Labeled property graphs: SQL or NoSQL?

Dmitry Anikin
Ivannikov Institute for System Programming of the RAS
Higher School of Economics
Moscow, Russia
anikin@ispras.ru

Oleg Borisenko
Ivannikov Institute for System Programming of the RAS
Plekhanov Russian University of Economics
Moscow, Russia
al@somestuff.ru

Yaroslav Nedumov
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
yaroslav.nedumov@ispras.ru

***Abstract*—There are two main approaches to graph databases: based on RDF model and based on labeled property graph model. RDF is well known and studied, but modern graph dataabases with labeled property graph model are studied much lesser.**

In this paper we evaluated several possible solutions for storing and querying graph data using Gremlin – general purpose graph query language from Apache TinkerPop. We used LDBC Graphalytics framework and compared NoSQL-based setups with SQL-based setups. We evaluated JanusGraph on HBase both on single machine and cluster and SQLG on top of PostgreSQL and H2. We used datasets from the different domains and of different sizes up to tens of millions vertices and edges.

Evaluation results show that for the used workload SQLG with PostgreSQL is about ten times faster than JanusGraph on HBase and SQLG with H2 performance is in between.

***Index Terms*—graph databases, labeled property graph, postgresql, hbase, TinkerPop**

I. Introduction

Graph form is a natural choice for data representation in many fields. We can meet them in many areas of our lives – citation graphs, social networks, linked WWW pages, and so on. Graph data modeling is easy to understand and easy to use so this data model is of particular interest for research.

Graph DBMS are developed since 1980th [1], but then with the rise of relational databases the activity almost disappeared. Since 2000, there has been rapid growth in data worldwide. Processing this amount of data is becoming more and more difficult, so the technology is beginning to gain popularity, contributing to the distribution of computing and data storage to hundreds or thousands of servers. This was one of the reasons for the growth of popularity of NoSQL databases and graph databases in particular [2].

In the modern world there are two well-known concepts: SPARQL-based DBMS for RDF [3] data and Apache TinkerPop-based DBMS for labeled property graphs (LPG).

The RDF-model has been studied in many papers [4], [5], which cannot be said about the LPG model.

In LPG model vertices denote discrete objects and edges denote relationships between vertices. Both vertices and edges can have a number of key/value-pairs called properties. The key can be age, name, weight and so on. In fact, LPG and RDF models are equivalent, but LPG looks more convenient to use.

While LPG DBMS looks useful for general graph data processing, their performance is not studied enough (we will show in related work section). So in our work we tried to partially close the gap and tested JanusGraph graph database backed by two different storage engines, HBase and H2, and PostgreSQL wrapped by external Gremlin graph query language translator: SQLG. We wanted to cover different use cases so we need test data from different domains and various sizes.

In the next section we will overview existing approaches to graph DBMS testing. Then we will describe our testing method and obtained results.

II. Related work

In this paper we investigate the performance of graph databases using labeled property graph as the data model. Before we begin, we looked at previous research to find a suitable testing methodology. We were able to find some interesting benchmarks and looked at each one in detail.

The LinkBench [6] is a framework developed in Facebook and its workload is suitable for social networking graphs. The authors used the data generator that generated a synthetic social graph with characteristics similar to the real data. Tests include obtaining edges and vertices by ID, simple creation, removal and updating of edges and vertices, edge range queries, edge count queries. The LinkBench is able to generate a synthetic graph only in the area of social networks, so we tried to find more general solution.

The GraphMath [7] framework was developed at Intel and is designed for large-scale graph analysis. This framework uses sparse matrixes and multi-core scalability to achieve high performance. Page Rank, Breadth First Search, Collaborative Filtering, Triangle Counting, and Single Source Shortest Path are used as testing algorithms. Before we tested the graph databases, we had to understand what methods could be used to do this. The GraphMath provides a really good testing methodology, but unfortunately it does not support testing of different databases.

The YCSB [8] is a framework for comparing the performance of cloud data serving systems. The YCSB supports many systems: Cassandra, HBase, MongoDB and many others. The workload includes four operations: inserting a new

978-1-7281-4623-2/19/$31.00 ©2019 IEEE
DOI 10.1109/IVMEM.2019.00007

record, updating an existing record, reading randomly records, and scanning records in the specified order. The framework supports a large number of systems, but all of them are ”key-value” and ”cloud” storages, so we can not use it as a benchmark for graph databases directly. But it was used as a base for next one platform: XGDBench.

