Experimental Comparison of Graph Databases*

Vojtěch Kolomičenko
Department of Software Engineering,
Charles University in Prague, Czech Republic
kolomicenko@gmail.com

Martin Svoboda
Department of Software Engineering,
Charles University in Prague, Czech Republic
svoboda@ksi.mff.cuni.cz

Irena Holubová (Mlýnková)
Department of Software Engineering,
Charles University in Prague, Czech Republic
holubova@ksi.mff.cuni.cz

ABSTRACT

In the recent years a new type of NoSQL databases, called *graph databases* (GDBs), has gained significant popularity due to the increasing need of processing and storing data in the form of a graph. The objective of this paper is a research on possibilities and limitations of GDBs and conducting an experimental comparison of selected GDB implementations. For this purpose the requirements of a universal GDB benchmark have been formulated and an extensible benchmarking tool, called *BlueBench*, has been developed.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and Text Editing – languages, document management

General Terms

Measurement, Algorithms

Keywords

graph databases, NoSQL databases, benchmarking, experimental comparison

1. INTRODUCTION

Recently, there has been a huge increase of importance to store and analyze data in the form of a graph. Be it social networks, Web graphs, recommendation systems or biological networks, these graphs have at least two things in common – they are highly interconnected and huge, and so is the complexity of algorithms used for their processing and analysis of their characteristics. Unlike other types of data, networks contain lots of information in the way how particular objects are connected; in other words, the *relations* between the objects.

*This work was supported by the GAUK grant no. 4105/2011 and SVV grant no. SVV-2013-267312.

Graph database management systems (GDBs) have emerged to fill the gap in the market by providing solutions to the problem of storing and working with large graphs. A variety of GDB implementations has been appearing, and most of the systems are usually accompanied with biased proprietary tests claiming that their performance is somewhat superior. Nevertheless, a large scale unbiased benchmark comprising the most substantial graph database functionality and putting a higher number of GDBs into a comparison has not been conducted yet. We believe that this is most likely to be the reason of the versatility of interfaces of particular GDB engines, their configuration specifics and different capabilities.

The objective of this paper is to summarize the background behind the graph databases family and describe a chosen set of GDB representatives. Next, a benchmarking suite, called *BlueBench*, is created and the performance of a selected set of GDBs experimentally evaluated. The tested databases are carefully chosen so that *BlueBench* compares not only the most popular GDB engines, but less conventional representatives as well. Hence, we have implemented a simple wrapper around a selected relational database, which provides the basic functionality expected from GDBs.

The rest of the paper is structured as follows: In Section 2 we describe the existing graph database systems. In Section 3 we overview related work. In Section 4 we describe the architecture of *BlueBench*. In Section 5 we provide results of the tests. And we conclude in Section 6.

2. GRAPH DATABASE SYSTEMS

In recent years a new family of databases known as *NoSQL* has gained lots of popularity, particularly because of the need of storing and effectively retrieving huge volumes of data. Graph databases belong into this family despite their slightly higher complexity. In contrast to the other NoSQL implementations, in a graph database the relations between the objects are of primary importance. Graph databases support a *graph model* which allows for a direct persistent storing of the particular objects in the database together with the relations between them. In addition, a GDB should provide an access to query methods that not only deal with the stored objects, but also with the graph structure itself. The best known example of such an operation is *traversal*, which in its most simple form can be used to obtain the neighbors of a specified object, that is, the objects that the specified object is directly related to.

The advantage of having a direct access to heavily interconnected data comes at the cost of very complicated partitioning of the database. To be able to efficiently partition the graph data onto a number of separate machines, the graph must be reorganized so that the smallest possible amount of relations crosses the boundaries of a single machine in the cluster. Algorithms for performing this operation and then keeping the database in the same state after further data changes have not been successfully put into practice; therefore, the problem of efficient partitioning of graph data remains open [13].

2.1 *TinkerPop* Stack

TinkerPop[1] has been developing a stack of applications designed to simplify and unite the way of working with graph databases engines. Most of the software is created in Java, which is also meant to be the primary accessing language; nevertheless, interfaces in other languages are provided as well. Majority of the best known GDBs support the *TinkerPop*. A brief description of the most significant parts follows.

Blueprints[2] is a set of interfaces enabling applications to take advantage of the complete functionality provided by *TinkerPop*.The interfaces are designed to be very transparent and offer mostly elementary methods which can, however, be combined to build much more complicated queries. *Blueprints* works with the property graph model; consequently, if a GDB exposes a stronger model, that extra functionality can only be utilized via its own native methods.

Rexster[3] is a configurable graph server which exposes any *Blueprints*-compliant graph database via the REST HTTP protocol. A binary protocol, called *RexPro*, is also supported for better performance on a single machine. *Dog House* is a browser-based user interface also offered by *Rexster*.

TinkerGraph[4] is a lightweight in-memory implementation of the property graph model, and is part of the *Blueprints* package. *TinkerGraph* does not have any support for transactions; it primarily serves only as a reference implementation for *Blueprints* and its graph model. However, indexing of elements based on their properties is implemented using a *HashMap* collection.

Finally, *Gremlin*[5] is graph query language designed for iterative traversing of the graph which is controlled by the application in a procedural manner. In addition, it supports graph manipulation and all other functionality from *Blueprints*, because it directly utilizes its methods.

2.2 *DEX*

DEX[6] is a closed-source commercial graph database written in Java with a C++ core. It was first released in 2007 and its goal is to be a high-performance and scalable solution for working with very large graphs [9]. *DEX* is currently the third most popular graph DBMS today [10].

The graph model this database exposes is called "Labeled and directed attributed multi-graph" because the edges can be either directed or undirected, all elements belong to arbitrary types and there can exist more than one edge between two vertices. *DEX* provides a native API for Java, C++ and .NET platforms; in addition, it is compliant with *Blueprints* interface and the database server can be remotely accessed via REST methods. The database can therefore be used in a variety of applications.

DEX uses its own bitmap-based highly compressed native persistent storage which should be very effective and have a small memory footprint thanks to the light and independent data structures [11]. The fundamental data structure used is a "Link" which is a combination of a map and a number of bitmaps that enables fast conversion between an object identifier and its value and vice versa. When object's value is needed, the map is used; whereas the bitmap can return all object identifiers associated with a specified value. The whole graph can then be stored as the following combination of *links*: one *link* for each element type, one *link* for each element attribute, two *links* for each edge type.

