GraphBuilder: A Scalable Graph ETL Framework

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

Guangdeng Liao
Systems Architecture Lab
Intel Corporation
Hillsboro, OR 97124
guangdeng.liao@intel.com

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

ABSTRACT

Graph abstraction is essential for many applications, from finding a shortest path to executing complex machine learning (ML) algorithms like collaborative filtering. However, constructing graphs from relationships hidden within large unstructured datasets is challenging. Since graph construction is a data-parallel problem, MapReduce is well-suited for this task. We developed GraphBuilder, an open source scalable framework for graph Extract-Transform-Load (ETL), to offload many of the complexities of graph construction, including graph construction, transformation, normalization, and partitioning. GraphBuilder is written in Java, for ease of programming, and it scales using the MapReduce model. In this paper, we describe the motivation for GraphBuilder, its architecture, MapReduce algorithms for graph processing, and a performance evaluation of the framework. Since large graphs should be partitioned over a cluster for storing and processing and partitioning methods have a significant impact on performance, we develop several graph partitioning methods and evaluate their performance.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems and Performance evaluation

General Terms

Algorithms, Design, Measurement, Distributed System

Keywords

GraphBuilder, Graph Construction, Graph Analytics, Graph Partitioning, Graph ETL, MapReduce, Hadoop

1. INTRODUCTION

Large graphs appear in a number of contexts, including internet connectivity models, complex scientific and engineering problems, social networks, and protein networks [6, 14]. Many combinatorial algorithms, such as graph coloring, shortest path, and clique analysis, as well as machine learning algorithms, such as Loopy Belief Propagation, Co-EM, and LASSO are used to perform sophisticated analysis on graph-structured data [14]. Recently, various tools have emerged for graph analytics, such as:

- SNAP [11] and PEGASUS [8] to study basic graph characteristics
- GraphLab [14], Pregel [15], Hama [20] that provide runtime environments for graph-parallel computation
- Stinger [19] for stream graph processing
- Neo4j [17] and Titan [23] graph databases

In order for data scientists to use these frameworks, they must have tools to construct large graphs with arbitrary edge and vertex relationships and data types. Unfortunately, tools do not exist today to efficiently and easily construct graphs with billions or trillions of vertex and edges from unstructured or semi-structured data. A standard solution is to partition large graphs across a cluster of machines. Graph partitioning affects load balancing and communications traffic, and thus significantly affects graph processing performance. While one may program a MapReduce framework, such as Apache Hadoop, to perform parallel graph construction and partitioning, the programmer must possess a deep understanding of not only their application and relevant analytics algorithms but also distributed systems concepts. As a result of this burden, many data scientists spend most of their time preparing data using scripts or application-specific MapReduce programs, leaving little time for analysis.

Motivated by the above challenges, we propose GraphBuilder, a scalable graph Extract-Transform-Load (ETL) framework. It provides a collection of algorithms for parallel graph construction, tabulation, transformation, normalization, partitioning, output formatting, and serialization. We describe its architecture and demonstrate its utility by constructing graphs for two distributed graph-based ML algorithms. In order to achieve good graph partitioning quality, we develop six partitioning methods in the framework and perform detailed analysis.

Our contributions in this paper include:

- Design of MapReduce algorithms for graph ETL including tabulation, transformation, normalization and partitioning
- Design and development of several graph partitioning algorithms and a detailed analysis of partitioning quality
- Performance evaluation of GraphBuilder

2. BACKGROUND AND MOTIVATION

All of the graph analytic tools described in Section 1 assume a pre-constructed graph that is ready for use. Unfortunately, graph-structured data is not always available for them. Graph needs to be created from raw data sources by extracting features relevant to the application. Current solutions are based on custom scripts, which are hard to maintain and extend, and are only short-term solutions. Since many applications call for graphs to be constructed from large volumes of data, resulting in graphs with billions

GRADES – SIGMOD/PODS'13, Month 6, 2013, New York, NY, USA.

of edges and vertices, a parallel graph construction method is needed.

