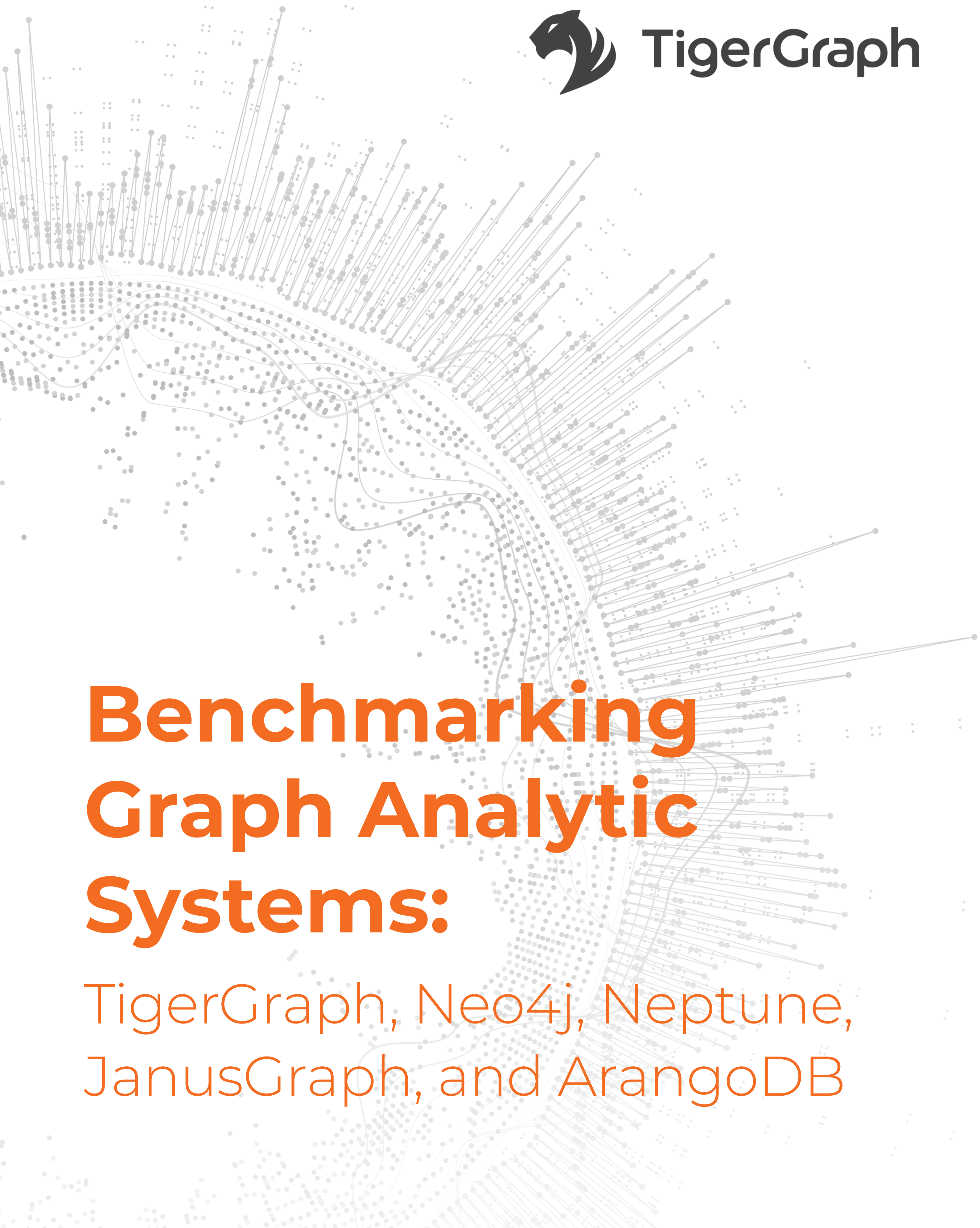
TigerGraph



Benchmarking Graph Analytic Systems:

TigerGraph, Neo4j, Neptune, JanusGraph, and ArangoDB

EXECUTIVE SUMMARY

As the world's first native, real-time, and MPP (Massively Parallel Processing) graph database, the TigerGraph^{TM} system loads, stores, and queries data faster than other graph databases.

- TigerGraph is 2x to more than 8000x faster at graph traversal and query response times compared to other graph databases tested, running on a single server. TigerGraph's advantage increases as the number of hops increases.
- TigerGraph's query performance increases almost linearly with the number of machines. Even on PageRank, requiring massive node-to-node communication, TigerGraph still achieves 6.7x speedup with 8x machines.
- TigerGraph's inherent parallelism enables it to load data online 1.8x to 3x faster than Neo4j's offline mode and 12x to 58x faster than other databases, even when using a single machine.
- TigerGraph's high data compression lets it store a large graph in 5x to 13x less disk space than that of other graph databases tested.