The structure used for the maps is a B+ tree[7], the values are stored as UTF-8 strings and the element identifiers are 37 bit unsigned integers. The identifiers are grouped and compressed in order to significantly reduce the size of the structure [11]. Attribute values can be indexed when required by the application to accelerate the speed of the element scan based on properties; this is ensured by adding the index on the values in the appropriate attribute *link*.

DEX offers a partial ACID transaction support, called "aCiD", because the isolation and atomicity cannot be always guaranteed [9]. The transaction concurrency model is based on the N-readers 1-writer model.

2.3 *InfiniteGraph*

InfiniteGraph[8] is another commercial graph database written in Java with a C++ core. It was initially released in 2009. The priorities of this graph database system lie in scalability, distributed approach, parallel processing and graph partitioning [12].

InfiniteGraph uses a "Labeled directed multigraph" model which also includes bidirectional edges. The database can be exposed through an API in various languages (Java, C++, C#, Python) and provides a distinct Traverser interface which can also be used for distributed traversal queries. This is accompanied with *Blueprints* support; as a consequence, the database can also be accessed using *Rexster* or *Gremlin* facilities.

The database system conforms to a full ACID model. In addition, it also offers relaxing of the consistency for accelerated temporary ingests of data [14]. The graph database uses Objectivity/DB as a backend; and thus adopts its distributed properties allowing scaling and replication.

2.4 *Neo4j*

Neo4j[9] is written completely in Java, it is an open-source project and its first release dates back to 2007. *Neo4j* is a very well known graph database system, in fact currently the most popular one by a great margin [12].

Neo4j features a graph model called "Property graph" which is in reality very similar to the models the afore mentioned databases offer. The native API is exposed through a whole range of different languages, e.g. Java, Python, Ruby, JavaScript, PHP, .NET, etc. There also are many kinds of ways how to query the database, for example via

[1]http://www.tinkerpop.com/.
[2]https://github.com/tinkerpop/blueprints/wiki
[3]https://github.com/tinkerpop/rexster/wiki
[4]https://github.com/tinkerpop/blueprints/wiki/TinkerGraph
[5]https://github.com/tinkerpop/gremlin/wiki
[6]http://www.sparsity-technologies.com/dex
[7]http://www.seanster.com/BplusTree/BplusTree.html
[8]http://www.objectivity.com/InfiniteGraph
[9]http://www.neo4j.org/

the native Traverser API, or using *SPARQL* or Cypher query languages[10]. The database system also implements the *Blueprints* interface and a native REST interface to further expand the versatility of ways how to communicate with the database. *Neo4j* also supports custom indexes on elements' properties using external indexing engines, currently employing *Apache Lucene*[11] as the default engine.

Persistency of *Neo4j* database is provided by a native graph storage back-end mainly using adjacency lists architecture. The three main components of the graph (i.e. vertices, edges and element properties) are stored in three separate *store files*. Vertices are stored with a pointer to their first edge and the first property. Properties store is a group of linked lists, where there is one linked list per vertex. Finally, edges are stored as two double-linked lists (one for each endpoint of the edge) along with the edge's type and pointers to the current edge's endpoints and first property.

Full ACID transactions is supported by *Neo4j* achieved by having in-memory transaction logs and a lock manager applying locks on any database objects altered during the transaction.

Neo4j database can be distributed on a cloud of servers using the Master/Slave replication model and utilizing Apache Zookeeper to manage the process. As a consequence, the database consistency property is loosen to eventual consistency while the rest of ACID characteristics stays the same. However, the database distribution solution applies only to replication of the data which helps the system to handle higher read load; at the moment *Neo4j* does not support sharding of the graph [15].

2.5 *OrientDB*

OrientDB[12] is a wide-range NoSQL solution providing a key/value store, document-oriented database and a graph database. It is written solely in Java and has become very popular shortly after its initial release in 2010 [12].

The GDB system adopts "Property graph" as its graph model and provides API in many programming languages. Many approaches can also be used to query the database, namely the native *Traverser* API for Java embedded solutions, REST interface for remote access or an SQL-like query language which is called *Extended SQL* and has been developed alongside *OrientDB*. The database is also *Blueprints* compliant and thus several other ways of accessing the engine are available. In addition, unlike the majority of other GDB systems, *OrientDB* provides support for basic security management which is based on the users and roles model.

OrientDB's graph database is built atop the document database for which a native model of a persistent storage was created. It uses its own data structure also for indexing properties of elements. It is an innovative algorithm called *MVRB-Tree* which is a combination of a B+ tree and a Red-Black tree[13] and which consumes only half as much memory as a Red-Black tree while keeping the speed of balancing the structure after an insert or update [16]. The caching mechanism is also quite sophisticated and is spread over several levels; from caches exclusive to a single application thread to caches on the physical storage level.

Transactions in *OrientDB* have all ACID properties which are ensured by utilizing the *MVCC*[14] method. Thanks to this approach, there can be concurrent reads and writes on the same records without the need to lock the database; however, all the records must carry their version so that the age of the record can be checked on a transaction commit [17]. The resolution of any transaction conflicts is left up to the application.

This database system can also be distributed across a number of servers in a cluster, using *Hazelcast*[15] for the clustering management. Amongst the nodes in the cluster the *Multi Master*[16] replication method is supported. Therefore, all of the servers are allowed to perform read and write operations on their own replicas of the database while notifying the rest of the nodes in the cluster. Such notification can be synchronous or asynchronous, the latter being faster, however guaranteeing only *eventual consistency* property.

2.6 *Titan*

Titan[17] is one of the newest graph database systems as it has emerged very recently, in 2012. Similarly to other GDB engines, *Titan* is written in Java and it is an open-source project. The authors claim that it is a highly scalable solution specialized on handling graphs distributed across a cluster of many servers [19].

The "Property graph" model used by *Titan* can be accessed via two provided interfaces. *Titan* can either be run as a standalone server which should be queried by REST methods, or be embedded in the user's Java application for which case it supports the *Blueprints* interface. Compliance to *Blueprints* also naturally opens up the *Rexster* and *Gremlin* possibilities. Furthermore, to index elements by their properties users can choose between two external indexing engines depending on the application needs. Specifically, the engines are *Apache Lucene* and *ElasticSearch*[18] which can be used to perform effective full-text searches or numeric-range and geo searches respectively [20].

As a backend, *Titan* supports three particular key-value or column-family databases which have a contrasting influence on the transaction and scalability properties of the resulting system. Therefore, it is possible to choose between *Cassandra*[19], *HBase*[20] and *Berkeley DB*[21] depending on the application's business requirements.

