person.

Algorithms that are applied to a graph can also update profile properties, for example, by adding intermediate results: an algorithm that calculates the degrees of the vertices could write the results into each vertex profile for later reference.

The dynamics of a graph are reflected in each vertex profile. Whenever a profile is changed, either by altering a profile property or by altering the list of incoming and outgoing edges, a copy of the profile is created. As time passes all history data becomes available to each vertex in the graph.

This allows our system to perform operations on a snapshot of the graph. We are thus able to determine the state of each vertex at any point in time.

Middle-ware: A middle-ware layer is needed to manage all aspects that are not related to the graph, for example, managing user sessions and authentication. It provides a RESTful [19] interface for client applications and relays requests and their results from the client to the cluster and vice versa.

The main purpose of the middle-ware layer is to send queries to the cluster and wait for results from the individual compute nodes. The system basically two types of queries: static and dynamic. Static queries specify a time-stamp which refers to a point in the past, and the cluster responds to the query on the basis of a snapshot of the graph at that time.

The master node of the cluster distributes a static query to all compute, which try to answer the query given their local sub-graph. Each compute node sends its result back to the master node, which in turn passes it in larger packets on to the middle-ware.

A static query ends after a timeout specified in the query and depends on the application. It should be as short as possible since the middle-ware waits for the timeout to expire and then returns the result to the clients.

Dynamic queries are handled in a similar way, but do not specify a time stamp for the snapshot, and specify a much longer timeout or none at all.

The middle-ware and the cluster keep lists of the dynamic queries. Whenever the graph changes each compute node affected checks all currently active dynamic queries in order to adapt their results.

Clients: The purpose of a client in our system is to view and analyze the graph data stored in the cluster. This is the reason why the clients are given only limited abilities to change the graph.

Clients are able to connect to the middle-ware via a RESTful service. They open two communication streams to the middle-ware: one, opened on demand, sends commands and queries

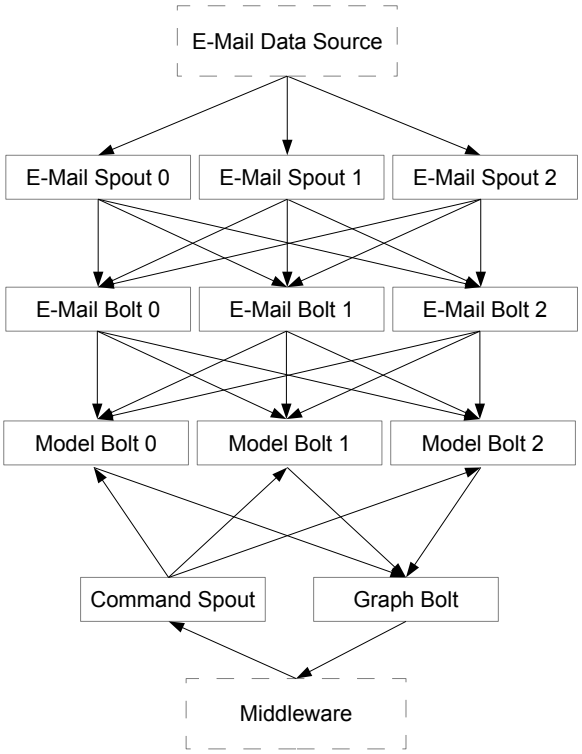

Fig. 2. Example Storm topology with three parallel users instantiated during setup, showing how data and commands are passed through the system. The dashed boxes are components outside the topology.

to the cluster, and the other is kept open for the complete duration of a user session; it is used by the middle-ware to direct query results back to a client.

The latter communication channel is especially important in cases where the client has sent a dynamic query to the cluster. Such queries are kept active until the user stops them, they time our, or the user session ends.

Data Sources: All the components presented so far access the system through the middle-ware layer. Data sources, however, are enabled to send data directly to the cluster. In our reference implementation, we allow data sources to be deployed on the very same cluster that also operates the graph processing engine. Hence, data sources can be used that would otherwise cause bottlenecks in the middle-ware layer.

