GraphBuilder – A Scalable Graph Construction Library for Apache™ Hadoop™

Theodore L. Willke
Systems Architecture Lab
Intel Corporation
Hillsboro, OR 97124
theodore.l.willke@intel.com

Nilesh Jain
Systems Architecture Lab
Intel Corporation
Hillsboro, OR 97124
nilesh.jain@intel.com

Haijie Gu
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213
gu.haijie@gmail.com

Abstract

The exponential growth in the pursuit of knowledge gleaned from data relationships that are expressed naturally as large and complex graphs is fueling new parallel machine learning algorithms. The nature of these computations is iterative and data-dependent. Recently, frameworks have emerged to perform these computations in a distributed manner at commercial scale. But feeding data to these frameworks is a huge challenge in itself. Since graph construction is a data-parallel problem, Hadoop is well-suited for this task but lacks some elements that would make things easier for data scientists that do not have domain expertise in distributed systems engineering. We developed GraphBuilder, a scalable graph construction software library for Apache Hadoop, to address this gap. GraphBuilder offloads many of the complexities of graph construction, including graph formation, tabulation, compression, transformation, partitioning, output formatting, and serialization. It is written in Java for ease of programming and scales using the MapReduce parallel programming model. We describe the motivation for GraphBuilder, its architecture, and present two case studies that provide a preliminary evaluation.

1 Introduction

The exponential growth in the study of graph-based data dependencies is fueling the need for large scale machine learning (ML) frameworks that can analyze these relationships. Recently, frameworks, such as Google's Pregel, Apache's Hama, and CMU's GraphLab, have emerged to perform these often iterative and data-dependent computations in a distributed manner at commercial scale [1], [2], [3]. But in order for data scientists to use these frameworks, they must have the tools to construct very large graphs with arbitrary edge and vertex relationships and data structures.

Unfortunately, tools do not exist today to *efficiently* and *easily* construct graphs with billions

or trillions of elements from unstructured or semi-structured data sources. While one may program MapReduce frameworks, like Apache Hadoop, to do this, the programmer must possess a deep understanding of not only their application and the relevant ML algorithms but also how to parallelize graph construction while conserving system resources. And, they must then partition the graph *effectively* for distributed computation and mining. As a result of this burden, many data scientists spend most of their time preparing data using scripts or application-specific MapReduce programs, with little time left over for analysis [4].

In this paper, we propose GraphBuilder, a scalable graph construction library for Hadoop MapReduce, which provides a simple Java library with algorithms for parallel graph construction, transformation, and verification that are useful for graph mining. GraphBuilder may also partition and serialize large-scale graphs for ingest by GraphLab and other parallel ML frameworks. We describe our open source implementation and use it to construct graphs for distributed versions of two graph-based algorithms. And, we briefly evaluate the library's performance on a small server cluster to demonstrate its potential. Our contributions include:

- A survey of applications that informs the library's architecture
- New MapReduce algorithms for graph normalization (compression), transformation, verification, and partitioning
- An open source implementation of GraphBuilder and initial assessment of its performance

2 Background

Large natural graphs appear in a number of contexts, including Internet connectivity models, online social networks, and genetic networks, among others [5], [3]. Graph analysis, like coloring, shortest path, and in-out edge statistics may be used to study basic characteristics, and structured machine learning algorithms, such as Loopy Belief Propagation, Gibbs Sampling, Co-EM, and LASSO, may be used to perform more sophisticated analysis [6]. And, a community of tools has emerged to store, query, and analyze these structures [7], [8], [9]. But, existing algorithms and tools assume that the problem starts with a constructed graph that is ready for use.

Unfortunately, the sheer scale of these graphs, with billions of edges and vertices, combined with the fact that many follow power-law degree distributions [3], [5], makes them difficult to construct and feed to the graph computational frameworks mentioned in the Introduction. Data scientists often write lengthy scripts or programs to do this, consuming valuable time and challenging the limits of their domain expertise, which we believe is largely unnecessary.

Several typical applications that use power-law graphs are shown in Figure 1. The applications require graphs and associated network information (i.e., vertex and edge data structures) to be constructed from a variety of unstructured and semi-structured data sources. The raw data is pre-processed using various extraction and tokenization methods and then formed into a graph abstraction with network information attached, which is often computed from straightforward tabulations. In a survey of applications, we observed that the stages of the pipeline become less application-specific and more generic as the processing progresses from raw start to completion. That is, although data connectors and pre-processing are application specific, only a handful of graphs and network statistics are used by many applications.

