A Performance Evaluation of Open Source Graph Databases

Robert McColl David Ediger Jason Poovey Dan Campbell David A. Bader
Georgia Institute of Technology

Abstract

With the proliferation of large, irregular, and sparse relational datasets, new storage and analysis platforms have arisen to fill gaps in performance and capability left by conventional approaches built on traditional database technologies and query languages. Many of these platforms apply graph structures and analysis techniques to enable users to ingest, update, query, and compute on the topological structure of the network represented as sets of edges relating sets of vertices. To store and process Facebook-scale datasets, software and algorithms must be able to support data sources with billions of edges, update rates of millions of updates per second, and complex analysis kernels. These platforms must provide intuitive interfaces that enable graph experts and novice programmers to write implementations of common graph algorithms. In this paper, we conduct a qualitative study and a performance comparison of 12 open source graph databases using four fundamental graph algorithms on networks containing up to 256 million edges.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Parallel databases

Keywords graph databases, graph algorithms, relational databases

1. Background

In the context of this paper, the term *graph database* is used to refer to any storage system that can contain, represent, and query a graph consisting of a set of vertices and a set of edges relating pairs of vertices. This broad definition encompasses many technologies. Specifically, we examine schemas within traditional disk-backed, ACID-compliant SQL databases (SQLite, MySQL, Oracle, and Microsoft SQL Server), modern NoSQL databases and graph databases (Neo4j, OrientDB, InfoGrid, Titan, FlockDB, ArangoDB, InfiniteGraph, AllegroGraph, DEX, GraphBase, and HyperGraphDB), distributed graph processing toolkits based on MapReduce, HDFS, and custom BSP engines (Bagel, Hama, Giraph, PEGASUS, Faunus), and in-memory graph packages designed for massive shared-memory (NetworkX, Gephi, MTGL, Boost, uRiKA, and STINGER). For all of these, we construct a table containing the package maintainer, license, platform, implementation language(s), features, cost, transactional capabilities, memory vs. disk storage, single-node vs. distributed, text-based query language support, built-in algorithm support, and primary traversal and query styles supported. An abridged version of this table is presented in Table 1.

For a selected group of primarily open source graph databases, we implement and test a set of representative fundamental graph kernels on each technology. Although this work focuses on open source software, proprietary technologies are included where appropriate for comparison. These kernels include shortest path algorithms, iterative propagation algorithms, and updates to the graph data structure. Using identical high-end computing hardware and input data sets, we measure the performance and scalability of these algorithms for each graph database. In the spirit of full disclosure, several authors are affiliated with the ongoing design and development of STINGER [7].

1.1 Experimental Design

Previous work has compared Neo4j and MySQL using simple queries and breadth-first search [11], and we are inspired by a combination objective/subjective approach to the report. Graph primitives for RDF query languages were extensively studied in [1] and data models for graph databases in [2], which are beyond the scope of this study.

1.2 Algorithms and Approach

Four fundamental graph kernels are selected with requirements placed on how each must be implemented to emphasize common programming styles of graph algorithms. The first is the Single Source Shortest Paths (SSSP) problem, which is specifically required to be implemented as a level-synchronous parallel breadth-first traversal of the graph. Single source shortest paths are used for routing and connec-

PPAA '14, February 16, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2654-4/14/02. . . $15.00.
http://dx.doi.org/10.1145/2567634.2567638