XGDBench [9] was the first benchmark suite aimed particularly to graph databases. It is based on the YCSB framework described above. This benchmark supports five testing operations: inserting a new vertex, deleting a vertex, updating all vertex properties, reading the vertex and its properties, getting all neighbors of a given vertex, as well as the Breadth First Search algorithm. According to the authors of XGDBench, it was designed to test social networks and does not cover other domains. In addition, the last update for the benchmark was more than three years ago, so we decided to look for some other solutions.

Finally we found LDBC Graphalytics [10] It is an industrial-grade benchmark for graph analysis platforms that supports flexible customization for each platform. It contains six algorithms for testing: Breadth-first search (BFS), PageRank (PR), Weakly connected components (WCC), Community detection using label propagation (CDLP), Local clustering coefficient (LCC), Single-source shortest paths (SSSP). Also, Graphalytics has a built-in generator of synthetic tests and many real data sets from different subject areas: citation graphs, games, social networks, and so on. This benchmark already has several ready-to-use drivers for testing, and it also allows developers to write their own drivers for their own platforms.

In this paper we used LDBC Graphalytics by writing two drivers for the JanusGraph graph database and the Apache Tinkerpop implementation over the PostreSQL relational database.

III. Test environment

In this section we will describe hardware and software we used to perform the performance testing.

A. Experimental stand

The following tools and hardware were used for benchmark: Java 8, JanusGraph 0.3.1, TinkerPop 3.4.1, HBase 1.4.10, PostgreSQL 11.3, SQLG 2.0.0, Snap 4.1, Docker 18.09.6.

All computations were performed on a computer with these characteristics: **CPU** 12 x Intel Core i7-8700 @ 3,20GHz, **RAM** 4 x DIMM DDR4 Synchronous 2666 MHz (0,4 ns, 16GiB), **HDD** Toshiba 2TB 7200rpm 64MB DT01ACA200, **SSD** KXG50ZNV512G NVMe TOSHIBA 512GB, **OS** Linux Ubuntu 18.04.

In one experiment, Hbase was deployed on a cluster. Each node in the cluster has the following characteristics: 2 **CPU** @ 2.6GHz, 16GB **RAM**, 55GB SAS **HDD** 7200 RPM.

B. Configuration options of databases

Below are parameters that affect performance only.

- **JanusGraph (hbase single node):** cache.db-cache=true; cache.db-cache-time=180000; cache.db-cache-size=0.5; cache.db-cache-clean-wait = 20; storage.batch-loading=true
- **JanusGraph (hbase cluster):** storage.hbase.region-count=300 storage.hbase.regions-per-server=50 cache.db-cache=true cache.db-cache-time=180000 cache.db-cache-size=0.5 cache.db-cache-clean-wait = 20 storage.batch-loading=true
- **HBase (single node):** hbase.client.write.buffer=8388608; hbase.client.scanner.caching=10000; hbase.hregion.max.filesize=1073741824; hbase.client.scanner.caching=10000; hbase.thrift.maxWorkerThreads=2000; hbase.thrift.maxQueuedRequests=2000; hbase.regionserver.handler.count=100
- **HBase (cluster):** hbase.cluster.distributed=true; hbase.zookeper.quorum=node-master; hbase.client.write.buffer=8388608; hbase.client.scanner.caching=10000; hbase.hregion.max.filesize=1073741824; hbase.client.scanner.caching=10000; hbase.thrift.maxWorkerThreads=2000; hbase.thrift.maxQueuedRequests=2000; hbase.regionserver.handler.count=100
- **H2:** cache_size=16777216; query_cache_size=64; early_filter=true; auto_server=true
- **PostgreSQL:** shared_buffers=16384MB; work_mem=256MB; maintenance_work_mem=256MB; checkpoint_timeout=20min; max_wal_size=1GB; min_wal_size=80MB; checkpoint_completion_target=0.9; effective_cache_size=4GB

C. Implemented driver details

Implementation of the driver represents the implementation of each of the platform-dependent stages.

