Research Directions for Big Data Graph Analytics

John A. Miller, Lakshmish Ramaswamy, Krys J. Kochut, and Arash Fard
Department of Computer Science
University of Georgia
Athens, GA, USA
{*jam, laks, kochut, ar*}*@cs.uga.edu*

***Abstract*—In the era of big data, interest in analysis and extraction of information from large data graphs is increasing rapidly. This paper examines the field of graph analytics from somewhat of a query processing point of view. Whether it be determination of shortest paths or finding patterns in a data graph matching a query graph, the issue is to find interesting characteristics or information content from graphs. Many of the associated problems can be abstracted to problems on paths or problems on patterns. Unfortunately, seemingly simple problems, such as finding patterns in a data graph matching a query graph are surprisingly difficult. In addition, the iterative nature of algorithms in this field makes the simple MapReduce style of parallel and distributed processing less effective. Still, the need to provide answers even for very large graphs is driving the research. Progress, trends and directions for future research are presented.**

***Keywords*-big data; graph analytics; graph databases, Semantic Web, social networks, graph paths, graph patterns;**

I. Introduction

Simply put, Big Data Analytics takes data on an unprecedentedly large scale to make predictions, find patterns and enhance understanding. In the past, the challenge was to create/obtain data, but now, and more so in the future, it will be what to do with all the available data. How will the data be stored, shared or made open? How can the right subsets of data be found for conducting data analytics? What advances in algorithms as well as parallel and distributed implementations will be possible? The challenges for big data analytics would be overwhelming if not for the progress already made in several disciplines: statistics, numerical linear algebra, machine learning, data mining, graph theory, graph mining, databases and parallel and distributed processing.

In many cases, the data is numerical in nature (or can be converted to this form). Often such data is captured in a matrix and used to estimate parameters in a predictive model. In other situations, the relationships between data items is what is of most importance. In such cases, the data may be captured in a graph. Many techniques have and are being developed for performing analytics on graphs.

Graph analytics has wide ranging applications in many diverse domains such as World Wide Web (WWW) data management, Internet and overlay management, road networks, online social networks and bio-chemistry. Most of these domains are characterized by massive, and in many cases dynamic graphs. Many routine tasks in these domains require analyzing the underlying graph via various types of queries. For example, the famous page rank algorithm for ranking Web search results is in essence a *link analysis* algorithm, and it works by iteratively propagating the weights (representing the importance of Web domains) through the edges (representing the hyperlinks) of a Web graph. As a second example, relationship analysis is a fundamental task in many social networks such as Facebook, Twitter, and LinkedIn. It is used for suggesting friends/products, and placing advertisements. Relationship analysis necessarily involves computing paths among the vertices (representing users) in a social network. Fan et al. [1] demonstrate how identifying suspects in a drug ring can be modeled as a subgraph pattern search problem. Driving direction computation in an online map application (e.g., Google maps, MapQuest, etc.), connectivity monitoring and root cause analysis in large-scale distributed systems, and identification of chemical structures and analysis of biochemical pathways in biological sciences are other examples of tasks requiring graph analytics.

Traditional graph computation algorithms, many of which are highly sequential in nature do not scale well to effectively support massive graphs. Two distinct approaches have been pursued in recent years to overcome the limitations of traditional graph analytics — (a) *designing paradigms to distribute the computation among the machines of a shared nothing cluster* and (b) *designing smart indexing techniques for on-demand execution of graph queries*. While MapReduce (MR) is a popular cluster computing paradigm, it is not well suited for graph analytics because many graph analytics tasks are iterative in nature. Recently, alternative paradigms based on the Bulk Synchronous Parallel (BSP) programming model [2] have been proposed. These include the "think like vertex" paradigm (exemplified by systems like Pregel [3], Giraph [4] and GPS [5]) and the "think like graph" paradigm. Many indexing schemes have been proposed for various types of graph queries including differential structures and G-String [6] (for pattern matching queries) and 2-Hop [7], GRIPP [8] and Dual-labeling [9] (for reachability queries).

Despite these recent advances, scalable graph analytics is still challenging on multiple fronts. First, designing parallel

graph algorithms whether in the vertex-centric or graph centric paradigms is not straightforward; certain problems such as subgraph pattern matching are notoriously difficult to parallelize. Second, the performance of cluster-based graph computation frameworks is dependent upon multiple factors such as vertex distribution among compute nodes, characteristics of the algorithm in terms of whether the computation is confined to subsets of compute nodes at various stages of the computation, and computation and communication capabilities of the cluster. Managing the inherent tradeoffs among these diverse factors so as to achieve close to optimal performance is a significant challenge. Third, many of the existing graph indexes are *brittle* with respect to graph changes, and hence are not cost-effective for dynamic graphs. Thus, for dynamic graphs, it is necessary to design indexing schemes that are more flexible and resilient to graph changes. Fourth, in many applications such as Linked Open Data, the graph data is geographically distributed (for example, in multiple data centers). This adds an additional layer of complexity. To our best knowledge very few of the existing research efforts consider data that is split amongst multiple locations.

The rest of this paper is organized as follows: Section II provides basic definitions and outlines key problems in the domain of graph analytics. Current and future applications of graph analytics are discussed in section III. Computational models and frameworks used for efficient parallel and distributed implementations are discussed in section IV. Finally, section V concludes the paper.

