Graph Databases: Their Power and Limitations

Jaroslav Pokorný(✉)

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
pokorny@ksi.mff.cuni.cz

Abstract. Real world data offers a lot of possibilities to be represented as graphs. As a result we obtain undirected or directed graphs, multigraphs and hypergraphs, labelled or weighted graphs and their variants. A development of graph modelling brings also new approaches, e.g., considering constraints. Processing graphs in a database way can be done in many different ways. Some graphs can be represented as JSON or XML structures and processed by their native database tools. More generally, a graph database is specified as any storage system that provides index-free adjacency, i.e. an explicit graph structure. Graph database technology contains some technological features inherent to traditional databases, e.g. ACID properties and availability. Use cases of graph databases like Neo4j, OrientDB, InfiniteGraph, FlockDB, AllegroGraph, and others, document that graph databases are becoming a common means for any connected data. In Big Data era, important questions are connected with scalability for large graphs as well as scaling for read/write operations. For example, scaling graph data by distributing it in a network is much more difficult than scaling simpler data models and is still a work in progress. Still a challenge is pattern matching in graphs providing, in principle, an arbitrarily complex identity function. Mining complete frequent patterns from graph databases is also challenging since supporting operations are computationally costly. In this paper, we discuss recent advances and limitations in these areas as well as future directions.

Keywords: Graph database · Graph storage · Graph querying · Graph scalability · Big graphs

1 Introduction

A *graph database* is any storage system that uses graph structures with nodes and edges, to represent and store data. The most commonly used model of graphs in the context of graph databases is called a (*labelled*) *property graph model* [15]. The property graph contains connected *entities* (the *nodes*) which can hold any number of *properties* (*attributes*) expressed as key-value pairs. Nodes and edges can be tagged with *labels* representing their different roles in application domain. Some approaches refer to the label as the *type*. Labels may also serve to attach metadata—index or constraint information—to certain nodes.

DOI: 10.1007/978-3-319-24369-6_5

Relationships provide directed, semantically relevant connections (*edges*) between two nodes. A relationship always has a *direction*, a *start node*, and an *end node*. Like nodes, relationships can have any properties. Often, relationships have quantitative properties, such as weight, cost, distance, ratings or time interval. Properties make the nodes and edges more descriptive and practical in use. Both nodes and edges are defined by a *unique identifier*.

As relationships are stored efficiently, two nodes can share any number or relationships of different types without sacrificing performance. Note that although they are directed, relationships can always be navigated regardless of direction. In fact, the property graph model concerns data structure called in graph theory *labelled and directed attributed multigraphs*.

Sometimes we can meet hypergraphs in graph database software. A *hypergraph* is a generalization of the concept of a graph, in which the edges are substituted by *hyperedges*. If a regular edge connects two nodes of a graph, then a hyperedge connects an arbitrary set of nodes.

Considering graphs as a special structured data, an immediate idea which arises is, how to store and process graph data in a database way. For example, we can represent a graph by tables in a relational DBMS (RDBMS) and use sophisticated constructs of SQL or Datalog to express some graph queries. Some graphs can be represented as JSON or XML structures and processed by their native database tools. A more general native solution is offered by graph databases.

One of the more interesting upcoming growth areas is the use of graph databases and graph-based analytics on large, unstructured datasets. A special attention is devoted to so-called *Big Graphs*, e.g. Facebook with 1 Billion nodes and 140 Billion edges, requiring special storage and processing algorithms [12].

Graph databases are focused on:

- processing highly connected data,
- be flexible in usage data models behind graphs used,
- exceptional performances for local reads, by traversing the graph.

Graph databases are often included among NoSQL databases[1].

We should also mention lower tools for dealing with graphs. They include frameworks, such as Google's Pregel [8] - a system for large-scale graph processing on distributed cluster of commodity machines, and its more advanced variant Giraph[2] suitable for analytical purposes. They do not use a graph database for storage. These systems are particularly suitable for OLAP and offline graph analytics, i.e. they are optimized for scanning and processing Big Graphs in batch mode. Also the notion of a *Big Analytics* occurs in this context.