- **[A] Verify-setup:** The benchmark verifies that the platform and environment are configured correctly based on the preconditions defined in the platform driver.
- **[B] Format-graph:** The benchmark minimizes ”input data” by removing unused nodes and edges properties.
- **[C] Load-graph:** The platform converts minimized data into any platform-specific data format and uploads the data into a storage system, which can be either a local file system, a shared file system or a distributed file system.
- **[D] Execute-run:** The platform runs a test run with a specific algorithm and dataset. All the tests using the same input dataset can use the already prepared data set obtained at the ”load-graph” stage.
- **[E] Delete-graph:** The platform unloads the data set from the storage system - this is a part of the process of cleaning after completion of all performance tests on a given dataset.

In addition to the steps described above, it is necessary to implement each of the six algorithms for a particular platform.

It should be noted that the driver sends queries to the database synchronously. That is, it waits for an answer before sending a new query.

D. Datasets and their characteristics

In order to ensure the objective results obtained, performance should be measured on datasets of different nature.

Therefore, in this paper we used datasets taken from the real world, and synthetic data sets obtained with the help of built-in graph generator Graphalytics. Datasets from the real world represent various areas: social networks, computer games, citation graphs and so on.

TABLE I
CHARACTERISTICS OF DATASETS

	D1	D2	D3	D4	D5	D6	D7
Vertices	0.63M	0.75M	13.18M	16.52M	1.38M	0.06M	2.39M
Edges	34.18M	42.16M	32.79M	41.02M	85.67M	50.87M	64.15M
P1	5	5	27	33	5	4	6
P2	107.9	111.8	4.9	4.9	123.4	1663.2	53.5
P3	0.087	0.086	0.125	0.12	0.08	0.31	0.06
P4	12B	15B	4B	5B	34B	182B	185B
P5	124M	157M	21M	27M	345M	16B	2B
P6	3.45	3.5	6.54	6.09	3.56	2.33	3.8

In addition to the vertices and edges, several other important characteristics can be identified:

- **The diameter of the graph (P1)** - is a number equal to the distance between the most distant nodes of the graph.
- **Average degree of vertex (P2)** - is a number equal to the sum of degrees of all vertices divided by the number of vertices.
- **Local clustering coefficient (P3)** is a measure of how "well" connected the two are the neighbors of this node. The local clustering coefficient is computed as the number of connections between neighbors of the node divided by the possible number of connections between neighbors.
- **Open Triads (P4)** - is the number of triads of vertices between which there are only two relations.
- **Closed Triads (P5)** - is the number of three vertices where any two vertices are connected by a relationship.
- **AnfEffDiam (P6)** - 90th percentile of the distribution of the length of the shortest chart path.

We expect that the working-time of the algorithms will depend on some of these characteristics.

The Graphalytics driver implements the specifics of each database. Graphalytics has several built-in drivers for different databases. Among them is one of the most popular graph database: Neo4j. Unfortunately, we failed to run the built-in driver for Neo4j over the current version of Graphalytics because the driver is outdated.

IV. EXPERIMENTS

Using experimental stands, datasets and workload specified by Graphalytics framework we conducted several experiments in order to measure performance of different setups.

Firstly we tested JanusGraph and Hbase hosted on a single machine as some easy to do baseline for NoSQL. Then we tested SQLG with PostgreSQL also on a single machine as equivalent baseline for SQL. We also tested H2 backend for SQLG in order to broaden the scope of SQL-based solutions. Finally we tested JanusGraph with Hbase using cluster with six virtual machines each with dedicated HDD.

SSDs are capable of achieving high speeds, but still cost far more than HDDs. Therefore, a typical usage scenario is the separation of database and data. This is the scenario we used in our experiments.

It would be interesting to conduct similar experiments entirely on SSDs. Perhaps a different way of physical data storage would show other dependencies and performance.

In the next subsections we will provide more detailed descriptions of the experiments and obtained results.

A. JanusGraph + Hbase

HBase is one of the most popular open source NoSQL class DBMS included in the list of supported JanusGraph graph database storages. Each algorithm was executed three times for each graph. Hbase was on the SSD and the stored data was on the HDD.

