GraphBuilder: 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. Graph construction from raw data for various applications is becoming challenging, due to exponential growth in data, as well as the need for large scale graph processing. Since graph construction is a data-parallel problem, MapReduce is well-suited for this task. We developed GraphBuilder, a scalable framework for graph Extract-Transform-Load (ETL), to offload many of the complexities of graph construction, including graph formation, tabulation, transformation, partitioning, output formatting, and serialization. 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, and performance evaluation of the framework. Since large graphs should be partitioned over a cluster for storing and processing and partitioning methods have significant performance impacts, we develop several graph partitioning methods and evaluate their performance. We also open source the framework at https://01.org/graphbuilder/.

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:

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

- SNAP [11], PEGASUS [8]: to study basic graph characteristics
- GraphLab [14], Pregel [15], Hama [20]: runtime environment for massive graph computation
- Stinger [19]: stream graph processing
- Neo4j [17], Titan [23]: graph database

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 structures. Unfortunately, tools do not exist today to efficiently and easily construct graphs with billions or trillions of vertex and edges from unstructured to structured data. While one may program MapReduce model, such as Apache Hadoop, to do this, the programmer must possess a deep understanding of not only their applications and the relevant analytics algorithms, but also know how to parallelize graph construction while using the resources efficiently. Moreover, modern graph datasets are huge and evolving so it is getting increasingly challenging to process using single commodity machine. A standard solution is to partition a large graph over a cluster. Graph partition determines load balance and communication traffics among machines, and thus has significant performance impacts on graph processing. 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 ETL framework. It provides a collection of algorithms for parallel graph construction, transformation, normalization, and partitioning. 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 detailed analysis of partition quality.
- Performance evaluation of GraphBuilder.
- Open source of the framework.

2. BACKGROUND AND MOTIVATION

All 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 exponentially growing raw data by extracting features relevant to the application. Current solutions are based on custom scripts, which are hard to maintain and extend, are only short-term solutions.

Since graphs are typically constructed from a large volume of raw data and resulted graphs also often have billions of edges and vertices, a parallel graph construction method is needed.

To understand functionality required for a graph ETL framework, we study machine learning applications with the need of graphs and illustrated four typical examples in Table 1. We find that input graphs might need TFIDF calculation or application specific edge value calculation known as tabulation. Therefore, a graph ETL framework should be flexible enough to allow users to add user-specific tabulation methods and also provide many popular built-in tabulation methods like frequency count, TFIDF etc.

Table 1: Graph-Based ML Applications Survey

Application	Algorithm	Graph	Edge Value
Topic Modeling	LDA	Bipartite	TFIDF
Ranking	PageRank	Directed	N/A
Recommendation System	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 also be partitioned over a whole cluster for efficient graph store and graph processing. What is more, many graphs follow power-law degree distribution [6, 14], randomly partitioning vertices in a cluster for these graphs easily incurs load imbalance and significant communication traffics among machines [3, 13]. Therefore more intelligent partitioning methods should be provided by a Graph ETL framework to ensure balanced computation and minimum communication across clusters for power-law graphs.

In this paper, we design and develop GraphBuilder, a scalable Graph ETL framework to address the varying requirements of graph applications, nature of graph and need to support different kinds of tabulation. To the best of our knowledge, GraphBuilder is the first open source large scale graph ETL framework.

3. GRAPHBUILDER ARCHITECTURE

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

GraphBuilder ETL architecture is based on configurable Directed Acyclic Graph (DAG) MapReduce job model, which makes it easy to tailor the ETL pipeline for the individual graph construction tools. 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, has flexibility and allows user to easily extend the framework such as user-specific vertex/edge value transformation. GraphBuilder provides a command line tool for easy use. Additionally, it exposes its interface at both job- and API-level for applications.