Data sources are able to issue insert, update, and delete commands to the cluster. The master node of the cluster ensures that the commands are routed to the correct compute nodes.

B. Prototype

To demonstrate the feasibility of our approach, we created a reference implementation.

Our prototype implementation already contains many of the building blocks needed. It is able to store the model of a large dynamic graph and to expose this model to clients via a simple query interface.

The current implementation is based on Storm, a distributed and fault-tolerant framework for real-time computation. We

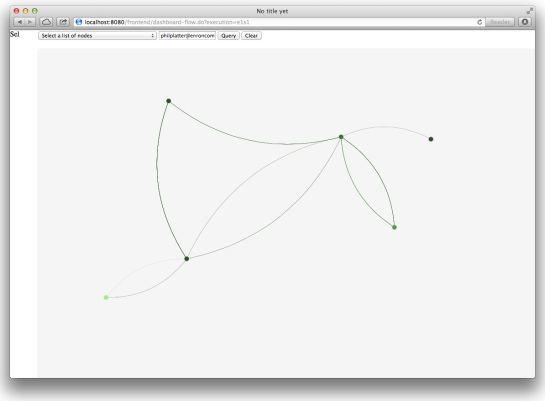

Fig. 3. Screen-shot of our web-based visualization.

use the storm concept of *spouts* to stream graph data and selection commands to the system. In Storm, very small program blocks called *bolts* are used to perform computations. In a Storm topology spouts and bolts can be connected.

Storm is capable of processing streaming big data by deploying a topology built of spouts and bolts on a compute cluster. Storm dynamically distributes spouts and bolts to optimize the overall system for throughput.

We leverage Storm's cluster computing capabilities in order to distribute our graph model and graph processing in the Cloud.

Its cluster capabilities make Storm an excellent example platform. It handles load-balancing automatically and adjusts the cluster to the workload on each compute node.

Storm is also very fault-tolerant. A Storm cluster can be resized dynamically by adding or removing nodes. In some application scenarios, where the cluster does not maintain a persistent state, some compute nodes can even be turned off without affecting system stability.

Figure 2 illustrates the topology of our prototype system and how spouts and bolts, parse e-mail messages. The e-mail data source sends a stream of plain e-mail messages to the e-mail spouts. Each e-mail spout separates the individual messages and passes them on to the e-mail bolt, where the plain message is parsed for sender and recipient address, subject, and message id. This tuple is then sent to the model bolts, where the dynamic model is stored. The middle-ware can send queries to the cluster via the command spout, which distributes the commands to all model bolts. The graph bolt accepts query results and forwards them to the middle-ware layer.

Our current query interface supports three simple query types: the first is able to query the complete graph currently stored in the system, the second allows specification of a list of node names that are to be loaded from the cluster, and the third provides the option to load a list of nodes and their direct neighbors. All queries can be submitted either as static or as dynamic queries.

Figure 3 the results of querying some specific profiles from

the Enron e-mail corpus [20] in order to investigate their communication relationship. Within the graph visualization, the user can modify the graphical representation of the graph by rearranging individual vertices, zooming in an out, and panning. Double licking a vertex loads all its neighbors. This mechanism allows close examination of the data by drilling deeper into the model.

V. Evaluation and Lessons Learned

Only a few data sets that are both very large and dynamic are available for research. We used two data sets to test our system with respect to both criteria.

We used the Enron e-mail corpus [20]. to test our system's handling of dynamic graph properties, that is, its ability to adapt the results to reflect changes in the graph. This corpus, which was made available after the Enron scandal [21] in October 2001, consists of 517,425 individual messages in 150 mailboxes. It is stored as a 2.5 GB large MBox directory, where each message, stripped of its attachments, is contained in an individual file.

To explore dynamic properties of graphs, we loaded the e-mail corpus. We followed the time stamps in the messages to recreate the history of the communication graph. Thus, we are able to observe how the communication network in Enron evolved over time.