TABLE II
TRIPLE-RUN AVERAGE TESTING TIME IN HOURS FOR HBASE

	D1	D2	D3	D4	D5	D6	D7
BFS	1.44	1.48	2.77	4.19	4.20	0.91	4.12
PR	4.00	4.33	5.37	7.40	10.89	2.10	9.15
WCC	1.89	2.50	2.62	3.69	5.97	1.15	4.56
LCC	2.69	3.15	2.59	4.23	7.94	1.94	8.60
CDLP	1.72	2.18	2.78	4.75	5.35	1.28	5.01
SSSP	2.94	3.00	5.47	7.59	7.55	1.80	8.35

Each of the algorithms repeatedly performs the operation "get all the neighbors". We have considered the dependence of the average execution time of this operation for vertices on the average degree of the graph nodes.

The graph in the Fig. 1 show that the execution time of the algorithms depends on the average degree of vertices.

The same dependence can be observed for the other five algorithms. Dependencies on other characteristics were not found.

B. SQLG + PostgreSQL

SQLG is a partial implementation of the TinkerPop framework over relational database systems. SQLG allows you to use many basic functions of TinkerPop, which will be quite enough to implement the algorithms of the benchmark.

PostgreSQL is a free object-relational database management system. With the help of SQLG we managed to use PostgreSQL as a storage and perform testing. Each algorithm was executed three times for each graph. PostgreSQL was on the SSD and the stored data was on the HDD.

Let's consider the dependence of execution time of algorithms on the number of vertices.

The graphs in the Fig. 2 show that for BFS the execution time depends on the number of nodes in the graph. For LCC a similar dependence is maintained for graphs with less than two

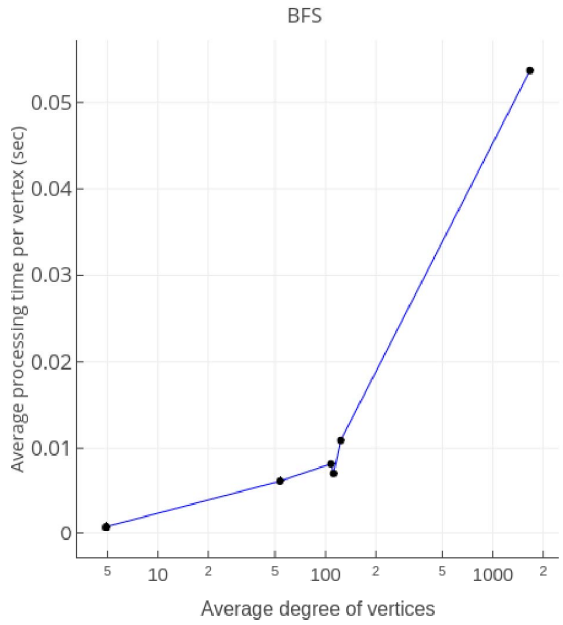

Fig. 1. Dependence of average processing time of one vertex on average degree of vertices for BFS

TABLE III
TRIPLE-RUN AVERAGE TESTING TIME IN HOURS FOR POSTGESQL

	D1	D2	D3	D4	D5	D6	D7
BFS	0.1	0.1	0.6	0.7	0.2	0.1	0.2
PR	0.2	0.3	1.2	1.5	0.5	0.1	0.5
WCC	0.1	0.1	0.7	0.9	0.3	0.1	0.3
LCC	0.9	1.2	0.7	0.9	1.6	0.8	2.8
CDLP	0.1	0.2	0.6	0.8	0.4	0.1	0.3
SSSP	0.2	0.2	0.3	0.5	0.4	0.1	0.6

million vertices. For graphs with more than two million nodes, you can see an anomaly: a rapid decrease in the execution time of the algorithm.

Let's consider the dependence of the time of the algorithm of computation of the local clustering coefficient on the value of this coefficient.

The graphs in the Fig. 3 show that the higher clustering coefficient, the shorter the execution time of the algorithm.

In this experiment, the dependencies of time on two characteristics were found: the number of vertices and the local clustering coefficient. In addition, an anomaly was found: a rapid acceleration of the algorithm with an increase in the number of vertices.

In an attempt to explain this anomaly, we formed a hypothesis and tested it in the next experiment. During the benchmark working, the dataset is first loaded into the database, and then it executes all the algorithms sequentially. We assumed that PostgreSQL could cache some data. Therefore, we decided to run each algorithm in the Docker container containing PostgreSQL.