In order to store the graph to a persistent storage *Titan* uses adjacency lists data structure. Namely, one column family in the underlying backend represents one vertex's adjacency list with a row key equal to the vertex's identifier. Moreover, each element property and edge is stored in a single column with the edge direction, label, or key saved as the column prefix [22]. An index can be created on any vertex property so that the vertex can be retrieved faster when the property's key-value pair is provided.

[10]`http://docs.neo4j.org/chunked/stable/cypher-query-lang.html`

[11]`http://lucene.apache.org/core/`

[12]`http://www.orientdb.org/`

[13]`http://cs.wellesley.edu/~cs231/fall01/red-black.pdf`

[14]`http://clojure.org/refs`

[15]`http://www.hazelcast.com/`

[16]Multi server architecture where all servers have an equal role and are allowed to modify the stored data.

[17]`http://thinkaurelius.github.com/titan/`

[18]`http://www.elasticsearch.org/`

[19]`http://cassandra.apache.org/`

[20]`http://hbase.apache.org/`

[21]`http://www.oracle.com/technetwork/products/berkeleydb`

3. RELATED WORK

Naturally, the idea of comparing and benchmarking of graph databases is not completely new. In this section we describe and compare various existing works.

3.1 *GraphDB-Bench*

GraphDB-Bench[22] is an extensible graph database benchmarking framework built on the *TinkerPop* stack, heavily using *Blueprints*, and *Gremlin* in order to be easily executed on any *Blueprints* enabled graph database system. It provides an interface for user-defined graph database testing operations, automatically measures their execution time and logs all results. *GraphDB-Bench* also contains scripts for automatic generation of synthetic graphs and then plotting benchmark results, using the *iGraph*[23] library. In case the users want to provide own graph data, thanks to *Blueprints*, input files in the *GraphML*[24] notation are accepted.

In the currently accessible package, the framework comes out with a sample benchmark (and with its results) comprising breadth first search traversals, loading graph data from a GraphML file, performance of writing into an index, and reading from an index. All the tests are run against a data set of a varying size, from a tiny graph of a thousand vertices to a large graph consisting of a million of vertices. Nonetheless, the benchmark itself claims the results should not be taken too seriously but as the framework's proof of concept.

TinkerGraph, *OrientDB* and *Neo4j* were tested in this benchmark. *TinkerGraph*, which is only an in-memory implementation of the *Blueprints* interface, was much faster in almost all tests. Apart from that, *Neo4j* surpassed *OrientDB* in graph loading and reading from the index. On the other hand, *OrientDB* was faster during index writes. Interesting situation arrived during the test measuring the performance of traversals. *Neo4j* was faster in traversals when the depth was up to four and *OrientDB* began to dominate when the traversal depth was higher than five.

3.2 *HPC-SGAB*

In paper [1] the authors describe and implement guidelines from the *High Performance Computing Scalable Graph Analysis Benchmark* (*HPC-SGAB*) [2] which is normally used to assess the performance of graph algorithm libraries.

The benchmark consists of four testing scenarios where the execution time is measured – loading the graph into the database, query against the database which ignores any relations, a traversing query, and a query which traverses the whole graph. The graph data used in this benchmark do not come from a real environment but are artificially created using the *R-MAT generator* [3].

Four databases were selected for the benchmark, namely *Neo4j*, *DEX*, *Jena*[25] and *HypergraphDB*[26]. Regarding the results, *DEX* was shown to be the best performing of the analyzed engines in almost all of the tests. Apart from that, *Neo4j* performed much better than *Jena* during traversing operations, and *Jena* surpassed *Neo4j* in the first two tests (loading the graph and scanning the edges). *HypergraphDB* could not be assessed on most of the operations because it was not possible to load the graph into the database for majority of the initial data set sizes in time.

[22]https://github.com/tinkerpop/tinkubator/tree/master/graphdb-bench

[23]http://igraph.sourceforge.net/

[24]http://graphml.graphdrawing.org/

[25]http://jena.apache.org/

[26]http://www.hypergraphdb.org/

3.3 Benchmarking Traversal Operations

Paper [4] provides a thorough description of a benchmark created by the same authors. The benchmark is implemented in Java, uses *Blueprints* API, and most importantly, heavily utilizes *GraphDB-Bench*[27] framework. Graph data used for the testing are synthesized via the *LFR-Benchmark* [5] generator which produces graphs with properties similar to those of the real world graphs. It generates network data sets with power law distribution property and with artificial communities implanted within the networks [4].

The following tests are described in the paper and their execution time is measured by the benchmark: loading the graph into the database, computing the local clustering coefficient of ten thousand randomly chosen vertices, performing breadth first search traversals, and running an algorithm for detection of connected components.

The benchmark is run against these GDB systems: *Neo4j*, *DEX*, *OrientDB*, *NativeSail* and *SGDB* (which is their own research prototype of a GDB). The results of the benchmark are preliminary and not very clear, but *SGDB* seemed to perform significantly better than the rest of the systems. However, *SGDB* is only a research prototype with some concurrency issues and not a ready product [7]. Amongst the rest of the engines, *DEX* and *Neo4j* showed to be more efficient than *OrientDB* and *NativeSail* for most of the tested operations.

3.4 Graph vs. Relational Database

In paper [6] a comparison between a chosen graph database and a more traditional relational database is described in detail. Not only do the authors compare the systems objectively using a benchmark, they also provide a subjective view on other aspects of the systems, such as quality of documentation, support, level of security, ease of programming etc. For the data generation of the benchmark, they come up with their own generator of random directed acyclic graphs. Those graphs, however, do not seem to be satisfying any of the real world graphs' properties (e.g., power law distribution property, small diameter, high clustering coefficient).

The following operations were tested and measured in this benchmark. They can be divided into two groups, traversal and non-traversal queries: finding all vertices with the incoming and outgoing degree being equal to zero, traverse the graph to a depth of 4 from a single starting point, the same as the previous test with the depth of 128, count all nodes having their random integer payload equal to/lower than some value, and count all nodes whose string payload contains some search string.

For the non-traversal part of the queries, a few different types of payloads were tried one by one. The authors also considered various sizes of the random graph to point out how the database systems scale. In addition, they measured the physical size of the DB once all data were loaded; in other words they compared the systems in terms of disk usage efficiency. On the other hand, the loading times of the data into the DB were not covered at all.