This benchmark study examines the data loading and query performance of TigerGraph, Neo4j, Amazon Neptune, JanusGraph, and ArangoDB. The benchmark tests include the following:

1. Data Loading
   - Loading time
   - Storage size of loaded data
2. Querying
   - Query response time for K-hop path neighbor discovery
   - Query response time for Weakly Connected Components and PageRank
   - Scalability of query response time over a cluster of machines

1

BENCHMARK SETUP

This section describes the graph systems tested and the hardware platforms and datasets used.

Graph database/analytics systems

- TigerGraph Developer Edition 2.1.4
- Neo4j 3.4.4 Community Edition
- Amazon Neptune 1.0.1.0.200233.0
- JanusGraph 0.2.1, with Cassandra as the storage backend
- ArangoDB 3.3.13, with each of two storage engines, MMFiles and RocksDB[1]

1 ArangoDB offers two storage engine options: MMFiles (memory mapped files) and RocksDB. MMFiles works well for datasets that fit into main memory; RocksDB is better for larger datasets. https://www.arangodb.com/why-arangodb/comparing-rocksdb-mmfiles-storage-engines/

Hardware platform

All experiments were run on the same Amazon EC2 hardware, except Neptune where the top 2 most powerful DB instances were employed.

Table 1A - Cloud hardware and OS for each database

Database	EC2 Machine	vCPUs	Memory (GiB)	Provisioned IOPS	Max Bandwidth (Gbps)	OS
TigerGraph	r4.8xlarge[2]	32	244	25,600	7	Ubuntu 14.04.5 LTS
Neo4j	r4.8xlarge	32	244	25,600	7	Ubuntu 14.04.5 LTS
Neptune1	db.r4.4xlarge	16	122	160,000[3]	3.5	Amazon Linux AMI 2018.03
Neptune2	db.r4.8xlarge	32	244	323,000[2]	7	Amazon Linux AMI 2018.03
JanusGraph	r4.8xlarge	32	244	25,600	7	Amazon Linux AMI 2018.03
ArangoDB	r4.8xlarge	32	244	25,600	7	Ubuntu 14.04 LTS

In addition, Neptune requires a client machine from which to submit data loading jobs.

Table 1B - Neptune client machine

EC2 Machine	vCPUs	Memory (GiB)	OS	Maximum Bandwidth
r4.xlarge	4	30.5	Amazon Linux AMI 2018.03	850 Mbps

Datasets

The benchmarking study uses two datasets, one synthetic and one real.

Table 2 - Datasets

Name	Description and Source	Vertices	Edges
graph500	Synthetic Kronecker graph http://graph500.org	2.4 M	67 M
twitter	Twitter user-follower directed graph http://an.kaist.ac.kr/traces/WWW2010.html	41.6 M	1.47 B

2 For r4.8xlarge, we attached a 512GB EBS-optimized Provisioned IOPS SSD (IO1) for all I/O traffic with 50 IOPS/GB.
3 Amazon Neptune IOPS is observed from the Amazon Cloudwatch console.

For each graph, the raw data are formatted as a single tab-separated edge list:

```
U1	U3
U1	U4
U2	U3
```

Vertices do not have attributes, so there is no need for a separate vertex list.

2

DATA LOADING TESTS

The data loading tests examined these two areas:

1. Loading time and speed
2. Storage size of loaded data

Loading Methodology

For each graph database, we selected the most favorable method for bulk loading of initial data:

Table 3 - Loading methods for each database

Name	Loading API or Method
TigerGraph	GSQL declarative loading job
Neo4j	neo4j-import (offline loader), then build index on vertices to speed up querying
Neptune	Neptune loader which loads external file (stored in S3) directly to Neptune DB instance
JanusGraph	Java program which uses TinkerPop API to add vertices and edges
ArangoDB	arangoimp bulk importer tool

We discovered that some of the graph databases require either some pre-processing or post-processing effort. Our datasets consist of edge lists only, but other systems either must (Neo4j, Neptune, ArangoDB) or benefit from (JanusGraph) loading vertices before loading edges. Therefore, we wrote a simple script to generate a vertex file which contains all the unique vertex IDs in the edge lists. Neptune has stricter data format requirements than the others. To avoid memory problems with JanusGraph, we split the data file into several small datasets, then loaded them sequentially. For Neo4j, users need to build an index as a separate step after loading, in order to have good query performance.

We broke down the overall data loading into four stages and measured the wall clock time for each stage.

- **Vertex File Preparation:** The time to generate the vertex file.
- **Vertex Loading:** The time to load the vertices to the graph.

- **Edge Loading:** The time to load edges (and possibly vertices) to the graph.
- **Index building:** Most of the graph databases build a primary ID index during loading. Neo4j requires that this be done as a separate step. Without an index, query performance is significantly slower.

Loading Time and Speed

The tables below show the total loading time as well as the per-stage time for each database, for the two datasets.