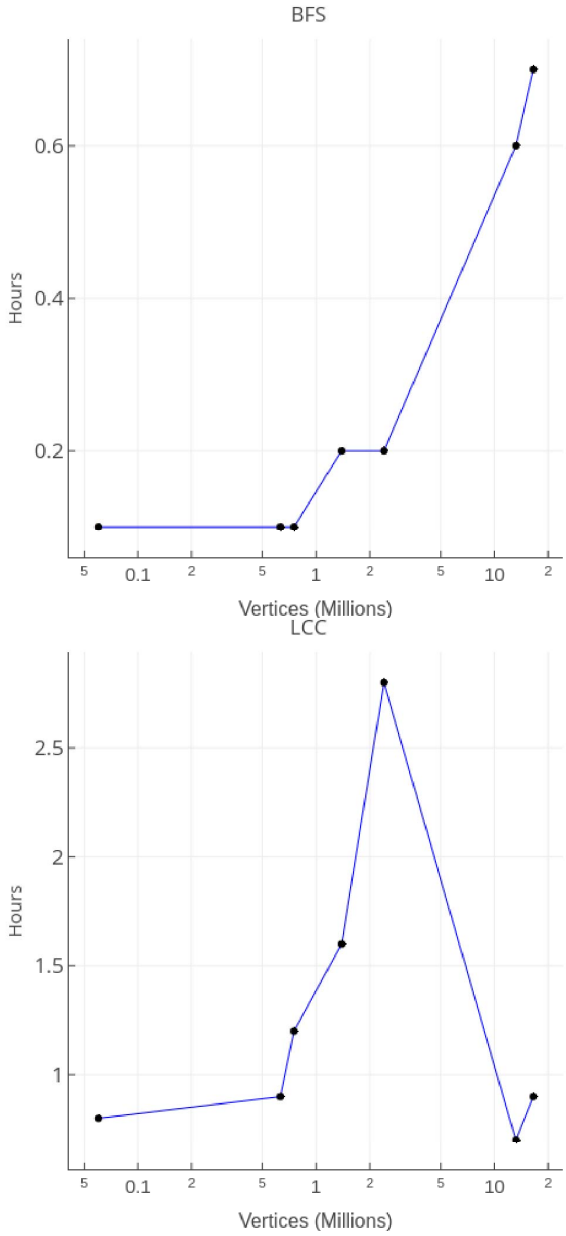

Fig. 2. (a) Dependence of execution time on the number of nodes for the BFS (b) Dependence of execution time on the number of nodes for the LCC

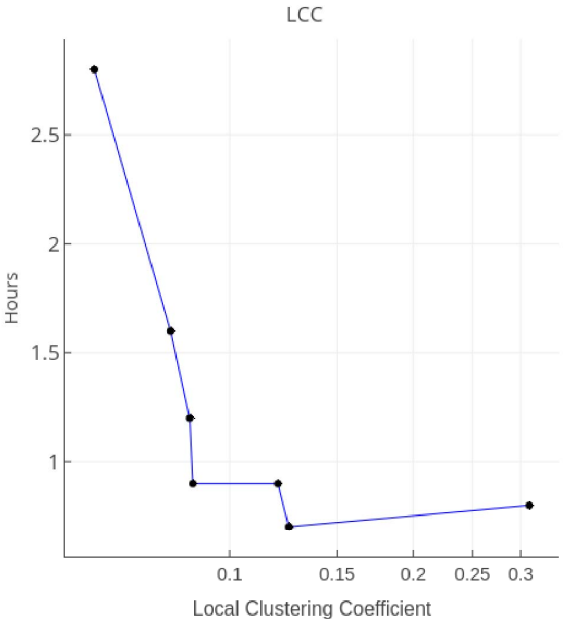

Fig. 3. Dependence of execution time on the local clustering coefficient for LCC (SQLG + PostgreSQL)

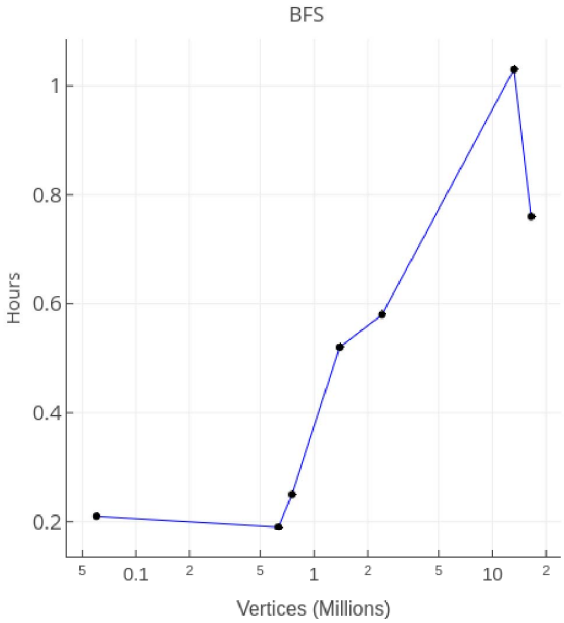

Fig. 4. Dependence of execution time on the number of nodes for the BFS (h2)

