Ringo: Interactive Graph Analytics
on Big-Memory Machines

Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison,
Pararth Shah, Jure Leskovec
Stanford University
{yperez, rok, arijitb, rohanp, mraison, pararth, jure}@cs.stanford.edu

ABSTRACT

We present Ringo, a system for analysis of large graphs. Graphs provide a way to represent and analyze systems of interacting objects (people, proteins, webpages) with edges between the objects denoting interactions (friendships, physical interactions, links). Mining graphs provides valuable insights about individual objects as well as the relationships among them.

In building Ringo, we take advantage of the fact that machines with large memory and many cores are widely available and also relatively affordable. This allows us to build an easy-to-use interactive high-performance graph analytics system. Graphs also need to be built from input data, which often resides in the form of relational tables. Thus, Ringo provides rich functionality for manipulating raw input data tables into various kinds of graphs. Furthermore, Ringo also provides over 200 graph analytics functions that can then be applied to constructed graphs.

We show that a single big-memory machine provides a very attractive platform for performing analytics on all but the largest graphs as it offers excellent performance and ease of use as compared to alternative approaches. With Ringo, we also demonstrate how to integrate graph analytics with an iterative process of trial-and-error data exploration and rapid experimentation, common in data mining workloads.

Categories and Subject Descriptors: H.2.4 [**Information Systems**]: Database Management—*Systems*
General Terms: Algorithms, Performance
Keywords: Graphs, networks, graph processing, graph analytics

1. INTRODUCTION

Detecting expert users in a question-answering forum, tracing the propagation of information in a social network, or reconstructing the Internet network topology from a set of traceroutes are examples of tasks faced by today's data scientists. A common theme to all these examples is that they involve input data manipulation as well as graph analytics, where graphs are analyzed using various graph algorithms. To solve such problems and extract valuable insights, data scientists must be able to quickly construct graphs from input data, and analyze the graphs using various graph algorithms.

Thus, in order to support the work of data scientists, one requires a system that offers a rich set of *graph manipulation and analysis algorithms*. As graphs are rarely given as input, but have to be *constructed* from input data, a modern graph analytics system has to support easy manipulation of input data in order to build a desired graph. Arguably the most common form of input data are *relational tables* and while in principle tables can be used to represent graphs, dedicated graph structures where the neighbors of each node are easily accessible are more efficient for most graph computations. Thus, the system also needs to provide a way to *convert* between graphical and tabular data structures and data representations.

And last, the system also needs to provide *fast execution times* suitable for *interactive* use.

In summary, the desiderata for a modern data science oriented graph analytics system are:

(1) Ability to process large graphs, on the order of hundreds of millions of nodes and billions of edges,

(2) Fast execution times that allow for interactive, exploratory use (as opposed to batch-mode use),

(3) Easy to use front-end that provides many graph algorithms in a commonly used high-level programming language,

(4) Large number of efficient ready-to-use graph algorithms,

(5) Rich support for transformations of input data to graphs.

There are many challenges in building such systems. For example, what underlying hardware infrastructure shall one use? A cluster or a big server? How does one design data structures for tables and graphs that are efficient, flexible and fast? What operations are needed for building graphs from input data tables? What are the considerations for end-to-end graph analytics systems?

Ringo: Graph analytics on a big-memory machine. We present Ringo, an in-memory interactive graph analytics system that scales to large graphs. Ringo combines an easy-to-use Python front-end and a scalable parallel C++ back-end, which is responsible for rapid data handling and manipulation. Ringo provides functionality for efficiently building graphs from input data tables, for converting the tables to an efficient graph data structure, and for analyzing graphs using over 200 different graph functions through its core graph analytics package SNAP[1]. Ringo source code is open[2].