In traditional database terminology, we should distinguish a *Graph Database Management Systems* (GDBMS) and a *graph database*. Unfortunately, the latter substitutes often the former in practice. We will also follow this imprecise terminology.

[1] http://nosql-database.org/

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

There are a lot of papers about graph models, graph databases, e.g. [7], [12], [16], and theory and practise of graph queries, e.g. [4]. Now the most popular book is rather practically oriented work [15]. A performance comparison of some graph databases is presented, e.g., in [6], [9].

In this paper, a lot of examples from the graph database technology will be documented on the most popular graph database Neo4j[3], particularly in its version 2.2. In Section 2 we describe some basic technological features of graph databases. Section 3 presents an overview of graph databases categories as well as some their representatives, i.e., some commercial products. Section 4 presents some facts concerning the paper title and offers some research challenges. Finally, Section 5 concludes the paper.

2 Graph Database Technology

According to other DBMS, we can distinguish a number of basic components of graph database technology. They include graph storage, graph querying, scalability, and transaction processing. We will discuss them in the following subsections.

2.1 Graph Storage

An important feature of graph databases is that provide native processing capabilities, at least a property called *index-free adjacency*, meaning that every node is directly linked to its neighbour node. A database engine that utilizes index-free adjacency is one in which each node maintains direct references to its adjacent nodes; each node, therefore acts as an index of other nearby nodes, which is much cheaper than using global indexes. This is appropriate for local graph queries where we need one index lookup for starting node, and then we will traverse relationships by dereferencing physical pointers directly. In RDBMS we would probably need joining more tables trough foreign keys and, possibly, additional index lookups.

Obviously, more advanced indexes are used. For example, it is desirable to retrieve graphs quickly from a large database via *graph-based indices*, e.g. path-based methods. The approach used in [17] introduces so called *gIndex* using frequent substructures as the basic indexing features. Unfortunately, most of these techniques are usable only for small graphs.

Some graph stores offer a graph interface over non-native graph storage, such as a column store in the Virtuoso Universal Server[4] in application for RDF data. Often other DBMS is used as back-end storage. For example, the graph database FlockDB[5] stores graph data, but it is not optimized for graph-traversal operations. Instead, it is optimized for very large adjacency lists. FlockDB uses MySQL as the basic database storage system just for storing adjacency lists.

[3] http://www.neo4j.org/ (retrieved on 9.3.2015)

[4] http://virtuoso.openlinksw.com/ (retrieved on 9.3.2015)

[5] https://github.com/twitter/flockdb (retrieved on 9.3.2015)

2.2 Graph Querying

Query capabilities are fundamental for each DBMS. Those used in graph databases, of course, come from the associated graph model [2]. The simplest type of a query preferably uses the index-free adjacency. A node $v_k \in V$ is said to be at a *k-hop distance* from another node $v_0 \in V$, if there exists a shortest path from v_0 to v_k comprising of k edges. In practice, the basic queries are the most frequent. They include look for a node, look for the neighbours (1-hop), scan edges in several hops (layers), retrieve an attribute values, etc. Looking for a node based on its properties or through its identifier is called *point querying*.

Retrieving an edge by *id*, may not be a constant time operation. For example, Titan[6] will retrieve an adjacent node of the edge to be retrieved and then execute a node query to identify the edge. The former is constant time but the latter is potentially linear in the number of edges incident on the node with the same edge label.

As more complex queries we meet very often *subgraph* and *supergraph queries*. They belong to rather traditional queries based on exact matching. Other typical queries include *breadth-first/depth-first search*, *path* and *shortest path finding*, *finding cliques* or *dense subgraphs*, *finding strong connected components*, etc. Algorithms used for such complex queries need often iterative computation. This is not easy, e.g., with the MapReduce (MR) framework used usually in NoSQL databases for BigData processing. But the authors of [14] show for finding connected components that some efficient MR algorithms exist. In Big Graphs often *approximate matching* is needed. Allowing structural relaxation, then we talk about *structural similarity queries*.