C. SQLG + PostgreSQL + Containers

Create and deploy new applications faster with Docker/Podman containers. Containers combine software and its dependencies into a standardized, easy-to-develop software block that includes everything it needs to run: code, runtime, system tools and libraries. This guarantees that the application will always work in the same way. This experiment used an official Docker container with the same PostgreSQL version and configuration as the previous experiment. Each algorithm was executed ten times for each graph. The Docker was stored on the SSD and the data was stored on the HDD.

TABLE IV
OVER TEN RUNS TESTING TIME AVERAGING IN HOURS FOR POSTGESQL AND DOCKER

	D1	D2	D3	D4	D5	D6	D7
BFS	0.12	0.15	0.79	0.99	0.3	0.08	0.31
PR	0.27	0.33	1.64	2.03	0.67	0.17	0.64
WCC	0.14	0.17	0.87	1.1	0.34	0.34	0.32
LCC	1.06	1.37	0.79	1.06	1.89	0.97	3.01
CDLP	0.17	0.21	0.86	1.09	0.43	0.11	0.37
SSSP	0.25	0.24	0.55	0.74	0.53	0.14	0.63

In a previous experiment, an anomaly was discovered. In this experiment, we tested our hypothesis and, unfortunately, our hypothesis was wrong. When using the Docker container, the anomaly also persists. We also found that additional overheads are imposed on the launch of the container from the docker.

D. H2

H2 is an open-source lightweight Java RDBMS. It can be embedded in Java applications or run in the client-server mode. The main feature is the ability to store data in-memory. H2 was on the SSD and the stored data was on the HDD.

TABLE V
TRIPLE-RUN AVERAGE TESTING TIME IN HOURS FOR H2

	D1	D2	D3	D4	D5	D6	D7
BFS	0.19	0.25	1.03	0.76	0.52	0.21	0.58
PR	0.39	0.49	1.96	1.31	1.07	0.37	1.04
WCC	0.21	0.32	1.05	0.83	0.72	0.31	0.83
LCC	1.12	1.39	1.03	1.20	0.5	1.15	0.48
CDLP	0.21	0.27	0.97	0.71	0.6	0.21	0.57
SSSP	0.45	0.40	0.55	0.72	0.64	0.43	0.88

Let's consider the dependence of execution time of algorithms on the number of vertices. On the Fig. 4 shows the chart only for BFS, because all the other algorithms behave in the same way.

The graphs in the Fig. 4 show that for BFS the execution time depends on the number of nodes in the graph. H2 shows the time dependence of only one characteristic.

E. Hbase on the cluster

The Hbase storage supports cluster work using Hadoop. We have deployed Hbase on seven machines: six workers and one master-node. JanusGraph was used as the database again. Unfortunately, when we ran benchmark on the cluster, we found that it was very slow. By varying the various configurations, we were unable to achieve a significant increase in performance.

The time of our research is limited, so the table below shows the results only for the BFS algorithm.

TABLE VI
THE RUN TESTING TIME IN HOURS FOR H2

	D1	D2	D3	D4	D5	D6	D7
BFS	8.37	8.59	16.4	25.43	25	7.19	24.6

V. CONCLUSION AND DISCUSSION

In this article we have tested many SQL and NoSQL solutions on graph-specific use-cases. SQL solutions are related to: PosgreSQL, H2. NoSQL solutions are related to: Hbase, JanusGraph. Additionally, we tested Hbase and JanusGraph on a cluster of 7 machines. The benchmark includes the following graph-specific use-cases: Breadth-first search, PageRank, Weakly connected components, Community detection using label propagation, Local clustering coefficient, Single-source shortest paths.

