Performance Introspection of Graph Databases

Peter Macko
Harvard University
pmacko@eecs.harvard.edu

Daniel Margo
Harvard University
dmargo@eecs.harvard.edu

Margo Seltzer
Harvard University
margo@eecs.harvard.edu

ABSTRACT

The explosion of graph data in social and biological networks, recommendation systems, provenance databases, etc. makes graph storage and processing of paramount importance. We present a performance introspection framework for graph databases, PIG, which provides both a toolset and methodology for understanding graph database performance. PIG consists of a hierarchical collection of benchmarks that compose to produce performance models; the models provide a way to illuminate the strengths and weaknesses of a particular implementation. The suite has three layers of benchmarks: primitive operations, composite access patterns, and graph algorithms. While the framework could be used to compare different graph database systems, its primary goal is to help explain the observed performance of a particular system. Such introspection allows one to evaluate the degree to which systems exploit their knowledge of graph access patterns. We present both the PIG methodology and infrastructure and then demonstrate its efficacy by analyzing the popular Neo4j and DEX graph databases.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Measurement techniques, Modeling techniques

General Terms

Measurement, Performance

Keywords

graphs; databases; performance introspection

1. INTRODUCTION

The explosion in social networking over the past five years has produced interest and demand for systems that store and query large graphs. However, the importance of graph data is not limited to social networking: telecommunications companies have been studying phone and network graphs for years; biologists study protein interactions and neural pathways, frequently represented as graphs; marketing professionals mine data about relationships between people, the media they use, and the products they buy. Often these data are too large to manipulate exclusively in main memory without large compute clusters. As these applications have grown in importance, so has interest in graph databases.

We use the term *graph database* to embody two characteristics: First, a graph database is a storage system whose native representation of data is in terms of objects (vertices) and inter-object relationships (edges). Second, a graph database supports single-object access via indexed lookup or iteration. That is, in addition to enabling whole-graph analysis, we require that a graph database be able to efficiently answer queries about the attributes and relationships of specific elements. Although graph representations are flexible enough to embed virtually any data set, in practice they are most useful for applications with "graph-like" queries. For example, in social networking, finding the relationship distance between two people is a shortest path query; listing all your friends is a neighborhood query; and testing the six degrees of separation hypothesis [11, 17] compares the results of a 6-hop neighborhood query to the original graph. These queries arise in many disciplines, and the algorithms underlying them are well studied by graph theorists.

This combination of large data and complex analyses drives the need for high-performance graph storage and query. In this context graph database performance tools are valuable for both researchers and end-users. Analysis tools should help researchers identify what costs aggregate, amortize, and dominate system performance and where it is possible to significantly improve performance. Such tools should also help users deconstruct and expose performance relationships between low-level database operations and higher-level access patterns, such that they can estimate performance for their own data sets and queries. For example, shortest-path-finding via breadth-first search is composed of operations such as per-vertex marking and neighborhood retrieval. The extent to which a graph database exploits shortest-path-like access patterns can be characterized in part by how much it amortizes the costs of these aggregate operations. In turn, shortest-path-like access patterns are a building block that partially determines the performance of many complex analyses, such as betweenness centrality.

These observations lead to our introspection approach, providing an analytical framework in which to relate primitive database operations (microbenchmarks) to graph algo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SYSTOR '13, June 30 - July 02 2013, Haifa, Israel
Copyright 2013 ACM 978-1-4503-2116-7/13/06 ...$15.00.

rithms (macrobenchmarks). Performance Introspection for Graphs (PIG) is a suite of benchmarks and a model of inter-benchmark performance relationships.

We demonstrate the use of PIG by analyzing DEX and Neo4j, the industry-standard graph databases. We reveal, for example, the performance boost due to Neo4j's object cache on small graphs and DEX's ability to perform fast inserts due to a feature that allows its users to enable weaker durability guarantees. We also find that properties are not first-class citizens and that k-hop neighborhood retrievals are not well amortized.

The rest of this paper is structured as follows. The next section discusses existing work on graph benchmarking. Section 3 explains our framework, and Section 4 presents the PIG specification. We provide implementation details in Section 5, examples of PIG in use on the Neo4j and DEX graph databases in Section 6, and conclude in Section 7.

2. BACKGROUND

PIG is based on the *property graph* model [18], in which a graph is a set of edges and vertices in which each element is uniquely identifiable and has an collection of key/value pairs called *properties* (or *attributes*). Each edge has a *label* that identifies the type of relationship it represents. While many graph analyses focus exclusively on structure, today's applications also require attributes. For example, in a social network, vertices might represent people with different names, ages, etc., and also different entity types such as organizations or articles. A database implementation that supports efficient selection of vertices by these properties will perform better than one that does not.

Our methodology is inspired by work in deconstructing operating system performance [8]. That work presented a composable micro-benchmark suite that proved invaluable in illustrating critical efficiencies and bottlenecks in different operating systems on different platforms. Our goal here is to apply a similar methodology to understand the behavior, strengths, and weaknesses of today's graph databases, and provide guidance about how to approach development of a next generation graph database.

This work is also informed by earlier graph processing benchmarks. These benchmarks, such as the HPC Scalable Graph Analysis Benchmark suite [3, 6, 9], Graph500 [2], and BG [1] produce end-to-end latency results and are complementary to PIG. PIG provides a framework designed to explain *why* a system performs as it does on these other benchmarks. LDBC [13], an EU project whose goal is to develop a realistic industry-standard graph benchmark and provide an independent authority for evaluating graph databases, and PBBS [19], a set of benchmarks for evaluating and comparing parallel systems (not just graph databases), are two more recent benchmarks under development. Neither is released, but like BG and Graph500 both appear to focus on applications rather than a framework for understanding the resulting behavior, so we expect our efforts to be synergistic.

3. METHODOLOGY

We begin our discussion by defining different levels of the framework, identifying the operations at each level, and discussing how the operations on one level relate to those on another. Then we discuss workloads including graph size and the underlying structural model describing each graph.

3.1 Benchmark Structure

In their simplest form, graph algorithms are usually described in terms of imperative, primitive per-element "get" and "set" operations. At a higher level, there are complex analytical operations such as centrality measurements [10], clustering coefficients [20], and substructure finding. Such metrics may be easy to describe declaratively, but implementing them efficiently is a subject of considerable research. These are the "representative", demanding queries that existing benchmarks measure but do not decompose.