II. Graph Analytics

When relationships between data items take center stage (e.g., social networks), big data analytics often takes the form of graph analytics, in which the data items are represented as labeled vertices, and the relationships as labeled edges. Many problems in graph analytics may be formulated in terms of labeled multidigraphs. A labeled multidigraph allows multiple directed edges between any two vertices, so long as they are differentiably labeled. More formally, a *labeled multidigraph* may be defined as a 4-tuple $G(V, E, L, l)$ where

$$\begin{array}{ll} V \ = \ \text{set of vertices} & \\ E \subseteq V \times V \times L & \text{(set of labeled edges)} \\ L \ = \ \text{set of labels} & \\ l : V \rightarrow L & \text{(vertex labeling function)} \end{array} \quad (1)$$

The connections between vertices are characterized by a set of edges. When not considering edge labels, $E \subseteq V \times V$ and the multidigraph becomes a digraph. For a digraph, $uv \in E$ mean that there is a directed edge from vertex u to v. The same notation will be used for multidigraphs, rather than the more detailed and precise projection $uv \in \pi_{12}(E)$.

A simple way to characterize the connectivity is in terms of children and parents, as defined by the following two set-valued functions.

$$child(u) = \{v : uv \in G.E\}$$
$$parent(u) = \{w : wu \in G.E\}$$

Many of the problems in graph analytics involve finding paths, patterns or partitions in very large data graphs (e.g., graphs with a billion edges). These problems are strongly interrelated. A path may be viewed as a simple linear pattern and partitioning is needed for both path and pattern problems, when graphs become too large to store or process on a single machine or single thread.

A. Path Problems

1) Reachability: Path problems involve asking questions about paths between vertices in graph G. The simplest is given two vertices, $u, w \in G.V$, find a path (set of edges) connecting them.

$$path(u, w) \ = \ uv_1 l_{i_1}, v_1 v_2 l_{i_2}, \ldots, v_n w l_{i_{n+1}} \in G.E$$

This can be generalized to return all paths between u and w.

$$a\text{-}paths(u, w) \ = \ \{p : p = path(u, w)\}$$

The arguments may also be generalized to sets of vertices. Reachability is simply

$$reach(u, w) \ = \ \exists path(u, w)$$

Reachability analysis has applications in many domains including XML indexing and querying, homeland security, navigation in road networks and root causes analysis in large-scale overlay-based distributed systems. A straightforward approach to this problem is to do an on-demand traversal (breadth-first or depth-first) on the graph. However, graph traversal is $O(v + e)$ where v (e) is the number of vertices (edges) in the graph. This makes traversal-based approaches unsuitable for very large graphs especially when the query loads are high. An alternate choice is to compute the *Transitive Closure (TC)* of the graph. But the storage costs of TC are too high ($O(v^2)$). To address these issues, several indexing-based approaches have been proposed. As the name suggests, these approaches rely upon certain indexes (sometimes stored in a relational database) for speeding up the reachability query evaluation. The indexes are constructed by doing a breadth-first or depth-first traversal (a one-time cost), and harnessed to answer many reachability queries. Examples of index-based reachability analysis include 2-Hop, Duallabeling, and Gripp.

Future Directions: While reachability analysis in static graphs has received considerable research attention in recent years, surprisingly, there is very little work on reachability analysis in dynamic (time-evolving) graphs. Many of the approaches cannot be extended to dynamic graphs in a straightforward manner because they are too brittle to handle graph

changes. In other words, even minor changes in the graph require massive updates to the index structures. Developing robust reachability analysis frameworks for dynamic graphs poses many important challenges. First, there can be multiple temporal classes of reachability queries including *version-specific reachability queries* (where reachability testing is done a specific version of the graph), *inverse version-specific queries* (finding the first/n^{th}/all version(s) satisfying a given reachability test) and *continual reachability queries* (trigger queries that require continuous monitoring of reachability status). Each class has unique requirements and hence needs very distinct approaches. Second, the straight forward approach of re-indexing the graph on every change is very costly, and hence impractical. Thus, we need a framework that manages the tradeoffs between the indexing costs and query latencies. Third, we need better (and probably simpler) indexing strategies that can be *incrementally maintained* as the underlying graph changes. Fourth, most of the existing studies on reachability analysis use Relational Databases (RDBs) or main-memory indexing structures. However, both of them have inherent limitations. While traditional RDBs are often too bulky (and thus perform poorly especially for ingesting large amounts of indexing data), main memory indexing schemes are limited by the main-memory availability. An important and interesting question in this regard is whether recent research on No-SQL databases such as Cassandra, BigTable, MongoDB and DynamoDB can be harnessed for storing reachability indexes.

In two recent research projects we demonstrated how the interval-based indexing paradigm can be extended for answering snapshot-specific and continuous reachability queries in dynamic hierarchies and graphs [10], [11]. However, we believe that the research on reachability analysis in dynamic graphs is in very nascent stages, and much more work needs to be done to address the above challenges.

Finding paths constrained by a formal language, i.e., where labels of edges forming a path must form a string from a formal language over an alphabet Σ, have recently gained significant attention This can involve a single path (e.g., shortest) or all paths between u and w. The problem of finding simple paths constrained by regular expressions has been studied quite intensively [12], [13]. Formal language constrained graph problems were discussed in [14], who showed that shortest path problems, when constrained by a context-free language can be solved in polynomial time. However, finding simple paths between a source and a given destination, constrained by a regular language, is $\mathcal{NP}$-hard, unless the graph itself is treewidth bounded, when it can be solved in polynomial time.

More research is needed in this area, especially in regard to very large and distributed graphs, including the very large data sets within the Linking Open Data project, discussed in section III.B, later in this paper.

2) Shortest Path: The purpose of shortest path problems is to find a path with the minimum distance (cumulative edge weight) that includes all k vertices in the path. Versions exists for both directed and undirected graphs. When $k = 2$, Dijkstra's Algorithm [15] or the Bellman-Ford algorithm [16] may be used. For a digraph, let the edge label $l(e)$ represent an edge weight, then given vertices u and w, find $s\text{-}path$.

$$s\text{-}path(u,w) \;=\; \underset{p \in a\text{-}paths(u,v)}{\operatorname{argmin}} \Big[\sum_{e \in p} l(e) \Big]$$

For $k = 3$, three applications of Dijkstra's Algorithm (or equivalent) will suffice to find the short path connecting all three vertices. The all-pairs short path problem [17] is also of interest in Big Data Analytics.