Table 4A - Data loading times for graph500 (in seconds)

	TigerGraph	Neo4j	Neptune1	Neptune2	JanusGraph	ArangoDB.m	ArangoDB.r
edge	56.0	37.3*	1,571.0	1,052.0	574.6	599.6	944.9
vertex	-	- *	45.0	30.0	43.9	11.5	44.5
vertex file prep	-	54.4	93.2	93.2	54.4	54.4	54.4
index	-	7.99	-	-	-	-	-
total	56.0	99.7	1,709.2	1,175.2	672.9	665.5	1,043.8
relative	**1**	**1.8**	**30.5**	**21.0**	**12.0**	**11.9**	**18.6**

Table 4B - Data loading times for Twitter (in seconds)

	TigerGraph	Neo4j	Neptune1	Neptune2	JanusGraph	ArangoDB.m	ArangoDB.r
edge	785	685*	42248	21700	12192	N/A[4]	20137
vertex	-	- *	733	503	839	206	272
vertex file prep	-	1270	2259	2259	1270	-	1270
index	-	126	-	-	-	-	-
total	785	2081	45240	24462	14301	N/A	21679
relative	**1**	**2.7**	**57.6**	**31.2**	**18.2**	**N/A**	**27.6**

4 ArangoDB.m's edge loading ran for 24.4 hours, loaded 54% of the data, and then ran out of memory.

* The Neo4j bulk loader works on vertex files and edge files together. We recorded the total time of the bulk loader.

The figures below illustrate these loading times graphically.