Inspired by the SQL language, graph databases are often equipped by a declarative query language. Today, the most known graph declarative query language is Cypher working with Neo4j database. Cypher commands are loosely based on SQL syntax and are targeted at ad hoc queries of the graph data. A rather procedural graph language is the traversal language Gremlin[7].

The most distinctive output for a graph query is another graph, which is ordinarily a transformation, a selection or a projection of the original graph stored in the database. This implies that graph visualization is strongly tied to the graph querying [13].

2.3 Scalability

Sharding (or *graph partitioning*) is crucial to making graphs scale. Scaling graph data by distributing it across multiple machines is much more difficult than scaling the simpler data in other NoSQL databases, but it is possible. The reason is the very nature way the graph data is connected. When distributing a graph, we want to avoid having relationships that span machines as much as possible; this is called the *minimum point-cut problem*. But what looks like a good distribution one moment may no longer be optimal a few seconds later. Typically, graph partition problems fall under the category of NP-hard problems. Scaling is usually connected with three things:

[6] http://thinkaurelius.github.io/titan/ (retrieved on 9.3.2015)
[7] http://gremlindocs.com/

- scaling for large datasets,
- scaling for read performance,
- and scaling for write performance.

In practice, the former is most often discussed. Today, it is not problem in graph databases area. For example, Neo4j currently has an arbitrary upper limit on the size of the graph on the order of 10^{10}. This is enough to support most of real-world graphs, including a Neo4j deployment that has now more than half of Facebook's social graph in one Neo4j cluster.

Scaling for reads usually presents no problem. For example, Neo4j has historically focused on read performance. In master-slave regime read operations can be done locally on each slave. To improve scalability in highly concurrent workloads, Neo4j uses two levels of caching.

Scaling for writes can be accomplished by scaling vertically, but at some point, for very heavy write loads, it requires the ability to distribute the data across multiple machines. This is the real challenge. For example, Titan is a highly scalable OLTP graph database system optimized for thousands of users concurrently accessing and updating one Big Graph.

2.4 Transaction Processing

As in any other DBMS, there are three generic use cases for graphs:

- CRUD (create, read, update, delete) applications,
- query processing - reporting, data warehousing, and real-time analytics,
- batch mode analytics or data discovery.

Graph databases are often optimized and focused on one or more of these uses. Particularly, the first two uses are focused on transactions processing, i.e. OLTP databases. When dealing with many concurrent transactions, the nature of the graph data structure helps spread the transactional overhead across the graph. As the graph grows transactional conflicts typically falls away, i.e. extending the graph tends to the more throughputs. But not all graph databases are fully ACID. However, the variant based on the BASE properties often considered in the context of NoSQL databases is not too appropriate for graphs.

In general, distributed graph processing requires the application of appropriate partitioning and replication strategies such as to maximise the locality of the processing, i.e., minimise the need to ship data between different network nodes.

For example, Neo4j uses master-slave replication, i.e. one machine is designated as the master and the others as slaves. In Neo4j, all writes directed towards any machine are passed through the master, which in turn ships updates to the slaves when polled. If the master fails, the cluster automatically elects a new master.

Neo4j requires a quorum in order to serve write load. It means that a strict majority of the servers in the cluster need to be online in order for the cluster to accept write operations. Otherwise, the cluster will degrade into read-only operation until a quorum can be established. Emphasize, that today’s graph databases do not have the same level of write throughput as other types of NoSQL databases. This is a consequence of master-slave clustering and proper ACID transactions.

Some more complex architectures occur in the world of graph databases. Typically, a simple database is used to absorb load, and then feed the data into a graph database for refinement and analysis. The architecture Neo4j 2.2 contains even a bulk loader which operates at throughput of million records per second.