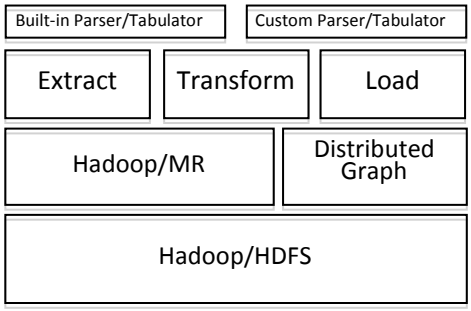

Figure 1: Architecture Overview of GraphBuilder

3.1 Graph Formation and Tabulation

Users write application-specific parsers for their data source and short routines in 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 among vertices, 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 vertices list 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. In GraphBuilder, it supplies a set of built-in tabulation functions, such as TF (term frequency), TFIDF, WC (word count), ADD, MUL, and DIV, that may be used to tabulate both vertex values and edge values. Moreover, it also provides plug in interface on both source and destination vertex [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 present the input graph in a required format for computation. GraphBuilder supports a filter for directionality conversion, duplication, dangling, and self edge removal. We design them using MapReduce model in the following manner:

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 $x_i \neq y_i$ and achieve 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 described below. Phase 1 builds a dictionary to map raw IDs to integers and chunks it into smaller segments for efficient load in phase 2. After phase 1, phase 2 first sorts edge lists based on raw source vertex IDs and then reads dictionary segments to normalize source vertex IDs. Similarly, GraphBuilder applies the same method to normalize target 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 IDs 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 pairs (R_{im}, r_{im}) where $m \in (0 \dots j)$.

Then we apply a new MapReduce job to chunk dictionary into smaller segments by applying the following hash function in MapReduce shuffling 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: We apply a MapReduce job to sort edge lists according to raw IDs. Then apply the following new MapReduce job to do normalization:

Map (x_i, y_i): Read sorted source ids in the edge lists and Load corresponding dictionary segment $hash(x_i) \% num\_chunks$, find normalized integers then 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 communication across machines while maintaining the load balance. Unfortunately, most large-scale graph processing tools such as Pregel, HAMA, Trinity [21] and Kineograph [5], have not yet explored graph partitioning methods carefully, and resort to simple graph partitioning by using random assignment of vertices or edges. These methods are simple, and result in close-to-balanced partitions. However, these methods lead to much higher communication overheads than sophisticated partitioning algorithms. Gonzalez et al. [6] shows 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 approach of offline methods is to construct a balanced k-way cut in which subgraphs are balanced over machines and communication between machines is minimized. Offline methods, such as spectral clustering [18], METIS [16], k-partitioning [10] collect full graph information to perform offline partitioning and achieve good cut, but fail to scale to large scale graphs 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 based on the information they have. Their goal is to find a close-to-optimal balanced partitioning with minimum memory usage and computational overhead.

Given the complexity of offline partitioning, limited parallelism and the evolving nature of graphs, we decided to support online graph partitioning in GraphBuilder. Online algorithm supports either edge or vertex cut, 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 its multiple partitioning heuristics.

We denote a graph to be $G = (V, E)$, 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). With vertex cut, each vertex v spans a set of machines $A(v)$, where $A(v) \subseteq \{1,2,..,p\}$ containing its adjacent edges. Similar to power graph [6], we define the partitioning objective as follows:

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

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

Where α ($\alpha > 1$) is a load balance factor. $|A(v)|$ represents the number of copies (replication) of vertex v in the cluster (a.k.a replication factor). The first equation minimizes the replication factor to reduce the communication cost, and the second equation ensures the load balance with a small relaxation factor α. With the above partitioning objective, we design and analyze six partitioning methods as follows

Random Vertex-cuts (A1): It is the simplest method, and randomly assigns edges to the 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 aim at minimizing the replication factor of vertices by using the following heuristic 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 a history of the edge assignments to take the next decision. Our MapReduce implementation runs this heuristic in reduce tasks independently without task coordination and achieves good partitioning performance.

The above two methods don't constrain the assignment of vertex and 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 of replication factors, constrained-based approaches can potentially have 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. Constrained sets intersect with
2. No constrained set is a superset of another constrained set
3. All constrained set has 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 a 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 intersected shards with 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 intersected 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.