To understand the ETL functionality required, we studied a variety of graph-based machine learning applications, as illustrated in Table 1. We found that users may desire a set of popular edge value tabulations, like frequency counts and TF-IDF calculations, as well as user-defined and application-specific tabulation methods.

Table 1: Survey of Graph-Based ML Applications

Application	Algorithm	Graph	Edge Value
Topic Modeling	LDA	Bipartite	TFIDF
Ranking	PageRank	Directed	N/A
Recommendation Systems	ALS	Bipartite	User Rating
Medical Diagnosis	Belief Propagation	Bipartite	Application Specific

In addition, the constructed graphs often have billions of edges and vertices and have a substantial amount of data associated with them. They must often be partitioned across machines for parallel graph processing and storage. What is more, many graphs follow power-law degree distributions [6, 14] and simple random partitioning leads to load imbalances and significant communications traffic among machines [3, 13]. Therefore more intelligent partitioning methods should be provided to ensure balanced computation and minimize cluster communications for power-law graphs.

To the best of our knowledge, GraphBuilder is the first open source large-scale graph ETL framework that addresses these requirements.

3. GRAPHBUILDER ARCHITECTURE

Since MapReduce is well-suited for the algorithms used in large-scale graph construction, we developed GraphBuilder as a library for Hadoop MapReduce. The major components of GraphBuilder and their relation to Hadoop are shown in Figure 1. GraphBuilder uses MapReduce to construct graphs and stores graphs in HDFS. GraphBuilder provides services for graph analytics that are similar to classic Extract-Transform-Load (ETL) services for databases.

GraphBuilder's architecture is based on the configurable Directed Acyclic Graph (DAG) MapReduce job model, which makes it easy to tailor the pipeline for individual graph construction applications. The complete ETL pipeline provides the following functions (for details, we refer to our GraphBuilder white paper [26]):

- **Extract**: feature extraction, graph formation and tabulation
- **Transform**: graph transformation, checks and normalization
- **Load**: graph partitioning and serialization

The framework is designed to support different data parsers and tabulators, and it allows users to easily add application-specific vertex/edge tabulations. GraphBuilder provides a command line interface for interactive use. Additionally, it exposes its interface at both the job- and API-level for flexibility.

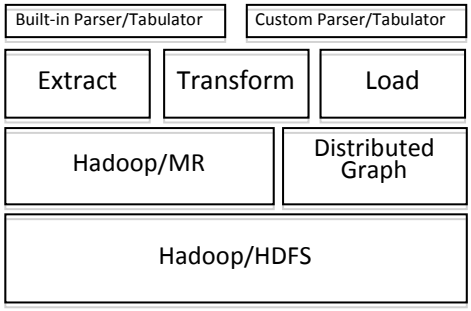

Figure 1: Elements of GraphBuilder

3.1 Graph Formation and Tabulation

To form a graph, users write application-specific parsers for their data source and short routines in the Map function of a MapReduce job to extract and tokenize the features they are interested in analyzing. The output of Map task is a set of vertices and edges, 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. Reduce tasks combines corresponding edge lists and vertex lists from all map tasks to build a complete graph.

As shown in Table 1, several tabulation methods are required for different machine learning applications to calculate edge values. GraphBuilder supports a set of built-in tabulation functions, such as TF (term frequency), TF-IDF, WC (word count), ADD, MUL, and DIV, that may be used to tabulate both vertex values and edge values. It also provides a plug-in interface to source and destination vertex sets [26] to allow users to customize tabulation methods.

3.2 Graph Transformation and Checking

Graph mining and machine learning algorithms often require selective filtering of the input graph (e.g., directionality conversion) to prepare the input graph for subsequent analysis. GraphBuilder supports a filter for directionality conversion, duplicate edge removal, dangling edge removal, and self-edge removal. These filters are programmed in MapReduce as follows:

Objective: Given a list of edges $(x_i, y_i), x_i, y_i \in X \; where \; X$ is a list of vertex IDs, obtain all unique edges (x_i, y_i) where $x_i \neq y_i$ while completing the user-specified graph transformation.