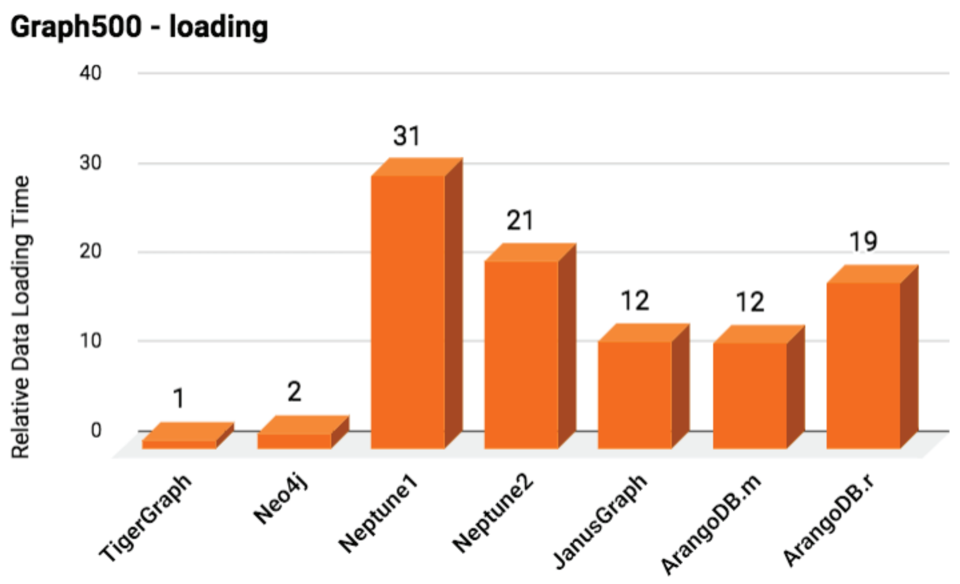


Figure 1A - graph500 data loading times

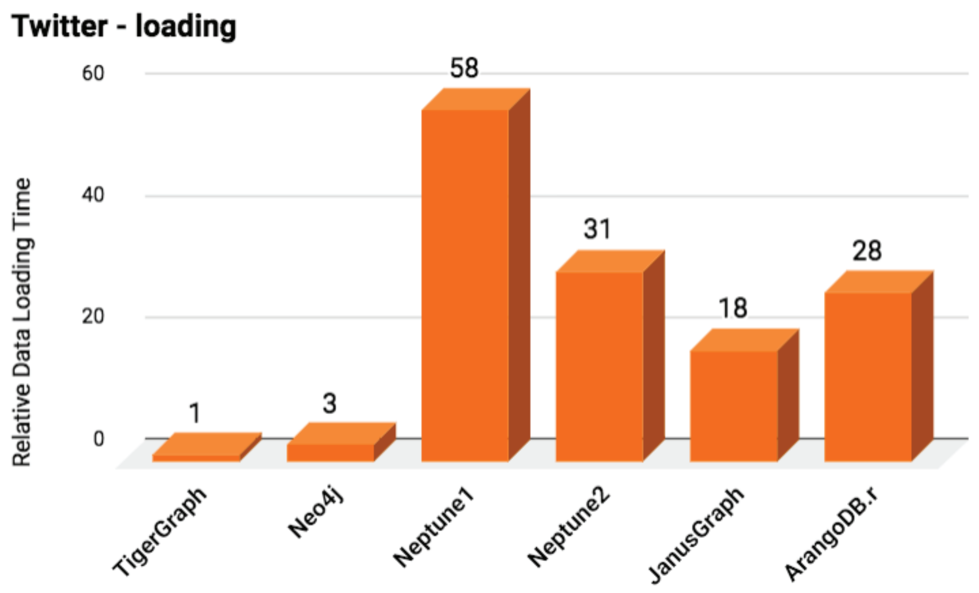


Figure 1B - Twitter data loading times

- ***Only Neo4j's loading time is comparable to TigerGraph's (1.8x to 2.7x slower than TigerGraph). We used Neo4j's offline loading method (where concurrent database operations are not allowed); its online loading method is much slower.[5]***
- ***Aside from Neo4j, TigerGraph loads the smaller dataset 12x to 31x faster than the other graph databases. It loads the larger dataset 18x to 58x times faster.***
- ***TigerGraph has the simplest loading process, with no pre- or post-processing.***

5 We benchmarked Neo4j's offline and online loading methods against TigerGraph in 2017: https://doc.tigergraph.com/report/WP_NPGbenchmark_Sep17_web.pdf

Storage Size of Loaded Data

The size of the loaded data is an important consideration for system cost and performance. All other things equal, a compactly stored database can store more data on a given machine and has faster access times because it gets more cache and page hits. TigerGraph automatically encodes and compresses data, reducing the raw data size to less than half its original size. In contrast, all the other databases magnify the input data. We measured the database storage size after loading. For Neptune we use the Cloudwatch console to get the storage size. The Amazon user guide says they automatically replicate 6 times, so we divided our measured number by 6.

Table 5 - Loaded data storage sizes (MB)

Dataset	Raw Data	Tiger Graph	Neo4j	Neptune	JanusGraph	ArangoDB
graph500	967	482	2,300	3,850	2,764	6,657
twitter	24,375	9,500	49,000	91,140	55,296	126,970

The chart below compares the graph storage sizes, normalized to the raw input data size.

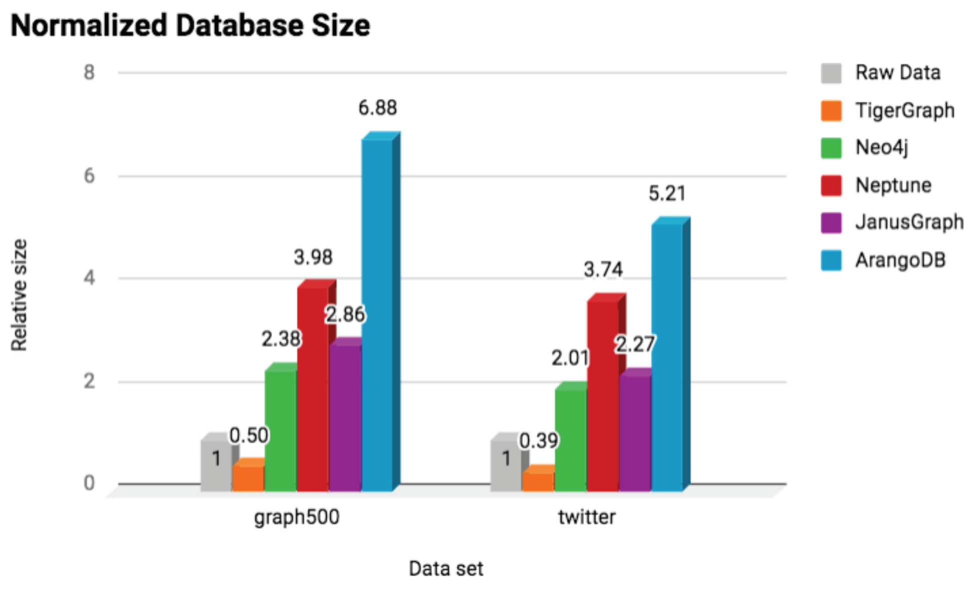


Figure 2 - Normalized database size after loading

Data Storage Summary

- ***Raw data is compressed and stored in TigerGraph resulting in 50% storage savings for the Graph 500 dataset and 61% storage savings for the Twitter dataset.***
- ***Other graph databases create graphs that are larger than the raw data, demanding 5x to 13x more storage space than TigerGraph.***

3

QUERY PERFORMANCE TESTS

Data retrieval, or querying, is one of the essential functions of any database system. A data analytics system must be able to perform the types of queries needed for the user's application, and they must be performed fast.

The Query performance tests examined these three areas:

- Query response time for K-hop-path neighbor count query
- Query response time for Weakly Connected Components and PageRank (full graph queries)
- Scalability of query response time

K-Neighborhood Query

The k-hop-path neighbor count query, which asks for the total count of the vertices which have a k-length path from a starting vertex, is a good measure of graph traversal performance. For each dataset we measure the query response time for the following queries:

- Count all 1-hop-path endpoint vertices for 300 fixed random seeds, with timeout set to 3 minutes/query.
- Count all 2-hop-path endpoint vertices for 300 fixed random seeds, with timeout set to 3 minutes/query.
- Count all 3-hop-path endpoint vertices for 10 fixed random seeds, with timeout set to 2.5 hours/query.
- Count all 6-hop-path endpoint vertices for 10 fixed random seeds, with timeout set to 2.5 hours/query.