	Owner / Maintainer	License	Platform	Language	Distribution	Cost	Transactional	Memory-Based	Disk-Based	Single-Node	Distributed	Graph Algorithms	Text-Based Query Language	Embeddable	Software	Data Store	Type
MySQL	Oracle	GPL/Proprietary	x86	C/C++	Bin	Free	X	-	X	X	-	-	SQL	X	X	X	SQLDB
Oracle	Oracle	Proprietary	x86	C/C++	Bin	$180-$950	X	-	X	X	X	-	SQL	X	X	X	SQLDB
SQL Server	Microsoft	Proprietary	x86-Win	C++	Bin	$898-$8592	X	-	X	X	-	-	SQL	-	X	X	SQLDB
SQLite	Richard Hipp	Public Domain	x86	C	Src/Bin	Free	X	X	X	X	-	-	SQL	X	X	X	SQLDB
AllegroGraph	Franz, Inc.	Proprietary	x86	Likely Java	Bin	Free-ish/$$$$	X	-	X	X	-	X	SPARQL,RDFS++,Prolog	-	X	X	GDB
ArangoDB	ArangoDB	Apache	x86	C/C++/JS	Src/Bin	Free	-	-	X	X	-	-	AQL	-	X	X	GDB/KV/DOC
DEX	Sparsity-Technologies	Proprietary	x86	C++	Bin	Free Personal/Commercial $$	X	-	X	X	-	X	Traversal	X	-	X	GDB
FlockDB	Twitter	Apache	Java	Java,Scala,Ruby	Src	Free	-	-	X	X	X	-	-	-	X	X	GDB
GraphBase	FactNexus	Proprietary	Java	Java	Bin	Free,$15/mo,$20,000	?	-	X	X	-	-	Bounds	X	X	X	GDB
HyperGraphDB	Kobrix Software	LGPL	Java	Java	Src	Free	MVCC	X	X	X	X	-	HGQuery,Traversal	X	-	-	HyperGDB
InfiniteGraph	Objectivity	Proprietary	x86/Java	Java/C++	Bin	Free Trial/$5,000	Both	-	X	X	X	-	Gremlin	X	X	X	GDB
InfoGrid	Johannes Ernst	AGPL/Proprietary	Java	Java	Src/Bin	Free + Support	-	X	X	X	X	-	-	X	X	-	GDB
Neo4j	Neo Technology	GPL/Proprietary	Java	Java	Src/Bin	Free,$6,000-$24,000	X	-	X	X	-	X	Cypher	X	X	X	GDB/NoSQL
OrientDB	NuvolaBase Ltd	Apache	Java	Java	Src/Bin	Free + Support	Both	X	X	X	X	-	Extended SQL, Gremlin	X	X	X	GDB/NoSQL
Titan	Aurelius	Apache	Java	Java	Src/Bin	Free + Support	Both	-	X	X	X	-	Gremlin	X	X	-	GDB
Bagel	UC Berkley	BSD	Java	Java/Scala/Spark	Src	Free	-	X	-	X	X	X	-	X	-	-	BSP
BGL	Boost / IU	Boost	x86 / C++	C++	Src/Bin	Free	-	X	-	X	-	X	-	X	-	-	Library
Faunus	Aurelius	Apache	Java	Java	Src	Free + Support	Both	-	X	X	X	-	Gremlin	X	X	-	Hadoop
Gephi	Gephi Consortium	GPL/CDDL	Java	Java,OpenGL	Src/Bin	Free	-	X	-	X	-	X	-	X	X	-	Toolkit
Giraph	Apache	Apache	Java	Java	Src	Free	-	X	-	X	X	X	-	X	-	-	BSP
GraphLab	GraphLab, Inc.	Apache	x86	C++	Src	Free	-	X	X	X	X	X	-	X	-	-	BSP
GraphStream	University Le Havre	LGPL/CeCILL-C	Java	Java	Src/Bin	Free	-	X	-	X	-	X	-	X	-	-	Library
Hama	Apache	Apache	Java	Java	Src	Free	-	X	-	X	X	X	-	X	-	-	BSP
MTGL	Sandia NL	BSD	x86/XMT	C++	Src	Free	-	X	-	X	-	X	-	X	-	-	Library
NetworkX	Los Alamos NL	BSD	x86	Python	Src/Bin	Free	-	X	-	X	-	X	-	X	-	-	Library
PEGASUS	CMU	Apache	Java	Java	Src/Bin	Free	-	-	X	X	X	X	-	-	X	-	Hadoop
STINGER	GT / GTRI	BSD	x86/XMT	C	Src	Free	-	X	-	X	-	X	-	X	X	X	Library
uRiKA	Yarc Data / Cray	Proprietary	XMT	Likely C++	Bin	$$$$	?	X	-	X	-	-	SPARQL	-	X	X	Appliance

Table 1: Graph database comparison matrix

tivity, and all-pairs shortest paths computations are building blocks for algorithms such as betweenness centrality [4, 8].