In order to test our system with large graphs, we chose a Facebook sub-graph that was extracted from the Facebook system in 2009. Although it is possible to query Facebook for graph data, the company strictly limits the amount of data that can be extracted from their system. The data set was augmented to larger size using the Metropolis-Hastings algorithm. It contains 957,000 unique user profiles and their relationships [22].

The Facebook graph we used does not contain any time stamps. During the import, the data set was read in a sequentially, and the time stamps assigned indicate the time of import. We used this data set to experiment with large sparse graphs as typically found in social networks.

In our experiments, we were able to load both data sets to a cluster, and users were able to query the graph model.

The data source, cluster and middle-ware software components were deployed on our OpenNebula private cloud in a variety of configurations. We used virtual machine instances configured with two virtual CPUs and 4096 MB of RAM. We created clusters of different sizes to see if cluster size affects the time required to load a data set.

Our OpenNebula cloud resources are reserved for research and it is thus possible to dedicate the whole system exclusively to single experiments. It is equipped with two compute nodes which both feature two 6-core Intel(R) Xeon(R) X5690 CPUs running at a clock rate of 3.47 GHz. Both machines are also equipped with 36 GB RAM.

To avoid overprovisioning of resources, our experiments were limited to 24 virtual CPUs and 72 GB of RAM. However, the machines may encounter input output channel limitations, which are currently not monitored.

TABLE I
Load time of the Enron e-mail corpus with different configurations

Number of Nodes	Average	Fastest	Slowest
1	1091	942	1169
2	539	497	611
3	619	578	677
4	419	395	436
5	458	436	477
6	423	393	444
7	452	419	519
8	458	447	465
9	452	440	462

For each run of the experiment, we created all necessary virtual machines from scratch. Disk images and virtual machines from previous runs were not reused to avoid any effects due to file system caching which was enabled in the Linux kernels used.

For our tests with the Enron e-mail corpus, we used a designated virtual machine as data source. This designated machine was running software that can read any e-mail database stored in MBox format and exposes the plain e-mail messages on a TCP/IP socket. The measurements were embedded to the data source. The virtual machine stored a time stamp when the first client connected and requested to read the e-mail database. It then measured the elapsed time until all e-mail messages were sent to the processing cluster.

The cluster ran a Storm spout that was able to read from this socket and passed the plain e-mail messages to the cluster for further processing. This spout was designed to launch many instances in parallel to avoid reduces a potential bottle neck in input output operations.

Our results show that the massively parallel approach of our architecture leads to faster load times compared to sequential: on average, loading the Enron e-mail corpus sequentially into the model takes 1091 seconds whereas our fastest parallel load with 6 compute nodes took 393 seconds.

Table I lists the fastest, the slowest, and the average load times for Storm clusters containing 1 to 9 compute nodes. We recorded 5 runs to average out distortions.

Our results show that the load times become considerably shorter when the number of nodes increases from 1 to 4. Beyond this number, no significant changes in load time were measured. The configurations with 5 and 8 compute nodes showed load times of nearly 4 minutes.

This result is contrary to the naive expectation that a larger number of compute nodes leads to shorter load times. In fact, larger clusters lead to more complexity in message passing and a higher amount of network communication.

Since our current platform constitutes only a proof of concept, it lacks mechanisms for performance measurements inside the cluster. Therefore, it was not bench-marked against the current state of the art of graph databases.

Our system differs architecturally from other systems in

three main ways: (1) Frameworks such as Pregel [9] and the open source implementation Golden Orb [10] are designed for distributed offline processing of static graph data, whereas our approach performs online processing of dynamic graph data. (2) Although pure graph databases that rely on individual and multiple distributed Neo4j [7] instances have been presented [14], they both lack our system's ability to process dynamic graph data, and the former does not support distributed storage and processing of graphs. (3) Finally, systems such as HyperGraphDB [8] and Trinity [12] focus on distributed processing of graph data, but - unlike our system - they are also limited to static graph processing algorithms.