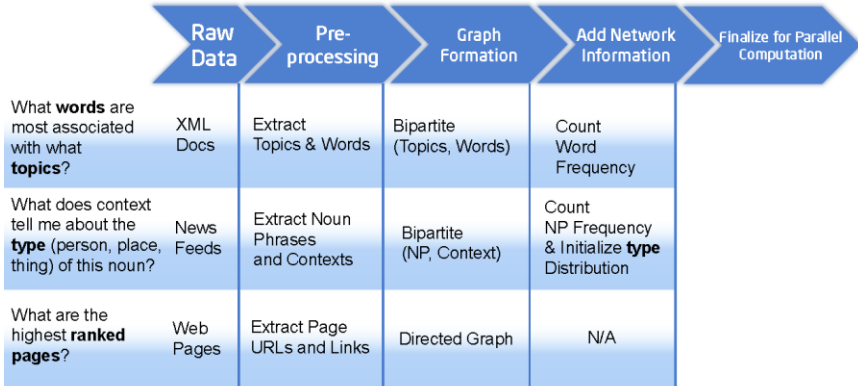


Figure 1: Building Graphs for Practical Applications

Constructed graphs must also be finalized for parallel computation and mining. This step permits distributed frameworks to rapidly extract value from data sources by:

- Minimizing the use of system resources, like cluster memory and storage
- Ensuring the computational effort is load balanced for power-law graphs
- Ensuring that the graph and associated data are correct, clean, and complete

We believe that application programmers would greatly benefit from a library that provides the above functionality in a manner that is easy to program, performant, and scalable. GraphBuilder was developed to fill this important hole in the emerging graph analytics ecosystem and offload domain expertise.

3 The GraphBuilder Library

GraphBuilder is written in Java for easy use and integration with Hadoop MapReduce. MapReduce is well suited for the algorithms used in large-scale graph construction and Hadoop provides cluster resource abstraction and fault tolerance. The major components of the GraphBuilder library, and its relation to Hadoop MapReduce, are shown in Figure 2.

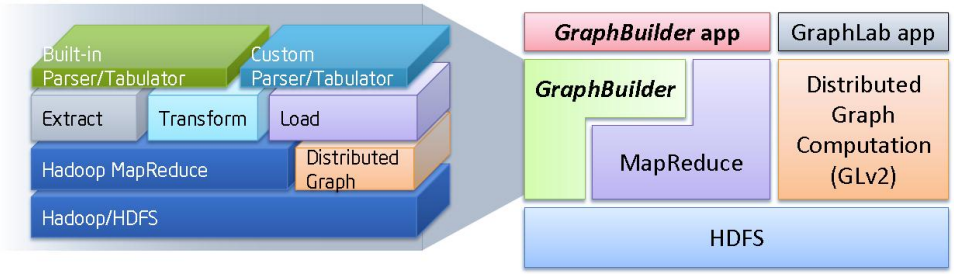


Figure 2: The GraphBuilder software stack

The GraphBuilder library implements the graph construction pipeline shown in Figure 3. The data flow is analogous to the *extract*, *transform*, and *load* sequence commonly used to populate relational databases. We briefly describe the stages below.