Neo4j was selected as a representative of graph databases

[27]http://code.google.com/p/graphdb-bench/

and *MySQL*[28] (accessed via JDBC) was chosen for relational database systems. Regarding the results, *Neo4j* clearly outperformed *MySQL* in the traversal tests. Performance results of the non-traversing queries were mixed because *MySQL* was faster with integer payload handling, in contrast to string payloads where *Neo4j* was faster in most cases.

4. *BLUEBENCH* ARCHITECTURE

BlueBench is implemented in Java and the database engines are run in the same JVM as the benchmark itself if possible. In addition, *BlueBench* heavily utilizes *Blueprints* as the main but not only interface to access the tested GDB systems. The latest version of *Blueprints* at the time of writing the benchmark suite was 2.2.0.

BlueBench is divided into three different standalone benchmarks which, however, share most of the input data and most of the tested operations. That way the individual operations in the particular benchmarks can be compared across benchmarks while the consistency and fairness is guaranteed. Each benchmark consists of operations which have clearly defined boundaries and execution time of which is always measured. Each operation gets the *Blueprints* graph (hiding the actual database system implementation) and when it is done, it must leave the database in a consistent state. Every operation can receive any number of arguments which are prepared by the benchmark itself. The set of operations has been selected to reflect most of the requirements from a graph database system and to incorporate operations from very complicated ones to very trivial one:

1. *DeleteGraph.* This operation completely clears the database along with the data and indexes from the disk to ensure that the database is in its initial state.

2. *CreateIndexes.* Indexing of vertices by selected property key to be performed before any elements are loaded into the database.

3. *LoadGraphML.* This operation inserts elements into the database according to a GraphML input file.

4. *Dijkstra.* The shortest paths between a randomly selected vertex and all other reachable vertices in the graph are computed.

5. *Traversal.* A simple breadth first search traversal to the depth of five from a given starting vertex is conducted.

6. *TraversalNative.* As opposed to the *Traversal* operation, the native API of the underlying GDB engine is used instead of the standard *Blueprints* API whenever possible.

7. *ShortestPath.* This operation computes the shortest path between two vertices. An important difference between *Traversal* and *ShortestPath* operations is that the former one uses *getVertices() Blueprints* method whereas the latter utilizes *getEdges()*, whose implementation and thus performance might be significantly different.

[28]`http://www.mysql.com/`

8. *ShortestPathLabeled.* The same as the previous operation; however, only edges with a certain label are considered.

9. *FindNeighbors.* This is the most primitive traversal operation – it only finds the closest neighbors of a randomly selected vertex.

10. *FindEdgesByProperty.* This is the first of the non-traversing operations and it browses the graph database while looking for all edges with a certain property equal to some string value. The value is not random but it is chosen from a dictionary of all the values that the edges could posses.

11. *FindVerticesByProperty.* Precisely the same as the previous operation, with the difference that the search is conducted for vertices instead of edges.

12. *UpdateProperties.* This operation tests how efficiently the database engine updates properties of elements. A predefined set of vertices is firstly selected, then this set is divided into pairs, and finally, every two vertices in each pair swap their properties.

13. *RemoveVertices.* By contrast, this operation tests the performance in deleting vertices.

The three benchmarks that are a part of *BlueBench* constitute of individual operations as it is depicted in Table 1. Each benchmark focuses on measuring of different aspects of the GDB system's performance and the choice of operations reflects it. As most of the operations are shared amongst the benchmarks, eventually the results could be compared *BlueBench* wide.

Operation	Labeled	Property	Indexed
DeleteGraph	Yes	Yes	Yes
CreateIndexes			Yes
LoadGraphML	Yes	Yes	Yes
Dijkstra		Yes	Yes
TraversalNative	Yes	Yes	Yes
Traversal	Yes	Yes	Yes
ShortestPath	Yes	Yes	Yes
ShortestPathLabeled	Yes	Yes	Yes
FindNeighbors	Yes	Yes	Yes
FindEdgesByProperty		Yes	Yes
FindVerticesByProperty		Yes	Yes
UpdateProperties		Yes	Yes
RemoveVertices	Yes	Yes	Yes

Table 1: Operations performed in the benchmarks

Labeled Graph Benchmark.

As the name of the benchmark suggests, an input graph with only labels (i.e. without properties) is accepted. The performance of basic traversals on the graph where elements have no payload is measured and an analysis of speed of loading and deleting the vertices is included. Labels are taken into account during the traversing.

Property Graph Benchmark.

Property graph benchmark accepts the same format of input data; however, it expects that the elements have certain properties to be able to run tests based on them. These tests include traversing queries working with properties (e.g. *Dijkstra*) and also non-traversing queries where the relationships between vertices could be ignored (e.g. *FindEdgesByProperty*). The *UpdateProperties* operation is also included not only to measure how quickly the DB systems can read properties, but also write. All the traversing tests from the Labeled graph benchmark are left in place to be able to observe whether the presence of properties in the graph will have any effect on the performance.

Indexed Graph Benchmark.

This benchmark will be run on the precisely same data as the Property graph benchmark. The only noticeable, although substantial, difference is the inclusion of the *CreateIndexes* operation which sets the database engines to start indexing values on some of the properties. This step should presumably influence the performance of most of the successive operations, either positively or negatively. For example, tests working directly with properties (e.g. *FindEdgesByProperty*) should be completed much faster; on the other hand, tests directly changing the indexed data (e.g. *LoadGraphML*) will have to take the indexes into consideration and thus be noticeably slowed down. When the execution of the benchmark is finished, it is possible to compare the results.

Blueprints supports two ways of indexing elements' properties in the database through *IndexableGraph* and *KeyIndexableGraph* interfaces [8]. The former one is less automatic and supports specific querying techniques with added parameters, e.g. to achieve case insensitive searching. Nonetheless, it is implemented by fewer database systems than *KeyIndexableGraph*, which is why we had to select *KeyIndexableGraph* interface to facilitate indexing of the elements in the graph database.

5. *BLUEBENCH* RESULTS

All the *BlueBench* tests were executed on a computer equipped with a single core Intel Xeon E5450 running at 3.00 GHz and 16GB of RAM, with Ubuntu Server 12.10[29]. The Java Virtual Machine of version 1.7 was started using the default parametrization except for the starting and maximum heap size, set to 12GB. The remaining 4GB of memory were left for other running system processes and, most importantly, for two of the tested DB systems which could not be run inside the JVM.