Map: compute hash h_i over (x_i, y_i) and distribute edges to reduce tasks according to hash value.

Reduce: since MapReduce framework generates $(h_i, \{(x_{i1}, y_{i1}), (x_{i2}, y_{i2}) \ldots\})$ for each reduce task, we can remove duplicate edges and also apply different optional functions to edge lists like removing self- or bi-directional edges, directionality conversion etc.

3.3 Graph Normalization

Vertex IDs generated from raw data often have arbitrarily long sparse labels (e.g., URLs). The sparse nature of labels causes high utilization of memory and storage. GraphBuilder addresses this problem by normalizing raw vertex IDs to integers. It does normalization in two phases as described below. Phase 1 (see below) builds a dictionary to map raw IDs to integers and chunks them into segments

for efficient loading in phase 2. Phase 2 (also below) sorts edge lists based on raw source vertex IDs and then reads dictionary segments to normalize the vertex IDs. GraphBuilder applies the same method to normalize destination vertex IDs.

Phase 1 (Dictionary creation and chunk)

Objective: Given a list of n raw vertex IDs $X = \{x_0, \dots, x_{n-1}\}$, creates a one-to-one mapping dictionary to $N = \{0, \dots, n-1\}$ *where* N is a list of integers and breaks the dictionary into smaller chunks by hashing raw IDs.

Input: A list of raw IDs: $\{x_1, \dots, x_{n-1}\}$, where $x_i \in X$

Output: A chunked dictionary: $x_i \rightarrow n_i$ where $n_i \in N$

Initialization: Configure each map task to process a fixed number (K) of key value pairs.

Map (x_j): Let x_j be the j^{th} raw ID processed by this map task, emit key-value pair (j, x_j).

Reduce $(k_i, [r_{i1}, r_{i2}, \dots r_{ij}])$: calculate corresponding new integers according to $R_{im} \rightarrow k_i + K * (m-1)$ and emit pair (R_{im}, r_{im}) where $m \in (0 \dots j)$.

Then we apply a new MapReduce job to chunk the dictionary into smaller segments by applying the following hash function in the MapReduce shuffle phase:

$$Hash(raw\ IDs) \% num\_chunks$$

Phase 2 (Normalization)

Objective: Given a list of edges (x_i, y_i) *where* $x_i, y_i \in X$ and a dictionary $D: X \rightarrow N$, normalize each pair (x_i, y_i) into $\big(D(x_i), D(y_i)\big)$.

Input: A list of edges $(x_i, y_i) \in X$, and a dictionary $D: X \rightarrow N$

Output: A list of edges: $\big(D(x_i), D(y_i)\big)$

Initialization: Apply a MapReduce job to sort the edge lists according to raw IDs. Then apply the following job to perform normalization:

Map (x_i, y_i): Read sorted source IDs in the edge lists, load the corresponding dictionary segment $hash(x_i) \% num\_chunks$, find normalized integers, and emit a new pair $(y_i, D(x_i))$.

Reduce $(y_i, \{D(x_{i_1}), D(x_{i_2}), \dots\})$: Similar to map function, load dictionary segment $hash(y_i) \% num\_chunks$, and then emit key-value pairs $\big(D(x_{i_1}), D(y_i)\big), \big(D(x_{i_2}), D(y_i)\big), \dots$.

3.4 Graph Partitioning

Large-scale graph processing requires efficient partitioning of the graph to minimize communications across machines while balancing the computational effort. Unfortunately, most large-scale graph processing tools such as Pregel, HAMA, Trinity [21], and Kineograph [5], do not yet offer sophisticated graph partitioning methods and typically resort to simple graph partitioning based on placement of vertices or edges. These methods are simple and result in well-balanced partitions. However, these methods lead to much higher communications than sophisticated partitioning algorithms. Gonzalez et al. [6] showed that a sophisticated partitioning method could achieve ~60% graph processing performance speedup over the random method.