By "fixed random seed," we mean we make a one-time random selection of N vertices from the graph and save this list as the repeatable input condition for the tests. Each path query starts from one vertex and finds all the vertices which are at the end of a k-length path. The N queries are run sequentially. Each starting vertex has a different neighborhood size; we report the average size. To focus on the graph traversal and to minimize the network output time, we output only the size of the k-hop-path neighborhood, instead of the complete list of vertices. We cross-validated all the test query results among the graph databases benchmarked.

Table 6 - Average k-hop neighborhood sizes

dataset ↓ \ hops→	1	2	3	6	Total Vertices
graph500	5.13E+03	4.89E+05	1.43E+06	1.58E+06	2.40E+06
twitter	1.06E+05	3.25E+06	2.07E+07	3.50E+07	4.16E+07

We implemented the query in the native query language of each database: GSQL for TigerGraph, Cypher for Neo4, Gremlin for Neptune and JanusGraph, and AQL for ArangoDB. The tables below report the actual and normalized query times for all the databases. Recall that ArangoDB.m was not able to load the twitter data.

Table 7 - One-hop path query time

Dataset	Measure	TigerGraph	Neo4j	Neptune1	Neptune2	JanusGraph	ArangoDB.r	ArangoDB.m
graph500	time (ms)	6.3	21.0	15.8	13.5	27.2	91.9	23.0
	normalized	1	3.3	2.5	2.1	4.3	14.6	3.7
twitter	time (ms)	24.1	205.0	335.8	289.2	394.8	1674.7	N/A
	normalized	1	9	14	12	16	69	N/A

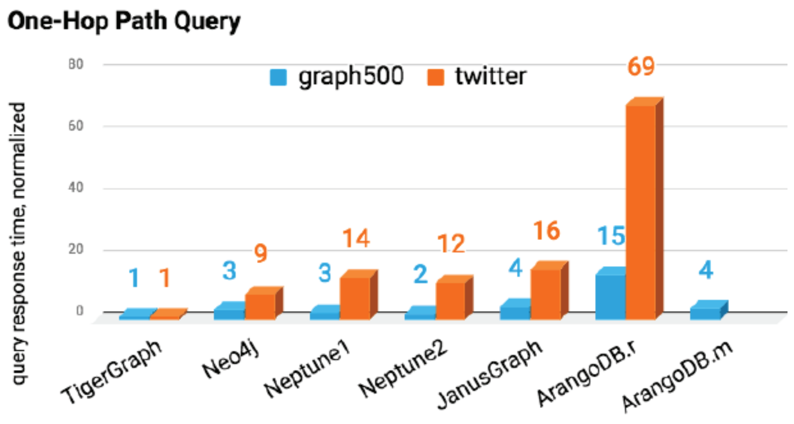


Figure 3 - Normalized query response times for 1-hop queries.

When the path length increases to 2 hops, some of the databases time out for some of the 300 trials. (We set a time limit of 3 minutes for each seed when k is 1 and 2.) In the table below, we report the average query response time over the successful cases, the corresponding normalized query time, and the number of trials that timed out.

Table 8 - Two-hop path query time

Dataset	Measure	TigerGraph	Neo4j	Neptune1	Neptune2	JanusGraph	ArangoDB.r	ArangoDB.m
graph500	time (sec)	0.071	4.18	7.77	8.25	15.39	24.05	16.65
	normalized	1	59	109	116	216	337	234
	timeouts	0	0	1	0	3	49	20
twitter	time (sec)	0.46	18.34	26.17	27.40	27.78	28.98	N/A
	normalized	1	40	57	60	60	63	N/A
	timeouts	0	0	63	60	48	101	N/A

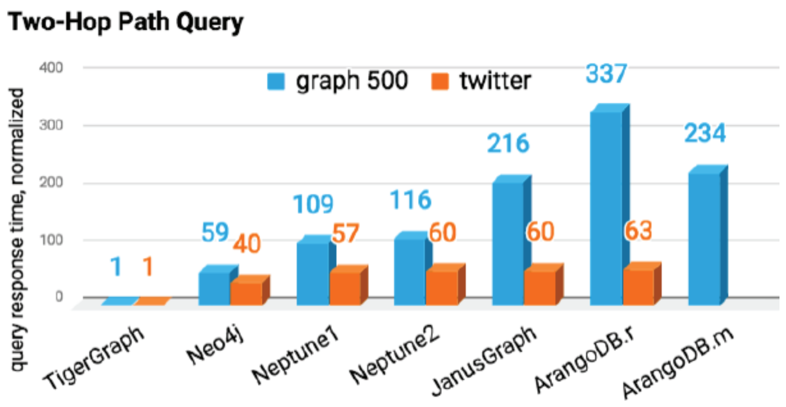


Figure 4 - Normalized query response times for 2-hop queries.