Recent research in graph analytics systems has been focused on distributed computing environments [8, 9, 10, 18, 21, 23, 24] or single-machine systems utilizing secondary storage [11, 13, 14]. Such systems offer scalability in the number of cores or in available throughput and size of the secondary storage, but these benefits

---

[1]SNAP is currently downloaded about a thousand times per month and actively used in our research group, as well as by over 500 students in Stanford University courses.
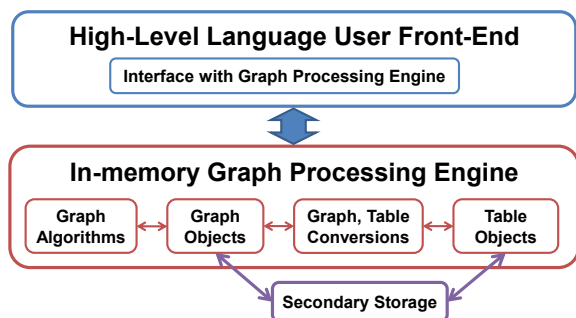[2]http://snap.stanford.edu/ringo

Figure 1: Ringo system overview.

Number of Edges	Number of Graphs
<0.1M	16
0.1M – 1M	25
1M – 10M	17
10M – 100M	7
100M – 1B	5
>1B	1

Table 1: Graph size statistics of 71 graphs publicly available in the Stanford Large Network Collection. 90% of graphs have less than 100M edges. Only one graph has more than 1B edges.

come at a high price of increased communication cost, increased system complexity, and challenges when programming non-trivial graph algorithms. On the other hand, big-memory, multi-core machines are becoming affordable and widely available; a machine with 1TB of main memory and 80 cores costs around $35K.

We observe that most graphs being analyzed today comfortably fit in the memory of one such "big-memory" machine. Graph based computations require random data access patterns and exhibit notoriously poor data locality, so a single big-memory machine seems a natural hardware choice for analytics of all-but-largest graphs.

Ringo is built on the assumption that graphs being analyzed fit in memory of a single machine. This approach has significant benefits in that there is no network communication overhead, no need for managing the secondary storage, and that the programming model and the system use are straightforward. Even though the raw input data might not fit into the main memory initially, data cleaning and manipulation often result in significant data size reduction, so that the "interesting" part of the data nicely fits into the main memory.

Ringo showcases that a single multi-core machine offers a suitable platform for interactive graph analytics, while matching the performance of the fastest distributed graph processing systems (Section 3). Figure 1 illustrates Ringo.

The key features of Ringo are as follows:

- A system for interactive and exploratory analysis of large graphs with hundreds of millions or even billions of edges,
- Tight integration between graph and table processing and efficient conversions between graphs and tables,
- Powerful operations to construct various types of graphs,
- Ringo runs on a single machine with a large main memory, simplifying programming significantly and out performing distributed systems on all but the largest graphs.

The rest of the paper is organized as follows. Section 2 provides a system overview. System evaluation and benchmarks are presented in Section 3. Usage scenarios are described in Section 4. Finally, we conclude in Section 5.

2. SYSTEM OVERVIEW

2.1 Design Choices

In designing Ringo for interactive graph analytics, we were informed by two insights. The first insight is that all but the largest graphs being analyzed today fit comfortably in memory of a big-memory machine. The second insight is that graphs are not analyzed in isolation, but are part of a larger data mining workflow, which requires that, in addition to graph operations, table analytics operations are integrated in the system as well. We discuss those insights and design choices in more detail next.

Graph sizes. Our goal is to provide fast execution time for commonly analyzed, average size graphs rather than provide a scalable solution for extremely large (and rare) graphs.

Classifying graphs from a widely used repository [4] of publicly available real-world networks (covering a wide range of areas, such as the Internet, social networks, road networks, Web networks) according to the number of edges shows that 90% out of 71 graphs have less than 100 million edges and only one graph has more than 1 billion edges (Table 1). Most graphs fit in a few GB of RAM, and the largest graph requires only about 30GB of RAM. Furthermore, very few real-world, non-synthetic graphs with more than one billion edges are discussed regularly in present research literature, Twitter2010 with 1.5B edges and Yahoo Web graph with 6.6B edges being the most common. Assuming 20 bytes of storage per edge, even the Yahoo Web graph requires only about 135GB and fits easily in a 1TB RAM machine. Other studies confirm our observation that most analytics datasets are limited in size [5, 22].