3 Categories of Graph Databases

There are a lot of graph databases. The well-maintained and structured Web site[8] included 20 products belonging among GDBMSs in 2011. The development of graph databases until 2011 is described in [1]. Wikipedia[9] describes 45 such tools. One half of them are ACID compliant.

We distinguish general purpose GDBMs, like Neo4j, InfiniteGraph[10], Sparksee[11], Titan, GraphBase[12], and Trinity[13], and special ones, e.g. the Web graph database InfoGrid[14] and FlockDB, or multimodel databases such as document-oriented databases enabling traversing between documents. For example, OrientDB[15] brings together the power of graphs and the flexibility of documents into one scalable database even with an SQL layer. HyperGraphDB[16] stores not only graphs but also hypergraph structures. All the graph information is stored in the form of key-value pairs.

An interesting question is which graph databases are most popular today. In June 2015, the web page DB-Engines Ranking of GDBMS[17] considering 17 graph products presented Neo4j, OrientDB, and Titan on the first three places. GDBMS Sparksee is on the 6th place.

In Section 3.1 we present two typical representatives of the general purpose category. From those special ones, more attention will be devoted to RDF triplestores in Section 3.2. The framework Pregel is explained in Section 3.3.

3.1 General Graph Purpose Databases - Examples

We describe shortly two successful graph GDBMSs - Neo4j and Sparksee - in some detail. In both GDBMSs a graph is a labelled directed attributed multigraph, where edges can be either directed or undirected.

[8] http://nosql-database.org/ (retrieved on 9.3.2015)
[9] http://en.wikipedia.org/wiki/Graph_database#cite_1 (retrieved on 12.6.2015)
[10] http://www.objectivity.com/infinitegraph#.U8O_yXnm9I0 (retrieved on 9.3.2015)
[11] http://sparsity-technologies.com/#sparksee (retrieved on 9.3.2015)
[12] http://graphbase.net/ (retrieved on 9.3.2015)
[13] http://research.microsoft.com/en-us/projects/trinity/ (retrieved on 9.3.2015)
[14] http://infogrid.org/trac/ (retrieved on 9.3.2015)
[15] http://www.orientechnologies.com/ (retrieved on 9.3.2015)
[16] http://www.hypergraphdb.org/index
[17] http://db-engines.com/en/ranking/graph+dbms (retrieved on 12.6.2015)

Example 1: Neo4j
Neo4j (now in version 2.2) is the world's leading GDBMS. It is an open-source, highly scalable, robust (fully ACID compliant) native graph database.

Neo4j stores data as nodes and relationships. Both nodes and relationships can hold properties in a key-value form. Values can be either a primitive or an array of one primitive type. Nodes are often used to represent entities, but depending on the domain the relationships may be used for that purpose as well. The nodes and edges have internal unique identifiers that can be used for the data search. Nodes cannot reference themselves directly [5]. The semantics can be expressed by adding directed relationships between nodes

Graph processing in Neo4j entails mostly random data access which can be unsuitable for Big Graphs. Graphs that cannot fit into main memory may require more disk accesses, which significantly influences graph processing. Big Graphs similarly to other Big Data collections must be partitioned over multiple machines to achieve scalable processing (see Section 2.3).

Example 2: Sparksee
In addition to the basic graph model, Sparksee also introduces the notion of a virtual edge that connects nodes having the same value for a given attribute. These edges are not materialized. A Sparksee graph is stored in a single file; values and identifiers are mapped by mapping functions into B+-trees. Bitmaps are used to store nodes and edges of a certain type.

The architecture of Sparksee includes the core, that manages and queries the graph structures, then an API layer to provide an application programming interface, and the higher layer applications, to extend the core capabilities and to visualize and browse the results. To speed up the different graph queries and other graph operations, Sparksee offers these index types:

- attributes,
- unique attributes,
- edges to index their neighbours, and
- indices on neighbours.

Sparksee implements a number of graph algorithms, e.g. shortest path, depth-first searching, finding strong connected components.