B. Pattern Problems

A simple and common form of pattern query, is to take a query graph Q and match its labeled vertices to corresponding labeled vertices in a data graph G.

$$pattern(Q,G) \;=\; \Phi : Q.V \rightarrow 2^{G.V} \text{ such that } \forall u' \in \Phi(u), l(u') = l(u)$$

One may think of vertex u in the query graph Q having a set of corresponding images $\{u'_i\}$ in the data graph G.

1) Graph Simulation: In addition to the labels of the vertices matching, patterns of connectivity should match as well; e.g., child match. Given, a possible match between $u \in Q.V$ and $u' \in \Phi(u)$, it is accepted iff for each vertex v in $child(u)$ there is a vertex in $\Phi(v)$ that is present in $child(u')$ as well.

$$match_c(u,u') = \forall v \in child_Q(u), \exists v' \in \Phi(v) \text{ such that } u'v' \in G.E$$

Algorithms for graph simulation typically work as follows: For each vertex $u \in Q.V$, initially compute the mapping set $\Phi(u)$ based on label matching. Then, repeatedly check the child match condition, $match_c$, for all vertices to refine their mapping sets until there is no change. For example, in Figure 1, $\Phi(2_Q) = \{2_G, 7_G\}$, so both vertices must undergo a child match, $match_c(2_Q, 2_G)$ and $match_c(2_Q, 7_G)$. The $match_c(2_Q, 7_G)$ condition is evaluated as follows:

$$match_c(2_Q,7_G) = \forall v \in \{1_Q, 3_Q, 4_Q\}, \exists v' \in \Phi(v) \text{ such that } 7_G v' \in G.E$$

The $match_c$ is true, since $8_G \in \Phi(1_Q)$ and $7_G\, 8_G \in G.E$, $5_G \in \Phi(3_Q)$ and $7_G\, 5_G \in G.E$, and $9_G \in \Phi(4_Q)$ and $7_G\, 9_G \in G.E$. If the $match_c$ evaluated to false, vertex 7_G would be removed from $\Phi(2_Q)$.

Similarly, one may wish to match parents. Given, a possible match between $u \in Q.V$ and $u' \in \Phi(u)$, it is accepted iff for each vertex in w in $parent(u)$ there is a vertex in $\Phi(w)$ that is present in $parent(u')$ as well.

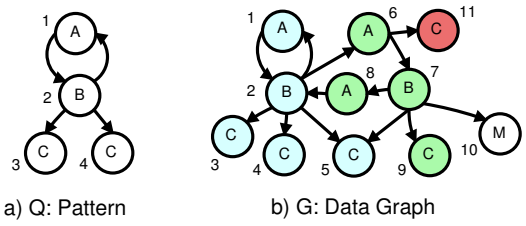

Figure 1. An Example for explaining the graph simulation algorithm

$$match_p(u, u') = \forall w \in parent_Q(u), \exists w' \in \Phi(w) \text{ such that } w'u' \in G.E$$

When the connectivity constraint is $match_c$, the pattern matching model is referred to as *graph simulation* [18], while when both $match_c$ and $match_p$ are used it is referred to as *dual simulation* [19].

To further restrict the matches, one may wish to eliminate solutions that contain large cycles which are possible to appear with dual simulation. Various locality restrictions may be added to dual simulation for this purpose. For *strong simulation* [19], any solution (match in G) must fit inside a ball of radius equal to diameter of the query graph Q.

Strict simulation [20] is based on strong simulation, but applies dual simulation first to reduce the number of balls. This also reduces the number of solutions.

A further restriction that reduces the number of balls and makes the balls smaller, is called *tight simulation* [21]. First the center of the query graph Q, call it u_c, is found and then balls are created for $u' \in \Phi(u_c)$. In addition, the radius of these balls is equal to the radius, not the diameter, of the query graph.

Tight simulation can be modified to produce results closer to subgraph isomorphism by using cardinality restrictions on child and parent matches to push results towards one-to-one correspondences. This modification is referred to as *Cardinality Restricted (CAR)-tight simulation* [22]. For $match_c(u, u')$ to be true, in addition to the constraints for tight simulation, the child count for each label must be at least as large for vertex $u' \in G.V$ as it is for vertex $u \in Q.V$. For example, while tight simulation evaluates $match_C(2_Q, 13_G)$ to true, as 14_G is used to match both of 2_Q's children, CAR-Tight simulation evaluates it to false, as 14_G has only one C-labeled child, while vertex 2_Q has two.

2) Graph Morphisms: More complex and often more constrained forms of pattern matching occur when a complete correspondence between edges is required.

$$match_e(Q, G) = \forall uv \in Q.E, \exists u'v' \in \Phi(u) \times \Phi(v) \bigcap G.E$$

This requires that for any edge $uv \in Q.E$, there must be a corresponding edge $u'v' \in \Phi(u) \times \Phi(v) \bigcap G.E$. In such case, the $\Phi(\cdot)$ set-valued function may be decomposed into a set of mapping functions $\{f_i(\cdot)\}$ that map a vertex $u \in Q.V$ to a vertex $u' \in G.V$. This form of pattern matching is called *graph homomorphism* [23]. If we further require the mapping functions $\{f_i(\cdot)\}$ to be bijections between $Q.V$ and $G'.V$, where G' is a subgraph of G ($G' \subseteq G$), then the form of pattern matching is called *subgraph isomorphism* [24]. (Some authors make a distinction between subgraph isomorphism and graph monomorphism (injective mapping), by requiring for subgraph isomorphism that G' to be induced by the selected vertices, i.e., include all edges having both endpoints in $G'.V$ [25].) The difference between graph homomorphism and subgraph isomorphism is that the former requires a correspondence between vertices, while the latter requires a one-to-one correspondence.