In between low-level database operations and high-level algorithms exists an extensive set of intermediate "operations" emerging from the graph literature. For example, *Khops* is the retrieval of a vertex's neighborhood within k traversals. Many higher-level analyses can be described and implemented in terms of Khops, such as triangle finding, which is equivalent to finding vertices that are both 1 and 2-hop neighbors of the same vertex. The local clustering coefficient (LCC) is the ratio of a vertex's triangles to the size of its 2-hop neighborhood, and the average LCC across all vertices partially characterizes small-world networks [12]. Similarly, power-law graphs exhibit exponential growth in the size of their Khop neighborhoods (the "hop-plot exponent"). Thus, the ability of a graph database to handle Khops access patterns predicts its performance on higher-level analyses.

In summary, graph data access patterns divide into three approximate levels: *Micro-operations*: Low-level database API operations, such as *GetVertex/Edge* or *SetProperty*; *Graph operations*: Intermediate "operations" that aggregate API calls, such as *Get(Khop)Neighbors* or *Ingest*; and *Algorithms* that express fundamental concepts in graph algorithms, but are often not part of a graph API, such as *FindShortestPath*, *ComputeClusteringCoefficient*. Figure 1 presents the three layers of PIG, the relationships between them, and how several applications can be composed from them.

Measurements from each layer are designed to be useful in and of themselves as well as to model performance of the layers above. This lets us identify a system's precise points of success and failure in the access pattern hierarchy. For ex-

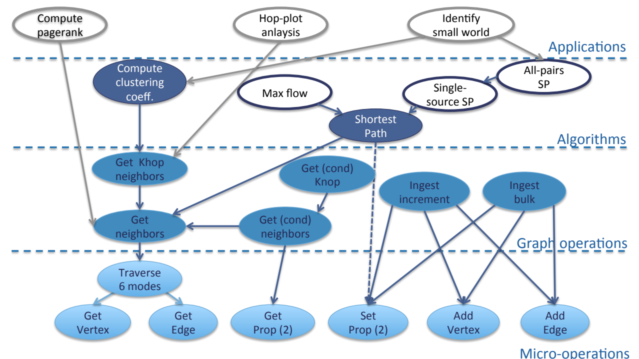

Figure 1: PIG Structure. The components at each layer compose into a model relating their performance to those in the level directly above it. Discrepancies between the model and measured performance identify strengths and weaknesses of a system. The filled ellipses show the operations supported by the benchmark.

Symbol	Description	Model
	Micro Operations	
$R_{j\in v,e,p}$	Get vertex, edge, property	primitive
$W_{j\in v,e,p}$	Set vertex, edge, property	primitive
$T_{j\in i,o,b}$	Traverse the first incoming/outgoing/either edge	$R_e + R_v$
$T^p_{j\in i,o,b}$	Traverse the incoming/outgoing/either edge conditioned on an edge property	$n(R_e + R_v + R_p)$
$T^l_{j\in i,o,b}$	Traverse the incoming/outgoing/either edge with label l	$n(R_e + R_v)$
	Graph Operations	
$N_{j\in i,o,b}$	Get all neighbors via incoming, outgoing, both edges	$n(R_v + R_e)$
$N^l_{j\in i,o,b}$	Get all neighbors via incoming, outgoing, both edges with label l	$n(R_v + R_e)$
$N^p_{j\in i,o,b}$	Get all neighbors via incoming, outgoing, both edges with property p	$n(R_v + R_e + R_p)$
$K^k_{j\in i,o,b}$	Get all Khop neighbors via incoming, outgoing, both edges	$N_j K$
$K^{kl}_{j\in i,o,b}$	Get all Khop neighbors via incoming, outgoing, both edges with label l	$N^l_j K$
$K^{kp}_{j\in i,o,b}$	Get all Khop neighbors via incoming, outgoing, both edges with property p	$N^p_j K$
S	Insert subgraph with i vertices, j edges, and k vertex and l edge properties	$iW_v + jW_e + (k + l)W_p$
	Algorithms	
SP	Find the shortest path from source to destination	K^k_b
CC	Compute the local clustering coefficient	K^2_b

Table 1: Benchmark Model. We use these equations for modeling operations at the two higher levels of the benchmark. In the formulas in the Model column, n indicates the number of neighbors a vertex has and K indicates the number of invocations of *GetNeighbors*.

ample, if a database performs poorly on microbenchmarks but "mysteriously" well on complex queries, PIG's framework should help identify precisely where amortization occurs. Throughout the text, we explain how we model various operations, but a summary of the entire model is in Table 1.

3.2 Graph Models

PIG comes with three types of graph workloads (Table 2): Barabasi graphs [17], Kronecker graphs [14], and real-world graphs from the SNAP [4] web site. Barabasi graphs represent traditional small-world, unpartitionable graphs. The Kronecker model is representative of natural graphs, and the SNAP data sets ground everything in real data.

For the Barabasi and Kronecker graphs, we use a range of graph sizes (in vertices): one thousand, one million, two million, and (Barabasi only) ten million. These sizes are designed to map to typical database operating points: in the database buffer cache, outside the cache, but in-memory, and out of memory. We configure a system to use a 1 GB cache, intending that the 1000 and 1 million vertex datasets fit in the database's cache, the 2M vertex datasets exceed the cache, but fit in memory, and the 10M vertex dataset far exceeds memory. In actuality, the graphs do not fall so neatly in these buckets, because the systems we study use fairly different representations. We generated all Barabasi graphs with $m = 5$ (five edges per vertex) and the stochastic Kronecker graphs from a 2×2 seed matrix.

The PIG graph generator can generate typed properties, distributed according to one of many distributions. We generated three properties for each vertex; one property has integer values uniformly distributed between 18 and 65 (e.g., ages), the second has unique integer values, and the third has approximately 100,000 string values between 15 and 20 characters long, uniformly distributed.

For edges, we generated both labels (types) and a property. The property is an integer between 0 and 10,000 described by a truncated normal distribution. In the graphs marked *1el*, we assigned every edge in the graph the same edge label. In the graphs marked *2el*, we generated two labels distributed uniformly across the edges.

We selected the `Amazon0302` and `Amazon0312` graphs from the SNAP collection, representing the product co-purchasing network, because they were relatively large (262,111 and 400,727 vertices respectively) but cacheable, and they had both graphical structure and properties. We imported only the title, category, and sales-rank properties, even though the original dataset comes with more properties.