5.1 Input Data

In *BlueBench* both artificially generated data and real graph data sets were used to fill up the tested databases. Specifically, the first benchmark which completely ignores how the databases handle properties was run twice. Once on synthesized data and then on data collected from *Amazon's co-purchasing network*[30] as it looked like in the year 2003. This network forms a directed graph consisting of more than 260.000 vertices and about 1.234.000 edges, where vertices represent products and a directed edge from product a to product b means that product a was frequently purchased together with the product b.

For the rest of the benchmarks, which also work with vertex and edge properties, no real data sets of suitable size and containing applicable property types were found, so artificially generated graphs were used. The generation was performed with the *iGraph* library, namely the synthesizer implementing the *Barabasi-Albert model*. String labels were added to all edges once the graph had been constructed. Afterwards, for all the benchmarks with the exception of the first one, several properties were appended to the edges and vertices of the graph so that all elements had one string and one integer property.

All the benchmarks were executed on artificial directed graphs having 1, 50, 100 and 200 thousand vertices and an approximate mean vertex degree equal to 5 and 10. This gives the total of eight different graphs with sizes ranging from only about 6.000 elements to as many as 2.200.000 elements.

5.2 Assessed DB Systems

During the first benchmark, where properties are ignored, the following DB systems were assessed: *DEX* 4.7.0, *InfiniteGraph* 3.0.0, *MongoDB* 2.2.3[31], our own prototype *MysqlGraph*, *NativeSail* 2.6.4, *Neo4j* 1.8.1, *OrientDB* 1.3.0, *TinkerGraph* 2.2.0 and *Titan* 0.2.0. Thanks to the size of this set none of the best known DB systems are left out, and several different technologies are represented (e.g. typical GDB, RDF[32] store, Document database, ...). In the rest of *BlueBench* all of these databases except *NativeSail* were benchmarked as it is not possible to naturally work with element properties in *NativeSail* DB. As with any other RDF store, element properties would have to be simulated by adding custom vertices and connecting those with the original element required to contain the property.

5.3 Results of the Tests

We examined the performance of the graph database systems on the operations listed in Section 4, with the exception of *DeleteGraph* and *CreateIndexes*. These two operations were excluded from *BlueBench* assessments for a number of reasons. First, it would be complicated to run the tests multiple times, which is necessary to guarantee a better precision. Or, such functionality is not needed to be run very often in any kind of application; and when it is run, the execution speed is not a factor. Each operation except *LoadGraphML* was executed ten times, two fastest and two slowest times were discarded, and then the rest was averaged.

Graph Loading.

The insertion speed was measured as *loaded objects per second* (LOPS) so that results from data sets of different sizes can be directly compared.

Figure 1 depicts the LOPS value as it was measured in the *Labeled Graph Benchmark*. *DEX* was the fastest system, closely followed by *Neo4j*, both steadying at about 35.000 LOPS. *InfiniteGraph* and *OrientDB* placed on the other side of the spectrum, with performance an order of magnitude lower. This shows quite a difference in insertion speed of the assessed systems; however, the results also show that most of the GDBs scale regularly with growing size of the network – the decrease of LOPS is only sub-linear.

The situation changes when element properties are in-

[29]`http://www.ubuntu.com/`

[30]`http://snap.stanford.edu/data/amazon0302.html`

[31]`http://www.mongodb.org/`

[32]`http://www.w3.org/RDF/`

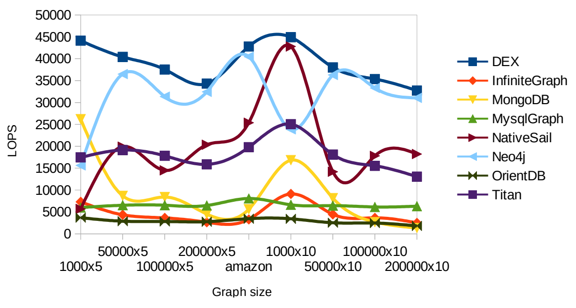

Figure 1: LOPS in *Labeled Graph Benchmark*.

volved in the second benchmark as displayed in Figure 2. The overall speed understandably diminishes by nearly half and *Neo4j* surpasses the rest of the systems by a big margin. There is no noticeable slowdown of *Titan*, and thus it gets on par with *DEX*. The rest of the GDBs remain much slower. Another change occurs when the graph elements are set to be indexed before the start of the loading (*Indexing Benchmark*). Specifically, *Neo4j* gets behind *Titan* and *DEX* which both do not seem influenced by the indexes at all. The reason of *Titan*'s result is trivial, it does not support indexing over properties of edges; therefore the insertion is constrained with much less burden. By contrast, it will be seen in the forthcoming test results that *DEX* does not take advantage of the indexes at all – and they evidently are ignored during the insertion phase as well.

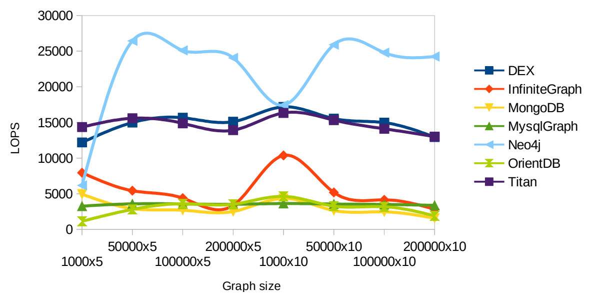

Figure 2: LOPS in *Property Graph Benchmark*.

Traversal.

The performance of a breadth-first search traversal which follows the edge direction was monitored. To better estimate how the GDBs scale with growing data size, the results were expressed as the *traversed edges per second* (TEPS). Figure 3 shows an absolute dominance of *Neo4j* on sparser graphs with TEPS getting as high as 300.000. It is followed by *DEX*, *Titan* and *NativeSail*. On the other hand, *MongoDB* and *InfiniteGraph* do not seem to be optimized for this kind of query at all, giving out TEPS of only around 1.000 which is further decreasing with the growing graph size.

The set of denser graphs brought *Neo4j* much closer to the rest of the systems, mainly to the benefit of *DEX*. As it can be seen in the plot, *Neo4j*'s TEPS values have a decreasing tendency as the data get larger, whereas *DEX*'s performance is gradually growing. We would probably see *DEX* surpassing *Neo4j* if the operation was tested on even bigger data. Meanwhile, *Titan* and *NativeSial* became roughly twice slower than *DEX*, but still clearly separated from the remaining GDBs.