According to [26], the tightest upper bound known for such pattern matching algorithms is

$$O(N_Q {N_G}^{N_Q})$$

where $N_Q = v_Q + e_Q$ (the number of vertices and edges in the query graph) and $N_G = v_G + e_G$ (same for data graph). As query graphs increase in size, the complexity of pattern matching goes up rapidly. Unless there is a fixed upper bound on N_Q, finding subgraphs matching the query graph is $\mathcal{NP}$-hard.

Figure 2 shows an example of a query graph Q and data graph G, and all eight forms of pattern matching. In the example, loosely inspired from Amazon's product co-purchasing network, if a product familty u is frequently co-purchased with product family v, the graph contains a directed edge uv from vertex u to v. Here, each letter inside the vertex is the category of the product and represents its label. Moreover, each number beside a vertex represents its ID number. The subgraph matching results of this example are displayed in Table I. For the first two rows, the set-valued Φ function is given, while for the next four, results are segmented into balls, and for the last two, mapping functions are given. The column *Count* displays the total number of vertices appearing in the results.

A more flexible type of morphism called *graph homeomorphism* [27] can be thought of as representing a topological match. The idea is that is does not matter whether vertices u and v are connected directly, i.e., $uv \in G.E$ or indirectly. A sequence of edge subdivision and smoothing operations can be performed as part of the topological match. Subdivision occurs when a vertex $w \in G.V$ is inserted between u and v, replacing the edge uv with uw and wv. Smoothing goes the other direction, replacing $uw, wv \in G.E$ with uv, so long as w is connected to nothing else ($indegree(w) = outdegree(w) = 1$).

Table II shows the complexity results for the nine graph pattern matching models discussed. The ones based on graph simulation are in $\mathcal{P}$, while those based on morphisms are $\mathcal{NP}$-hard. The table also indicates the containment hierarchy. In many cases the results of one model are strictly contained within that of another. In some cases, they are incomparable, e.g., CAR-tight simulation and graph

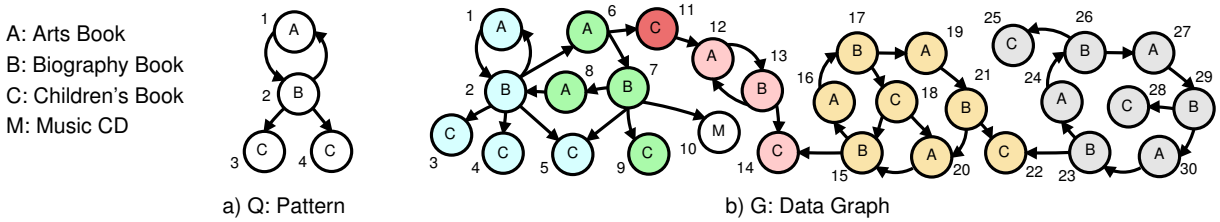

Figure 2. An Example of Subgraph Pattern Matching that shows the difference of the results for different Models

Table I
RESULTS OF DIFFERENT PATTERN MATCHING MODELS DISPLAYED IN FIGURE 2

Model	Subgraph Results	Count
Graph Simulation	$\Phi(1,2,3,4) \rightarrow (\{1,6,8,12,16,19,20,24,27,30\},\{2,7,13,15,17,21,23,26,29\}$ $,\{3,4,5,9,11,14,18,22,25,28\},\{3,4,5,9,11,14,18,22,25,28\})$	29
Dual Simulation	$\Phi(1,2,3,4) \rightarrow (\{1,6,8,12,16,19,20,24,27,30\},\{2,7,13,15,17,21,23,26,29\}$ $,\{3,4,5,9,14,18,22,25,28\},\{3,4,5,9,14,18,22,25,28\})$	28
Strong Simulation	$\Phi(1,2,3,4) \rightarrow (\{1,6,8\},\{2,7\},\{3,4,5,9\},\{3,4,5,9\}),(12,13,14,14)$ $,(\{16,19,20\},\{15,17,21\},\{14,18,22\},\{14,18,22\})$	20
Strict Simulation	$\Phi(1,2,3,4) \rightarrow (\{1,6,8\},\{2,7\},\{3,4,5,9\},\{3,4,5,9\}),(12,13,14,14)$	12
Tight Simulation	$\Phi(1,2,3,4) \rightarrow (1,2,\{3,4,5\},\{3,4,5\}),(12,13,14,14)$	8
CAR-Tight Simulation	$\Phi(1,2,3,4) \rightarrow (1,2,\{3,4,5\},\{3,4,5\})$	5
Graph Homomorphism	$f(1,2,3,4) \rightarrow (1,2,3,4),(1,2,3,5),(1,2,4,5),(1,2,3,3),(1,2,4,4),(1,2,5,5),(12,13,14,14)$	8
Subgraph Isomorphism	$f(1,2,3,4) \rightarrow (1,2,3,4),(1,2,3,5),(1,2,4,5)$	5

Table II
COMPLEXITY CLASS OF DIFFERENT PATTERN MATCHING MODELS

Model	Complexity Class	Source	Results Contained in
Graph Simulation	Quadratic	Henzinger et al. 1995 [18]	-
Dual Simulation	Cubic	Ma et al. 2011 [19]	Graph Simulation
Strong Simulation	Cubic	Ma et al. 2011 [19]	Dual Simulation
Strict Simulation	Cubic	Fard et al., 2013 [20]	Strong Simulation
Tight Simulation	Cubic	Fard et al., 2014 [21]	Strict Simulation
CAR-Tight Simulation	Cubic	Fard et al., 2014 [22]	Tight Simulation
Graph Homeomorphism	$\mathcal{NP}$-hard	Fortune et al., 1980 [28]	-
Graph Homomorphism	$\mathcal{NP}$-hard	Hell and Nesetril, 1990 [23]	Graph Homeomorphism and Tight Simulation
Subgraph Isomorphism	$\mathcal{NP}$-hard	Garey and Johnson, 1979 [29]	Graph Homomorphism and CAR-Tight Simulation

homomorphism.