3.2 Triplestores

Some graph-oriented products are intended for special graph applications, mostly RDF data expressed in the form of *subject* (*S*) - *predicate* (*P*) – *object* (*O*). RDF graphs can be viewed as a special kind of a property graph. At the logical level, an RDF graph is then represented as one table. For example, AllegroGraph[18] works with RDF graphs. BrightStarDB[19], Bigdata[20] and SparkleDB[21] (formerly known as

[18] http://franz.com/agraph/ (retrieved on 9.3.2015)
[19] http://brightstardb.com/ (retrieved on 9.3.2015)

Meronymy) serve for similar purposes. These *triple stores* employ intelligent data management solutions which combine full text search with graph analytics and logical reasoning to produce deeper results. Sometimes, *quad stores* are used holding a fourth attribute - the *graph name* (*N*) corresponding normally with the namespace of the ontology. AllegroGraph deals even with quints (*S*, *P*, *O*, *N*, *ID*), the ID can be used to attach metadata to a triple.

Now, GraphDB™[22] is the world's leading RDF triple store that can perform semantic inferring at scale allowing users to create new semantic facts from existing facts. GraphDB™ is built on OWL (Ontology Web Language). It uses ontologies that allow the repository to automatically reason about the data. AlegroGraph also supports reasoning and ontology modelling.

However, existing triple store technologies are not yet suitable for storing truly large data sets efficiently. According to the W3C Wiki, AllegroGraph leads the largest deployment with loading and querying 1 Trillion triples. The load operation alone took about 338 hours.

We remind also that triple stores create only a subcategory of graph databases. Rather a hybrid solution is represented by Virtuoso Universal Server[23]. Its functionality covers not only processing RDF data, but also relations, XML, text, and others.

A list of requirements often required by customers considering a triple store is introduced in [10]:

- inferring,
- integration with text mining pipelines,
- scalability,
- extensibility,
- enterprise resilience,
- data integration and identity resolution,
- semantics in the cloud,
- semantic expertise.

3.3 Pregel and Giraph

Pregel and Giraph are systems for large-scale graph processing. They provide a fault-tolerant framework for the execution of graph algorithms in parallel over many machines. Giraph utilizes Apache MR framework implementation to process graphs.

A significant approach to the design, analysis and implementation of parallel algorithms, hardware and software in Pregel is now the *Bulk Synchronous Processing* (BSP) model. BSP offers architecture independence and very high performance of parallel algorithms on top of multiple computers connected by a communication network.

[20] http://www.systap.com/ (retrieved on 9.3.2015)
[21] https://www.sparkledb.net/ (retrieved on 9.3.2015)
[22] http://www.ontotext.com/products/ontotext-graphdb/ (retrieved on 9.3.2015)
[23] http://virtuoso.openlinksw.com/ (retrieved on 9.3.2015)

BSP is a powerful generalization of MR. A subclass of BSP algorithms can be efficiently implemented in MR [11]. BSP is superfast on standard commodity hardware, orders of magnitude faster than the MR alone. It is an easy parallel programming model to learn, it has a cost model that makes it simple to design, analyse, and optimize massively parallel algorithms. It can be considered as a strong candidate to be the programming model for parallel computing and Big Data in the next years. For example, Google is already moving in its internal infrastructure from MR to BSP/Pregel.

4 Limitations of Graphs Databases

Despite of the long-term research and practice in this area, there are many important and hard problems that remain open in graph data management. They have influence on functionality restrictions of graph databases (Section 4.1). Others are specifically related to Big Analytics (Section 4.2). Challenges concerning some specific problems of graph database technology are summarized in Section 4.3.

4.1 Functionality Restrictions

Declarative querying: Most commercial graph databases cannot be queried using a declarative language. Only few vendors offer a declarative query interface. This implies also a lack of query optimization abilities.

Data partitioning: Most graph databases do not include the functionality to partition and distribute data in a computer network. This is essential for supporting horizontal scalability, too. It is difficult to partition a graph in a way that would not result in most queries having to access multiple partitions.