Based on this evidence we conclude that big-memory machines allow for storing and processing large graphs in-memory. As we show later in the experimental section, such machines also have sufficient compute power to execute graph analytics at speeds comparable to the fastest distributed graph processing systems.

Graph analytics workflow. Graph analytics often follow the workflow presented in Figure 2. Raw data is stored in a big data repository and handled by a system like Hadoop to extract the initial data for analysis. The extracted data is organized in a set of relational tables and a major part of the graph analytics workflow is then to construct many different graphs from the tables. Once graphs are built, we require graph specific operations, such as PageRank, connected components, shortest paths. Although graphs can be represented as relational tables and graph operations can be implemented using relational operations, we find it is more efficient (in terms of memory as well as speed) to have optimized graph-specific data structures. Thus, given graphs represented as tables, the next step is to convert these tables to a graph representation. We then execute graph operations and integrate the results back to tables. The result is an iterative process, where data can be rapidly converted from tables to graphs and vice versa.

To provide fast execution speed in Ringo, we implemented table processing as part of the system, which allows us to tightly integrate table and graph processing and to rapidly convert large datasets from one representation to another.

2.2 Graphs in Ringo

Ringo graph representation. A critical operation for fast execution of graph algorithms is quick access to neighboring nodes and edges. Additionally, we require that our graph representation is dynamic, so in existing graphs nodes and edges can be quickly added or removed. A challenge is thus to strike a balance between opposing requirements for fast access to a node's neighborhood and having a dynamic graph structure.
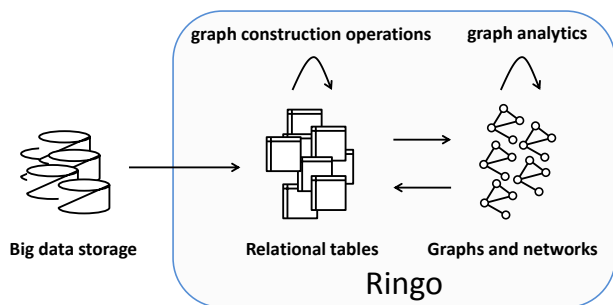
Figure 2: Graph analytics workflow. Using a system like Hadoop, data of interest is extracted from a big data storage into a set of relational tables. Graphs are built using graph construction operations on tables; the results are converted to graphs. Graph analytic operations are then applied to the graphs. Results of graph operations are added back to tables.

One possible approach for efficient graph representation would be to use the Compressed Sparse Row format [3]. This format uses two vectors, a vector for nodes and a vector for edges. The edge vector is sorted by source nodes and indexed by the node vector for fast access. While this approach offers high performance for traversal operations, it does not perform well for dynamic graphs, since graph updates cause prohibitive maintenance costs of the single big edge vector (e.g., deleting a single edge requires time linear in the total number of edges in the graph).

Ringo supports dynamic graphs by representing a graph as a hash table of nodes. Each node maintains sorted adjacency vector of neighboring nodes. Space requirements for this representation are similar to those of the Compressed Sparse Row format. However, we found that representing graphs as a hash table of nodes and the associated adjacency vectors does not dramatically impact the performance of graph algorithms (e.g., deleting a single edge only requires time linear in the node degree).

Ringo graph operations. To provide a rich set of graph constructs and algorithms, Ringo builds on the publicly available Stanford Network Analysis Platform (SNAP) [2]. SNAP provides more than two hundred out-of-the-box graph constructs and algorithms that are available for use on Ringo in-memory graph data structure.