The second test case is an implementation of the Shiloach-Vishkin connected components algorithm (SV) [10] in the edge-parallel label-pushing style. Connected components is a global graph metric that touches all edges in the graph. The Shiloach-Vishkin algorithm is not theoretically work efficient, but the edge-parallel access pattern is representative of a broad class of graph algorithms. This algorithm is read-heavy with sparse writes and a high degree of random access.

The third test case is PageRank (PR) [5] in the vertex-parallel Bulk Synchronous Parallel (BSP) power-iteration style. PageRank is a measure of influence and importance in the network. The algorithm for computing PageRank is representative of a broad class of iterative graph algorithms.

The final test case is the performance of a set of edge insertions and deletions in parallel to represent random access and modification of the structure. Real-world networks are in constant motion as new edges and vertices enter the graph. This test case measures the performance of change in the data structure.

When it is not possible to meet the algorithm requirements due to restrictions of the software framework, the algorithms are implemented in as close of a manner as possible. When the framework includes implementations of any of these algorithms, the implementation is used if it meets the requirements. The implementations are intentionally written in a straightforward manner with no manual tuning or optimization to emulate non-hero programmers.

Many real-world networks demonstrate "scale-free" characteristics [3, 12]. To emulate these characteristics in graphs that are consistent, can be easily created and recreated, and can be built to a specified size for the purposes of our benchmarking, we chose to use the Recursive MATrix (R-MAT) generator. Four initial sparse, static graphs and corresponding sets of edge insertions and deletions are created using an R-MAT [6] generator with parameters $A = 0.55$, $B = 0.1$, $C = 0.1$, and $D = 0.25$). These graphs contain 1K (tiny), 32K (small), 1M (medium), and 16M (large) vertices with 8K, 256K, 8M, and 256M undirected edges, respectively. Graphs with over 1B edges were considered, but while a small number of the packages examined can handle graphs at that scale, it was determined that many could not handle that scale. This was even true of those claiming to reach massive scale through distributed disk-based storage. As such, for the purposes of our comparison, there was no need to use these larger graphs. Edge updates have a 6.25% probability of being deletions uniformly selected from existing edges where deletions are handled in the default manner for the package. For the tiny and small graphs, 100K updates were performed, and 1M updates were performed otherwise.

The quantitative measurements taken are initial graph construction time, total memory in use by the program upon completion, update rate in edges per second, and time to completion for Single Source Shortest Paths, Shiloach-Vishkin, and PageRank. Tests for single-node platforms and databases are run on a system with four AMD Opteron 6282 SE processors and 256GB of DDR3 RAM. Distributed system tests are performed on a cluster of 21 systems with minimum configuration of two Intel Xeon X5660 processors with 24GB DDR3 RAM connected by QDR Infiniband. When applicable, the Hadoop Distributed File System (HDFS) was run in-memory.

2. Qualitative Observations

2.1 User Experience

A common goal of open source software is widespread adoption. Adoption promotes community involvement, and an active community is likely to contribute use cases, patches, and new features. When a user downloads a package, the goal should be to build as quickly as possible with minimal input from the user. A number of packages tested during the course of this study did not build on the first try. Often, this was the result of missing libraries or test modules that do not pass, and was resolved with help from the forums. This was specifically true of Giraph, which failed a test during the build process. The easiest to install were those that were self-contained. Even packages that relied on a package management system (NetworkX in Python, most of the Java packages which relied on Maven) were more difficult to install than their authors intended as the machines they were installed on inside of our cluster did not have direct internet access. Build problems may reduce the likelihood that users continue with a project.