So far the edge labels have been largely ignored. They simply further restrict the edge matching, i.e., the edges' endpoints must correspond and their labels must match.

$$match_e(Q,G) = \forall uv\lambda \in Q.E, \exists u'v'\lambda' \in G.E \text{ and } u'v' \in \Phi(u) \times \Phi(v) \text{ and } \lambda = \lambda'$$

Labels may be integers, reals, vectors, strings or tuples, depending upon the application. For query processing, integer representations may often be used for speed (e.g., unique identifiers or hash codes for strings, etc.). For more generalized queries, one may relax the label matching for vertex/edge labels in query graph Q to include the following:

- Wildcards: Special characters are used to match zero or more characters/one character.
- Regular Expressions: String patterns may be specified using ranges, unions and closures.
- Variables: The use of variables, e.g., `?x hasColor 'red' and ?x numTires 4`, allows the label to be unspecified, other than by the relationships it is in. Query languages for graph databases and RDF triplestores, allow variables for the relationship as well, e.g., `?x ?y 'red'`.
- Predicates: Implicit in label matching is the equality predicate. Most practical query languages will at least include the common six (`=, !=, <, <=, >, >=`).

Regarding the fact that answering pattern queries on massive graphs can be very time consuming, devising tech-

niques to improve their response time is an active field of research. An important technique is design and implementation of distributed algorithms to harness the power of Big Data platforms for this purpose [30], [20]. Also, a very recent thread of research investigating usage of view and caching techniques with respect to pattern queries [31], [22]. Moreover, real-world data graphs are evolving over time; i.e., there are minor changes in their structure through the time. Hence, it should be possible to design incremental algorithms for pattern problems in many applications [32].

Another area of research involves situations where one is interested in incomplete or inexact matches of Q in G. For example, one could find maximum (or maximal) partial matches of Q in G. Maximum can be measured in terms of missing vertices or missing edges. The former problem is called Maximum Common Subgraph (MCS), while the latter is called Maximum Common Edge Subgraph (MCES). A graph C is a common subgraph to graphs Q and G, when it is isomorphic to subgraphs of each.

$$common(Q, G) = C \text{ such that } C \text{ isomorphic to } Q' \text{ and } G'$$

where $Q' \subseteq Q$ and $G' \subseteq G$. An MCS is a common subgraph with the maximum number of vertices [33], while an MCES is a common subgraph with the maximum number of edges [34]. These types of pattern matching are not the focus of this paper, but the following paper [35] provides a good survey.

The long term trend for research in graph pattern matching is to the attack the problem of $\mathcal{NP}$-hardness (e.g., Subgraph Isomorphism and Graph Homomorphism, see Table II) from two directions. Effective techniques for *indexing*, *ordering* evaluations and *pruning* away vertices have provided huge speed-up, e.g., compare the performance recent algorithms, DualIso [36] and TurboIso [37], to that of the original algorithm for subgraph isomorphism, Ullmann's Algorithm [24]. The other direction, is to create more sophisticated polynomial algorithms that produce results more closely resembling the results produced by Subgraph Isomorphism. As shown in Table I, the move from graph simulation to dual to strong to strict to tight to CAR-tight simulation, illustrates the progress in this research direction. Although more complex, an extension beyond dual simulation to also check grandchildren could be tested. Many combinations of checking grandchildren (or grandparents) could be added to all the simulation models described above. The polynomial-time algorithms developed could be closer to the results produced by subgraph isomorphism. Unfortunately, providing absolute or relative error bounds is complicated by the fact that related inexact problems like MSC and MCES are Approximable APX-hard [38]. The other avenue is to apply more computational power through parallel and distributed techniques, see section IV.

III. Applications

A. Graph Databases

Graph databases [39] have existed form some time. Recently, with the emergence of NoSQL databases [40] as an alternative to traditional Relational Databases for big data applications requiring greater storage and performance, graph databases, along with document databases, are gaining in momentum. Some of the popular graph databases are Neo4j [41], OrientDB [42] and Titan [43].

In this paper, the focus is not on graph databases, but rather how advances in graph pattern matching could be used in graph database engines to improve query processing. Neo4j supports two query languages Cypher and Gremlin [44]. Consider the following query in the Cypher language.

```
MATCH (x: Lawyer, y: Doctor, z: Lawyer,
       x-[:FRIEND]->y,
       x-[:COMPETES_WITH]->z,
       y-[:FRIEND]->z)
```

Given two lawyers and one doctor, where the first lawyer is a friend of the doctor and competes with the second lawyer, whom the doctor is friends with, find all (or a sufficient number of) occurrences of the query graph in the large data graphs making up the graph database. Typically, graph database query engines will solve such pattern matching queries using (i) subgraph isomorphism, (ii) graph homomorphism or (iii) graph homeomorphism algorithms.

GraphQL [45] defines graph pattern matching in terms of subgraph isomorphism. The paper defines a Φ function similar to ours, but generalizes to matching a predicate f_u rather than a label l. Given a vertex $u \in Q.V$, the initial matches in G are defined as follows:

$$\Phi(u) \;=\; \{u' : u' \in V.G \text{ and } f_u(u')\}$$

The pattern matching algorithm used in GraphQL first computes Φ for all vertices in Q (these are called the feasible mates) and then narrows down the choices by checking the correspondence of edges.