2.3   Tables in Ringo

Table representation. In addition to graph objects, Ringo implements its own native relational table objects to allow for efficient and flexible parallel implementations of operations important for graph construction, to support fast conversions into graph objects, and to avoid any performance overheads related to frequent transitions to and from external systems during the iterative data analysis process. Tables in Ringo have a schema, which defines table columns and their types (integer, floating point, or string). Since tables have been studied extensively [25], we only describe some Ringo specific details here.

As most tabular operations in Ringo are graph related and primarily use iterations over columns, Ringo table representation optimizes this use by implementing tables with a column based store. In Ringo each row has a persistent unique identifier. This allows for fast in-place grouping, filtering and selection. Moreover, identifiers allow for fine-grained data tracking, so the user can identify data records even after they undergo a complex set of operations.

Ringo provides basic relational operations on table objects, such as select, join, project, group & aggregate, set operations, order (sort), and similar. In addition, Ringo also provides a number of advanced graph construction operations, described next.

Graph construction. In order to construct a graph, we first manipulate input data tables into an *edge table* that has two columns, a column with edge source nodes and a column with edge destination nodes. Once the edge table is constructed we transform it into a Ringo graph in-memory data structure.

In some cases, the edge table can be constructed using basic relational operations, such as join and select. However, often graph construction requires advanced operations unique to Ringo. Ringo allows for creating edges based on node similarity or temporal order of nodes. Ringo implements *SimJoin*, which joins two records if their distance is smaller than a given threshold, and *NextK*, which joins predecessor-successor records.

2.4   Converting Between Tables and Graphs

Fast conversions between graph and table objects are essential for data exploration tasks involving graphs. Without the loss of generality we limit our discussion to conversions of directed graphs.

Conversion of tables to graphs. The goal is to generate a directed graph $G = (V, E)$ from table T, with edge source column S and edge destination column D. Nodes V in the graph are defined by unique values in columns S and D. And each row $r \in T$ defines an edge $e \in E$ with the source node provided in column S and the destination node provided in column D, $e = (r_S, r_D)$.

A directed graph in Ringo is represented as a node hash table, where each node contains two sorted adjacency vectors providing its in-neighbors and out-neighbors. The problem of converting a table to a graph is how to transform an edge list given by two columns in a table to a node hash table with sorted neighbor vectors. The challenge is to transform tables with hundreds of millions of rows at speeds that make the system suitable for interactive use.

We experimented with several approaches and found that a "sort-first" algorithm works the best. The algorithm builds a graph representation from a table by first making copies of the source and destinations columns, then sorting the column copies, computing the number of neighbors for each node, and then copying the neighbor vectors to the graph hash table. Advantages of our method are that sorting can be done in parallel and that it does not require any thread-safe operations on vectors and hash tables. While concurrent access is still performed, there is no contention among the threads, which minimizes locking and allows fast execution on multi-core machines. Since the number of neighbors is calculated explicitly, there is also no need to estimate the size of the hash table or neighbor vectors in advance.

Conversion of graphs to tables. The conversion from a graph to a table involves building a node table or an edge table. This conversion can be easily preformed in parallel by partitioning the graph's nodes or edges among worker threads, pre-allocating the output table, and assigning a corresponding partition in the output table to each thread. The threads iterate over nodes or edges in their graph partitions and write the output to their assigned partitions in the output table.

2.5   Ringo Implementation

We highlight aspects of Ringo implementation that allow for high-performance graph processing in an interactive environment.

High-level language front-end. The user interacts with Ringo through a Python module. Ringo front-end utilizes our graph processing engine to execute time critical parts. We use SWIG [1] to connect the Python front-end with the parallel C++ table and graph processing back-end engine.