Among both commercial and open source software packages, we find a lack of consistency in approaches to documentation. It is often difficult to quickly get started. System requirements and library dependencies should be clear. Quickstart guides are appreciated. We find that most packages lack adequate usage examples. Common examples illustrate how to create a graph and query the existence of vertices and edges. Examples are needed that show how to compute graph algorithms, such as PageRank or connected components, using the provided primitives. The ideal example demonstrates data ingest, a real analysis workflow that reveals knowledge, and the interface that an application would use to conduct this analysis. In this regard, NetworkX is very well documented. Titan's documentation is extremely complete in terms of API documentation, but lacking in examples of real-world usage, which was a common theme. Some packages lack formal documentation or have fragmented documentation (STINGER, Bagel).

While not unique to graphs, there are a multitude input file formats that are supported by the software packages in this study. Formats can be built on XML, JSON, CSV, or other proprietary binary and plain-text formats. Each has a trade-off between size, descriptiveness, and flexibility. The

number of different formats creates a challenge for data interchange. Formats that are edge-based, delimited, and self-describing can easily be parsed in parallel and translated between each other.

2.2 Developer Experience

Object-Oriented Programming (OOP) was introduced to enforce modularity, increase reusability, and add abstraction to enable generality. These are important goals; however, OOP can also inadvertently obfuscate logic and create bloated code. For example, considering a basic PageRank computation in Giraph, the function implementing the algorithm uses approximately 16% of the 275 lines of code. The remainder of the code registers various callbacks and sets up the working environment. Although the extra code provides flexibility, it can be argued that much is boilerplate. The framework should support the programmer spending as little time as possible writing boilerplate and configuration so that the majority of code is a clear and concise implementation of the algorithm.

A number of graph databases retain many of the properties of a relational database, representing the graph edges as a key-value store without a schema. A variety of query languages are employed. Like their relational cousins, these databases are ACID-compliant and disk-backed. Some ship with graph algorithms implemented natively. While these query languages may be comfortable to users coming from a database background, they are rarely sufficient to implement even the most basic of graph algorithms succinctly. The better query languages are closer to full scripting environments, which may indicate that query languages are not sufficient for graph analysis.

In-memory graph databases focus more on the algorithms and rarely provide a query language. Access to the graph is done through algorithms or graph-centric APIs. This affords the user the ability to write nearly any algorithm, but presents a certain complexity and learning curve. The visitor pattern in MTGL is a strong example of a graph-centric API. The BFS visitor allows the user to register callbacks for newly visited vertices, newly discovered BFS tree edges, and edges to previously discovered vertices. Given that breadth-first traversals are frequently used as building blocks in common graph algorithms, providing an API that performs this traversal over the graph in an optimal way can abstract the specifics of the data structure while giving performance benefit to novice users. Similar primitives might include component labelings, matchings, independent sets, and others (many of which are provided by NetworkX). Additionally, listening mechanisms for dynamic algorithms should be considered in the future.

2.3 Graph-specific Concerns

An important measurement of performance is the size of memory consumed by the application processing a graph of interest. An appropriate measurement of efficiency is the