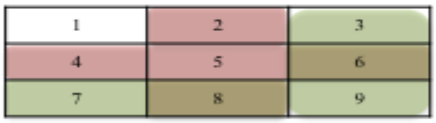

Figure 2 Grid based constrained solution

Grid-based Constrained Greedy Vertex-cuts (A4): Similar to the above approach except we use a 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 2D torus topology as shown in **Figure 3**, each constrained set is generated by all shards in the same column and $\frac{l}{2} + 1$ shards in the same raw, 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 intersected shard 1. The torus-based approach ensures that constrained sets intersect with other constrained sets at least one shard. If there are more than one intersected shards, we randomly select one for edge assignment. The upper bound of replication factor is $1.5\sqrt{n} + 1$.

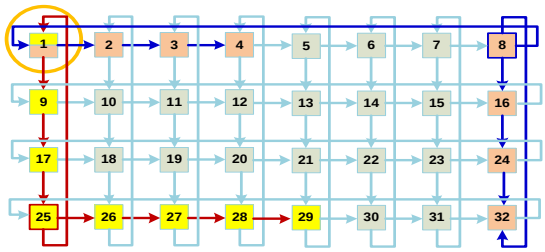

Figure 3 Torus based constrained solution

Torus-based Constrained Greedy Vertex-cuts (A6): Similar to the above method, but applied greedy heuristic to select a shard from the intersected shards.

4. EVALUATION

We evaluated our GraphBuilder framework using full Wikipedia dataset [25], by constructing page-link graph and 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 an Intel® 10G Ethernet and switch. Page-link and word-page graphs are constructed for PageRank and topic modeling analysis, respectively.

We constructed the above graphs on 8 nodes and 16 nodes clusters and presented results in Figure and 5. Our results show that extract phase is most time-consuming. Our analysis reveals that the performance is dominated by parsing XML files, which is user-specific. Partitioning phase also takes a large portion of time and normalization phase has various overheads, up to applications. Since topic modeling requires TFIDF edge weight, 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 parallelism in different phases, we believe that our framework can scale to larger clusters.