A path length of 3 hops is much more challenging for most of the databases. We raised the timeout threshold per query from 3 minutes to 2.5 hours, but we still see significant problems. Some systems run out of memory. To keep the total test manageable, we reduced the total number of trials from 300 to 10. Besides the query times, we also report how often a database ran out of memory (OOM) or timed out. TigerGraph is the only database with no OOM or timeout instances.

Table 9 - Three-hop path query time

Dataset	Measure	TigerGraph	Neo4j	Neptune1	Neptune2	JanusGraph	ArangoDB.r	ArangoDB.m
graph500	time (sec)	0.41	51.38	2.12	2.27	1846.71	3461.34	1854.87
	normalized	1	125	5	5	4477	8391	4497
	% OOM	0	0	80%	80%	0	0	0
	% timeouts	0	0	0	0	20%	50%	30%
twitter	time (sec)	6.73	298.0	38.2	38.7	4324.0	3901.6	N/A
	normalized	1	44	6	6	642	580	N/A
	% OOM	0	0	90%	90%	0	0	N/A
	% timeouts	0	60%	0	0	50%	80%	N/A

Because the range of query times varies by three orders of magnitude, the vertical axis in the figure below uses a log scale.

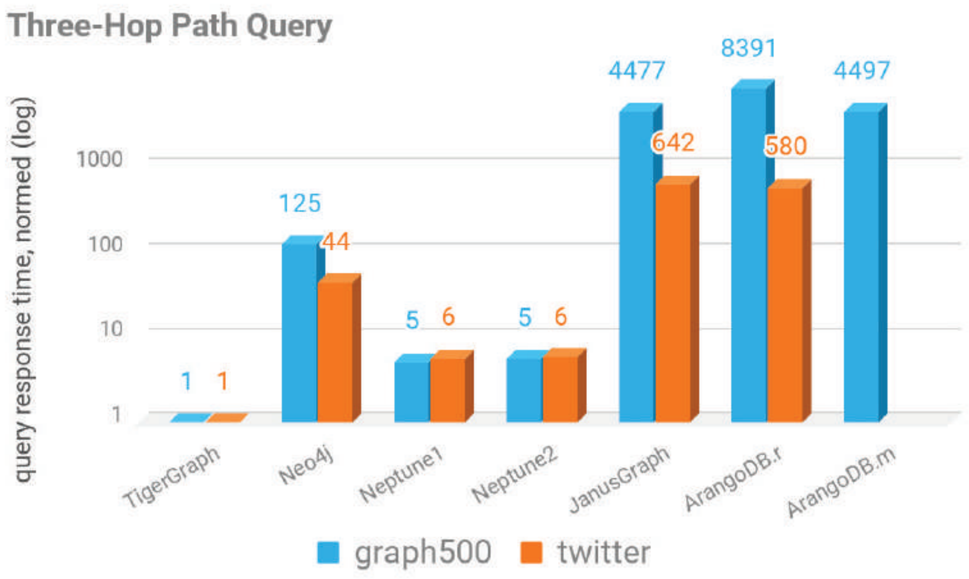


Figure 5 - Normalized three-hop path query time

To really see the limits of the databases, we increase the path length to 6 hops. TigerGraph ran the 6-hop queries on both datasets with no problem. Both Neptunes ran out of memory. Queries on JanusGraph and ArangoDB failed due to timeout.

Table 10 - Six-hop path query time

Dataset	Measure	TigerGraph	Neo4j	Neptune1	Neptune2	JanusGraph	ArangoDB.r	ArangoDB.m
graph500	time (sec)	1.78	1312.7	N/A	N/A	N/A	N/A	N/A
	normalized	1	737	N/A	N/A	N/A	N/A	N/A
	% OOM	0	0	100%	100%	0	0	0
	% timeouts	0	80%	0	0	100%	100%	100%
twitter	time (sec)	63.06	N/A	N/A	N/A	N/A	N/A	N/A
	normalized	1	N/A	N/A	N/A	N/A	N/A	N/A
	% OOM	0	0	100%	100%	0	0	N/A
	% timeouts	0	100%	0	0	100%	100%	N/A

- *TigerGraph is 2x to more than 60x faster than the other graph DBs on the 1-hop path query.*
- *On the 2-hop path query, Neptune, JanusGraph, and ArangoDB sometimes failed due to timeout. Looking only at the trials where a database was fast enough to complete the test, TigerGraph is 40x to over 300x faster than other graph DBs.*
- *On the 3-hop path query, only TigerGraph and Neo4j could complete all trials on the smaller dataset. Only TigerGraph could complete all trials on the larger dataset. Looking only at the cases where a database completed at least 50% of the trials, TigerGraph is 125x to over 4000x times faster than others.*
- *Amazon Neptune ran out of memory in 17 out of 20 test cases for 3-hops and in 100% of test cases for 6-hops.*
- *Only TigerGraph could complete the 6-hop path query (in 1.8 secs on the small graph and 63 secs on the large graph).*

Weakly Connected Component and PageRank Queries

Full graph queries examine the entire graph and compute results which describe the characteristics of the graph. Two such queries are weakly connected component labeling and PageRank. A weakly connected component (WCC) is the maximal set of vertices and their connecting edges which can reach one another, if the direction of directed edges is ignored. The WCC query finds and labels all the WCCs in a graph. This query requires that every vertex and every edge be traversed.

