Socrates

A System For Scalable Graph Analytics

C. Savkli, R.Carr, M. Chapman, B. Chee, D. Minch
Johns Hopkins University, Applied Physics Laboratory
Mail Stop MP6-N312
Laurel, MD 20723-6099 USA
cetin.savkli@jhuapl.edu

***Abstract*— A distributed graph processing system that provides locality control, indexing, graph query, and parallel processing capabilities is presented.**

Keywords—graph, distributed, semantic, query, analytics

I. INTRODUCTION

Graphs provide a flexible data structure that facilitates fusion of disparate data sets. The popularity of graphs has shown a steady growth with the development of internet, cyber, and social networks. While graphs provide a flexible data structure, processing and analysis of large graphs remains a challenging problem.

Successful implementation of graph analytics revolves around several key considerations: rapid data ingest and retrieval, scalable storage, and parallel processing. In this paper we present a graph analytics platform that is particularly focused on facilitating analysis of large-scale attribute rich graphs.

Recently, NoSQL systems such as Hadoop have become popular for storing big data; however these systems face several fundamental challenges that make analyzing that data difficult:

- Lack of secondary indexing, which leads to poor performance of attribute queries (e.g. "What flights have we seen moving faster than 500mph?").
- Lack of locality control, which can lead to unnecessary movement of data.
- Lack of well-defined schema, which makes database maintenance challenging.

More traditional relational databases (a.k.a. RDBMS) do not have these problems, but face their own set of challenges when dealing with big data:

- Table structures not flexible enough to support new kinds of data easily.
- Poor parallelization & scalability.

We have developed a graph processing system called Socrates that is designed to facilitate the development of scalable analytics on graph data. Socrates combines the key features of these two approaches and avoids all of the above problems. It features several advances in data management for parallel computing and scalable distributed storage:

- Models data as a property graph, which supports representation of different kinds of data using a simple, extensible database schema.
- Implements the open-source Blueprints graph API[1], allowing analytics to be easily adapted to and from other Blueprints-enabled storage systems.
- Distributed storage & processing: Functionality to execute algorithms on each cluster node and merge results with minimal inter-node communication.
- Indexing: Every attribute is indexed for fast random access and analytics.
- Distributed management: Nothing in the cluster is centrally managed. This includes communication as well as locating graph elements.
- Locality control: Graph vertices can be placed on specific machines, a feature essential for minimizing data movement in graph analytics.
- Platform independence: Runs in the Java Virtual Machine.

II. RELATED WORK

Socrates is designed for attribute rich data – having many labels on both nodes and edges. Twitter's Cassovary, FlockDB, and Pegasus are fast at processing structural graphs. However, they do not leverage or provide facilities to store or use labels on edges or nodes besides edge weights [1][2][3]. Current graph benchmarking tools such as HPC Scalable Graph Analysis Benchmark (HPC-SGAB) generate tuples of data with the form <StartVertex, EndVertex, Weight> with no other attributes, which plays well to the strengths of Cassovary or Pegasus [4]. However, such benchmarks tend to ignore the functionality we specifically aim for within this work.

[1] http://www.tinkerpop.com/

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

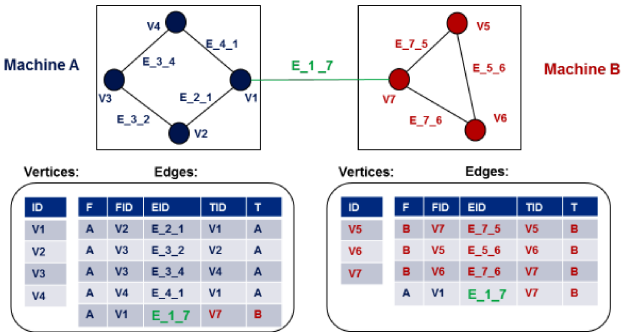

Fig. 1: Graph structure representation on the cluster.

Knowledge Discovery Toolbox (KDT) is a python toolbox built on Combinatorial BLAS, a linear algebra based library [5]. A linear algebra adjacency matrix approach is similar to Pegasus or GraphLab, however, KDT enables the use of attributes through filtering. Users create queries or filters on an input graph that generates a new adjacency matrix that is used by Combinatorial BLAS. However, this filtering process on large graphs can be expensive and lead to storage issues; KDT also requires that the adjacency matrix fit into distributed main memory [5].