Graph partitioning has been studied for decades, and is an NP-hard problem with many applications in different domains. Numerous solutions have been proposed. Broadly, they are categorized into two groups: 1) offline graph partitioning [9] and 2) online (streaming) graph partitioning [22, 24]. A common offline approach is to construct a balanced k-way cut in which subgraphs are balanced over machines and communications between machines is minimized. Offline methods, such as spectral clustering [18], METIS [16], and k-partitioning [10], collect full graph information to perform offline partitioning and achieve good cuts, but exhibit poor scaling due to high computation and memory costs [1]. These algorithms perform poorly on power-law graphs and are difficult to parallelize due to frequent coordination of global graph information [1]. Online partitioning methods are proposed to address these challenges [22, 24]. These algorithms assign edges and vertices without a priori knowledge of the graph. Their goal is to find a close-to-optimal balanced partitioning while minimizing memory usage and computational complexity.

Given the complexity of offline partitioning and its limited parallelism, GraphBuilder supports online graph partitioning. Online algorithms support either edge or vertex cuts, where edges or vertices may span multiple machines, respectively. Percolation theory suggests that power-law graphs have good vertex-cuts [2], and research has shown that any edge cut can directly construct a vertex cut which requires strictly less communications and storage [6]. Given the advantages of the vertex cut approach for power-law graphs, we decided to analyze heuristic partitioning methods based on it.

We denote a graph as $G = (V, E)$, where V is a set of vertices and E is a set of edges. In vertex-cut methods, each edge e is assigned to a machine $A(e)$, where $A(e) \in \{1,2,..,p\}$ and p is the number graph partitions (shards). Each vertex v spans a set of machines $A(v)$, where $A(v) \subseteq \{1,2,..,p\}$ contains its adjacent edges. Similar to PowerGraph [6], we define the partitioning objective as follows:

$$min_A \frac{1}{|V|} \sum_{v \in V} |A(v)| \quad (1)$$

$$\text{s.t. } max_m |\{e \in E \mid A(e) = m\}| < \alpha \frac{|E|}{p} \quad (2)$$

Where α ($\alpha > 1$) is a load balance factor. $|A(v)|$ represents the number of copies (or replications) of vertex v in the cluster (a.k.a. the *replication factor*). The first equation minimizes the replication factor to reduce the communications cost, and the second equation ensures load balancing, with a small relaxation factor α. With the above partitioning objective, we designed and analyzed the following six partitioning strategies:

Random Vertex-Cuts (A1): This is the simplest method and randomly assigns edges to machines. This approach has little computational overhead and achieves good balance, but it has a high replication factor.

Greedy Vertex-Cuts (A2): improves the random algorithm by introducing heuristics to edge assignments. We aimed to minimize the replication factor by using the following heuristics to assign a new edge $e\ (u, v)$*:*

- **Heuristic 1**: if $A(u) \cap A(v) \neq \emptyset$, select a machine $a \in A(u) \cap A(v)$ to assign the edge $e(u, v)$ to. The current load $l(a)$ is increased by 1.
- **Heuristic 2**: if $A(u) \cap A(v) = \emptyset$, $A(u) \neq \emptyset$ and $A(v) \neq \emptyset$, select a machine $a = argmin_k l(k), k \in A(u) \cup A(v)$ to assign the edge $e\ (u, v)$ to. Then increase $l(a)$ by 1 and add a to $A(u)$ if a is not in $A(u)$ and $A(v)$ if a is not in $A(v)$.

- **Heuristic 3**: if $A(u) = \emptyset$, $A(u) \neq \emptyset$, or $A(v) = \emptyset$, $A(u) \neq \emptyset$, select a machine $a \in A(u)$ or $a \in A(v)$ to assign the edge $e\ (u, v)$ to. Then increase $l(a)$ by 1 and add a to $A(v)$ and $A(u)$.
- **Heuristic 4**: if $A(u) = A(v) = \emptyset$, select a machine $a = argmin_k l(k), k \in \{1,2,..p\}$ to assign the edge $e(u, v)$ to. Then increase $l(a)$ by 1 and add a to $A(u)$ and $A(v)$.