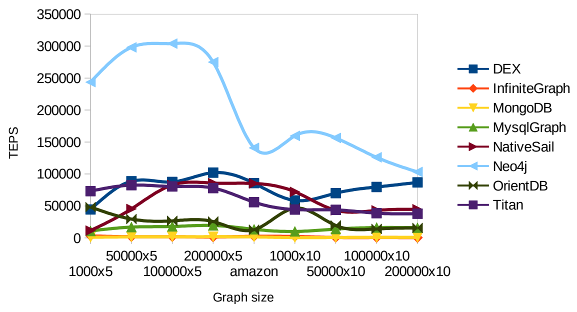

Figure 3: TEPS in *Labeled Graph Benchmark*.

The native implementation of the traversal algorithm provided by a subset of the tested engines was executed with the same parameters within the *TraversalNative* operation. The performance is compared to the results of the conventional methods in Figure 4. Apart from *Neo4j*, the GDBs showed some improvement after using the native method; especially *InfiniteGraph*, TEPS values of which are up to four times higher. A healthy margin in favor of the native method is noticeable in *DEX*'s results also. However, we still conclude that once an application uses *Blueprints* to work with the database, it is not worth making the effort to accommodate the native interface, because the differences in performance are not extensive.

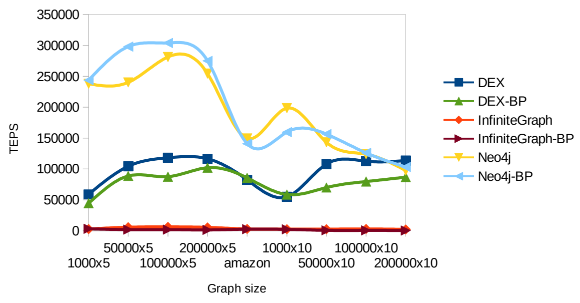

Figure 4: Native traversal implementations vs. their counterparts in *Blueprints* (*Labeled Graph Benchmark*).

The execution times of the *Traversal* operation when run in the three benchmarks were very similar; in other words, it was shown that the presence of element properties or custom indexes on these properties does not have any noteworthy effect on the efficiency of traversal queries.

A very similar situation occurred in the *FindNeighbors* operation; after all, the two operations are implemented using the same *getVertices()* method from the *Blueprints* interface. Only *NativeSail* struggled and yielded surprisingly high results reaching hundreds of milliseconds for the largest graphs. This behavior was most probably caused by the *FindNeighbors* operation fetching all neighbors, even in the opposite direction, in contrast to *Traversal* operation where edge direction is respected. Besides that, the rest of the engines showed little or none growth of the response time with the increasing graph data size.

Shortest Path.

The breadth-first search and the shortest path algorithms both count as traversals and so they could be expected to exhibit a similar performance diversity. However, *getVertices()* method was used for *Traversal* operation implementation, whereas *ShortestPath* employs *getEdges() Blueprints* API. The mechanism underlying these two methods can be separate; in addition, the different algorithms imply different query patterns, and thus dissimilar caching techniques can be used. Two versions of the shortest path operation were executed, the first without considering edge labels and the second following only edges having a label which was previously selected at random. We again cannot confirm any degradation of performance caused by properties or indexes attached to the elements; consequently, only results from the first benchmark are discussed.

Figure 5 depicts the results of *ShortestPathLabeled* operation. Their difference from those of *ShortestPath* is negligible, with the exception of *Titan*, which performed much better in the labeled version due to its vertex-centric indexes optimization (the index could be used to quickly obtain only the edges having the right label). In general, the results are distinct from those of *Traversal* operation; namely, *DEX* is not the nearly best performing system as it was with search traversals, as opposed to *Titan* which clearly improved. On the other hand, *InfiniteGraph*, *MongoDB* and *MysqlGraph* remained to be very inefficient, like they were in the *Traversal* operation.

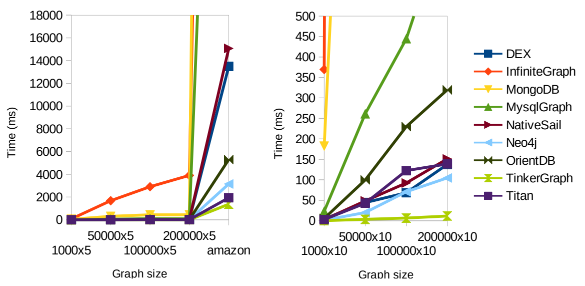

Figure 5: Run times of *ShortestPathLabeled* in *Labeled Graph Benchmark.*

The *ShortestPath* test results yield another interesting observation. As it is clearly noticeable from the plotted run times, the GDBs took much longer to execute the operation on the Amazon data set than on the synthesized graph data, in spite of the graph size being of similar magnitude. This was caused by the fact that there mostly is not an existing oriented path between any two vertices randomly selected from a graph generated by the Barabasi-Albert model. Therefore, the executed algorithms usually finished prematurely and rather quickly. By contrast, the network obtained from Amazon is well connected, and thus the operations often ran until the path was completed, taking much more time to return. This suggests that the graphs created by the selected generator do not necessarily resemble real networks in every possible aspect.

As opposed to *ShortestPath*, *Dijkstra* operation represents a complex query requiring the traversal of the complete graph, and making use of both element properties and edge labels. Therefore, it is more complicated for the GDBs to take advantage of their caches as they would normally do during simpler traversals. This is likely to be the reason of *MysqlGraph*'s acceptable performance in relation to the other systems and, equally importantly, in relation to its performance in *ShortestPath*.

Neo4j is the most efficient engine for this task; it even outperforms *TinkerGraph* by a recognizable margin (by almost 30%). Given that *TinkerGraph* works only in memory and can avoid any delays caused by persistent storages, an explanation could be that *Neo4j* managed to load the entire graph into its caches and calculated the algorithm there. The results of *Dijkstra* operation are plotted in Figure 6.

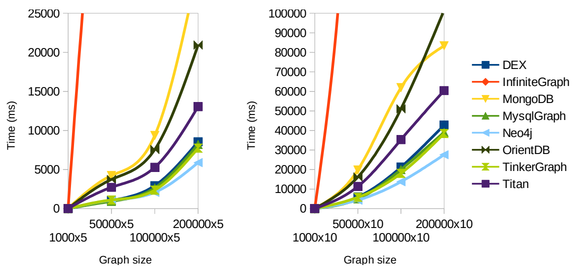

Figure 6: Performance of Dijkstra's algorithm in *Property Graph Benchmark.*

Only *TinkerGraph*'s efficiency was hindered by the presence of element indexes; otherwise the GDBs were practically unaffected. Besides, with respect to the understandable sovereignty of *TinkerGraph* in all above operations, its performance in *Dijkstra* is surprising. Apparently, the algorithms and data structures in *TinkerGraph* engine are optimized for less demanding tasks. However, the Dijkstra's algorithm appeared not to be the best possible choice of a complex operation for GDB assessment. Considerable portions of the run time were spent inside the algorithm itself, leaving only limited room for the differences in the underlying systems to be fully revealed.