PageRank is an iterative algorithm which traverses every edge during every iteration and computes a score for each vertex. After several iterations, the scores will converge to steady state values. For our experiment, we run 10 iterations.

For TigerGraph, we implemented each algorithm in the GSQL language.

For Neo4j, we use the Neo4j native WCC and PageRank implementations provided in the Neo4j AOPC procedures (release 3.4.0.1).

For Amazon Neptune, we did not find any way to run algorithmic queries like WCC and PageRank because Neptune currently only supports declarative queries. Specifically, Neptune does not support VertexProgram from Gremlin OLAP API. So, without a library of built-in graph functions or a language which supports procedural computation, Neptune could not be included in this part of the benchmark testing.

JanusGraph has a native PageRank implementation, but not for WCC. So, we used the Gremlin OLAP API to write WCC.

For ArangoDB, we use their build-in WCC and PageRank queries.

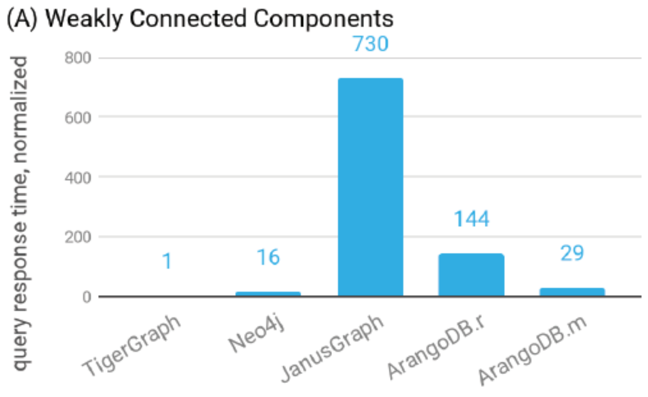
Table 11 - Weakly connected component query time

Dataset	Measure	TigerGraph	Neo4j	JanusGraph	ArangoDB.m	ArangoDB.r
graph500	time (sec)	3.08	49.97	2246.54	87.77	442.89
	relative time	1	16.2	729.8	28.5	143.9
twitter	time (sec)	74.06	1093.26	too slow[6]	no data[7]	too slow[8]
	relative time	1	14.8			

6 The first iteration did not complete in 24 hours.
7 The twitter graph could not be loaded.
8 Did not complete in 24 hours.

Table 12 - PageRank query response time

Dataset		TigerGraph	Neo4j	JanusGraph	ArangoDB.m	ArangoDB.r
graph500	time (sec)	12.52	30.27	2610.72	119.43	582.62
	relative time	1	2.4	208.6	9.5	46.6
twitter	time (sec)	265.36	614.96	too slow[9]	no data[10]	too slow[11]
	relative time	1	2.3			



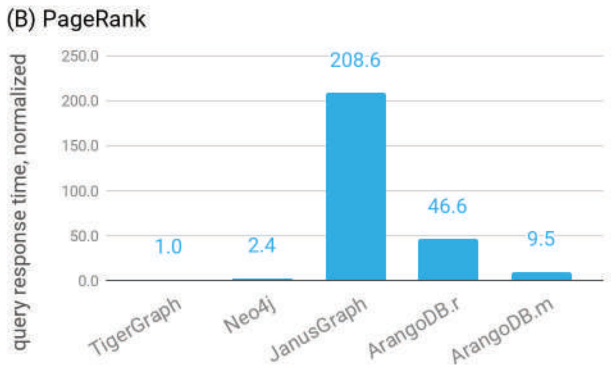


Figure 5 - Normalized query response times for (A) WCC and (B) PageRank

- ***Amazon Neptune does not provide native capability to run analytical queries.***
- ***JanusGraph and ArangoDB could not finish WCC or PageRank within 24 hours on the larger graph.***
- ***TigerGraph is about 15x faster than Neo4j for WCC, and 28x to more than 700x faster than the other graph DBs, if they finished.***
- ***TigerGraph is about 2.3x faster than Neo4j for PageRank, and 10x to more than 200x faster than the other graph DBs, if they finished.***

Scalability of Query Response Times

In the previous tests, we considered single-machine performance. We now look at how TigerGraph's performance scales as the data are distributed across a cluster. In this test, we examine how the increasing the number of compute servers affects query performance. For this test, we used a more economical Amazon EC2 instance type (r4.2xlarge: 8 vCPUs, 61GiB memory, and 200GB attached GP2 SSD). When the twitter dataset (compressed to 9.5 GB by TigerGraph) is distributed across 1 to 8 machines, 60GiB memory and 8 cores per machine is more than enough. For larger graphs or for higher query throughput, more cores may help; TigerGraph provides settings to tune memory and compute resource utilization. Also, to run on a cluster, we switched from the TigerGraph Developer Edition (v2.1.4) to the equivalent Enterprise Edition (v2.1.6).