This method uses the history of edge assignments to make the next assignment decision. Our MapReduce implementation runs this heuristic in reduce tasks independently without task coordination and achieves good partitioning performance.

The above two methods can potentially assign a vertex to any machine in the cluster. By allowing a vertex v to be only replicated over a small subset of machines, or shards (denoted as s_i, $s_i \subset s$, where s is the complete set of shards), we are able to control the upper bound of the replication factor. By limiting the upper bound, so-called constraint-based approaches can potentially offer lower replication factors than random and greedy approaches.

In order to successfully assign an edge $e\ (u, v)$, constrained sets s_i and s_j corresponding to u and v should overlap. In order to get good constrained sets, we formulate the requirements of constrained sets below:

$$\forall i, j, 1 \leq i \leq n, 1 \leq j \leq n, i \neq j, s_i \cap s_j \neq \infty,\ s_i \not\subset s_j,$$

$$|s_i| = |s_j|$$

The above formulation requires that:

1. Any two constrained sets are not disjointed or have some share items
2. No constrained set is a superset of another constrained set
3. All constrained sets are the same size

The biggest challenge for this approach is how to find these constrained sets. We illustrate the following approaches:

Grid-Based Constrained Random Vertex-Cuts (A3): Vertex v is mapped into shard i in shard-grid G by using a simple hash function. Then, s_i is generated by selecting an arbitrary column and row in shard i. Following this construction, no matter which column and row we choose, constrained sets are ensured to have at least two shards that intersect any other constrained set.

For example, Figure 2 shows a 3x3 grid. If a vertex v is mapped to shard 5 according to a hash function, then its corresponding constrained set is $s_5 = \{2,5,8,4,6\}$. If a vertex u is mapped to shard 9, then its corresponding constrained set is $s_9 = \{3,6,9,7,8\}$. Given a new edge $e\ (u, v)$, it will be assigned to one of intersecting shards 6 and 8. In this method, we randomly select one shard for edge assignment. The upper bound of replication factor obtained with this approach is $2\sqrt{n} - 1$, where n is the number of shards in the cluster.

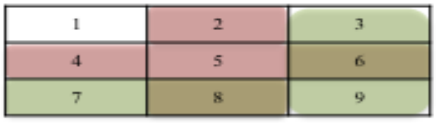

Figure 2 Grid-based constrained solution

Grid-Based Constrained Greedy Vertex-Cuts (A4): This approach is similar to the above approach, except we use greedy vertex assignment for shard selection.

Torus-Based Constrained Random Vertex-Cuts (A5): To further reduce the upper-bound of the replication factor, we used a 2D torus topology as shown in Figure 3, in which each constrained set is generated by all shards in the same column and $\frac{l}{2} + 1$ shards in the same row, where l is the number of shards in each row. For example, if a vertex v is mapped to shard 25, then its corresponding constrained set is $s_{25} = \{1, 9,17, 25,26,27,28,29\}$. If a vertex u is mapped to shard 8, then its corresponding constrained set is $s_8 = \{1, 2, 3, 4, 8,16,24,32\}$. Given an edge $e\ (u, v)$, it will be assigned to one of shards intersecting shard 1. The torus-based approach ensures that constrained sets intersect with other constrained sets in at least one shard. If there are more than one intersecting shards, we randomly select one for edge assignment. The upper bound of replication factor is $1.5\sqrt{n} + 1$.

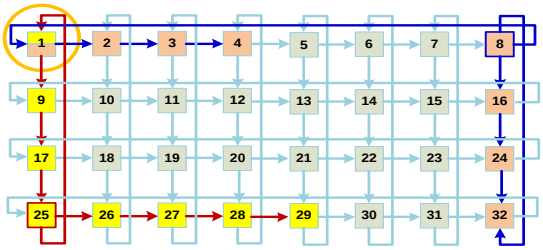

Figure 3 Torus based constrained solution

Torus-Based Constrained Greedy Vertex-Cuts (A6): This approach is similar to the above approach, except we use a greedy heuristic to select a shard from the intersecting shards.