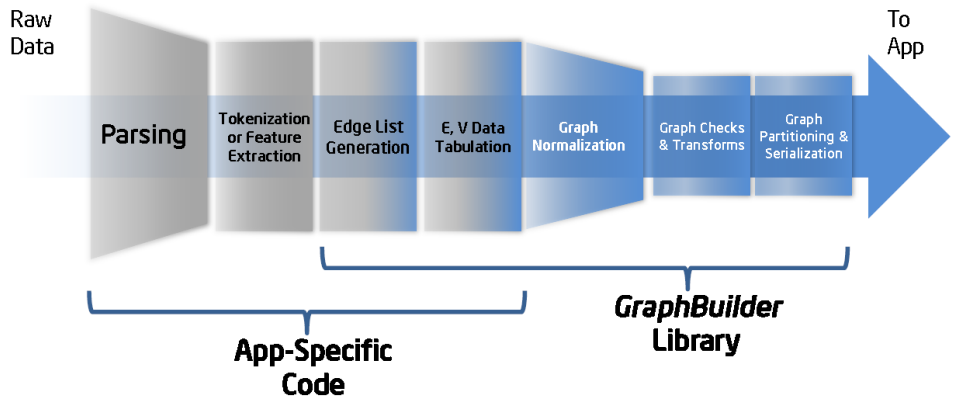


Figure 3: The GraphBuilder graph construction pipeline

3.1 Graph formation and tabulation of network information

Users write application-specific parsers for their data source and short routines to extract and tokenize the features they are interested in analyzing. MapReduce parallelizes the routines for the programmer using *map tasks* and optional *reduce tasks*. An example of XML parsing and extraction is shown in Figure 4.

```
conf.set(XMLInputFormat.START_TAG_KEY, START_TAG);
conf.set(XMLInputFormat.END_TAG_KEY, END_TAG);
new XMLRecordReader((FileSplit) split, conf);
```
Read vertex object

```
Document doc = builder.parse(new InputSource(new
StringReader(s)));
title = xpath.evaluate("//page/title/text()", doc);
```
Parse

```
title = title.replaceAll("\\s", "_");
id = xpath.evaluate("//page/id/text()", doc);
String text =
xpath.evaluate("//page/revision/text/text()", doc);
parseLinks(text);
```
Extract Features

Figure 4: XML parsing and feature extraction

Next, the programmer calls library-supplied APIs to generate an edge list for the graph, connecting the members of one or more classes of features to one another using application-specific rules. Edges are defined using a vertex adjacency list and vertices may be assigned arbitrary string names.

The library supplies a set of built-in functions, such as TF (term frequency), TFIDF, WC (word count), ADD, MUL, and DIV, that may be used to tabulate vertex values and edge weights. This network information is stored separately from the adjacency list to simplify manipulation of the dataset.

3.2 Graph transformation and checking

Once the graph is constructed, library filters may be selectively applied to clean up the adjacency list or even transform the graph from one type to another. Depending on the problem, the data scientist may want to remove duplicate, dangling, and/or self-edges. Library map tasks use distributed hashing algorithms to ensure that all edges appearing between the same two vertices are assigned to the same reduce task while load balancing the reducers. This scheme permits the easy removal of duplicate edges and conversion of a directed graph into an undirected one, if desired.

Following this step, the graph may be arbitrarily partitioned and mined. However, the graph may be further optimized for parallel ML processing, as described below.

3.2 Graph compression and partitioning

The library offers load optimization for graphs that will be subsequently used in parallel ML frameworks like GraphLab. Memory and storage may be conserved by applying dictionary-based compression to the graph and its network information, and the library includes the simple MapReduce compression algorithm shown in Figure 5. A map task creates a key-value dictionary, indexed by ordered integers and alphabetically, of all string values appearing in the adjacency list and edge/vertex data structures. The dictionary is then sharded across machines and co-located with shards of the unconverted edge list sorted alphabetically by *source vertex*. The local dictionaries are then used to convert the source vertex labels to integers. The edge list is then reshuffled to alphabetically shard it by *destination vertex* and the integer substitution repeated. A similar process is applied to compress network information.

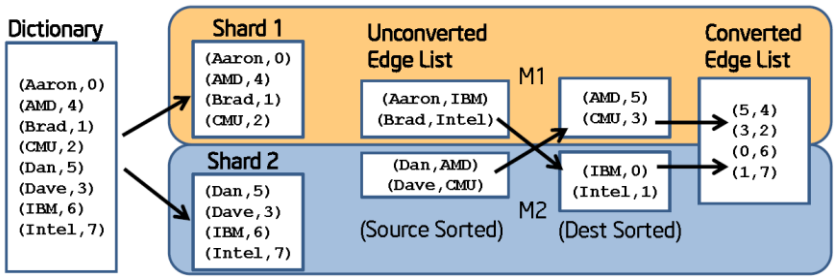


Figure 5: Dictionary-based compression using MapReduce

Following compression, the graph is partitioned. Balanced partitioning of arbitrarily connected graphs, like power-law graphs, is NP hard [10]. Furthermore, traditional graph partitioning algorithms perform poorly on power-law graphs [11]. We implement the balanced p-way vertex cut scheme described in [3] using heuristic MapReduce algorithms. These algorithms partition the graph in a way that will: 1) minimize cluster communications by minimizing the number of machines each vertex spans and 2) load balance the work by placing roughly the same number of edges on each machine during subsequent parallel ML processing.