Name	Vertices in 1000s	Edges in 1000s	Neo4j Size	Dex Size
Barabasi Graphs				
Small	1	5	1.1 MB	1.2 MB
Medium 1	1,000	5,000	877 MB	723 MB
Medium 2	2,000	10,000	1.8 GB	1.4 GB
Large	10,000	50,000	8.7 GB	7.2 GB
Kronecker Graphs				
Small	1	3	0.8 MB	0.9 MB
Medium 1	1,049	7,054	1.2 GB	0.9 GB
Medium 2	2,097	15,519	2.5 GB	2.0 GB
Amazon Graphs				
`amazon0302`	262	1,235	128 MB	138 MB
`amazon0312`	401	3,200	239 MB	250 MB

Table 2: Database Sizes and Characteristics

4. PIG SPECIFICATION

PIG consists of a set of test components and a web-based visualization tool. The latter enables comparisons between different graphs and databases and between measured and modeled performance. The components record both timings and operational data needed to relate results across layers.

The following discussion presents PIG according to its structure (Figure 1) rather than the order in which the individual benchmarks execute. The "Applications" layer in this figure illustrates how a user might evaluate application performance using PIG results, but we do not include such

applications in the framework. Thus, PIG has three layers: Micro-Operations, Graph Operations, and Algorithms.

4.1 Level 1: Micro-Operations

4.1.1 Reads

PIG's basic read operations retrieve individual elements from the database: a vertex, an edge, or a property. The test components are: *GetVertex*, *GetEdge*, *GetVertexProperty*, and *GetEdgeProperty*. We test both integer and string vertex properties, measuring each independently. We time these operations in a loop of statistically significant size that gets elements either by ID or by an indexed property value.

Traversal is the retrieval of one vertex from another via an edge. In theory, traversal should be the aggregate of one *GetEdge* and one *GetVertex*. However, we expect databases will amortize traversal with respect to random *Get* operations, because it is the fundamental expression of graph locality and an essential operation for graph algorithms (in fact, the HPC-SGAB benchmark measures all performance in terms of "traversals per second"). In a directed database, it is also important to measure how reverse traversal compares to forward and whether it amortizes *Get* operations similarly.

PIG tests six traversals: for each of incoming, outgoing, or either edges, we measure retrieval time for the vertex of the first edge, either unconditionally or conditionally based upon the edge label (i.e., a relation type such as *friend*).

4.1.2 Writes

Our fundamental write operations mirror our fundamental read operations: *AddVertex*, *AddEdge*, *SetEdgeProperty* and *SetVertexProperty*. (There is nothing analogous to traversal.) At the next level, we use these fundamental operations to evaluate bulk and incremental graph ingest.

4.2 Level 2: Graph Operations

Level two components express fundamental graph operations and can be modeled from the microbenchmark results.

4.2.1 Reads

The common use for traversal is neighborhood exploration: visiting a vertex's neighbors and evaluating them to direct further traversal. We query for neighboring nodes via incoming, outgoing, or either edges, and we repeat each test conditioned and not conditioned on an edge label. This is represented in Figure 1 by *GetNeighbors* and its conditional variants. We model each neighborhood query by counting the number of vertices accessed and multiplying by the sum of corresponding *GetEdge* and *GetVertex* time.

Khop retrieval generalizes the *GetNeighbors* query, retrieving sequential neighborhoods from a starting vertex to collect all vertices k or fewer edges away from the point of origin. *GetKhopNeighbors* is a common component of clustering and power-law analysis algorithms. We model *GetKhopNeighbors* as a linear function of *GetNeighbors* queries. The extent to which a database amortizes *GetKhopNeighbors* over random retrieval can be interpreted as a measure of the extent to which it takes local vertex clustering into consideration in its layout and cache.

4.2.2 Writes

PIG analyzes two forms of data ingest: bulk and incremental. Bulk ingest represents the initial loading of a dataset into the database's persistent storage from a canonical external representation (e.g., CSV). Incremental ingest represents the addition of vertices and/or edges to an existing graph using individual *AddVertex* and *AddEdge* operations. Unsurprisingly, this piecewise insertion is too slow for large graphs, and systems generally provide a mechanism for bulk ingest. In both cases, it is instructive to express the cost of ingest as a function of the cost of incrementally inserting nodes and edges to expose the efficiency of a system's bulk ingest.

We measure bulk ingest time on the first 90% of an input file, and then measure incremental ingest on the remaining 10% using per-object operations. We model incremental ingest as a function of *AddVertex*, *AddEdge*, and *SetProperty*.

4.3 Level 3: Algorithms

Many graph database queries are instances of well-known graph algorithms. Currently, PIG benchmarks two.

4.3.1 Shortest Paths

Dijkstra's algorithm is a stateful breadth-first search that finds the shortest path between a source vertex and all other vertices in the graph. If we wish to find the shortest path between two particular nodes, then we use Dijkstra's algorithm, terminating as soon as we locate the target vertex. The basic algorithm is quadratic in the number of vertices, but with a Fibonacci heap can be reduced to running time $O(|E|+|V|log|V|)$ for graphs with non-negative weights. For simplicity, our benchmark does not consider edge weighs and thus uses the basic algorithm.

Some databases provide their own implementation of the shortest path finding routine, sometimes using algorithms other than Dijkstra's. We use our implementation of the Dijkstra's algorithm for the purposes of this paper in order to evaluate the "cleverness" of the core graph database, not the "cleverness" of the choice of the algorithm.

In theory, Dijkstra's algorithm is a heterogeneous mix of *SetVertexProperty* and *Get(Cond)Neighborhood* operations, assuming that we use properties to keep track of which vertices we visited. Unfortunately, this simple implementation is impractical. None of the systems we have analyzed to date support transient properties, and clearing all the marked vertices between iterations of the benchmark proved prohibitively expensive. Instead, we maintain marking information in heap storage.

The extension of shortest path to single-source shortest path is straightforward. We expect that the performance of both will be dominated by the length of the (longest) path. At the current time, we include only a simple shortest path algorithm and do not support variants such as single-source shortest path or all-pairs shortest path.