4. EVALUATION

We evaluated the GraphBuilder framework using a full copy of the Wikipedia dataset [25], from which we constructed a page-link graph and a bipartite word-page graph on an Intel® Xeon® E5 cluster. Each node had dual sockets with 8 cores each, 64GB memory, 4x1TB SATA HDDs, and Intel® 10G Ethernet cards and switches. The page-link and word-page graphs were constructed for PageRank and topic modeling analysis, respectively.

We constructed the above graphs on both 8-node and 16-node clusters. The results are presented in Figure 4 and 5. Our analysis shows that the extract phase is the most time-consuming phase. It also reveals that its effort is dominated by the parsing of XML files for this particular data source. The partitioning phase also takes a large portion of time and the normalization phase has varying overhead that strongly depends on the application. Since topic modeling requires TF-IDF edge weights, this tabulation is included in Figure 5. Moreover, our results reveal that construction time scales linearly up to 16 nodes, and drops by almost half as we increase the cluster size from 8 to 16 nodes. Given the high degree of parallelism exhibited by each phase, we believe that our framework will scale well to larger clusters.

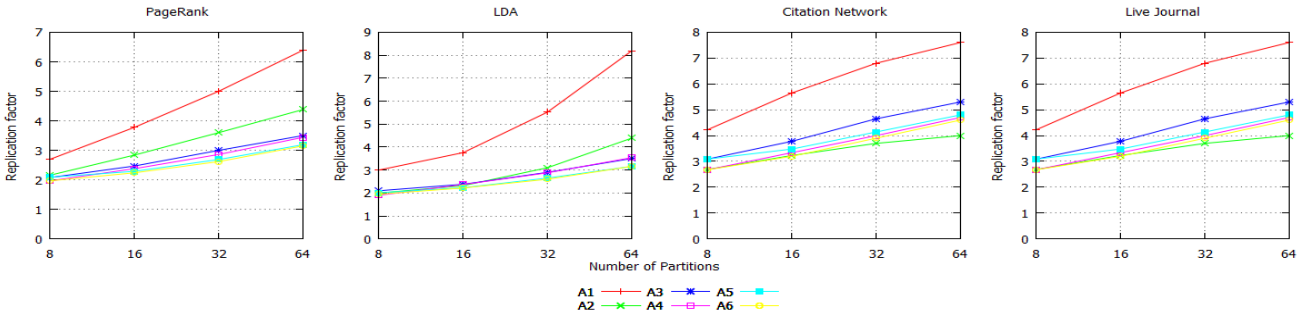

Figure 6: Replication Factor Scaling for Real-World Graphs

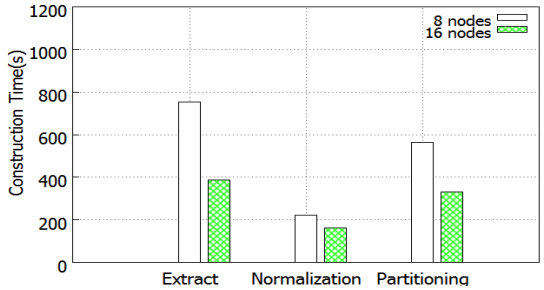

Figure 4: Page-Link Graph Construction Time

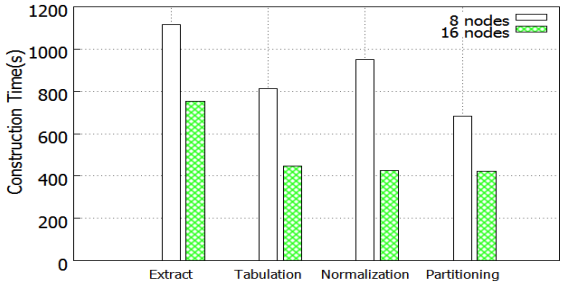

Figure 5: Word-Page Graph Construction Time

In addition to graph construction time, we also evaluated partitioning algorithms by studying their replication factor (Equation 1 in section 3.4) and load balancing (Equation 2 in section 3.4) using four real-world graphs shown in Table 2. Besides page-link and word-page graphs, we added two pre-constructed graphs from SNAP datasets [11]. Detailed graph statistics are shown in Table 2.