The GrGen [26] uses graph homomorphism to match query and data graphs. Although graph pattern matching queries are much faster in graph databases than in relational databases [44], current and new research ideas could be incorporated for further speed-up. Graph databases can also benefit from the considerable amount of research performed on indexing techniques [46].

B. Semantic Web

The concept of Semantic Web has been introduced by Tim Berners-Lee as an evolution of the World Wide Web to enable data sharing and reuse "across application, enterprise, and community boundaries". The Semantic Web is based on a number of standards, including the Resource Description Framework (RDF), the Web Ontology Language (OWL) and SPARQL. Conceptually, data encoded using RDF is

represented as a directed labeled multigraph, making RDF similar in many respects to graph database models. In an RDF graph, vertices are the resources (IRIs), blank nodes, or literals and edges are formed from RDF triples (the triple's predicate, which is an IRI, is the edge's label). Strictly speaking, blank nodes may have no identifiers and therefore no corresponding labels, which makes an RDF graph slightly different than a multigraph introduced in section II. However, it is a minor problem, since most RDF serialization formats require blank node identifiers.

While many implementations of RDF triple-stores rely on some form of a relational database, in some cases, RDF triple-stores are organized as graphs [47]. Some other implementations are quad stores, as RDF data sets may include multiple graphs and the graph to which a triple belongs is the fourth element, making it a quadruple. SPARQL is the query language for RDF data sets, recommended by the World Wide Web Consortium. The example Cypher query from section III.A looks very similar when expressed in the in the SPARQL query language:

```
SELECT ?x, ?y, ?z
WHERE { x a Lawyer . y a Doctor . z a Lawyer .
        x friend y .
        x competes_with z .
        y friend z }
```

There has been a considerable amount research conducted to optimize query engines for processing SPARQL queries [48]. Much of the progress involved development of sophisticated indexing strategies and graph-based storage models.

Recently, a Linking Open Data (LOD) project [49] has been initiated to provide a method of publishing a variety of structured data sets as interlinked RDF data sets. As of 2014, the LOD project comprised 1014 interlinked RDF data sets spanning a multitude of knowledge areas, such as life sciences, geographic, government, social networking, publications, media, and linguistics. At the center of it is DBpedia, an RDF representation of the Wikipedia, which is interlinked with a high number of other data sets. Overall, the size of the interlinked RDF graph in the LOD cloud is measured in tens of billions of RDF triples and therefore edges (over 80 billion as of this writing).

As the sizes of individual RDF data graphs continue to grow dramatically, optimization of processing of SPARQL queries becomes even more important, especially in view of the need for complex, hypothesis-driven [50] and analytics-related queries. Much effort must be dedicated to distributed processing of SPARQL queries [51], [52]. Furthermore, processing of federated SPARQL queries (introduced in SPARQL 1.1) on the LOD graph is challenging and requires vigorous research.

As the individual data sets dramatically increase in size, RDF graph partitioning and its impact on distributed processing of SPARQL queries and RDF graph analytics [53] is of significant importance. SPARQL query processing was formulated in terms of subgraph isomorphism and related to graph databases in [54]. A SPARQL implementation based on graph homomorphism is given in [55]. Even though SPARQL's OPTIONAL graphs and the UNION operator offer much flexibility in query formulation, many RDF analytics tasks may be expressed much easier with the addition of a other query types. For example, it will be important to include query forms based on graph simulation and other graph morphisms discussed in section II.B, which are not directly available in SPARQL. This will require providing additional query forms and/or relaxing the strict subgraph isomorphism semantics of the current query language.

C. Social Networks/Media and Web Mining

Graphs are employed heavily in online social networks/media (Facebook, Twitter, LinkedIn, etc.) and online retailers (e.g., Amazon). The reason for this popularity is that graphs offer a natural way of representing various kinds of relationships that are important for these applications. The friendship graph in Facebook, the follower graph in Twitter, endorsement graph in LinkedIn and product affinity graph in Amazon are some examples of social network and media graphs. The characteristics and properties of graphs vary significantly from one application to another. For example, the follower-following relationship graph in Twitter is a directed graph with various users as its vertices. A directed edge from vertex u to vertex v signifies that the user represented by u is a follower of user v. Note that most of the graphs in most online social networks and e-commerce companies are not only massive but also dynamic.

Social media companies are keen to derive business intelligence by running various kinds of analytics on these graphs. Computing various path related statistics is among the most common type of graph analytics. For example, social networking companies are interested in finding the most "influential" persons amongst their user-base. A popular metric for quantifying influence is the number of vertices within n hops of a given person. Thus, computing the exact/approximate number of n-hop neighbors of all or a subset of vertices is a common analytics task. Interestingly, there are two problems embedded in this task – computing the number of n-hop neighbors from scratch and maintaining the statistics as the graph undergoes changes. A variant of this problem is to estimate the influence as a weighted sum of n-hop neighbors (for instance, $\text{Influence}(v) = \sum_{j=1}^{n} \frac{\text{\# of j-hop vertices}}{j}$). In this equation, the contribution of a vertex to the influence score of another vertex diminishes as the distance between them increases. Other commonly employed path-related graph analytics tasks include: (a) computing shared n-hop neighbors between a given pair of vertices (used for suggesting friends), (b) computing one or more paths between a given pair of vertices (for illustrating how a suggested friend is related to a given user), and (c) computing graph centrality measures.