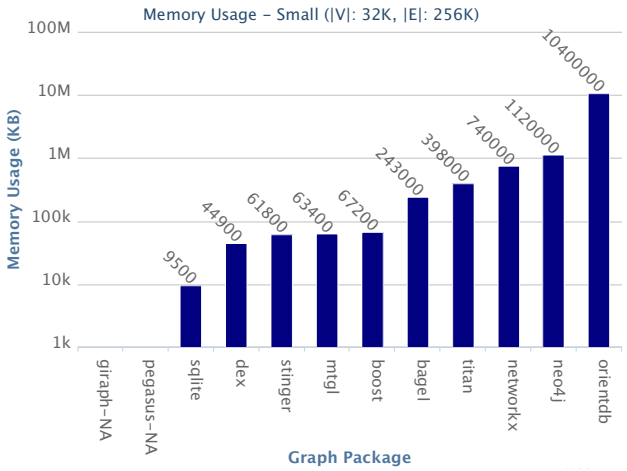
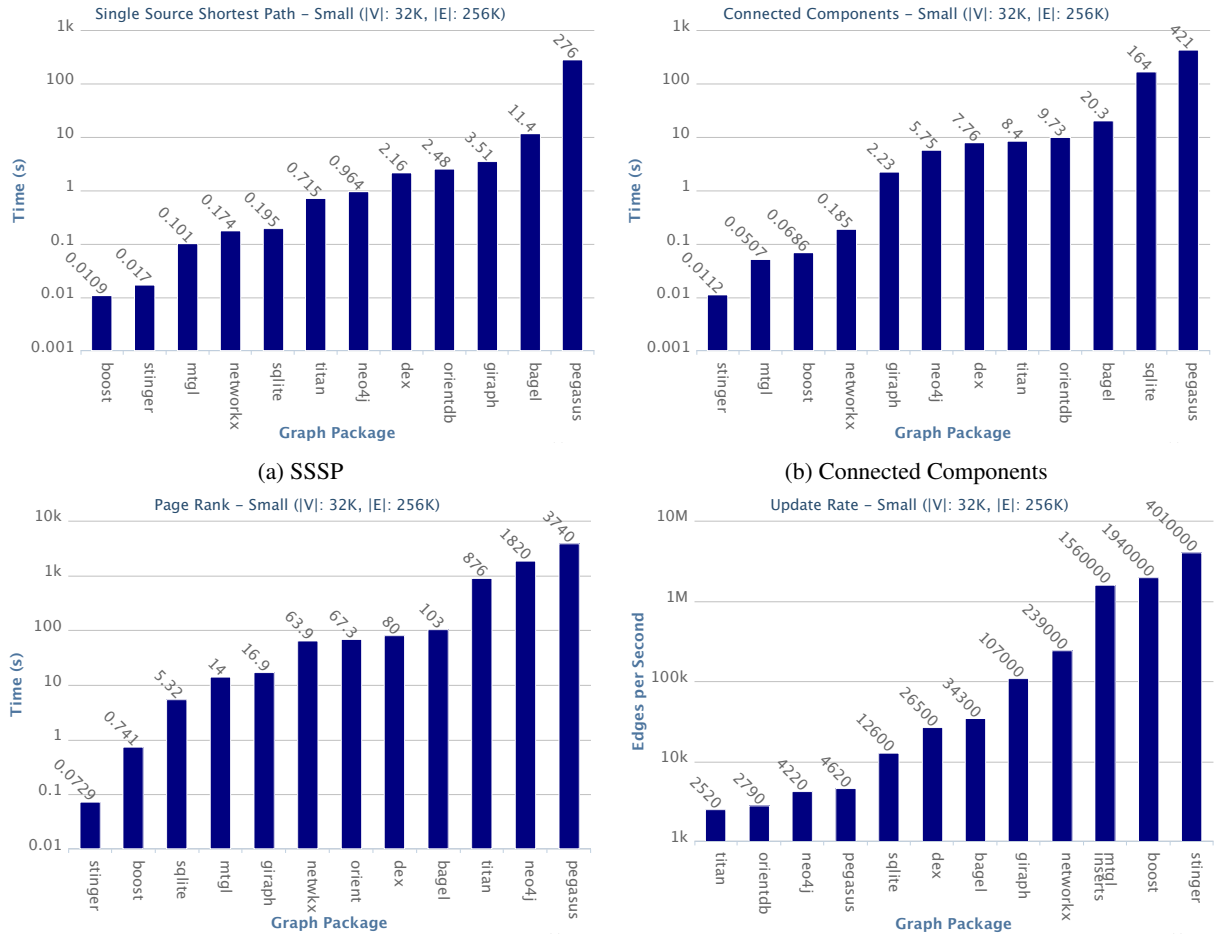


Figure 1: The memory occupied by the program after performing all operations on the small graph (32K vertices and 256K undirected edges) for each graph analysis package.

number of bytes of memory required to store each edge of the graph. The metadata associated with vertices and edges can vary by software package. The memory usage of each graph database on the small graph with 256K edges is plotted in Figure 1. The most efficient package, SQLite consumes only 37 bytes per edge. OrientDB uses over 40KB per edge. Boost represents the median at 950 bytes per edge. The largest test graph contained more than 128 million edges, or about one gigabyte in 4-byte tuple form. Graph databases were run on machines with 256GB of memory. However, a number of software packages could not run on the largest test graph (neo4j, OrientDB).