Vectored operations: They support a procedure which sequentially writes data from multiple buffers to a single data stream or reads data from a data stream to multiple buffers. Horizontally scaled NoSQL databases support this type of data access. It seems that it is not the case in graph databases today.

Model restrictions: Possibilities of data schema and constraints definitions are restricted in graph databases. Therefore, data inconsistencies can quickly reduce their usefulness. Often the graph model itself is restricted. Let us recall, e.g., Neo4j nodes cannot reference themselves directly. There might be real world cases where self-reference is required.

Querying restrictions: For example, FlockDB overcomes the difficulty of horizontal scaling the graph by limiting the complexity of graph traversal. In particular, FlockDB does not allow multi-hop graph walks, so it cannot do a full "transitive closure". However, FlockDB enables very fast and scalable processing of 1-hop queries.

4.2 Big Analytics Requirements

Graph extraction: A question is how to efficiently extract a graph, or a collection of graphs, from non-graph data stores. Most graph analytics systems assume that the graph is provided explicitly. However, in many cases, the graph may have to be constructed by joining and combining information from different resources which are not necessarily graphical. Even if the data is stored in a graph database, often we only need to load a set of subgraphs of that database graph for further analysis.

High cost of some queries: Most real-world graphs are highly dynamic and often generate large volumes of data at a very rapid rate. One challenge here is how to store the historical trace compactly while still enabling efficient execution of point queries and global or neighbourhood-centric analysis tasks. Key differences from temporal DBMSs developed in the past are the scale of data, focus on distributed and in-memory environments, and the need to support global analytics. The last task usually requires loading entire historical snapshots into memory.

Real time processing: As noted, graph data discovery takes place essentially in batch environments, e.g., in Giraph. Some products aimed at data discovery and complex analytics that will operate in real-time. An example is uRIKA[24] – a Big Data Appliance for Graph Analytics. It uses in-memory technology and multithreaded processor to support non-batch operations on RDF triples.

Graph algorithms: More complex graph algorithms are needed in practice. The ideal graph database should understand analytic queries that go beyond k-hop queries for small k. Authors of [9] did a performance comparison of 12 open source graph databases using four fundamental graph algorithms (e.g. simple source shortest path problem and Page Rank) on networks containing up to 256 million edges. Surprisingly, the most popular graph databases have reached the worst results in these tests. Current graph databases (like relational databases) tend to prioritize low latency query execution over high-throughput data analytics.

Parallelisation: In the context of Big Graphs there is a need for parallelisation of graph data processing algorithms when the data is too big to handle on one server. There is a need to understand the performance impact on graph data processing algorithms when the data does not all fit into the memory available and to design algorithms explicitly for these scenarios.

Heterogeneous and uncertain graph data: There is a need to find automated methods of handling the heterogeneity, incompleteness and inconsistency between different Big Graph data sets that need to be semantically integrated in order to be effectively queried or analysed.

---

[24] http://www.cray.com/products/analytics/urika-gd

4.3 Other Challenges

Other challenges in the development of graph databases include:

Design of graph databases: Similarly to traditional databases, some attempts to develop design models and tools occur in last time. In [3], the authors propose a model-driven, system-independent methodology for the design of graph databases starting from ER-model conceptual schema.

Need for a benchmark: Querying graph data can significantly depend on graph properties. The benchmarks built, e.g., for RDF data are mostly focused on scaling and not on querying. Also benchmarks covering a variety of graph analysis tasks would help towards evaluating and comparing the expressive power and the performance of different graph databases and frameworks.

Developing heuristics for some hard graph problems: For example, partitioning of large-scale dynamic graph data for efficient distributed processing belongs among these problems, given that the classical graph partitioning problem is NP-hard.

Graph pattern matching: New semantics and algorithms for graph pattern matching over distributed graphs are in development, given that the classical subgraph isomorphism problem is NP-complete.

Compressing graphs: Compressing graphs for matching without decompression is possible. Combining parallelism with compressing or partitioning is also very interesting.