As a testing system we used LDBC Graphalytics. We also implemented two drivers that allowed us to launch a benchmark for JanusGraph and SQLG. Each of the algorithms has been tested on different datasets of different sizes and domains. Each of the datasets belongs to one of the areas: social networks, games or synthetically generated graphs. The sizes of the graphs vary in the following ranges: from 61,170 to 16,521,886 vertices and from 32,791,267 to 64,155,735 edges. The best performance was shown by SQLG + PostgeSQL. For example, on a dataset with 16 million nodes and 41 million edges, the BFS algorithm finished its work in 0.7 hours. The worst time was shown for a graph with 2 million vertices and 64 million edges, the LCC algorithm completed its work in 2.8 hours. The performance of the H2 database is almost at the same level as that of PostgreSQL, as can be seen in the Fig. 5.

As a NoSQL solution we used JanusGraph + Hbase. This solution showed performance about 10 times worse than PostgreSQL. For example, on a dataset with 16 million nodes and 41 million edges, the BFS algorithm finished its work in 4.9 hours. The worst time was shown for a graph with 1.3 million vertices and 85 million edges, the PR algorithm completed its work in 10.89 hours. We also tested NoSQl's JanusGraph + Hbase solution on a cluster of 7 machines, where 6 were workers-nodes and 1 was a master-node. The cluster showed very low performance compared to single node. For example, for the smallest dataset with 633 thousand vertices and 34 million edges, the BFS algorithm has completed its work in 8.37 hours.

Amount of performed experiments is too limited to make broad conclusions but we showed that specific graph-manipulation language could be efficiently implemented over SQL database running on the single machine. Graph with sizes up to tens of millions vertices could be processed in such way much faster than using the cluster. Any graphs in our samples can be processed faster on a single machine than on a cluster. Perhaps there are other cluster settings that can increase performance, so cluster-based NoSQL setup needs further research.

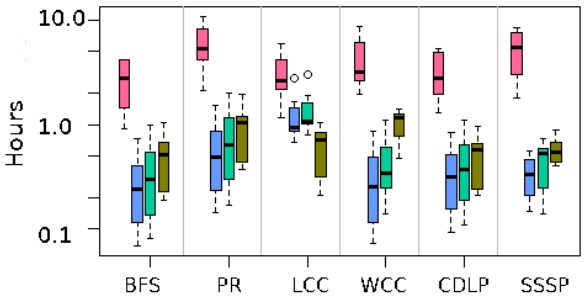

Fig. 5. The general chart for all launches. Red - JanusGraph + HBase (single node), blue - SQLG + PostgreSQL, light green - SQLG + PostgreSQL + Docker, dark green - H2

VI. FUTURE WORK

In the future work we are going to further research graph DBMS performance and check also Neo4j and OrientDB. Also we are going to increase size of stored graphs and try to use Microsoft Academic Graph [11] with hundreds of millions edges. In addition, the performance of graph databases on clusters is of particular interest. Cassandra provide an opportunity to run on a cluster, we would like to know what performance we can get on several machines.

ACKNOWLEDGMENT

This work is funded by the Minobrnauki Russia (grant id RFMEFI60417X0199, grant number 14.604.21.0199).

REFERENCES

[1] RENZO ANGLES, CLAUDIO GUTIERREZ, "Survey of Graph Database Models"
[2] Yu Xu, "Graph Databases Burst into the Mainstream", 2018, https://www.kdnuggets.com/2018/02/graph-databases-burst-into-the-mainstream.html
[3] W3C, "RDFa 1.1 Primer - Third Edition", 2015
[4] Eugene Inseok Chong ; Matthew Perry ; Souripriya Das, "Improving RDF Query Performance Using In-memory Virtual Columns in Oracle Database", 2019
[5] Franck Ravat, Jiefu Song, Olivier Teste, "Improving the performance of querying multidimensional RDF data using aggregates", 2019
[6] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, Mark Callaghan, "LinkBench: a Database Benchmark Based on the Facebook Social Graph", 2013
[7] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, "GraphMat: High performance graph analytics madeproductive", 2015
[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears, "Benchmarking Cloud Serving Systems with YCSB", 2010
[9] Miyuru Dayarathna, Toyotaro Suzumura, "XGDBench: A Benchmarking Platform for Graph Stores in Exascale Clouds", 2012

[10] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Perez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, Peter Boncz, "LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms", 2016

[11] Microsoft, Microsoft Academic Graph, https://academic.microsoft.com/home