It is often possible to store large graphs on disk, but the working set size of the algorithm in the midst of the computation can overwhelm the system. For example, a breadth-first search will often contain an iteration in which the size of the frontier is as many as half of the vertices in the graph. We find that some graph databases can store these graphs on disk, but cannot compute on them because the in-memory portion of the computational is too large.

We find differing semantics regarding fundamental operations on graphs. For example, when an edge inserted already exists, there are several possibilities: 1) do nothing, 2) insert another copy of that edge, 3) increment the edge weight of that edge, and 4) replace the existing edge. Graph databases may support one or more of these semantics. The same can be said for edge deletion and the existence of self-edges. Given that there is no consensus among the packages on how these operations should operate on the graph, the documentation of all packages should be more explicit about how insert, update, and remove operations affect the data structure itself and what the implications in terms of storage and traversal are. It may also be valuable for the community as a



(a) SSSP

(b) Connected Components

(c) PageRank

(d) Updates per second - Deletions for MTGL are ignored.

Figure 2: Time taken to compute benchmark algorithms on the small graph (32K vertices and 256K undirected edges).

whole to study the algorithmic and informatic implications of these design choices.

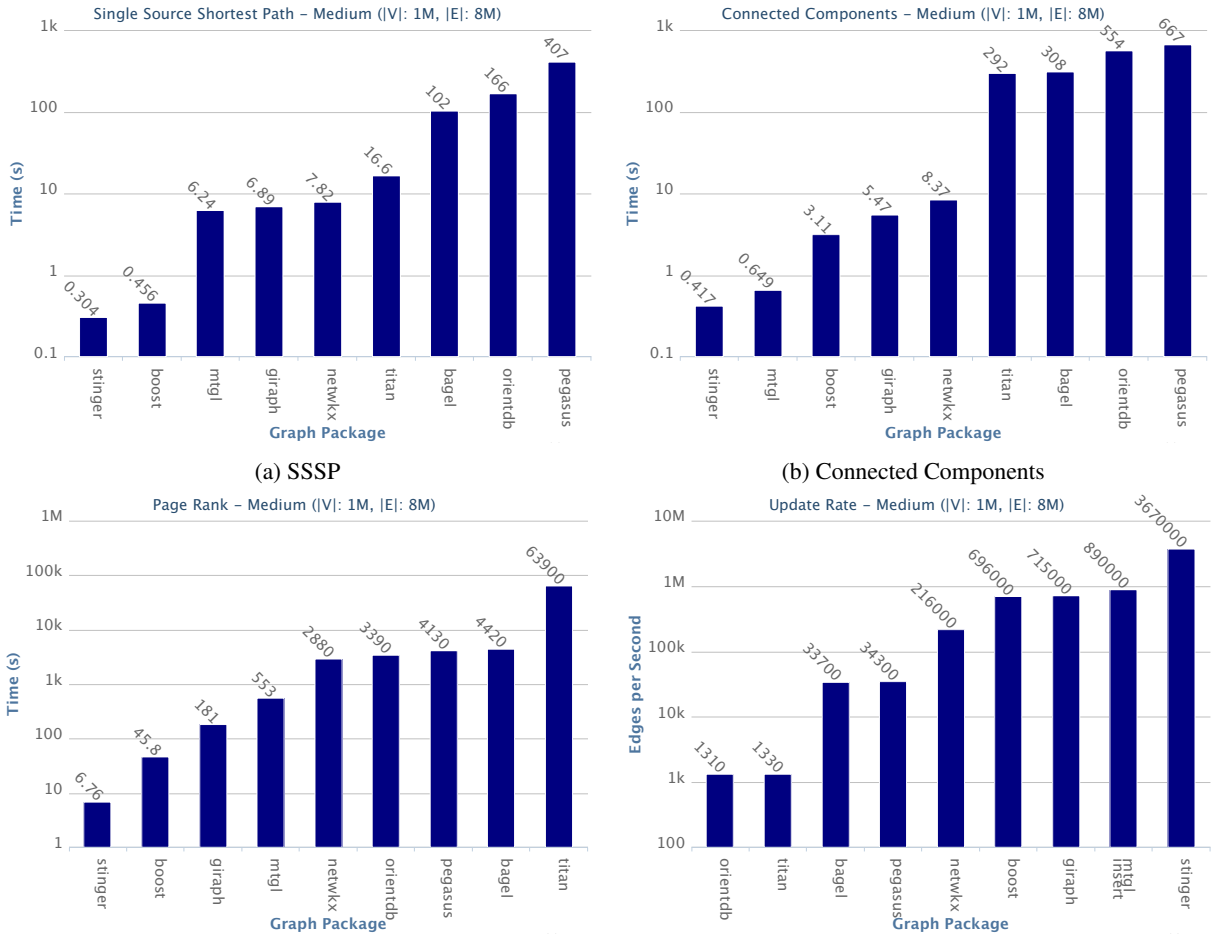
A common query access pattern among graph databases is the extraction of a subgraph, egonet, or neighborhood around a vertex. While answering some questions, a more flexible interface allows random access to the vertices and edges, as well as graph traversal. Graph traversal is a fundamental component to many graph algorithms. In some cases, random access to graph edges enables cleaner implementation of an algorithm with better workload balance or communication properties.