Graph pattern matching queries are also popular in social media applications. Besides the relatively controlled environments provided by graph databases and their cousins RDF triple-stores, there is a great deal of interest in graph pattern matching in social networks/media and mining the Web in general. As pointed out by [56], the data in such contexts are more noisy, so that exact matching, particularly of complex topology, may be less useful than inexact matching. For these types of applications, some form of graph simulation may be more useful than subgraph isomorphism.

For such applications, the use of graph homomorphism is discussed in [57]. Graph homomorphism is more flexible than subgraph isomorphism, as stated in Khan et al, 2013, "In contrast to strict one-to-one mapping as in traditional subgraph isomorphism tests, we consider a more general many-to-one subgraph matching function. Indeed, two query nodes may have the same match" [57]. Beyond that, the work reported in the paper also relaxes the strict label matching used in subgraph isomorphism [57]. Relaxations to both graph homomorphism and subgraph isomorphism are presented in [58]. The basic idea is similar to that of graph homeomorphism in which an edge in one graph is mapped/matched to a path in the other graph. A form of graph homeomorphism where edges are mapped to simple paths matching a regular expression is discussed in [59].

IV. Computational Models and Frameworks

For problems that have a few well-defined phases of computations, MapReduce style computations provide a means for highly parallel execution in large clusters with hundreds or more machines [60]. Some classical examples are word counting, statistics such as means and variances, and page rank. Frameworks, like Hadoop [61], put such capabilities within the hands of many programmers. Unlike the Message Passing Interface (MPI) [62], only a limited amount of specialized training is needed. The provision of fault-tolerant execution and a high-performance distributed file system further makes programming easier. Typically in Hadoop, data is read from the Hadoop Distributed File System (HDFS) by mappers based on a key values and written back to HDFS, and read by reducers, merged and again written back.

More complex algorithms, particularly iterative algorithms, are less amenable to the basic MapReduce style. Apache Storm [63] is similar to Hadoop, but focuses on more efficient stream processing, allowing data to be sent directly from one worker to another. Apache Spark [64] maintains intermediate results in main memory to reduce the number of slow page transfers to and from secondary storage and thereby, speed up computations. Hadoop 2 [65] adds the YARN resource manager, so that other programming models in addition to MapReduce can be supported.

An alternative to dividing computations into mappers and reducers for iterative algorithms, is to divide computations into a series of supersteps that involve receiving input messages, performing computations and sending output messages. Synchronization is system provided, since a task must wait for all tasks within a superstep to complete, before moving on the next superstep. This approach was made popular with Bulk Synchronous Parallel (BSP) [2]. In cases, where the number of superstep is not too large and work is well balanced among the tasks, BSP can be quite useful for implementing graph algorithms.

A special form of BSP, called vertex-centric, has become popular for big data graph analytics. In this programming model, each vertex of the graph is a computing unit which is conceptually a task in BSP. Each vertex initially knows only about its own status and its outgoing edges. Then, vertices can exchange messages through successive supersteps to learn about each other. When a vertex believes that it has accomplished its tasks, it votes to halt and goes to inactive mode. When all vertices become inactive the algorithm terminates. Several frameworks support this style of programming including Pregel [3], GPS [5] and Giraph [4].

Although the BSP computing model can be successfully used for graph, dual, strong, strict, tight, CAR-tight simulation, our work has found significant overhead in the synchronization. This is particularly true in the latter supersteps when many of the vertices have dropped out of the calculation. To obtain better performance, one may resort asynchronous frameworks such as GraphLab [66] and GRACE [67]. Unfortunately, this approach puts much of the burden for synchronization back on the programmer.

Future research may pursue two research directions. First, combining the ease of programming and high scalability potentials of BSP, with the performance advantages of asynchronous programming should be explored. Second, effective combination of multi-core parallel programming with cluster-based distributed programming, with minimal complexity overhead should be pursued as well.

V. Conclusions

With the increasing importance and growing size of graph stores and databases, recent research activity has increased substantially. Progress has also been substantial, but many challanges remain for future research.

References

[1] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 264–275, Sep 2010.

[2] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*. ACM, 2010, pp. 135–146.

[4] "Giraph website," http://giraph.apache.org/.

[5] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *SSDBM*. ACM, 2013, pp. 22:1–22:12.

[6] H. Jiang, H. Wang, P. S. Yu, and S. Zhou, "GString: A novel approach for efficient search in graph databases," in *ICDE*, 2007.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[8] S. Trisl and U. Leser, "Fast and practical indexing and querying of very large graphs," ser. SIGMOD. ACM, 2007, pp. 845–856.

[9] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *ICDE'06*. IEEE, 2006, pp. 75–75.

[10] P. R. Mullangi and L. Ramaswamy, "SCISSOR: scalable and efficient reachability query processing in time-evolving hierarchies," in *22nd ACM International Conference on Information and Knowledge Management, CIKM'13*, 2013.

[11] ——, "CoUPE: Continuous query processing engine for evolving graphs," in *2015 IEEE International Congress on Big Data*, 2015.

[12] A. O. Mendelzon and P. T. Wood, "Finding regular simple paths in graph databases," in *Proceedings of the 15th International Conference on Very Large Data Bases*, ser. VLDB '89. Morgan Kaufmann Publishers Inc., 1989, pp. 185–193.

[13] ——, "Finding regular simple paths in graph databases," *SIAM Journal on Computing*, vol. 24, no. 6, pp. 1235–1258, 1995.

[14] C. Barrett, R. Jacob, and M. Marathe, "Formal language constrained path problems," in *Algorithm TheorySWAT'98*. Springer, 1998, pp. 234–245.

[15] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[16] R. Bellman, "On a routing problem," DTIC Document, Tech. Rep., 1956.

[17] D. Z. Ghent, "On the all-pairs Euclidean short path problem," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, vol. 76. SIAM, 1995, p. 292.