4.3.2 Clustering Coefficient

The degree to which nodes in a graph tend to cluster together is its clustering coefficient. There are both local and global metrics of clustering. The global clustering coefficient is three times the fraction of triplets (sets of three vertices connected by at least 2 edges) that form triangles. The local clustering coefficient for a given vertex expresses what fraction of the possible edges in the vertex's neighborhood exist (i.e., the number of edges among a vertex and all its neighbors over the number of edges required to form a clique among a vertex and its neighbors). Our last algorithmic benchmark computes the local clustering coefficients.

We model clustering coefficient as a 2-hop neighborhood query: one hop to locate the neighbors and another to identify edges among neighbors.

5. IMPLEMENTATION

We began development of PIG using the *graphdb-bench* benchmark authoring framework [5], but we have since diverged from the original codebase extensively. During PIG development, we tested it on several databases: DEX [15], Neo4j [16], MySQL and Berkeley DB. These databases have highly heterogeneous data models and ensured that the benchmark was suitable to a wide range of implementations. The diversity of systems also forced us to think carefully about how to produce a fair, "apples-to-apples" comparison. In the interest of space we present results only for DEX and Neo4j, because these are the industry-standard dedicated graph databases.

We provide a reference implementation for each benchmark in the Blueprints [7] graph model API, which provides a standard access method to graph data stores from Java, similar in purpose to the relational JDBC. The API encapsulates the database backend in an object-oriented wrapper of standard graph elements (e.g., Graph, Vertex, Edge).

PIG can thus use any graph database that provides a Blueprints driver plus a few PIG-specific routines that are currently not specified by Blueprints, such as a method to return the total number of vertices in a graph and a method to select a vertex uniformly at random. Our implementations of these functions for DEX and Neo4j were each approximately 150 lines of code. However, we recommend that each benchmarked database implement the benchmarks in its native API to obtain the most accurate results, not affected by Blueprints' overhead. We implemented all benchmarks in DEX's and Neo4j's native APIs.

5.1 Configuring the Databases

We configured each database to use a total of 1 GB for all of its caches. This was straightforward for DEX; its API allows specifying the total cache size, and it decides how best to divide this space. Neo4j's API requires choosing the size of each of its caches individually. We allocated 75% of the memory to the buffer caches, splitting it proportionally between the node, relationship, property, string, and array stores based on their relative file sizes. (We used the defaults for the initial ingest.) We gave the remaining 25% to the garbage collection resistant (GCR) object cache, splitting it so that it can hold the same fraction of nodes and relationships.

On advice from the DEX team, we configured DEX to materialize neighbors, so that a vertex's edges and the neighboring vertices are co-located in the persistent storage. The rest of the database configuration is left in its default state. Most notably, this means in the case of DEX that it is tuned for performance at the cost of relaxed durability.

5.2 Running the Benchmarks

The benchmark begins by opening the database and initializing all operations, which usually involves selecting random nodes, edges, and/or property values. We use a uniform distribution in most cases, but we also have the ability to sample nodes proportionally to their degree. The next steps are to warm the file system's buffer cache by reading all the database files and to warm (and pollute) the database's caches by scanning all objects in the database. This ensures that the caches are warm and do not necessarily contain all objects touched during the initialization. Finally, we run all operations.

We run each operation enough times to warm up the Java code paths (so that we measure the performance with "JITed", not interpreted, bytecode), and then again enough times to get a statistically significant result. We only report the statistics for this second run. For each operation, we capture and log its execution time, memory overhead, time spent inside Java's garbage collector, and its self-reported statistics, such as the number of retrieved neighborhoods and vertices, and the number of unique vertices.

6. USING PIG

We used a single system for all benchmarking: Intel Core i3 3 GHz equipped with 4 GB RAM, running Ubuntu 12.04 and OpenJDK Java 6. We ran our benchmark on the Neo4j 1.8 and DEX 4.6.0 graph databases. For all tests, we allocated 1 GB of Java heap and 1 GB of database cache.

In the results below, our goal is to illustrate the analytical power of the benchmark. While we do sometimes compare results from Neo4j and DEX, we do so to highlight key issues that the benchmark uncovers and not to recommend one system over the other. As we shall see, the default configurations for the two databases target different workloads, and an evaluation designed to select a database would undoubtedly choose to configure the systems differently.

6.1 Level 1: Micro-Operations

We begin by examining the primitive operations of a graph database. As these operations are at the lowest level of the framework, we do not have models for their performance, but PIG reveals details of the underlying implementations that are not necessarily apparent when examining high-level application results. In the interest of space, we present only a subset of the results here, selecting those that provide the greatest insight. We observed very little difference between graphs using one edge label and those using two, so we show results only for the single edge-label graphs.

6.1.1 Get/Add Edge/Vertex

We begin by examining the read performance of Neo4j and DEX on the *GetEdge* and *GetVertex* micro-operations in Figure 2. We typically think of a database as operating in one of three regimes (in-cache, in-memory, on-disk). Neo4j maintains an object cache in the Java heap, producing a fourth performance regime illustrated by the particularly low latency results for 1K graphs and the 1M Barabasi; the Kronecker graphs most clearly illustrate three of the four operating points with the 10M graph providing the fourth point. As DEX does not have this additional cache, its performance corresponds more closely to the expected three-level hierarchy; the 1M and 2M graphs both fit in memory, while the 1K fits in cache and the 10M exceeds memory.

While Neo4j's *GetVertex* latency more than doubles when we fall out of memory (transitioning from the 1M to the 10M case), DEX's latency increases only by 50%, because its latency in the 1M case is already more than twice that of Neo4j, so the disk penalty is proportionally less. The results for *GetEdge* are more significant. DEX's latency increases by approximately a factor of four when it falls out of memory. Neo4j's latency increases by a factor of almost 50 as the

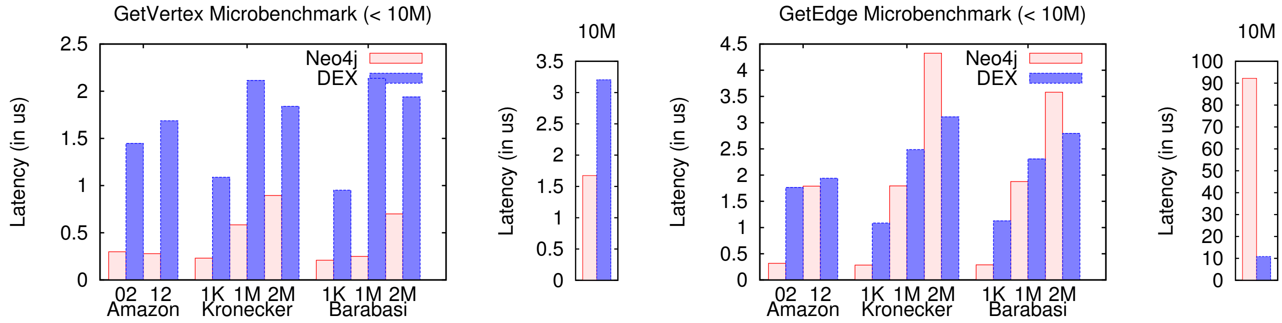

Figure 2: Get Edge/Vertex Microbenchmarks.