Table 2: Statistics for the Real-World Graphs

Graph	|V|	|E|	Power law factor
Page-Link	20M	128M	2.41
Word-Page	55M	1.4B	2.23
Citation Network	3.7M	16M	1.66
Live Journal	4M	34.6M	2.1

We studied replication factors of graph partitioning methods as a function of the number of partitions and present the results in Figure 6. The constraint-based methods scale significantly better than the greedy and random methods for all four graphs; our results show a 30% reduction in replication even with only 8 partitions. This is because constrained-based methods have a lower theoretical upper bound for replication factors (see Section 3.4). In addition to replication factor, we also captured load balancing factors for different numbers of partitions and observed that constrained-based random methods result in imbalanced partitions whereas other approaches achieve the same load balance as random. The load balance factors for 16 partitions are shown in Figure 7.

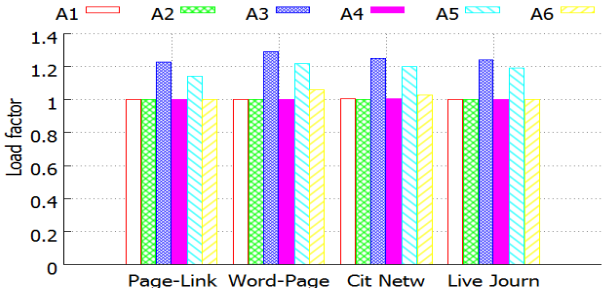

Figure 7: Graph Partition Load Balance

Partitioning has direct effect on the performance of distributed graph processing. We performed Wikipedia topic modeling over the word-page graph using GraphLab's topic modeling (LDA) toolkit. We measured both the time to load partitioned graphs into GraphLab and the computation time of the LDA algorithm on a 16-node cluster. We only used partitioned graphs generated by random (A1), greedy (A2), and {grid, torus} constrained-greedy (A4, A6) algorithms because A3 and A5 result in unacceptable load imbalance. We present the results in Figure 8 and observe the partitioning method has a large impact on performance. A6 with the least replication achieves the best execution time.

Based on our above analysis of replication factor, load balancing, and graph processing performance, we conclude that the A4 and A6 algorithms yield the best results and are two promising graph partitioning methods.

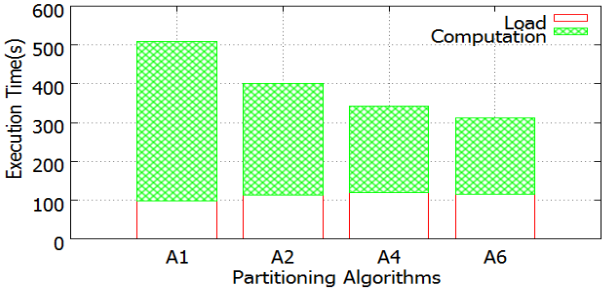

Figure 8: GraphLab Topic Modeling Toolkit Execution Time with Various Partitioning Algorithms

5. CONCLUSION

In this paper, we presented GraphBuilder, a framework that

provides large-scale graph ETL services. We conducted extensive experimental evaluations to understand the framework's performance. We analyzed several graph partitioning algorithms and evaluated their partitioning quality and impact on runtime performance. We found that two constraint-based greedy methods (A4 and A6) outperformed the other partitioning methods tested. GraphBuilder is available as an open source Java library for Hadoop MapReduce at www.01.org/graphbuilder.

In the future, we plan to: 1) study the scalability of GraphBuilder on larger clusters, 2) investigate additional graph partitioning algorithms, and 3) extend it to support stream-based graph construction.

6. ACKNOWLEDGMENTS

GraphBuilder was inspired by our collaboration with Carlos Guestrin (UW) and his extended team, namely Haijie Gu, Joseph E. Gonzalez, Yucheng Low, and Danny Bickson. This collaboration was made possible by the Intel Science and Technology Center for Cloud Computing. We would like to thank Xia (Ivy) Zhu and Kushal Datta of Intel Labs for helping us implement and demonstrate GraphBuilder, along with their team members, who provided invaluable input to the project.