[18] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 1995, pp. 453–462.

[19] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 310–321, 2011.

[20] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz, "A distributed vertex-centric approach for pattern matching in massive graphs," in *Big Data Conference*, Oct 2013, pp. 403–411.

[21] A. Fard, M. U. Nisar, J. A. Miller, and L. Ramaswamy, "Distributed and scalable graph pattern matching: Models and algorithms," *International Journal of Big Data (IJBD)*, vol. 1, no. 1, 2014.

[22] A. Fard, S. Manda, L. Ramaswamy, and J. A. Miller, "Effective caching techniques for accelerating pattern matching queries," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 491–499.

[23] P. Hell and J. Nešetřil, "On the complexity of h-coloring," *Journal of Combinatorial Theory, Series B*, vol. 48, no. 1, pp. 92–110, 1990.

[24] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[25] Y. Deville, G. Dooms, and S. Zampelli, "Combining two structured domains for modeling various graph matching problems," in *Recent Advances in Constraints*. Springer, 2008, pp. 76–90.

[26] R. Gei, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski, "GrGen: A fast SPO-based graph rewriting tool," in *Graph Transformations*. Springer, 2006, pp. 383–397.

[27] A. S. LaPaugh and R. L. Rivest, "The subgraph homeomorphism problem," in *Proceedings of the tenth annual ACM symposium on Theory of computing*. ACM, 1978, pp. 40–50.

[28] S. Fortune, J. Hopcroft, and J. Wyllie, "The directed subgraph homeomorphism problem," *Theoretical Computer Science*, vol. 10, no. 2, pp. 111–121, 1980.

[29] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of NP-completeness," *WH Freeman & Co., San Francisco*, 1979.

[30] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[31] W. Fan, X. Wang, and Y. Wu, "Answering graph pattern queries using views," in *ICDE*, March 2014, pp. 184–195.

[32] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller, "Towards efficient query processing on massive time-evolving graphs," in *CollaborateCom*, Oct. 2012, pp. 567–574.

[33] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18, no. 8, pp. 689–694, 1997.

[34] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of computer-aided molecular design*, vol. 16, no. 7, pp. 521–533, 2002.

[35] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[36] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "DualIso: An algorithm for subgraph pattern matching on very large labeled graphs," in *BigData Congress*. IEEE, 2014, pp. 498–505.

[37] W. S. Han, J. Lee, and J. H. Lee, "Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD*. ACM, 2013, pp. 337–348.

[38] V. Kann, "On the approximability of the maximum common subgraph problem," in *STACS 92*. Springer, 1992, pp. 375–388.

[39] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*. "O'Reilly Media, Inc.", 2013.

[40] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *Pervasive computing and applications (ICPCA)*. IEEE, 2011, pp. 363–366.

[41] J. Webber, "A programmatic introduction to NEO4j," in *Systems, Programming, and Applications: Software for Humanity*. ACM, 2012, pp. 217–218.

[42] C. Tesoriero, *Getting Started with OrientDB*. Packt Publishing Ltd, 2013.

[43] "TITAN Distributed Graph Database," http://thinkaurelius.github.io/titan/.

[44] F. Holzschuher and R. Peinl, "Performance of graph query languages," in *EDBT/ICDT*, 2013.

[45] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 405–418.

[46] S. Sakr and G. Al-Naymat, "Graph indexing and querying: a review," *International Journal of Web Information Systems*, vol. 6, no. 2, pp. 101–120, 2010.

[47] B. McBride, "Jena: Implementing the RDF model and syntax specification." in *SemWeb*, 2001.

[48] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "SPARQL basic graph pattern optimization using selectivity estimation," in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 595–604.

[49] T. Heath and C. Bizer, "Linked data: Evolving the web into a global data space," *Synthesis lectures on the semantic web: theory and technology*, vol. 1, no. 1, pp. 1–136, 2011.

[50] G. Gosal, K. J. Kochut, and N. Kannan, "Prokino: an ontology for integrative analysis of protein kinases in cancer," *PloS one*, vol. 6, no. 12, p. e28782, 2011.

[51] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[52] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao, "Processing SPARQL queries over linked data–a distributed graph-based approach," *arXiv preprint arXiv:1411.6763*, 2014.

[53] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gstore: a graph-based SPARQL query engine," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 23, no. 4, pp. 565–590, 2014.

[54] R. Angles and C. Gutierrez, "Querying RDF data from a graph database perspective," in *The Semantic Web: Research and Applications*. Springer, 2005, pp. 346–360.

[55] O. Corby and C. Faron-Zucker, "Implementation of SPARQL query language based on graph homomorphism," in *Conceptual Structures: Knowledge Architectures for Smart Applications*. Springer, 2007, pp. 472–475.

[56] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao, "Neighborhood based fast graph search in large networks," in *SIGMOD Conference*, 2011.

[57] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "Nema: Fast graph search with label similarity," *Proceedings of the VLDB Endowment*, vol. 6, no. 3, pp. 181–192, 2013.

[58] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph homomorphism revisited for graph matching," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1161–1172, 2010.

[59] P. Barceló Baeza, "Querying graph databases," in *Proceedings of the 32nd symposium on Principles of database systems*. ACM, 2013, pp. 175–188.

[60] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[61] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley, "Hadoop: A framework for running applications on large clusters built of commodity hardware," Wiki at http://lucene. apache. org/hadoop, Tech. Rep., 2005.

[62] W. Gropp, E. Lusk, N. Dossb, and A. Skjellumb, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[63] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," pp. 9–14, 2015.

[64] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–16.

[65] S. Radia and S. Srinivas, "Hadoop 2: What's new," pp. 12–15, 2014.

[66] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[67] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, 2013.