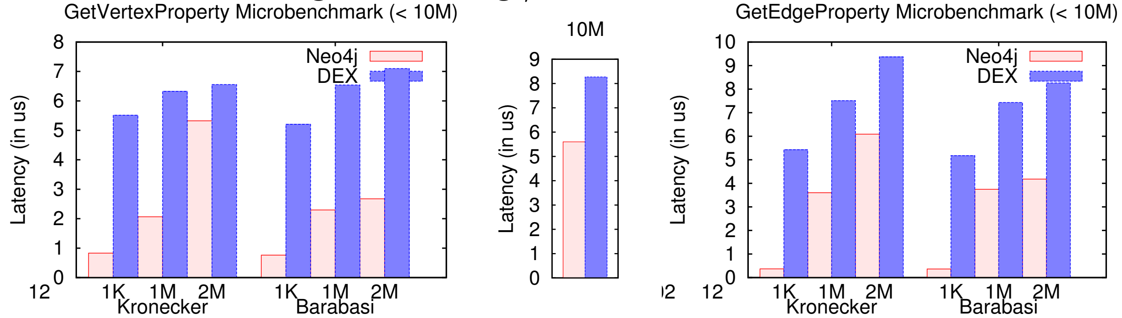

Figure 3: Get Edge/Vertex Property Microbenchmarks. We get the age property for vertices and the time property for Edges.

database becomes I/O-bound. This does not happen in the *GetVertex* case, because the vertex store is quite small (only 86 MB in the 10M case), relative to the edge store (4.6 GB).

The *GetVertex* and *GetEdge* benchmarks highlight the caching behavior of the systems as well as the relative sizes of the vertex and edge stores.

6.1.2 Get/Set Vertex/Edge Property

We also omit the Amazon datasets from this discussion for brevity; they show similar patterns as those discussed below. Figure 3 shows *GetVertexProperty* results for age (integer) property and the *GetEdgeProperty* for the time (integer) property. (The benchmark also generates results for the name property, but we omit it here in the interest of space.) Comparing these results to those in Figure 2, we can see that the latencies for these operations are significantly higher than those for vertices and edges. Although both systems implement property graphs, the performance suggests that they do not treat properties as “first class” objects. In Neo4j, getting a property in small graphs is up to six times more expensive than getting a vertex or an edge. We have omitted the results for getting edge properties for the 10M graph, because when the database falls out of the cache, Neo4j’s latency grows by three orders of magnitude. Neo4j links attributes off of the objects they describe, so in the out-of-memory case, retrieving an edge attribute is likely to incur disk I/Os for both the edge and then the property, and these are not explicitly placed near each other on disk. In contrast, DEX shows much more stable performance accessing properties, especially vertex properties. While still slower than retrieving vertices and edges, property retrieval is of the same order of magnitude, and scales only slightly with the database size.

The *GetVertexProperty* and *GetEdgeProperty* benchmarks reveal that properties are not of the same stature as edges or vertices in either system, but that DEX provides a significantly more scalable solution. Conversations with Neo4j developers confirmed that Neo4j’s design emphasizes in-memory performance over on-disk performance. The PIG results clearly illustrate this design decision.

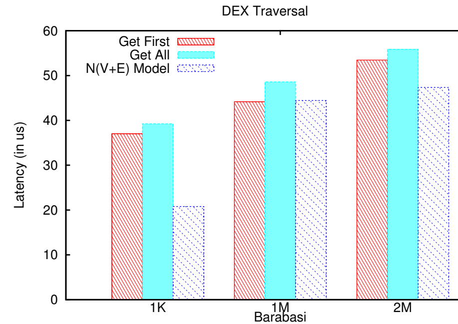

Figure 4: Traversal Results: The benchmark reveals that GetFirstNeighbor retrieves *all* neighbors before returning the first neighbor. The 10M DEX *GetFirstNeighbor* and *GetAllNeighbors* benchmarks force our JVM into swapping, and we have not yet diagnosed the cause.

6.1.3 Traversal

Our last read micro-operation is traversal. We intended traversal to be a primitive operation, measured by retrieving the first neighbor of a vertex. However, this produced a time significantly greater than what we expected based on *GetVertex* and *GetEdge* times. Instead, we found that

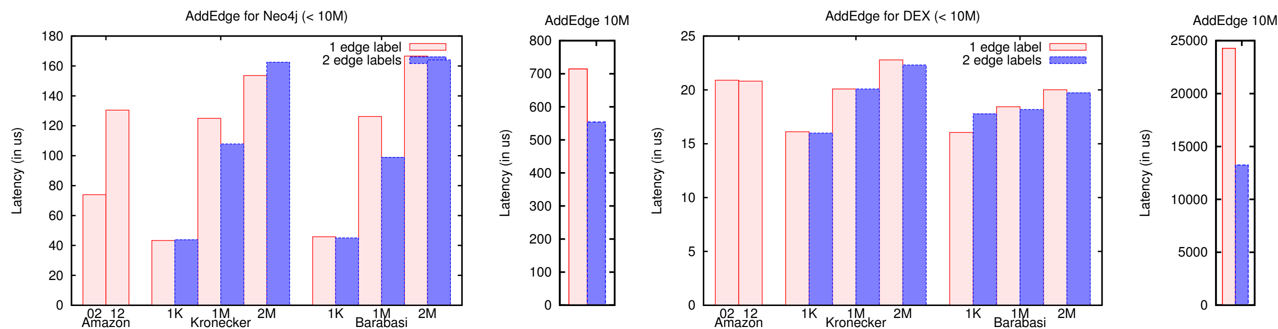

Figure 5: Add Edge Microbenchmark Results. The first two graphs show Neo4j's ACID *AddEdge* results while the second two graphs show DEX's non-transactional additions

retrieving the first neighbor performed nearly identically to retrieving all the neighbors. The results were similar across graph types and databases, so we present Figure 4, which depicts DEX performance on the 1K, 1M, and 2M Barabasi graphs as an example. We hypothesized that the database was retrieving all (or a sufficiently large number of) neighbors before returning the first. We tested this hypothesis by comparing the *GetFirstNeighbor* and *GetAllNeighbors* times (the first and second bars). The third bar in each set shows the time we model based upon the GetVertex and GetEdge times. The models are fairly close except for the 1K graphs, where the fixed overhead of assembling the Java objects dominates.