3. Performance Results

We compute four algorithms (single source shortest path (SSSP), connected components (SV), PageRank (PR), and update) for each of four graphs (tiny, small, medium, and large) using each graph package. Developer-provided implementations were used when available. Otherwise, implementations were created using the best algorithms, although no heroic programming or optimization were done. The source code used for these tests and raw results are available at `https://github.com/robmccoll/graphdb-testing`. For a complete collection of results plots, please refer to [9].

Tests for single-node platforms and databases are conducted on a system with four AMD Opteron 6282 SE processors and 256GB of DDR3 RAM. Distributed system tests are performed on a cluster of 21 systems with minimum configuration of two Intel Xeon X5660 processors with 24GB DDR3 RAM connected by QDR Infiniband. Apache HDFS was run in-memory. Memory usage for Java applications was taken from the MemoryMXBean memory management interface summing the heap and non-heap usage.

All 12 open source packages completed both tiny and small input datasets. The small graph is an undirected graph with 32K vertices and 256K edges. Execution times and performance results for all packages are plotted in Figure 2. There are four orders of magnitude difference in performance from top to bottom for each benchmark. STINGER, MTGL, and Boost consistently rank among the highest performers. NetworkX is the only Python package in the set, but



(a) SSSP

(b) Connected Components

(c) PageRank

(d) Updates per second - Deletions for MTGL are ignored.

Figure 3: Time taken to compute benchmark algorithms on the medium graph (1M vertices and 8M undirected edges).

outperforms the various Java-based packages in all but the PageRank test. SQLite is a traditional relational database, but performs well in the SSSP and PageRank tests. However, updates and connected components were among the lowest performers. Giraph was the highest performing Java implementation in all but the SSSP tests.

Increasing the number of edges by a factor of 32 to approximately 8 million edges, only 9 of the original 12 packages successfully completed all of the benchmark tests. Execution times and performance results for all packages are plotted in Figure 3. There are still four orders of magnitude difference in performance among the remaining packages. STINGER, MTGL, Boost, Giraph, and NetworkX represent the top five performers in each of the four benchmark tests.