Dynamic Distributed Dimensional Data Model (D4M) is a database and analytics platform built using Matlab built on various tuple stores such as Accumulo or HBase. The goal of D4M is to combine the advantages of distributed arrays, tuple stores, and multi-dimensional associative arrays [6]. D4M focuses on mathematical operations on associative arrays, which Graphs or adjacency matrices can be represented as. Socrates, however focuses primarily on property graphs and traditional edge or vertex operations.

Other graph databases focus more on content or tend to model specific relationship types such as those in an ontology. These databases include Jena, OpenLink Virtuoso, R2DF and other commercial offerings that use Resource Description Framework (RDF), which was originally designed to represent metadata [4][7][8]. RDF expressions consist of triples (subject, predicate and object) that are stored and queried against. The predicate in each triple represents a relationship between a subject and object. Intuitively, a general set of RDF tuples can be considered a graph although RDF is not formally defined as a mathematical concept of a graph [9].

Neo4J, Titan, HypergraphDB and DEX offer similar capabilities to Socrates. Socrates, Neo4J and Titan make use of the Blueprints API for interacting with graphs [10]. However, Socrates has extended this API to offer enhanced graph processing functionality that includes locality control, additional graph methods facilitating graph analytics, as well as a parallel processing capability.

Socrates is built upon a cluster of SQL databases, similar to Facebook's TAO and Twitter's deprecated FlockDB [11][3]. Facebook's social graph is currently stored in TAO, a data model and API specifically designed for social graphs. Facebook's social graph is served via TAO which was designed to support its workload needs: frequent reads with fewer writes, edge queries mostly having empty results and node connectivity and data sizes having distributions with long tails. Facebook's workload needs enable efficient cache implementations; since a large portion of reads are potentially in cache (e.g., people are interested in current events – time locality or alternatively a post becomes "viral" and is viewed by many people). TAO enables relatively few queries and stores its main data objects and associations as key-value pairs. Socrates stores attributes in a similar fashion, utilizing many tables in which each attribute is stored as value and the id (of a node or edge) as the key. This schema removes joins and associated memory and performance bottlenecks such as query planning or combinatorial explosions in the case of outer joins. TAO was developed with specific constraints in mind such as global scaling and time based social graphs, whereas Socrates is targeted at more general graph structures.

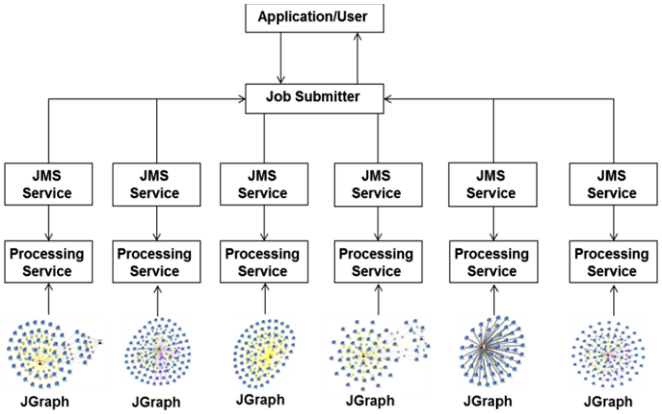

Fig. 2: The JGraph model of parallelism in Socrates. Users submit a processing job that is run in parallel on each node in the cluster, and given access to the subgraph stored on that node.

III. Design

A. Data Management & Locality

Figure 1 demonstrates Socrates' storage schema on a small graph in a two-node cluster. Structural graph information is stored in two tables on each machine: a vertex table containing only the IDs of the vertices stored on the machine, and an edge table containing, for every edge connected to a vertex in the vertex table: an edge ID; the IDs of both connected vertices; and identifiers specifying which machine each connected vertex is stored on. Thus, each vertex is stored on only one machine, while each edge is stored on either one machine (such as *E_4_1* or *E_7_5* in Fig. 1) or two (such as *E_1_7* in Fig. 1). Edges in Socrates are directed, but their direction can be ignored by undirected graph algorithms.

For each attribute present on any vertex or outgoing edge on a machine, Socrates also stores a two-column table consisting of <*id, val*> tuples, where *id* is either a vertex or edge identifier and *val* is the value of the attribute. This allows each attribute to be independently indexed and queried, while avoiding the complexity typically associated with table structure changes in relational databases.