As part of our future work, we will focus on the implementation of graph algorithms that are comparable to implementations in other systems. Further, we will instrument our source code extensively to measure its performance.

Considering profiles, our system still has weaknesses. Profile versioning currently creates deep copies of the last profile used, which makes storing profile history complex and performance poor. We will address this issue and use well-know versioning algorithms in order to store only differential data.

The performance of many algorithms could possibly be increased if the graph were intelligently distributed over the cluster. This should be done in a way that preserves data locality. However, finding good splits in large, dense graphs is a complex problem. Nevertheless, we are planning to extend our framework such that most vertices of a sub-graph end up on the same compute node. Our current implementation places vertices on compute nodes randomly.

VI. Conclusion

We have presented a platform for distributed data mining on large dynamic graphs. We have argued that by keeping the history of all changes to the graph, we are able to calculate its dynamic properties. The ability to evaluate the dynamics of a graph opens up exciting new applications.

An initial evaluation of our system showed that the distributed approach leads to better performance. Since the system is still a work in progress, we evaluated it by importing the Enron e-mail corpus to the system. When we measured the import time we found that a purely sequential import takes the system approximately 1091 seconds on average, whereas a parallel import with 4 compute nodes in the cluster takes only 393 seconds.

References

[1] J Huan, D Bandyopadhyay, and W Wang. Comparing Graph Representations of Protein Structure for Mining Family-Specific Residue-Based Packing Motifs — Abstract. *Journal of Computational Biology*, 2005.

[2] Wikipedia Article on Facebook. http://en.wikipedia.org/wiki/Facebook.

[3] C Lynch. Big data: How do your data grow? *Nature*, 455(7209):28–29, 2008.

[4] Storm, distributed and fault-tolerant realtime computation. http://storm-project.net.

[5] Alessandro Vinciarelli, Maja Pantic, Hervé Bourlard, and Alex Pentland. Social signal processing: state-of-the-art and future perspectives of an emerging domain. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, October 2008.

[6] Matthias Steinbauer and Gabriele Kotsis. Building an Information System for Reality Mining Based on Communication Traces. In *15th International Conference on Network-Based Information Systems*, Melbourne, June 2012.

[7] Neo4j Graph Database. http://www.neo4j.org.

[8] B Iordanov. HyperGraphDB: a generalized graph database. *Web-Age Information Management*, pages 25–36, 2010.

[9] Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*, 2010.

[10] Golden Orb, Massive-Scale Graph Analysis. http://goldenorbos.org.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107–113, January 2008.

[12] B Shao, H Wang, and Y Li. The Trinity graph engine. Technical report, 2012.

[13] InfiniteGraph Powered by Objectivity. http://objectivity.com.

[14] Li-Yung Ho, Jan-Jan Wu, and 2012 IEEE 5th International Conference on Pangfeng Liu Cloud Computing CLOUD. Distributed Graph Database for Large-Scale Social Computing. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*.

[15] Chen Avin, Michal Koucký, and Zvi Lotker. How to Explore a Fast-Changing World (Cover Time of a Simple Random Walk on Evolving Graphs). *Automata*, 5125(Chapter 11):121–132, 2008.

[16] Andrea Clementi, Riccardo Silvestri, and Luca Trevisan. Information Spreading in Dynamic Graphs. In *Symposium on Principles of Distributed Computing*, pages 37–46, New York, New York, USA, 2012.

[17] P Mell and T Grance. The NIST definition of cloud computing (draft). *NIST special publication*, 800:145, 2011.

[18] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). *Network Working Group*, 2006.

[19] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, December 2008.

[20] William W. Cohen. Enron email dataset. http://www.cs.cmu.edu/~enron/, 08 2009.

[21] Enron scandal at-a-glance. http://news.bbc.co.uk/2/hi/business/1780075.stm, 08 2002.

[22] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *Proceedings of IEEE INFOCOM '10*, San Diego, CA, March 2010.