The traversal benchmark and accompanying analysis correctly reveal that traverse is not a primitive operation in either database. Therefore, in higher level benchmarks, we model traversal as the sum of *GetVertex* and *GetEdge* times, because there is no way to isolate and measure a single traverse operation.

6.1.4 Add Vertex/Edge

The *AddVertex* and *AddEdge* results tell an entirely different story from the corresponding *Get* results. First, in all the systems and across all graph sizes, *AddVertex* is a constant time operation (producing exceedingly boring graphs, which are omitted here). Second, Neo4j's *AddVertex* is nearly two orders of magnitude slower than its *GetVertex*. The first point demonstrates the node-centric orientation of both databases; adding a vertex simply appends an entry to the node store. The second point highlights Neo4j's support for ACID transactions. Although PIG wraps many (100,000) operations in a transaction, persistence is expensive. As we did not configure DEX to guarantee durability, its *AddVertex* performance is approximately the same as the in-cache time for *GetVertex*; the new vertex is placed directly in the cache.

The *AddEdge* results in Figure 5 tells a different story yet. Once again, we see that Neo4j's transactional properties introduce significant overhead. However, DEX's performance significantly drops when it falls out of memory (the rightmost graph in the figure). DEX uses a compressed representation for a vertex's neighbors, so adding an edge requires uncompressing and recompressing neighborhoods for the vertices on either end of the edge. This is particularly burdensome for the 10M graph, where it is likely that each retrieval of a neighborhood requires an I/O.

The *AddVertex* and *AddEdge* microbenchmarks highlight important features in both system: Neo4j's ACID properties and DEX's compressed edge representation.

6.1.5 Set Edge/Vertex Property

The performance of Neo4j's *SetProperty* closely mirrors its *AddVertex* and *AddEdge* performance, displaying both the four operating points and its ACID guarantees. DEX *SetProperty* performance closely mirrors its *AddEdge* performance.

6.1.6 Summary

Taking all the microbenchmarks together, a picture of each system emerges. As it was designed to do, Neo4j performs exceptionally well when reading in-memory graphs (i.e., performing graph analysis), but its performance is brittle when the graph far exceeds memory. Its ACID transactions introduce high write times. In contrast, DEX provides low and stable write times in all cases except when performing read-modify-write operations on compressed neighborhood sets in the out-of-memory case. We expect to see benefits to the compressed neighborhood representation as we move into the next level of benchmarks.

6.2 Level 2: Graph Operations

Having examined an extensive set of graphs in the previous section and examined some implementation differences between the two systems, we now focus on characteristics that can be illuminated through PIG's analytical framework.

6.2.1 GetAllNeighbors

While Figure 4 presented the *average* cost of retrieving a vertex's neighbors, we expect that the time should, in fact, depend on the number of neighbors. However, as neighborhood traversal is such a fundamental operation, we also expect that a good implementation will make an effort to cluster neighbors, so that retrieving a neighborhood takes less time than retrieving the same number of vertices randomly. Thus, we hope that the actual *GetNeighbors* performance exceeds that of what we model based on random vertex and edge retrieval. The degree to which these numbers diverge demonstrates a database's "cleverness" of implementation.

Figure 6 shows Neo4j's actual and modeled performance of *GetNeighbors* as a function of the number of neighbors. The story is quite interesting – for both small (1K) and large graphs (10M), performance is relatively independent of neighborhood size, and as the graph grows from purely in-cache to just in-memory, we see the relationship break down. If PIG is doing its job, this should reveal important properties of the database implementation, and it does. For

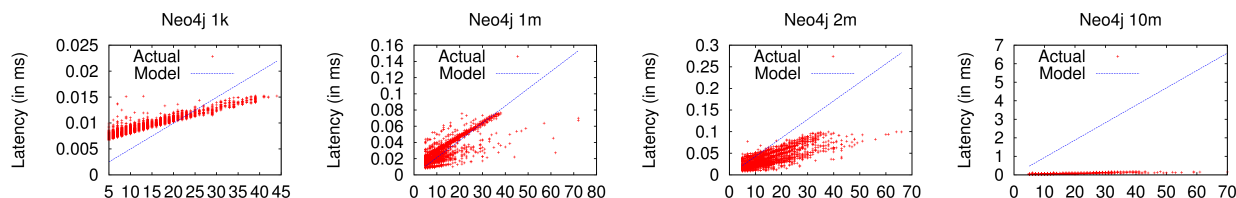

Figure 6: Get Neighbors Actual and Predicted performance as a function of the number of neighbors. The graphs correspond to the 1K, 1M, 2M, and 10M Barabasi graphs. The predictions are based on the sum of the *GetVertex* and *GetEdge* times the number of neighbors.

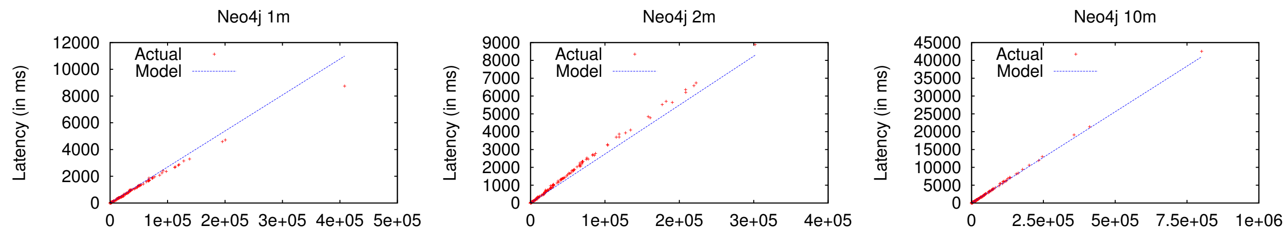

Figure 7: Neo4j GetKhopNeighbors Actual versus Predicted time versus the number of *GetNeighbor* invocations. Our predictions are based on the average measured cost of a *GetNeighbors* query times the number of such queries.