High-performance graph processing engine. Ringo graph processing engine is based on SNAP [2], a highly efficient C++ graph analysis library that implements complex graph classes and a rich

Graph Name	LiveJournal [7]	Twitter2010 [15]
Nodes	4.8M	42M
Edges	69M	1.5B
Text File Size	1.1GB	26.2GB
In-memory Graph Size	0.7GB	13.2GB
In-memory Table Size	1.1GB	23.5GB

Table 2: Experiment graphs. *Text Size* **is the size of the input text file,** *Graph Size* **is the size of the corresponding Ringo graph object, and** *Table Size* **is the size of the Ringo table object.**

Operation	LiveJournal	Twitter2010
PageRank	2.76s	60.5s
Triangle Counting	6.13s	263.6s

Table 3: Performance of parallel graph algorithms for PageRank and Triangle Counting on a single big-memory machine with 80 cores. For PageRank, ten iterations were timed.

set of graph constructs and algorithms. For Ringo, we have expanded SNAP with several new components, including table processing, conversions between tables and graphs, and parallel graph algorithms. OpenMP was used to parallelize critical loops in the code for full utilization of our target multi-core platforms.

Concurrent hash tables and vectors. The OpenMP layer relies on fast, thread-safe operations on concurrent hash tables and vectors, which are critical for achieving high performance of graph operations. We implemented an open addressing hash table with linear probing [16]. To support fast graph construction, we extended the node hash table with thread-safe insertions to a node's adjacency vector. Concurrent insertions to a vector are implemented by using an atomic increment instruction to claim an index of a cell to which a new value is inserted.

3. SYSTEM PERFORMANCE

In this section, we show Ringo performance on a single, big-memory, multi-core machine. Experiments demonstrate that such machines are able to execute critical operations at speeds that are needed for interactive graph analysis.

Experimental datasets and setup. For our benchmarks, we use two popular graphs, *LiveJournal* and the larger *Twitter2010* (Table 2), that have been widely used for benchmarks of other large-scale graph processing systems [6, 8, 11, 12, 14, 17, 18]. In many cases, graphs used for analyses are of sizes similar to *LiveJournal*. However, larger graphs do exist and can be easily processed on machines with 1TB RAM. For example, the *Twitter2010* graph takes only about 13GB of main memory in Ringo (Table 2), which means that the system could easily process graphs that are an order of magnitude larger.

Our measurements were performed on a machine with 1TB RAM and 4x Intel CPU E7-4870 at 2.40GHz, each CPU has 10 cores and supports 20 hyperthreads for a total of 80 hyperthreads, running CentOS 6.4. A similar machine costs $35K as of Nov 2014.

Parallel graph algorithms. We demonstrate graph analysis capabilities of big-memory machines by using PageRank [20] and undirected triangle counting—two key graph algorithms that are often used for benchmarking purposes. PageRank is an example of an iterative message-passing-like computation. We measured the runtime of 10 iterations. Triangle counting is directly related to relational joins, one of the central problems in database systems [19]. We ran each experiment 5 times, and report the average runtimes. Table 3 shows the results. It is worth comparing the performance of Ringo to the recently published results on the *Twitter2010* graph.

Dataset	LiveJournal	Twitter2010
Select 10K, in place	<0.2s	1.6s
Rows/s	405.9M	935.3M
Select all-10K, in place	<0.1s	1.6s
Rows/s	575.0M	917.7M
Join 10K	0.6s	4.2s
Rows/s	109.5M	348.8M
Join all-10K	3.1s	29.7s
Rows/s	44.5M	98.8M

Table 4: Ringo performance of Select and Join operations on tables. Numbers give measured times on our test datasets. The *Rows/s* **gives the processing rate in millions of table rows processed per second. The processing rate for** *Join* **takes the sizes of both input tables into account.**

For triangle counting, a recently published method [13] took 469s using 1 machine with 6 cores and SSD based secondary storage, while a different, distributed approach took 564s on a cluster of 200 processors [6]. Ringo takes 263s on 1 machine with 80 hyperthreads. The implementation of the algorithm in Ringo was simple, using a straightforward approach, similar to [6] and parallelizing the execution with a few OpenMP statements.