7. REFERENCES

1. Abou-Rjeili, A. and G. Karypis. *Multilevel algorithms for partitioning power-law graphs*. in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. 2006.
2. ALBERT, R., H. JEONG, and A.L. BARABSI. *Error and attack tolerance of complex networks.* 2000. In Nature.
3. Andreev, K., et al., *Balanced graph partitioning*, in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*2004, ACM: Barcelona, Spain. p. 120-124.
4. Chakrabarti, D., Y. Zhan, and C. Faloutsos. *R-MAT: A Recursive Model for Graph Mining*. 2004.
5. Cheng, R., et al., *Kineograph: taking the pulse of a fast-changing and connected world*, in *Proceedings of the 7th ACM european conference on Computer Systems*2012, ACM: Bern, Switzerland. p. 85-98.
6. Gonzalez, J.E., et al., *PowerGraph: distributed graph-parallel computation on natural graphs*, in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*2012, USENIX Association: Hollywood, CA, USA. p. 17-30.
7. Gremlin. *https://github.com/tinkerpop/gremlin/wiki*. 2012.
8. Kang, U., C.E. Tsourakakis, and C. Faloutsos. *PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations*. in *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*. 2009.
9. Karypis, G. and V. Kumar, *Parallel multilevel k-way partitioning scheme for irregular graphs*, in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*1996, IEEE Computer Society: Pittsburgh, Pennsylvania, USA. p. 35.
10. L, T.k., et al., *New spectral bounds on k-partitioning of graphs*, in *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*2001, ACM: Crete Island, Greece. p. 255-262.
11. Leskovec, J. *SNAP Library at* http://snap.stanford.edu/snap/index.html 2008.
12. Leskovec, J., et al., *Kronecker Graphs: An Approach to Modeling Networks.* J. Mach. Learn. Res., 2010. **11**: p. 985-1042.
13. Leskovec, J., et al., *Statistical properties of community structure in large social and information networks*, in *Proceedings of the 17th international conference on World Wide Web*2008, ACM: Beijing, China. p. 695-704.
14. Low, Y., et al., *Distributed GraphLab: a framework for machine learning and data mining in the cloud.* Proc. VLDB Endow., 2012. **5**(8): p. 716-727.
15. Malewicz, G., et al., *Pregel: a system for large-scale graph processing*, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*2010, ACM: Indianapolis, Indiana, USA. p. 135-146.
16. Metis. http://glaros.dtc.umn.edu/gkhome/views/metis/ 2012.
17. Neo4j. *Neo4j graph database at* http://neo4j.org.
18. Ng, A.Y., M.I. Jordan, and Y. Weiss. *On spectral clustering: Analysis and an algorithm*. 2002. Advances in neural information processing systems.
19. Riedy, J. and D.A. Bader, *Massive streaming data analytics: a graph-based approach.* XRDS, 2012. **19**(3): p. 37-43.
20. Sangwon, S., et al. *HAMA: An Efficient Matrix Computation with the MapReduce Framework*. in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. 2010.
21. Shao, B., H. Wang, and Y. Li. *The Trinity Graph Engine*. 2012. Microsoft Research Asia, Beijing, China.
22. Stanton, I. and G. Kliot, *Streaming graph partitioning for large distributed graphs*, in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*2012, ACM: Beijing, China. p. 1222-1230.
23. Titan. http://thinkaurelius.github.com/titan/ 2012.
24. Tsourakakis, C.E., et al. *FENNEL: Streaming Graph Partitioning for Massive Scale Graphs*. in *MSR Technical Report*. 2012.
25. Wikipedia. http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles1.xml-p000000010p000010000.bz2. 2012.
26. Willke, T.L., N. Jain, and H. Gu. *GraphBuilder – A Scalable Graph Construction Library for Apache Hadoop*. in *Big Learning WS at NIPS*. 2012. Las Vegas.