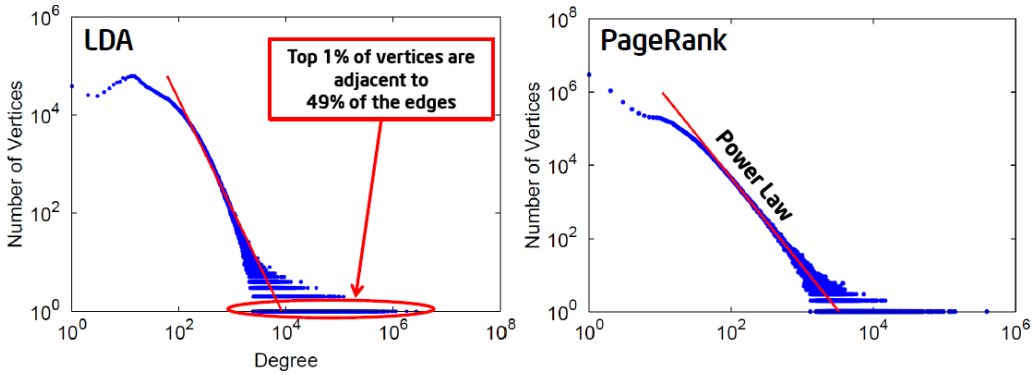
GraphBuilder implements *random edge placement*, where edges are placed randomly by each machine, and *oblivious greedy placement*, where edges are placed by each machine using a greedy heuristic algorithm. These algorithms are described in detail in [3]. The partitions are serialized and stored into files in HDFS using JSON, an extensible self-describing data format that offers compression. JSON makes it easy for computational frameworks and data mining tools to interpret the partition files.

3 Case studies

We used GraphBuilder to construct graphs for PageRank and LDA topic modeling from a XML dump of all pages in Wikipedia as of June 6, 2012. The PageRank and LDA algorithms were implemented on PowerGraph [3]. The PageRank application generated a list of all Wikipedia documents according to their hyperlink ranking within the Wikipedia site. LDA generated "word clouds" for each hidden topic, with words sized according to their prevalence in each topic.

The statistics for the two Wikipedia graphs are shown in Figure 6. The PageRank graph is a

directed graph with vertices representing documents and edges representing hyperlinks, whereas the LDA graph is a bipartite graph linking documents to words and with edge weights equal to the prevalence of each word in each respective document. Both graphs follow power-law degree distributions that make them difficult to partition.



Graph	|V|	|E|	α
LDA	**54M**	**1.4B**	**2.23**
PageRank	**20M**	**128M**	**2.41**

Figure 6: Statistics for the Wikipedia graphs

We implemented a beta version of our Apache 2.0-licensed GraphBuilder library on Apache Hadoop 1.0.1 and installed it on an 8-server cluster with Intel® Xeon® E5 processors, 64 GB of memory, four SATA HDDs, and Intel 10 GbE adapters and switch. We executed multiple runs to profile cluster performance, evaluate the resultant graphs, and assess ease of use. The results are shown in Table 1 and Figure 7.

Table 1: Various statistics for the case studies

Graph	Custom plug-in code	Compression	Vertex Replication Factor (Random)	Vertex Replication Factor (Oblivious)
PageRank	100 lines	60%	4.1	3.5
LDA	130 lines	5%	7.5	5.7

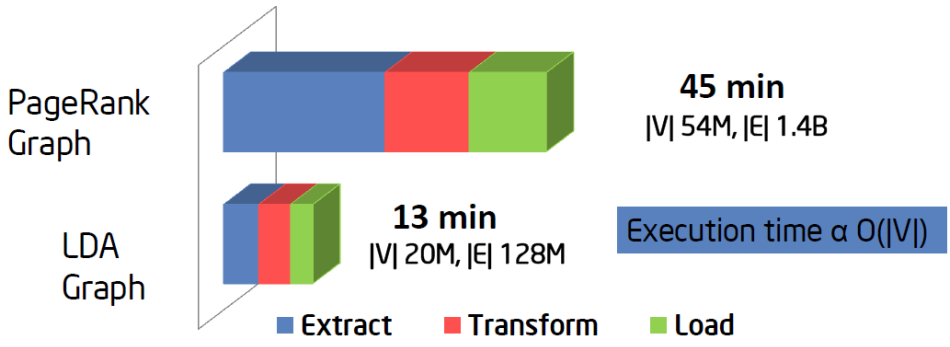


Figure 7: Breakdown of graph construction time

Table 1 shows that the stored size of the PageRank graph was reduced ~60% by the compression algorithm. This represents the savings achieved by eliminating repeated document names and hyperlinks with long string names. On the other hand, the LDA graph only benefited by ~5% because the document names were already encoded as integer IDs and the key words were relatively short.