Integration of graph data: In the context of Big Data, query formulation and evaluation techniques to assist users querying heterogeneous graph data are needed.

Visualization: Improvement of human-data interaction is fundamental, particularly a visualization of large-scale graph data, and of query and analysis results.

Graph streams processing: Developing algorithms for processing Big Graph data streams with goal to compute properties of a graph without storing the entire graph.

5 Conclusions

Graph databases are becoming mainstream. As data becomes connected in a more complicated way and as the technology of graph databases matures, their use will increase. New application areas occur, e.g. the Internet of Things, or rather Internet of Connected Things. Comparing to traditional RDBMS, there is a difficulty for potential users to identify the particular types of use case for which each product is most suitable. Performance varies greatly across different GDBMSs depending upon the size of the graph and how well-optimized a given tool is for a particular task. It seems that especially for Big Graphs and Big Analytics a lot of previous results and designs will have to be re-considered and re-thought in next research and development.

Acknowledgments. This paper was supported by the GAČR grant No. P103/13/08195S.

References

1. Angeles, R.: A comparison of current graph database models. In: Proc. of the 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012, pp. 171–177. IEEE Computer Society, Washington (2012)
2. Angeles, R., Gutierrez, C.: Survey of Graph Database Models. ACM Computing Surveys **40**(1), Article 1 (2008)
3. De Virgilio, R., Maccioni, A., Torlone, R.: Model-driven design of graph databases. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 172–185. Springer, Heidelberg (2014)
4. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In: Proc. of the Joint EDBT/ICDT 2013 Workshops, EDBT 2013, pp. 195–204. ACM, NY (2013)
5. Hurwitz, J., Nugent, A., Halper, F., Kaufman, M.: Big Data for Dummies. John Wiley & Sons, Inc. (2013)
6. Kolomičenko, V., Svoboda, M., Holubová – Mlýnková, I.: Experimental comparison of graph databases. In: Proc. of International Conference on Information Integration and Web-based Applications & Services, p. 115. ACM, NY (2013)
7. Larriba-Pey, J.L., Martínez-Bazán, N., Domínguez-Sal, D.: Introduction to graph databases. In: Koubarakis, M., Stamou, G., Stoilos, G., Horrocks, I., Kolaitis, P., Lausen, G., Weikum, G. (eds.) Reasoning Web. LNCS, vol. 8714, pp. 171–194. Springer, Heidelberg (2014)
8. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proc. of SIGMOD 2010 Int. Conf. on Management of data, pp. 135–146. ACM, NY (2010)
9. McColl, R., Ediger, D., Poovey, J., Campbell, D., Bader, D.A.: A performance evaluation of open source graph databases. In: Proc. of PPAA 2014, pp. 11–18. ACM, NY (2014)
10. Ontotext: The Truth about Triplestores. Ontotext (2014)
11. Pace, M.F.: BSP vs MapReduce. Procedia Computer Science **9**, 246–255 (2012)
12. Pallavi, M., Saxena, A.: Review: Graph Databases. Int. Journal of Advanced Research in Computer Science and Software Engineering **4**(5), 195–200 (2014)
13. Pokorny, J., Snášel, V.: Big graph storage, processing and visualization. In: Pitas, I. (ed.) Graph-Based Social Media Analysis, pp. 403–430. Chapman and Hall/CRC (in print, 2015)
14. Rastogi, V., Machanavajjhala, A., Chitnis, L., Sarma, A.D.: Finding Connected Components on Map-reduce in Logarithmic Rounds. CoRR, abs/1203.5387. ACM (2012)
15. Robinson, I., Webber, J., Eifrém, E.: Graph Databases. O'Reilly Media (2013)
16. Shimpi, D., Chaudhari, S.: An overview of Graph Databases. IJCA Proceedings on International Conference on Recent Trends in Information Technology and Computer Science 2012 ICRTITCS(3), 16–22 (2013)
17. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure –based approach. In: Proc. of SIGMOD 2004 Int. Conf. on Management of Data, pp. 335–346. ACM, NY (2004)