PowerGraph [8] is one of the highest performing graph processing systems. Using 64 machines with 512 cores, PowerGraph took 3.6s per iteration of the PageRank algorithm. On the other hand, Ringo takes around 6s per iteration, while using 6x fewer cores, only one machine and 13GB RAM. Similarly to the implementation of triangle counting, PageRank implementation in Ringo is based on a straightforward, sequential algorithm with a few OpenMP statements for parallel execution. Even though the purpose of Ringo is not to be the fastest graph engine, comparisons show the viability of big-memory machines for processing all-but-the-largest graphs.

In addition to fast execution times, Ringo graph processing also keeps a low memory footprint. The computation of 10 iterations of PageRank on the *Twitter2010* graph had a memory footprint of 18.3GB, and triangle counting on that same graph had a memory footprint of 22.6GB. In both cases the memory footprint was less than twice the size of the graph object itself.

Table operations. While the focus of Ringo is on graph analytics, table operations are necessary for processing the data prior to graph construction and are a fundamental part of the analysis process. To support interactive data exploration the table operations have to execute quickly and at speeds comparable to graph operations.

We benchmark Ringo on two essential table operations, *select* and *join* (Table 4). For select benchmarks, rows are chosen based on a comparison with a constant value. The value is determined so that it either selects 10,000 elements from the table or all elements except 10,000. The purpose of two measurements is to show performance when the output is either very small or similar in size to the input table. We show results for the select in-place operation, where the current table is modified.

For join benchmarks, the input table is joined with a second, single column table. The values in the second table are chosen so that the output table has either 10,000 elements or all elements from the input table except 10,000. Ringo join operation always produces a new table object. Overall, results in Table 4 demonstrate that Ringo offers robust performance over a range of scenarios.

Conversions between tables and graphs. Next we present performance of Ringo algorithms when converting tables to graphs and vice versa (discussed in Section 2.4).

Table 5 gives Ringo execution times for conversions of *LiveJournal* and *Twitter2010* datasets between table and graph representa-

Graph Name	LiveJournal	Twitter2010
Table to graph	8.5s	81.0s
Edges/s	13.0M	18.0M
Graph to table	1.5s	29.2s
Edges/s	46.0M	50.4M

Table 5: Execution times for converting tables to graphs and vice versa. The *Edges/s* row gives the processing rate in millions of edges processed per second.

Algorithm	Runtime
3-core	31.0s
SSSP	7.4s
SCC	18.0s

Table 6: Runtime of single-threaded implementations of commonly used graph algorithms on the *LiveJournal* graph: 3-core, single source shortest path (SSSP), and strongly connected component decomposition (SCC).

tions. For example, in the specific case of the *Twitter2010* graph, the conversion of the table containing a single large edge vector to a graph means that a table with 1.5B rows must be traversed, and 1.5B pairs of node identifiers are extracted. Each element of a pair is inserted to the node hash table in the graph and elements are added to the two corresponding adjacency vectors, which must be sorted. This results in a graph representation with a total of 42M nodes, and 84M vectors containing adjacent nodes.

Overall, the conversion from a table to a graph is performed at a rate of over 10M table rows or graph edges per second and about 50M edges per second in the opposite direction, so graphs with tens of millions of edges can be processed in seconds. The conversion scales well as the processing rate does not degrade for large graphs.

Sequential performance. For moderately large real-world graphs such as *LiveJournal*, even sequential implementations of graph algorithms are fast enough for interactive analysis. We measured the runtime of sequential implementations of 3 commonly used graph algorithms on the *LiveJournal* graph: *3-core* of the graph, *single source shortest path* (runtime averaged over 10 random sources), and finding *strongly connected components* (Table 6). All algorithms executed in about 30 seconds or less. For larger graphs, parallel implementations of graph algorithms are needed, and we are currently expanding the set of parallel algorithm implementations available through Ringo.

4. RINGO SCENARIOS

We demonstrate Ringo on real-world graph analytic scenarios, illustrating its applicability, ease of use, and performance.