Non-traversing Queries.

In this test the performance of two operations which ignore relations between vertices was analyzed; namely, *FindEdgesByProperty* and *FindVerticesByProperty*. Both the operations filter the graph's elements according to a string randomly chosen from a set of strings which are stored in the elements' attribute. Therefore, the operation requires the database systems to iterate through all the objects in the graph and perform string comparisons on the specified attribute of the objects. In *Indexed Graph Benchmark* the GDBs are encouraged to use their indexing mechanisms.

Analysis of *FindEdgesByProperty* operation's execution times and the comparison between plain and indexed benchmark versions is depicted in Figure 7. Once the indexes are used, an apparent performance improvement can be seen for almost all the systems except for *Titan*, which does not support indexing edges, *MysqlGraph*, where indexes are used permanently, and *DEX* – which is the only surprising case. The experiments clearly show that *DEX* completely ignores the assigned index. This problem, in addition to *DEX*'s already slow one-by-one filtering, renders the database system very impractical for this type of queries. In fact, the indexing problem can be caused only by the implementation of the *Blueprints* interface and not by the *DEX* engine itself, accuracy of which we could not verify.

The most efficient persistent GDB for this operation is *Neo4j*, being at least twice as fast as the other systems. Such a big difference is astonishing since this purely non-traversing query should definitely be handled at least equally efficiently by *MongoDB*, *OrientDB*, *InfiniteGraph* and *Titan*, which have their backends based on standalone NoSQL databases (as described in Section 2). *MysqlGraph*'s rapid growth of response time can be explained by the need of an execution of a *join* to merge the tables where the edges and their properties are stored.

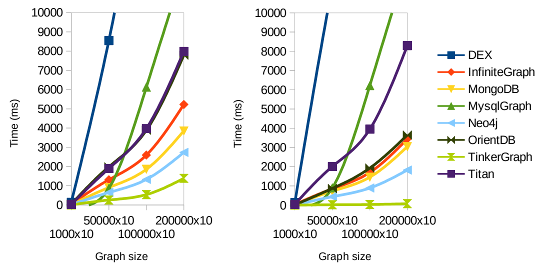

Figure 7: *FindEdgesByProperty* in *Property Graph* (left) and *Indexed Graph Benchmark* (right).

FindVerticesByProperty operation provided similar results in most cases. However, *MysqlGraph* does not need to *join* tables to return vertices; thus, it is slightly faster than most of the other engines in the *Indexed Graph Benchmark*. Moreover, *Titan* could use the index this time which made its performance considerably better. Finally, *Neo4j* again excelled amongst the persistent GDBs, being about twice faster than the second best performing engine.

Manipulation Queries.

Last but not least, extensive data manipulation queries were executed on the databases through *UpdateProperties* and *RemoveVertices* operations in order to observe how the systems cope with situations when data need to be changed, or deleted respectively.

Performance of modifying properties in the second and third benchmark is compared in Figure 8. The presence of indexes evidently has very negative influence on the efficiency of updating data in all the systems, especially in *InfiniteGraph*, *MongoDB*, *Neo4j* and also slightly on *OrientDB*. The other systems, however, did not have to consider indexes for various reason discussed above. In conclusion, *Neo4j* showed very good performance in updating data for being more than three times faster than the second best performing system; and it also handled indexes quite comfortably. *OrientDB* and *InfiniteGraph* ended up on the other end of the scale because the former was more than an order of magnitude slower when the data was indexed and the latter was very slow throughout the entire test.

Deleting a vertex is a rather expensive operation because it requires any edges incident with the vertex to be deleted also. Since *RemoveVertices* was always removing a significant part of the graph in one go, the execution times reached tens of seconds for the slower systems. Therefore, as it can be seen in Figure 9, the density of the graph substantially influenced the performance. *Neo4j*, *DEX* and *Titan* could cope with the situation much more efficiently than the other GDBs. When the same operation was run in *Indexed Graph Benchmark*, all engines apart from *MongoDB* and *Tinker-*

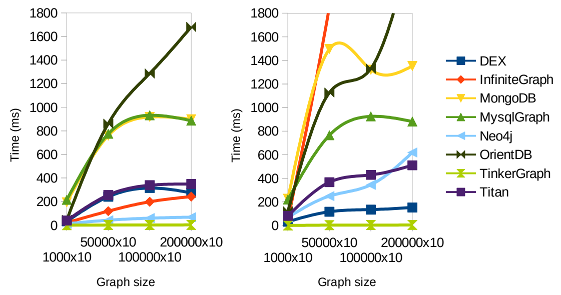

Figure 8: *UpdateProperties* in *Property Graph* (left) and *Indexed Graph Benchmark* (right).

Graph ended up suspiciously unaffected. This observation suggests that the indexes must have been updated in another thread after the method had returned, as opposed to *UpdateProperties*, where the deceleration was clear.

RemoveVertices was also executed in the first benchmark, where *InfiniteGraph* and *MysqlGraph* did not exhibit such big problems with graph density as they did in *Propery Graph Benchmark*; therefore, it is apparent that the process of deleting too many edge attributes was the true cause of their slowdown. Finally, *NativeSail* is part of the *Labeled Graph Benchmark* and was tested on this operation, too; only to expose a surprisingly inefficient implementation of the vertex removal method. To sum up, *Neo4j* was the fastest engine for deleting vertices from the sparser version of the graph, while *DEX* performed slightly better on the denser graph.

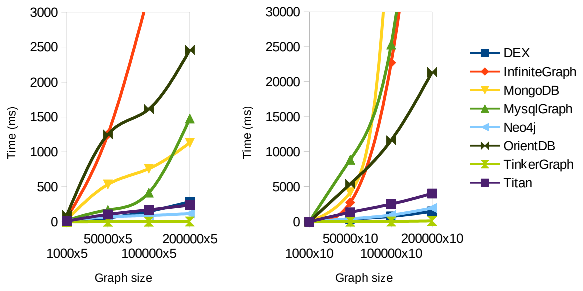

Figure 9: Run times of *RemoveVertices* in *Property Graph Benchmark.*

5.4 Summary of the Results

We divided the operations into several groups to make the analysis more comprehensible: graph loading, traversal queries, non-traversal queries and data manipulation. It is surprising to observe that the groups of experiments provided results that are quite alike; in other words, the GDBs' relative performance was similar although it was measured in entirely different scenarios.