9 The first iteration did not complete in 24 hours.
10 The twitter graph could not be loaded.
11 Did not complete in 24 hours.

We used the Twitter dataset and ran the PageRank query for 10 iterations. We repeated this three times and averaged the query times. We repeated the tests for clusters containing 1, 2, 4, 6, and 8 machines. For each cluster, the twitter graph was partitioned into equally-sized segments across all the machines being used.

Table 13 - Scalability of TigerGraph average query response times

No.of machines	1	2	4	6	8
average query time (sec)	969.8	535.5	263.4	209.1	144.8
speedup	1.0	1.81	3.68	4.64	6.70

PageRank is an iterative algorithm which traverses every edge during every iteration. This means there is much communication between the machines, with information being sent from one partition to another. Despite this communication overhead, TigerGraph's Native MPP Graph database architecture still succeeds in achieving a 6.7x speedup with 8 machines.

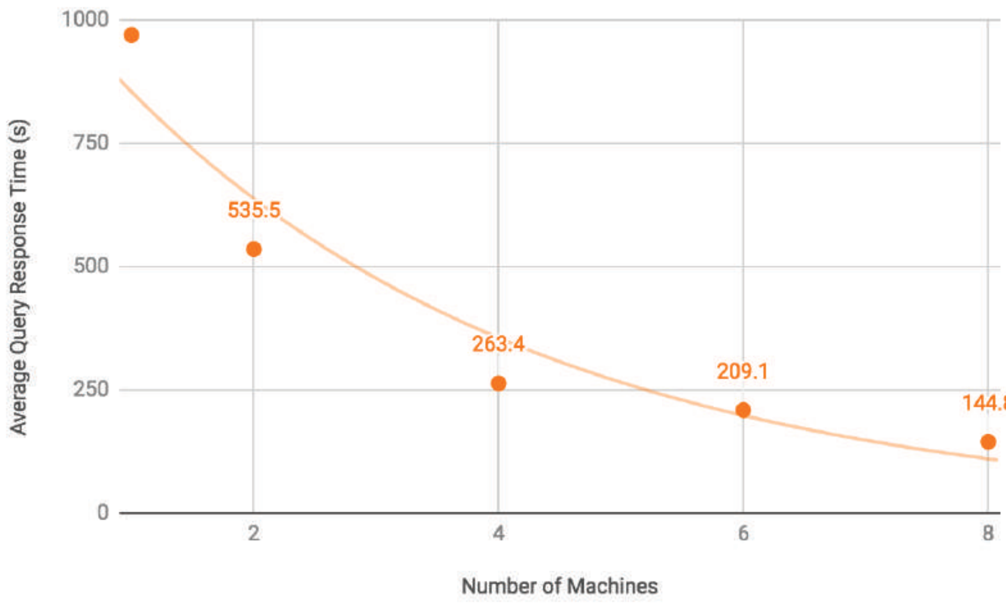


Figure 6 - TigerGraph query response time vs. Number of machines

We could not perform the scalability test on Neo4j or Amazon Neptune. Neo4j must store the full graph on a single server and cannot partition a graph across multiple machines. Amazon Neptune also cannot partition a graph across multiple machines, nor could we find a way to run PageRank. We have not yet attempted the scalability test on JanusGraph or ArangoDB Enterprise Edition[12].

12 ArangoDB Community Edition will not perform well when sharded, as noted here https://www.arangodb.com/why-arangodb/arangodb-enterprise/arangodb-enterprise-smart-graphs/

REPRODUCIBILITY

All of the files needed to reproduce the benchmark tests (datasets, queries, scripts, input parameters, result files, and general instructions) are available on GitHub:
https://github.com/tigergraph/ecosys/tree/benchmark/benchmark/

The machine and system software specifications are given in Section 1.

If you have questions or feedback regarding these tests, please contact us at benchmark@tigergraph.com.

About TigerGraph

TigerGraph is the world's fastest and most scalable graph analytics platform, powered by Native MPP Graph to deliver real-time deep link analytics (Massively Parallel Processing) technology. TigerGraph fulfills the true promise and benefits of the graph platform by tackling the toughest data challenges in real time, no matter how large or complex the dataset. TigerGraph supports applications such as IoT, AI and machine learning to make sense of ever-changing big data. For more information, follow the company on Twitter @TigerGraphDB or visit www.tigergraph.com.

Rev. Sep-2018-A

TigerGraph, Inc. | www.tigergraph.com | (650) 206-8888

3 Twin Dolphin Drive, Suite 225, Redwood City, CA 94065