Table 1 also lists the Vertex Replication Factor, which is the average number of splits required per vertex to partition the graph and is inversely proportional to performance. The oblivious partitioning algorithm outperformed random on both graphs.

We also evaluated the speed and scalability of GraphBuilder. Figure 7 shows the breakdown of total graph processing time. The execution time is proportional to |V|, with the LDA extraction phase taking longer than the others due to additional MapReduce iterations for text tokenization and TF tabulation.

Finally, the library reduced the programming effort by more than an order of magnitude in both case studies, calling on the data scientist to program just over 100 lines of Java to accomplish the entire *extract*, *transform*, and *load* operation.

4 Conclusions and future work

The growing interest in graphical data relationships is fueling the need for large scale machine learning (ML) frameworks that can analyze these relationships. But in order for data scientists to use these frameworks they must have the tools to construct very large information graphs. We have introduced GraphBuilder, a scalable graph construction library for Hadoop MapReduce, which allows data scientists to *efficiently* and *easily* construct graphs with billions or trillions of elements from unstructured or semi-structured data sources. GraphBuilder provides a simple Java library with algorithms for parallel graph construction, transformation, and verification that are useful for graph mining, as well as partitioning and serialization libraries to prepare graphs for ingestion by parallel ML frameworks. We described our open source implementation and used it to construct graphs for two case studies. And, we briefly evaluated the library's performance on a small server cluster to demonstrate its potential.

We continue to use GraphBuilder to understand the complexities of large-scale graph construction for a range of applications across a number of fields, including computational biology, Internet security, and online social networks. So far, we have focused our attention on applications that tolerate batch graph processing but we are moving toward more demanding applications that require the continual streaming of new information into graphs and involve graphs that evolve with time. We are also studying more advanced, "MapReduce-able" partitioning algorithms that take into account community structure within graphs [5], iterative in-memory MapReduce models that will speed up the entire pipeline, and storage architectures that will make it faster and easier to query and manipulate stored graphs. Finally, we believe that these systems will not reach their full potential until better tools are developed to visualize, interpret, and interact with large-scale graphical information. We are collaborating with academic partners within the Intel Science and Technology Centers to advance these capabilities.

Acknowledgments

We would like to thank Carlos Guestrin and his extended team, namely Joseph E. Gonzalez, Yucheng Low, and Danny Bickson, for inspiring us to create GraphBuilder and providing us with a first class education in graph-structured computational methods.

References

[1] Malewicz, G., et al., *Pregel: A system for large-scale graph processing*, in *ACM SIGMOD* 2010, ACM: Indianapolis, Indiana.

[2] Seo, S., et al., *HAMA: An efficient matrix computation with the MapReduce framework*, in *2nd IEEE International Conference on Cloud Computing Technology and Science* 2010: Indianapolis, Indiana USA.

[3] Gonzalez, J.E., et al., *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs*, in *10th USENIX Symposium on Operating Systems Design and Implementation* 2012, USENIX: Hollywood, California USA.

[4] Anonymous, 2012: Quote from Data Scientist in Jeffrey Heer's Stanford (Interview) Study.
[5] Leskovec, J., et al., *Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters*, 2008, Cornell University: Cornell University Library.
[6] Low, Y., et al., *GraphLab: A new framework for parallel machine learning*, in *The 26th Conference on Uncertainty in Artificial Intelligence* 2010: Catalina Island, California.
[7] Neo4j. *Neo4j graph database at http://neo4j.org/*. 2012 [cited 2012 November 6].
[8] Kang, U., S.E. Tsourakakis, and C. Faloutsos, *PEGASUS: A peta-scale graph mining system - implementation and observations*, in *IEEE International Conference on Data Mining* 2009: Miami, Florida USA.
[9] Leskovec, J. *SNAP System at http://snap.stanford.edu/snap/index.html*. 2012 [cited 2012 November 6].
[10] Andreev, K. and H. Racke, *Balanced graph partitioning*, in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* 2004: Barcelona, Spain.
[11] Abou-Rjeili, A. and G. Karypis, *Multilevel algorithms for partitioning power-law graphs*, in *20th International Conference on Parallel and Distributed Processing* 2006, ACM/IEEE: Rhodes Island, Greece.