4.1 End-to-end Graph Analytics

Our demo will showcase Ringo capabilities in an end-to-end graph analytics scenario. SIGMOD attendees will be able to interact with Ringo and observe ease of use and integration of table and graph operations.

Finding Java experts on StackOverflow. We will show a realistic and representative use case where the goal is to identify top Java experts in the StackOverflow user community. For demonstration we will use complete data from StackOverflow, which is the world's largest question-answering website, where users post questions, then others answer them. As the answers are given, the person posting the question has the option of picking the best answer by "accepting" it. In order to identify top experts our demo will start with complete StackOverflow data (8M questions, 14M

answers, 34M comments). The demo will then follow these steps: manipulate tables to build a graph which connects users providing Java related questions and answers, and use a graph algorithm to identify top Java experts.

Demonstrantion scenario. The SIGMOD attendee will first load complete StackOverflow data[3] in a form of relational tables. The attendee will then manipulate the input tables in order to build a graph representing the interactions in the forum's social network. For example, one way to build a graph is to connect users who answered the same question. A different way is to connect StackOverflow users that answered each other's questions. Ringo provides rich functionality for the SIGMOD attendee to build various kinds of graphs based on the StackOverflow data. Once the graph is built, the attendee will identify important nodes in the graph. Ringo implements over 200 different graph analytics algorithms (e.g., PageRank, Hits, and various other node centrality measures) that the attendee may try out to find Java experts.

Below, we show Ringo Python commands for the above demo[4]. The attendee will load the StackOverflow posts table P, extract all the Java posts JP, and build two new tables: questions table Q and answers table A:

```
P = ringo.LoadTableTSV(schema,'posts.tsv')
JP = ringo.Select(P,'Tag=Java')
Q = ringo.Select(JP,'Type=question')
A = ringo.Select(JP,'Type=answer')
```

Next, the attendee will build a graph and apply the PageRank algorithm on it. Ringo primitives for this task are:

```
QA = ringoJoin(Q,A,'AnswerId','PostId')
G = ringo.ToGraph(QA,'UserId-1','UserId-2')
PR = ringo.GetPageRank(G)
S = ringo.TableFromHashMap(PR,'User','Scr')
```

The *Answers* table on column *PostId* is joined with the *Questions* table on column *AnswerId*. The resulting *QA* table has two *UserId* columns, corresponding to the users that asked questions and the users whose answers were accepted. Using these user columns in *QA*, ToGraph() transforms the *QA* table into an optimized graph data structure *G*, where each node represents a user and an edge indicates that an answer by the destination node was accepted by the source node. GetPageRank() calculates the PageRank scores of all the nodes in *G*. The remaining line builds the final table *S* with the users' PageRank scores.

The above example clearly demonstrates that a system cannot treat graph analytics in isolation, but that graph analytics needs to be integrated with table operations. With Ringo we will demonstrate a system, where graph and table operations are tightly integrated. Besides implementing table operations as part of the system, in Ringo this integration involves fast conversions between table and graph data structures as well as customized table operations, suitable for graph analytics.

Ringo ease of use. We will also demonstrate several factors that contribute to ease of use of Ringo: integrated processing of graphs and tables, Python front-end, and execution on a single machine.

What the attendees will see and do. The attendees will be able to load the StackOverflow dataset in Ringo and perform graph analytics on the dataset. First, an attendee will be able to execute operations that identify the top Java experts on StackOverflow. Second,

---

[3]Freely available at http://data.stackexchange.com.
[4]For complete code for the demo, see http://snap.stanford.edu/ringo

the attendee will be able to vary the parameters to identify top experts on StackOverflow in other programming languages and topics of interest, or change the operations to explore alternative measures of expertise. Third, the attendee will be invited to an open exploration of the StackOverflow dataset by applying Ringo graph analytics capabilities.