Increasing the number of edges an additional factor of 16 to approximately 135 million edges, only 5 of the original 12 packages successfully completed all of the benchmark tests. Execution times and performance results for all packages are listed in Table 2. The performers of the large tests are four out of top five performers of the medium tests

	SSSP	SV	PR	Update
Boost	11.4	78.2	989	865,000
Giraph	29.6	47.6	2,950	280,000
MTGL	127	16.2	14,200	246,000
Pegasus	826	2,390	4,680	11,200
STINGER	7.35	5.52	161	2,530,000

Table 2: Algorithm performance results on the large graph (16M vertices and 128M undirected edges). Update results are in updates per second. All others in seconds. Packages not shown could not complete the benchmark.

and Pegasus, which runs on top of Apache Hadoop. In most circumstances, the PageRank computation is at least one order of magnitude slower than both other algorithms. Both PageRank and Shiloach-Vishkin are iterative algorithms, although the number of iterations for PageRank is limited to 30, while Shiloach-Vishkin often converges in fewer than 10 iterations on scale-free networks. All four algorithms are

known to have large numbers of irregular memory accesses, which can lead to poor cache performance.

4. Conclusions

Graph-based approaches to big data are emerging as a hot topic, and many open source efforts are under way. The goal of each of these efforts is to extract knowledge from the data in a more flexible manner than a relational database. Many approaches to graph databases build upon and leverage the long history of relational database research.

Our position is that graph databases must become more "graph aware" in data structure and query language. The ideal graph database should understand analytic queries that go beyond neighborhood selection. In relational databases, the index represents pre-determined knowledge of the structure of the computation without knowing the specific input parameters. A relational query selects a subset of the data and joins it by known fields. The index accelerates the query by effectively pre-computing a portion of the query.

In a graph database, the equivalent index is the portion of an analytic or graph algorithm that can be pre-computed or kept updated regardless of the input parameters of the query. Examples include the connected components of the graph, algorithms based on spanning trees, and vertex statistics such as PageRank or clustering coefficients. A sample query might ask for shortest paths vertices, or alternatively the top k vertices in the graph according to PageRank. Rather than compute the answer on demand, maintaining the analytic online results in lower latency responses. While indices over the properties of vertices may be convenient for certain cases, these types of queries are served equally well by traditional SQL databases.

While many software applications are backed by databases, most end users are unaware of the SQL interface between application and data. SQL is not itself an application. Likewise, we expect that NoSQL is not itself an application. Software will be built atop NoSQL interfaces that will be hidden from the user. The successful NoSQL framework will be the one that enables algorithm and application developers to implement their ideas easily while maintaining high levels of performance.

Visualization frameworks are often incorporated into graph databases. The result of a query is returned as a visualization of the extracted subgraph. State-of-the-art network visualization techniques rarely scale to more than one thousand vertices. We believe that relying on visualization for query output limits the types of queries that can be asked. Queries based on temporal and semantic changes can be difficult to capture in a force-based layout. Alternative strategies include visualizing statistics of the output of the algorithms over time, rather than the topological structure of the graph.

With regard to the performance of disk-backed databases, transactional guarantees may unnecessarily reduce performance. It could be argued that such guarantees are not necessary in the average case, especially atomicity and durability. For example, if the goal is to capture and analyze a stream of noisy data from Twitter, it may be acceptable for an edge connecting two users to briefly appear in one direction and not the other. Similarly, in the event of a power failure, the loss of a few seconds of data may not significantly change which vertices have the highest centrality on the graph. Some of the databases presented here seem to have reached this realization and have made transactional guarantees optional.

References

[1] R. Angles and C. Gutierrez. Querying RDF data from a graph database perspective. In *The Semantic Web: Research and Applications*, volume 3532 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26124-7.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008. ISSN 0360-0300.

[3] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, Oct. 1999.

[4] U. Brandes. A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163–177, 2001.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998. ISSN 0169-7552. Proceedings of the Seventh International World Wide Web Conference.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*, Orlando, FL, Apr. 2004. SIAM.

[7] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High performance data structure for streaming graphs. In *The IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, Sept. 2012. Best paper award.

[8] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[9] R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A brief study of open source graph databases. *CoRR*, abs/1309.2675, 2013.

[10] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

[11] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, pages 42:1–42:6, 2010.

[12] D. Watts and S. Strogatz. Collective dynamics of small world networks. *Nature*, 393:440–442, 1998.