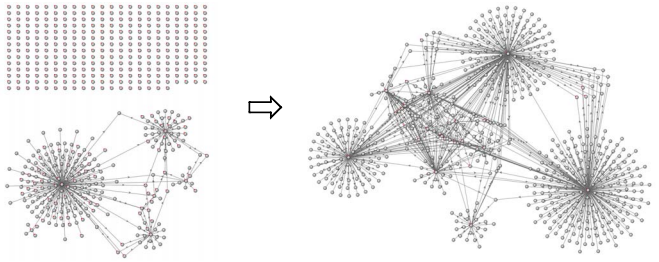

Fig. 3: The graph on the left represents one node of a four-machine cluster where data was archived without locality control. The vertices at the top with no edges have no neighbors on the local machine. In this case the probability that a neighbor resides on the same machine is ¼, which is reflected in the outcome where only about a quarter of vertices have their neighbors local. The graph on the right shows what a partial graph on the same machine looks like when archived using Socrates. The number of edges that connect to other machines is minimized, which enables more efficient computation of graph algorithms.

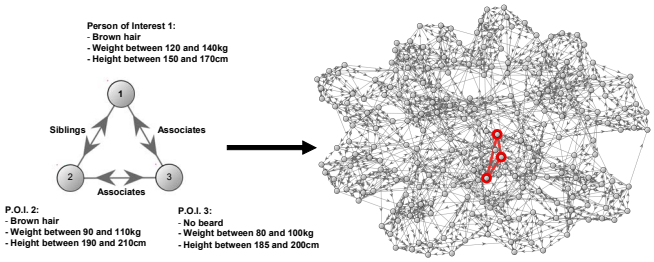

Fig. 4: An example of Socrates' sub-graph query capability. The left graph is the query graph, which contains both structural and semantic constraints. The right graph is the underlying data, with the instance of the query graph highlighted.

Users can access the data in the cluster via an augmented version of the Blueprints graph API. Besides the standard Blueprints methods, Socrates graph API provides additional methods that take advantage of its architecture for efficient implementation of analytics. One such method supports locality control, which involves adding vertices of the graph to specific machines. This machine-level access to the graph allows a user to partition the graph to minimize edge crossings between machines. The example in Fig. 3, which is built using the Brightkite data set [13], illustrates the benefit of locality control. The ability to place graph nodes in particular machines can also be used to automatically partition the data when attributes can be hashed to generate a machine ID, such as partitioning of graph based on latitude-longitude attributes of vertices.

Socrates also provides advanced query capabilities that have been efficiently implemented against our data model. For example, finding joint neighbors of a pair of vertices is a key operation for a variety of link discovery analysis. Socrates provides a function to do this with a single database query to the edge database on each machine (without any communication between cluster nodes), rather than iterating over neighbors of each vertex on the client.

Another advanced query capability Socrates provides is a function to query for all instances of a small sub-graph[2], which can include structural constraints (i.e., which vertices connect and which do not) as well as semantic constraints (i.e., constraints on vertex and edge attributes). See Fig. 4 for an example. This capability is based on an implementation of the VF2 algorithm [14], adapted to take advantage of Socrates' data model and the JGraph model of parallelism described in the next section.

[2] The maximum size of the sub-graph query varies depending on the strength of its constraints and the complexity of the underlying data Socrates' search algorithm aggressively prunes its search space, so generally if a query is too slow it is because the graph contains too many matches.

In addition to parallelized implementations of useful structural graph operations, Socrates also takes advantage of underlying SQL severs to efficiently perform various attribute operations in each cluster node without having to move data to the client.

B. *Parallel Processing*

A key feature for handling large-scale graphs is the ability to process the graph in parallel without having to move data. A primary challenge in parallel processing of graphs is that, for most nontrivial problems, analysis on each machine on the cluster requires access to data on other machines to be able to produce a result. This is an area where ability to control partitioning of the graph over the cluster becomes critical.

Another challenge in implementing parallelized graph analytics is designing an algorithm that takes advantage of distributed hardware wherever possible. Rather than limit users to a single parallel processing model, Socrates supports three models of parallel processing over the distributed graph: DGraph, JGraph, and Neighborhood. Each type provides a different tradeoff between ease of algorithm implementation, parallelism of client code, and network communication. This allows users to select the model which best suits the needs of their algorithm.

DGraph: In the first model, Socrates provides clients with access to the DGraph class, which implements the Blueprints API and abstracts away the distributed nature of the underlying graph. Methods of the DGraph class are implemented with parallel calls to the underlying database where possible, but all results are sent back to the client machine and no client code runs on the Socrates cluster. This model of parallelism is suitable for developing analytics that need a global view of the graph or make only simple queries, such as 'find me all vertices with attribute "last_name" equal to "Smith"'.