4.2 Performance Demonstration

This part of our demo will center around Ringo performance. Attendees will be able to observe system performance and resource utilization on a range of input datasets and hardware platforms.

Graph analytics operations. Several types of operations are critical for graph analytics: graph operations, table operations, conversions between tables and graphs, and data input/output. For the demo, we will show performance of a range of Ringo operations on the LiveJournal and the Twitter2010 datasets. LiveJournal is a typical representative of a graph dataset, while the Twitter2010 dataset is one of the largest publicly available real-world graph datasets.

Hardware used for demonstration. Big-memory machines provide a viable platform for interactive graph analytics. Large RAM on such machines can deliver fast random access, required by graph algorithms. In our demo, we will show how Ringo performs on a wide range of operations on a big-memory machine. For our demo, we will use a machine with 1TB RAM and 4x Intel CPU E7-4870 at 2.40GHz, running CentOS 6.5. Each CPU has 10 cores and supports 20 hyperthreads for a total of 80 hyperthreads.

Ringo also works well on standard personal computers, provided that the datasets fit in the RAM available. As discussed before, 90% of graphs in SNAP require only a few GB of RAM (see Table 1), so they can fit in memory of a typical desktop or laptop. We will demo Ringo also on a laptop with 8GB RAM and an Intel Core i5-4258U CPU at 2.40GHz with 2 cores, costing around $1,500.

What the attendees will see and do. The attendees will be able to use Ringo on a big-memory machine and on a personal computer. Initially, the attendee will choose either a big-memory machine or a laptop environment and load one of the datasets, LiveJournal or Twitter2010. (Only LiveJournal will be available on the laptop due to RAM limitations.) Next, the attendees will be able to execute a wide range of graph algorithms and observe their performance and the hardware utilization.

5. CONCLUSION

We presented Ringo—a system for interactive and exploratory analysis of large graphs with hundreds of millions of nodes and edges. Ringo exposes its functionality through high-level Python programming language, the language of choice of today's data scientists. Tight integration between graph and table processing and efficient conversions between graphs and tables allow for powerful operations that make it easy to perform complex analytic tasks.

Overall, we demonstrate that big-memory machines are suitable for performing interactive analyses of all, but the very largest graphs. In many real-world scenarios graph sizes are well below a terabyte and in such cases big-memory machines have significant benefit over large distributed clusters. Single machines are easier and more efficient to program, while the cost and complexity of cross-machine communication and scheduling are eliminated.

6. REFERENCES

[1] Simplified wrapper and interface generator SWIG. http://www.swig.org.

[2] SNAP: Stanford Network Analysis Platform. http://snap.stanford.edu.

[3] D. Gregor, J. Willcock, and A. Lumsdaine. Compressed Sparse Row Graph. http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/compressed_sparse_row.html

[4] J. Leskovec, and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection http://snap.stanford.edu/data

[5] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *SoCC*, 2013.

[6] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *CIKM*, 2013.

[7] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, 2006.

[8] J. Gonzalez, Y. Low, H. Gu, , D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[9] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[10] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. In *VLDB*, 2014.

[11] W. Han, S. Lee, K. Park, J. Lee, M. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *KDD*, 2013.

[12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *ASPLOS '12*,2012.

[13] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu. Opt: a new framework for overlapped and parallel triangulation in large-scale graphs. In *SIGMOD*, 2014.

[14] A. Kryola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[15] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[16] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *VLDB IMDM*, 2013.

[17] Y. Low, J. Gonzalez, A. Kryola, D. Bickson, C. Guestrin, and J. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[18] G. Malewicz, M. Austern, A. Bik, D. J., I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[19] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 2013.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[21] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in mapreduce. In *SIGMOD*, 2014.

[22] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *HotCDP*, 2012.

[23] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*, 2014.

[24] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[25] H. Garcia-Molina, J. Ullman, and J. Widom. Database Systems: The Complete Book. Prentice Hall Press 2nd edition, 2008.