tiny graphs that fit entirely in Neo4j's heap, performance is dominated by the fixed cost of initializing Neo4j's traversal data structures. Once initialized, the incremental cost of adding additional vertices to the neighborhood is minimal. This behavior is readily apparent from the comparison. At the other extreme (10M), we see that the actual performance is much better than what we model. Our model is based on retrieving random vertices and edges, which consequently expects multiple I/Os per neighborhood. However, in practice, the edges adjacent to the same vertex are often placed close together. This is can be an artifact of database ingest, as it is quite common for such edges to be stored together in the source data file as well, or a result of the database's on-disk representation. If we zoom in on the measured results, we see a small dependency on the neighborhood size, but this is dwarfed by the fact that fewer I/Os are necessary than if neighbors were randomly distributed.

In the mid-size region (1M and 2M) we see mild effects of the on-disk clustering. At 1M, the model predicts actual performance; as the graph grows and parts are evicted from memory, actual performance is better than the modeled performance, because neighborhoods cluster on-disk better than random vertices and edges.

PIG's comparative analysis helps us uncover important features of Neo4j's implementation and supports PIG's decision to model even higher-level algorithms in terms of these intermediate operations, rather than the primitive ones analyzed in the previous section.

6.2.2 Khop Traversal

As we move from simple neighborhood queries to Khop queries, we expect the number of nodes we visit to grow exponentially. The benchmark lets us compare Khop performance to predictions based on the total number of neighborhoods and vertices accessed, the number of unique vertices visited, and K, the number of hops outward. As expected, *average* time scales exponentially in K, but for each value of K, there is a large spread, because a Khop query can return dramatically different numbers of vertices, depending on the vertex from which it is launched. Although we examined all the relationships, we found that modeling Khops as a function of the number of *GetNeighbors* invocations works well. Figure 7 shows results for Neo4j; DEX has similar behavior.

Unlike immediate neighborhood queries, modeling Khops on *GetNeighbors* is a fairly accurate predictor of Khops performance, even for out-of-memory graphs. This, too, is a useful result: what it shows is that Neo4j's on-disk layout scheme doesn't significantly amortize placement costs beyond a vertex's immediate neighbors. Using PIG, a database designer might determine this to be the root cause of disappointing performance on a more complex Khops- based algorithm, and consider it evidence of a need for future research in on-disk layout schemes for $K >= 2$.

6.2.3 Ingest

Figure 8 presents the ingest results for Neo4j and DEX on medium (1M and 2M) Barabasi graphs (the other graphs exhibit a similar behavior and are thus omitted in the interest of space), comparing them to the model based on the individual costs of the *AddVertex*, *AddEdge*, and *SetProperty* operations. The bulk load ingests 90% of the graph and the incremental ingest the remaining 10%. Both systems at all database sizes demonstrate significant efficiency for bulk ingest, though it ranges from a factor of three (for DEX at 1M and 2M) to a factor of 17 (for Neo4j at 2M). At best, we might expect that the actual incremental ingest performance might exceed what we model for a naive implementation by similar factors.

This is an excellent example of how PIG's modeling offers insights not readily available through simple end to end measurements. Without modeling, all we might see is that DEX outperforms Neo4j significantly on ingest. However, we know that Neo4j's transactional guarantees are the major source of that difference. And the modeling also shows

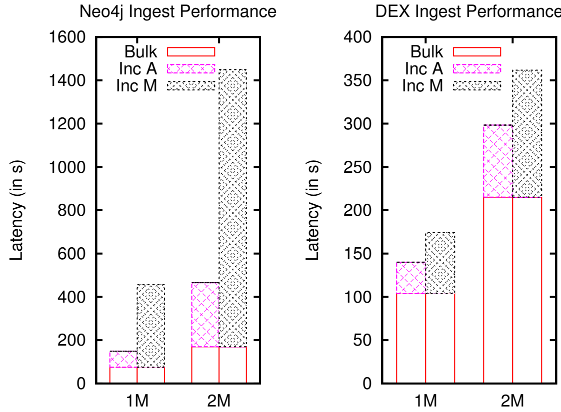

Figure 8: Neo4j (left) and DEX (right) Ingest Performance for medium graphs. 'A' stands for the actual, measured times, while those with 'M' are modeled.

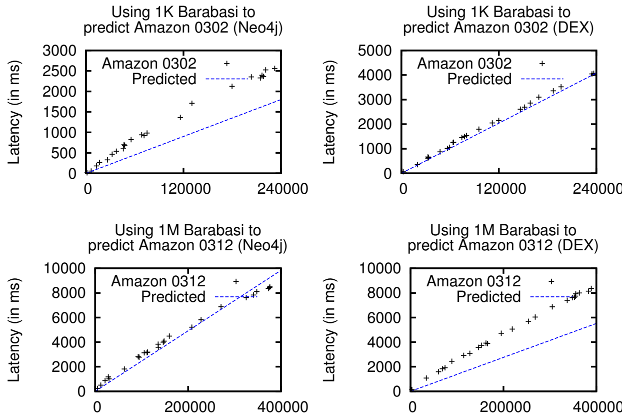

Figure 9: Predicting Amazon shortest path performance using the *GetKhopNeighbors* Barabasi prediction, which is itself based upon *GetNeighbors* benchmark results. The top two graphs use the 1K Barabasi graph results to predict the Amazon0302 data set and the bottom two use the 1M Barabasi to predict the Amazon0312 data set.

that DEX does not amortize its insertion costs as much: Its bulk load is only a factor of three better than its incremental load, and its incremental load is only twice that of the naive model at 1M and 2M sizes. In contrast, Neo4j's bulk ingest is always an order of magnitude faster than its incremental ingest, and its incremental ingest is five to ten times better than its modeled ingest. The former is due to the fact that Neo4j disables transactions during bulk ingest, but the latter speaks to the cost of DEX's compressed edge representation. The *Ingest* results provide insight into each systems bulk ingest efficiency and the degree to which each system amortizes repeated operations.