JGraph: In the second model, Socrates allows clients to create processing jobs that can be submitted to the cluster to run in parallel on each node; each job is given access to the JGraph local to the node it is being run on (See Fig. 2). A JGraph is another implementation of the Blueprints API that represents the partial graph stored on its local machine. Vertex

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

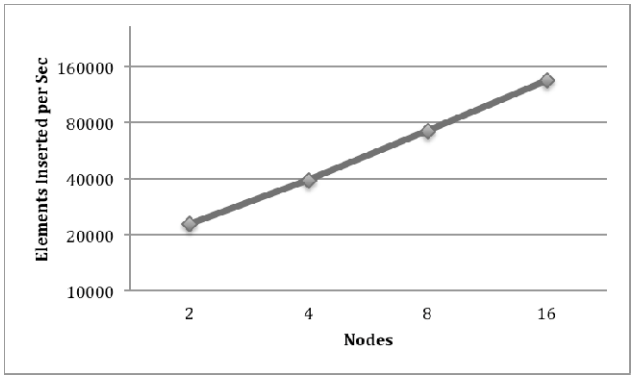

Fig. 6: Socrates insertion speeds (on a logarithmic scale) for an E-R graph with 10 million vertices and 100 million edges, with varying cluster sizes. On our cluster, Socrates has exhibited approximately linear ingestion speed-up as nodes are added.

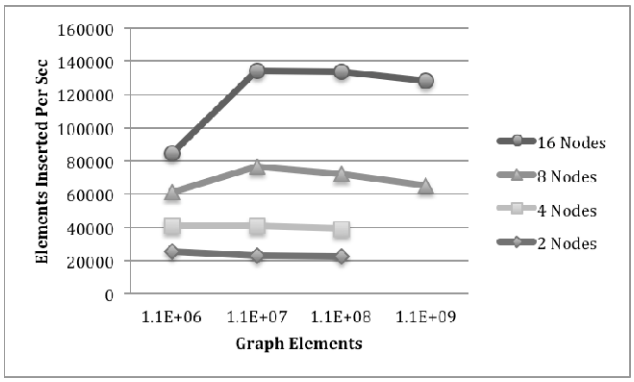

Fig. 5: Socrates insertion speeds for an ER graph with an average of 10 edges per vertex, with varying cluster sizes.

iterators used in parallel jobs on JGraphs iterate over vertices that are local in that machine. However any questions asked about local vertices, such as getNeighbors() operation, retrieves all matching results independent of where they are located. Therefore the implementation of parallel jobs is quite similar to a regular standalone program. User has a clear view of the boundaries of the local graph and can limit operations on the local graph based on this information.

This model of parallelism is suitable for developing analytics that can make use of a wide view of the graph as well as benefit from parallelism, such as sub-graph isomorphism (Fig. 4). It is also useful if the graph can be partitioned into disjoint sub-graphs that are small enough to fit on one machine; in this case, it is trivial to use Socrates' locality control features to make sure the entire sub-graphs are placed on the same machine, allowing any algorithm that runs on a DGraph in the previous model to be parallelized.

Neighborhood: The final model of parallelism supported by Socrates is intended for algorithms that perform local computation only, such as centrality measurements or PageRank. Socrates provides an interface that allows clients to define a function that will be run in batch on every vertex in the graph. When the function is called, its input is a TinkerGraph (an in-memory implementation of Blueprints) that contains one vertex labeled "root", and may contain other elements that the client specifies when the function processing job is submitted. The client is able to specify whether the TinkerGraph should contain the root vertex's immediate neighbors (or in/out neighbors only in the case of a directed graph) and their edges with the root, as well as any properties of vertices or edges that should be fetched. The client's function is then able to write out new property values for the root node or any of its neighboring edges (future implementations will allow new neighboring vertices and edges to be added as well).

This model of parallelism is intended to make it easy to write local-computation graph analytics that take full advantage of the computing power of a cluster. Under the hood, Socrates takes care of running the client function in parallel on each node using as many threads as the hardware on that node will support, optimizing SQL queries and inserts, caching frequently-used values, and minimizing network communication between nodes.