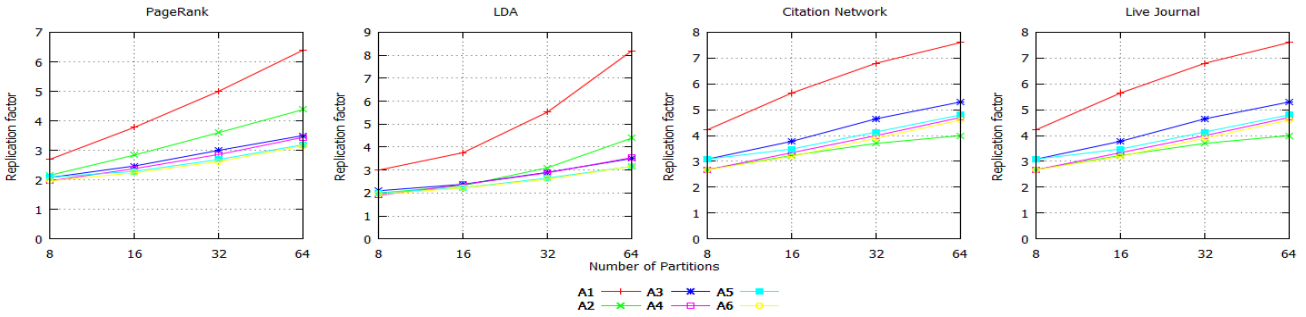

Figure 6: Replication factor scaling of real-world graph

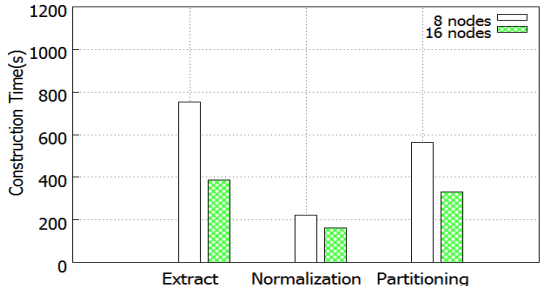

Figure 4: Page-Link Graph Construction Time

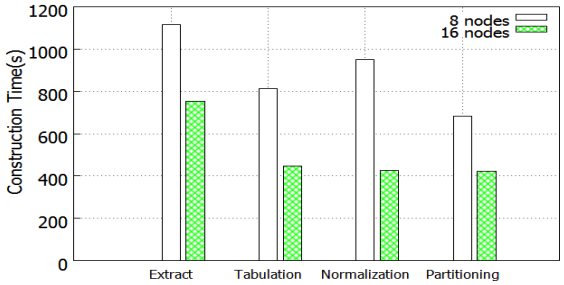

Figure 5: Word-Page Graph Construction Time

In addition to graph construction time, we also evaluated partitioning algorithms by studying their replication factor (Equation 1 in section 3.4) and load balance (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 along the number of partitions and present results in Figure 6. Figure reveals that constrained-based methods scale significantly better than greedy and random methods with the number of partitions for all four graphs; our results shows 30% reduction in replication even with 8 partitions. That is because constrained-based methods have theoretical upper bound of replication factors (see in subsection 3.4). In addition to replication factor, we also captured load balance factors with different number of partitions and observed that constrained-based random methods result in imbalanced partitions whereas other approaches achieve the same load balance as perfect random. A result of load balance factor across 16 partitions is shown in Figure 7.

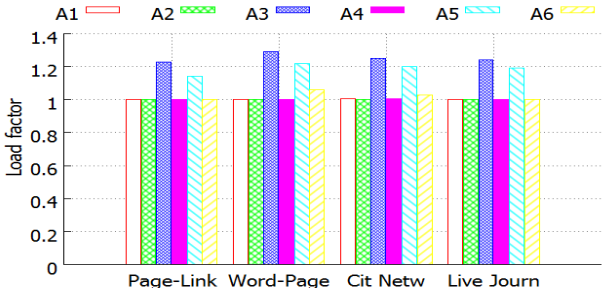

Figure 7: Graph Partition Load Balance

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 measure the time of loading partitioned graphs into GraphLab and total computation time of the LDA algorithm on a 16 nodes cluster. We only used partitioned graph generated by random (A1), greedy (A2), and {grid, torus} constrained-greedy (A4, A6) algorithms because A3 and A5 results in significant load imbalance and are excluded for comparison We present results in Figure 8 and observe that partitioning methods have large performance impacts and A6 with least replication achieves best execution time.

Based on our above analysis of replication factor, load balance, and graph processing performance, we concluded A6 and A4 algorithms yield the best result and are two promising graph partitioning methods.

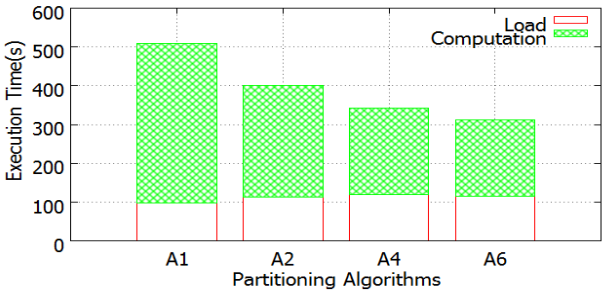

Figure 8: GraphLab Topic Modeling Toolkit Execution Time with Various Partitioning Algorithms

5. CONCLUSION

In this paper, we presented GraphBuilder, a framework that provides scalable services for large-scale graph ETL. We conducted extensive evaluations in our cluster to understand the framework's performance. We analyzed several graph partitioning algorithms and evaluated their partitioning quality and impact on runtime performance. We find that two constrained based greedy methods (A4, A6) outperform other methods for partitioning. We also open sourced GraphBuilder at www.01.org/graphbuilder.

In the future, we plan to extend our framework along the following avenues: 1) study scalability of our framework over a larger cluster; 2) investigate additional graph partitioning algorithms; 3) extend our framework 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 through Intel Science and Technology Center. We would like to thank Xia Zhu, Kushal Datta for helping us with GraphBuilder implementation, and other team members for valuable inputs.

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.