Arguably the strongest reason for GDBs to exist is the need of efficient implementation of traversal operations on persistent graph data. In this area *Neo4j* and *DEX* clearly outperform the rest of the systems, mainly because of the specialization of their backends for exactly this type of queries. *Neo4j* was constantly achieving the best performance even in the other tests, followed by *DEX* and *Titan*. These results are in contrast with [2] which was written four years ago and favored *DEX* over *Neo4j*. Clearly, *Neo4j* has made a lot of progress over the years.

On the other hand, it was shown that directly using a document-oriented database or even relational database for graph operations is not very efficient. *MysqlGraph* and especially *MongoDB* performed well only in tests which were either lightly or not graph related at all. Finally, *InfiniteGraph* was revealed to be the least performing implementation with notoriously slow traversals; often struggling with scenarios where the final execution time was barely acceptable. However, it must be stated that *InfiniteGraph* is focused on distributed solutions with horizontal scaling, not primarily addressing performance on a single machine.

The experiments also helped to discover an important difference between the synthesized and real-world data. The graphs generated by the Barabasi-Albert model exhibited low oriented interconnection of vertices and higher probability of nodes with an extensive degree; although the latter feature could be altered by setting an appropriate parameter before running the graph generation.

6. CONCLUSION

This paper addresses the need of creating a complex and fair benchmark for an experimental assessment of various GDB implementations. We have elaborated on the specific requirements of such a benchmark, and analyzed all possible ways of accessing the databases and conducting the experiments. The result of our efforts, *BlueBench*, can be run both on real and generated graph data sets and is composed of a number of testing scenarios which reflect the wide range of use-cases that GDBs have to face today.

Although lots of effort was put into selecting the set of aspects to be tested by *BlueBench*, there is much more work still to be done. For example, the performance of the database systems was measured only on a single machine under convenient conditions, i.e. the systems had as much memory as they needed. Many interesting results could be obtained from executing the benchmarks on a cluster of nodes where the GDBs would be replicated and a concurrent access (with intentional conflicts) performed.

7. REFERENCES

[1] Dominguez-Sal, Urbón-Bayes, Giménez-Vanó, Gómez-Villamor, Martínez-Bazán, Larriba-Pey. *Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark.* Springer Berlin Heidelberg, 2010, Pages 37–48. ISBN 978-3-642-16720-1.

[2] Bader, Feo, Gilbert, Kepner, Koester, Loh, Madduri, Mann, Meuse, Robinson. *HPC Scalable Graph Analysis Benchmark.* 2009. (`http://www.graphanalysis.org/benchmark/index.html`)

[3] Chakrabarti, Zhan, Faloutsos. *R-MAT: A Recursive Model for Graph Mining.* 2004. (`http://repository.cmu.edu/compsci/541/`)

[4] Ciglan, Averbuch, Hluchy. *Benchmarking traversal operations over graph databases.* 2012. (`http://ups.savba.sk/~marek/papers/gdm12-ciglan.pdf`)

[5] Lancichinetti, Fortunato. *Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities.* Phys. Rev. E 80, 016118, 2009. (`https://sites.google.com/site/andrealancichinetti/benchmark2.pdf?attredirects=0`)

[6] Vicknair, Macias, Zhao, Nan, Chen, Wilkins. *A comparison of a graph database and a relational database: a data provenance perspective.* 2010, ACM SE '10, Article No. 42. ISBN: 978-1-4503-0064-3.

[7] Ciglan. SGDB*3 – Simple Graph Database.* (`http://ups.savba.sk/~marek/sgdb.html`)

[8] *TinkerPop. Blueprints – Graph Indices.* (`https://github.com/tinkerpop/blueprints/wiki/Graph-Indices`)

[9] sparsity technologies. *Why* DEX. (`http://www.sparsity-technologies.com/dex`)

[10] DB-Engines. *Ranking of Graph DBMS.* (`http://db-engines.com/en/ranking/graph+dbms`)

[11] sparsity technologies. DEX *– A High-Performance Graph Database Management System.* (`http://www.sparsity-technologies.com/dex`)

[12] Angles Renzo. *Say hi to* GraphDB-Bench. 2012. (`http://dcc.utalca.cl/~rangles/files/gdm2012.pdf`)

[13] Robinson, Webber, Eifrem. *Graph Databases.* 2013, O'Reilly Media. (`http://graphdatabases.com/`)

[14] Objectivity inc.. *Understanding Accelerated Ingest.* (`http://wiki.InfiniteGraph.com/3.0/w/index.php?title=Understanding_Accelerated_Ingest`)

[15] Redmond, Wilson. *Seven Databases in Seven Weeks.* 2012, O'Reilly Media. ISBN: 978-1-934356-92-0. (`http://it-ebooks.info/book/866/`)

[16] NuvolaBase Ltd. OrientDB. (`https://github.com/nuvolabase/orientdb`)

[17] NuvolaBase Ltd. OrientDB *– Concepts – Record Version.* (`https://github.com/nuvolabase/orientdb/wiki/Concepts#record-version`)

[18] NuvolaBase Ltd. OrientDB *– Transactions.* (`https://github.com/nuvolabase/orientdb/wiki/Transactions`)

[19] Aurelius. Titan. (`https://github.com/thinkaurelius/titan/wiki`)

[20] Aurelius. Titan *– Indexing Backend Overview.* (`https://github.com/thinkaurelius/titan/wiki/Indexing-Backend-Overview`)

[21] Aurelius. Titan *– Storage Backend Overview.* (`https://github.com/thinkaurelius/titan/wiki/Storage-Backend-Overview`)

[22] Broecheler Matthias. *Big Graph Data.* (`http://www.slideshare.net/knowfrominfo/big-graph-data`)

[23] Rodriguez Marko. *The Rise of Big Graph Data.* (`http://www.slideshare.net/slidarko/titan-the-rise-of-big-graph-data`)

[24] Erdos, Renyi. *On random graphs.* Mathematicae 6, 1959, Pages 290–297.

[25] Leskovec, Lang, Dasgupta, Mahoney. *Statistical properties of community structure in large social and information networks.* ACM Press 2008. Pages 695–704. ISBN. 978-1-60558-085-2.

[26] Barabási, Albert. *Emergence of scaling in random networks.* Science. 2008. Vol. 286, no. 5439. Pages 509–512.