Communication for parallel processing is provided by Java Messaging Service (JMS) using publish-subscribe method. In order to eliminate centralized communication and potential bottlenecks, each machine operates its own message broker. Every machine on the cluster is therefore on an equal footing. Parallelizing message handling also eliminates a potential single point of failure in the system. Jobs are executed in parallel on each machine and the results are, optionally, returned back to the client submitting the job request.

IV. Performance

In this section, we provide preliminary performance benchmarks demonstrating the scalability of Socrates in terms of ingest and parallelized analytics.

A. Cluster configuration

Our cluster consists of 16 servers which are equipped with quad-core Intel Xeon E5-2609 2.5GHz processors, 64 GB 1600MHz DDR3 RAM, and two 4.0TB Seagate Constellation HDDs in RAID 0. The servers are running CentOS, and Socrates is using MySQL 5.5 with the TokuDB storage engine as its data store.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

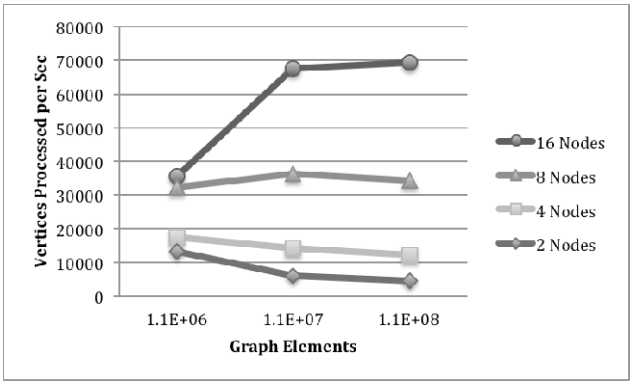

Fig. 7: Average processing speeds for an iteration of the connected component algorithm on an ER graph with an average of 10 edges per vertex.

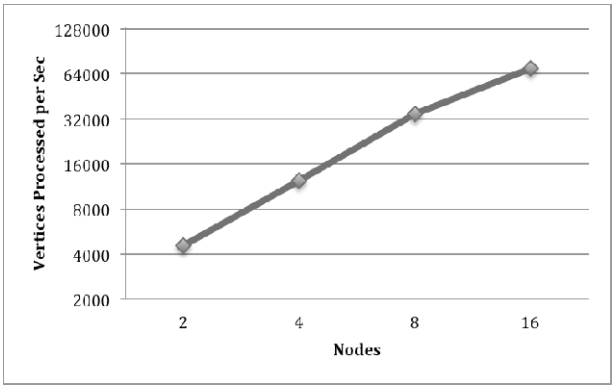

Fig. 8: Average processing speeds (on a logarithmic scale) for an E-R graph with 10 million vertices and 100 million edges, with varying cluster sizes. Again, we see approximately linear processing speed up as nodes are added.

B. *Ingest*

To measure our ingest speeds, we insert large randomly-generated Erdos-Renyi (E-R) graphs[3] into the Socrates cluster from an external machine. The E-R graphs we use here consist of 100-vertex connected components with an average of 1000 edges each, however the ingest speed of these graphs depends only on the number of vertices and edges, and not on the underlying structure of the graph.

We insert E-R graphs with a total size varying from 100,000 vertices and one million edges to 100 million vertices and one billion edges. In order to see how well our ingest speed scales as the size of a cluster grows, we have repeated the ingest benchmarks using only 2, 4, and 8 nodes in addition to the full 16.

Fig. 5 shows the ingest speeds, expressed as elements inserted per second, for graphs and clusters of varying sizes.

We can see that, although the 8- and 16-node cluster does not have time to reach its top speed when ingesting the smallest graph tested (which takes them 18 and 13 seconds, espectively), ingest speeds hold steady even as the input graphs grow to over a billion elements. Part of the reason for this is our use of TokuDB[4] as the storage engine for MySQL. Previous versions of Socrates used the standard InnoDB engine, and suffered from severe I/O bottlenecks when the graph grew above roughly 200 million edges, likely due to InnoDB no longer being able to fit the interior nodes of the B-Tree in its in-memory buffer pool. TokuDB uses cache-oblivious lookahead arrays in place of B-Trees, and seems to avoid this problem [12].

Fig. 6 shows the ingest speeds on a logarithmic scale for varying cluster sizes ingesting an E-R graph with 10 million vertices and 100 million edges.

We can see that Socrates ingest speed scales linearly to at least 16 nodes.