6.2.4 Level 3: Algorithms

PIG currently has native implementations only for Shortest Path and Local Clustering Coefficient. We started with these because they should be easily modeled, and because they are ubiquitous in graph analysis. We model local clustering coefficient computation as *GetKhopNeighbors* for $K = 2$. On both systems, the results for *GetKhopNeighbors* at $K = 2$ perfectly predicts the results for *ComputeClusteringCoefficient*. *ShortestPath* is a bounded Khops query, so we expect its performance to mirror that of *GetKhopNeighbors* for the "right" K for each query. In practice, *GetKhopNeighbors* for $K = LengthOfShortestPathFound$ perfectly models *ShortestPath*. As with Khops before, this implies the databases do not significantly amortize shortest path access patterns (beyond what already occurs in *GetNeighbors*).

We wrap up our discussion of PIG with an experiment simulating how an end-user might find it useful. Imagine that Amazon wanted to know how each of DEX and Neo4j might perform shortest path queries on its data. Rather than installing and running the two systems, such a user might want to use PIG results for a graph with similar characteristics to model that they might get. In this case, Table 2 suggests that the Amazon0302 data set is best modeled by a 1K Barabasi graph and the Amazon0312 data set is best modeled by the 1M Barabasi graph. We might then ask, how well we model Amazon's shortest path performance using PIG results from the Barabasi graphs. Rather than simply using the Barabasi shortest path results (which seemed like cheating and produced outstanding results), we decided to model Amazon's workload using graph primitives – in this case *GetKhopNeighbors*.

Figure 9 shows the results of using the Barabasi *GetKhopNeighbors* to model *ShortestPath* for each of the Amazon datasets. On each plot, the line shows the modeled performance, based on the appropriate Barabasi graph, and the scatter plot shows what we got when we ran shortest path on the Amazon data sets. The results are surprisingly good. Although Amazon0302 is 128 times larger than the 1K graph, because most of it fits in the Neo4j object cache, the model predicts its performance well. Similarly, even though Amazon0312 is about one-third the size of the 1M Barabasi graph, because it is a workload largely captured in the buffer cache, its performance is easily predicted by measurements in that operating regime. Both the modeled and actual results indicate that Neo4j provides better performance in one of the datasets, and DEX in the other.

7. CONCLUSION

In our efforts to develop and analyze PIG results, we learned a great deal about all the systems we've studied. Our conclusions fall into three categories: guidelines for features that are missing in both DEX and Neo4j but that would be useful in a next generation system, comments about the two systems we examined, and experience using PIG.

Neither DEX nor Neo4j appear to treat properties in the same "first-class" way that they treat vertices and edges. In our own work, we find that indices and queries on properties are common and that better performing indices would be beneficial. Similarly, it would have been both simpler and more aesthetically pleasing to use transient properties for marking vertices in *ShortestPath*. Maintaining mark information in the heap does not scale, but proved necessary; marking via *SetVertexProperty* was slow and clearing the resulting marks afterwards was extremely (unacceptably) slow. While both systems do amortize the cost of individual vertex operations when performing Khop queries, it appears

that there is still room for better co-location of related vertices and edges, when the data set falls out of memory. This is a challenging problem, because most natural graphs do not form neat, disjoint clusters; it should be possible to exploit knowledge of known access patterns, as relational databases do in constructing and selecting indices.

Both systems provide efficient bulk ingest, but also effectively amortize the cost of individual *AddVertex* and *AddEdge* operations on incremental ingest. Neo4j's transactional semantics add overhead for insertions, but protect from data corruption in the presence of failure. In exchange, DEX foregoes such guarantees, but provides better and more scalable write performance.

PIG proved invaluable in developing intuition and understanding about the two systems. We frequently made hypotheses, based on results and then confirmed those hypotheses with the system developers. We do not yet have extensive experience using PIG to model application performance, but it has been critical in driving the design of the next generation graph database we are developing.

8. AVAILABILITY

PIG is open source and available for free under a BSD license from the following website:

`https://code.google.com/p/pig-bench/`

To extend PIG to a new database one must implement a small 8-method interface corresponding to low-level graph operations. The implementations for DEX and Neo4j are about 150 lines of code each, not counting comments. We also recommend, but do not require, to reimplement each benchmarked operation in the database's own native API instead of using the reference Blueprints implementation in order to get more accurate results.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0937914 and Oracle Corporation.

We would like to thank Sparsity Technologies for providing us with a DEX license, and Dàmaris Coll Jimenez and Sergio Gómez Villamor in particular for helping us configure the database and interpret our initial results.

We would like to thank Alex Averbuch, Philip Rathle, and Jim Webber for helping us with Neo4j, and also to Alex for his GraphDB-Bench project [5] from which we initially forked our code base.

10. REFERENCES

[1] Bg: A multi-node social benchmark. `http://dblab.usc.edu/users/bg/index.htm`.

[2] Graph500. `http://www.graph500.org`.

[3] HPC graph analysis benchmark. `http://www.graphanalysis.org/benchmark/`.

[4] Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`.

[5] A. Averbuch. Graphdb-bench. `https://github.com/tinkerpop/tinkubator/tree/master/graphdb-bench`, April 2011.

[6] D. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC 2005)*, pages 465–476, December 2005.

[7] Blueprints. `https://github.com/tinkerpop/blueprints/wiki`.

[8] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: a case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '97, pages 214–224, New York, NY, USA, 1997. ACM.

[9] J. Feo, J. Gilbert, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2 graph analysis, 2006.

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

[11] F. Karinthy. Chain-links. *Everything is Different*, 1929.

[12] J. Kleinberg. The small-world phenomenon: an algorithm perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of Computing*, STOC '00, pages 163–170, New York, NY, USA, 2000. ACM.

[13] LDBC: Linked data benchmark council. `http://ldbc.eu`.

[14] J. Leskovic, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning*, 11:985–1042, March 2010.

[15] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey. Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of the sixteenth ACM conference on information and knowledge management*, CIKM '07, pages 573–582, New York, NY, USA, 2007. ACM.

[16] Neo4j: the graph database. `http://neo4j.org`.

[17] M. Newman, A.-L. Barabasi, and D. J. Watts, editors. *The Structure and Dynamics of Networks*. Princeton University Press, Princeton, NJ, USA, 2006.

[18] Property graph model. `https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model`.

[19] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *SPAA*, June 2012.

[20] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. In *Nature*, volume 393, pages 440 – 442. Macmillan Magazines Ltd., June 1998.