C. *Parallel Processing*

To measure our parallel processing capability, we use a naive connected component algorithm implemented in the Neighborhood parallelism model described above. On its initial iteration, the algorithm assigns each vertex a *component* attribute equal to the smallest vertex id among itself and its neighbors. On subsequent iterations, the algorithm examines the *component* attribute of itself and its neighbors, and updates its *component* to be the smallest value in the examined set. The algorithm terminates when no vertex's *component* changes in an iteration.

Not that this benchmark is not necessarily the fastest method for computing connected components, however it is useful as a benchmark because Socrates must fetch each vertex's in and out neighbors along with property information for each node. Thus, the speed at which an iteration of this algorithm runs can give us an idea of the batch processing capability of the Neighborhood parallelism model.

We have run this algorithm on the same graphs that were ingested in the previous experiment, i.e., Erdos-Renyi graphs consisting of connected components with 100 vertices and an average of 1000 edges each, varying in total size from 1.1 million to 1.1 billion elements. Once again, we have repeated the experiments using only 2, 4, and 8 nodes in our cluster in addition to the full 16.

Figs. 7 and 8 present our results of these experiments. The number of vertices processed per second, averaged over all iterations of the algorithm after the initial iteration, is on the y-axis. Note that processing a single vertex involves fetching its immediate neighborhood (an average of 10 edges and vertices), as well as the *component* property for each vertex in the neighborhood.

[3] http://www.citeulike.org/group/3072/article/1666220

[4] http://www.tokutek.com/products/tokudb-for-mysql/

These results are qualitatively similar to the ingest results. Fig. 7 demonstrates that, once the graph is large enough to allow the 8- and 16-node clusters to reach full speed, processing speed holds steady up to graphs of over a billion elements. Fig. 8 demonstrates that processing speeds also scale in an approximately linear fashion up to a cluster of 16 nodes.

ACKNOWLEDGMENT

Visualizations in the paper were produced using Pointillist, a graph visualization software developed by Jonathan Cohen. We acknowledge D. Silberberg and L. DiStefano for providing feedback and support during the development of Socrates.

REFERENCES

[1] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "WTF: The Who to Follow Service at Twitter," in *Proceedings of the 22Nd International Conference on World Wide Web*, Republic and Canton of Geneva, Switzerland, 2013, pp. 505–514.

[2] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *Ninth IEEE International Conference on Data Mining, 2009. ICDM '09*, 2009, pp. 229–238.

[3] P. Kalmegh and S. B. Navathe, "Graph Database Design Challenges Using HPC Platforms," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012, pp. 1306–1309.

[4] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey, "Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark," in *Proceedings of the 2010 International Conference on Web-age Information Management*, Berlin, Heidelberg, 2010, pp. 37–48.

[5] A. Buluç, A. Fox, J. R. Gilbert, S. A. Kamil, A. Lugowski, L. Oliker, and S. W. Williams, "High-performance Analysis of Filtered Semantic Graphs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2012, pp. 463–464.

[6] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. MCcabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (D4M) database and computation system," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012, pp. 5349–5352.

[7] J. P. Cedeño and K. S. Candan, "R2DF Framework for Ranked Path Queries over Weighted RDF Graphs," in *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, New York, NY, USA, 2011, pp. 40:1–40:12.

[8] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Comput Surv*, vol. 40, no. 1, pp. 1:1–1:39, Feb. 2008.

[9] J. Hayes and C. Gutierrez, "Bipartite Graphs as Intermediate Model for RDF," in *The Semantic Web – ISWC 2004*, S. A. McIlraith, D. Plexousakis, and F. van Harmelen, Eds. Springer Berlin Heidelberg, 2004, pp. 47–61.

[10] S. Jouili and V. Vansteenberghe, "An Empirical Comparison of Graph Databases," in *2013 International Conference on Social Computing (SocialCom)*, 2013, pp. 708–715.

[11] N. Bronson, Z. Amsden, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph," *USENIX Annu. Tech. Conf. ATC*, 2013.

[12] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious Streaming B-trees," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, NY, USA, 2007, pp. 81–92.

[13] E. Cho, S. A. Myers, J. Leskovec. Friendship and Mobility: Friendship and Mobility: User Movement in Location-Based Social Networks ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2011.

[14] Cordella, Luigi P., et al. "A (sub) graph isomorphism algorithm for matching large graphs." Pattern Analysis and Machine Intelligence, IEEE Transactions on 26.10 (2004): 1367-1372.