Evaluation of Parallel Graph Loading Techniques

Manuel Then
then@in.tum.de

Moritz Kaufmann
kaufmanm@in.tum.de

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT

For many exploratory graph workloads, the initial loading and construction of the graph data structures makes up a significant part of the total runtime. Still, this topic is hardly analyzed in literature and often neglected in systems and their evaluations. In this paper we analyze the whole graph loading process, including parsing, dense vertex identifier relabeling, and writing the final in-memory data structures. We present various loading strategies that take into consideration the properties of the input graph, e.g., partitioning, and evaluate them through extensive experiments.

CCS Concepts

•**Information systems** → **Network data models;** *Extraction, transformation and loading;* •**Networks** → Online social networks;

1. INTRODUCTION

Graphs are a natural way of representing many types of real-world data, e.g., relationships in social networks, links on the Web, and traversable paths in road networks. As a result, there is growing interest from both industry and academia in analyzing existing graphs to gain insights. This interest lead to the creation of many specialized systems including high-performance main memory graph analytics systems like GraphMat [14] and PGX [3, 4].

On a system level, most focus is usually put into a user-friendly query layer, and into providing an efficient runtime system for a wide variety of algorithms. This is complemented by algorithmic research on enabling more advanced graph analytics as well as on improving established algorithms by optimizing their parallelization, data structures, and data access patterns. The main metric for these improvements is the runtime of the actual algorithm.

In this paper we put the spotlight on a topic that is often neglected in system design, implementation and evaluation: *graph loading.* We show in our measurements that for many graph workloads, loading times are actually the dominant cost factor. Hence, for such workloads efficient loading is very important to minimize overall execution times.

This is the first paper that provides a systematic overview of how graph datasets in the commonly-used edge list format can be efficiently loaded into analytical in-memory graph data structures. We describe two distinct graph loading processes that optimize loading time or memory consumption, respectively (Section 2), and analyze the building blocks of these processes: parsing identifiers in the input data (Section 3), assigning dense vertex ids when required (Section 4), and generating the final graph data structure (Section 5). For this, we take into consideration properties of the input dataset, such as the existence of explicit vertex lists, or partitioning of the edge list, to improve the loading time. In addition, we apply optimizations to the final graph data structure, e.g., sorting of the neighbor lists, or degree-based vertex identifiers assignment, and measure how they affect the loading and analytics runtimes.

We provide an extensive experimental evaluation in which we measure the runtimes for various specialized graph loading processes for two well-known datasets, and show how each of the building blocks contributes to the respective loading time. Also, we compare against the loading times of two existing systems and show that our optimized loading process is more than an order of magnitude faster. Our paper shows that there is no one size that fits all, and as a consequence that systems should be adaptive to the properties of the input data as well as the analytics algorithm that should be run.

2. GRAPH LOADING PROCESS

Depending on the source and data format of the input graph, as well as on the properties of the created in-memory graph representation, the graph loading process comprises several stages. Graph datasets that are not in a system-specific binary format must first be *parsed* to extract the actual graph data, i.e., vertices, edges, and, if necessary, properties that are attached to them. Many systems then *relabel* the original vertex identifiers from the dataset to numbers in a dense range. This relabeling enables direct addressing in the internal graph representation as well as in temporary properties used by analytical algorithms. The relabeled graph data is then used to *construct* the final in-memory representation of the graph.

In each stage of the loading process, there are several possible approaches with different tradeoffs. Most approaches can be combined to compose a tailored graph loading process for a specific scenario. Besides minimizing the graph

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GRADES 2016 , June 24, 2016, Redwood Shores, USA.

© 2016 ACM. ISBN 123-4567-24-567/08/06…$15.00

DOI: 10.475/123_4

loading time, other factors can also influence the choice of an optimal approach in each stage. For example, in case of strict limits on the amount of available main memory, large graphs can only be loaded with approaches that have low memory overhead. Also, some approaches require reading the input data multiple times. While multiple iterations over the input are possible when the graph is loaded from a file, repeating a database query that generates the graph is potentially expensive. In this paper we distinguish two general graph loading processes which we describe in the following. They explore the tradeoffs between low loading times, memory consumption, and multiple data reads. In all variants we make full use of all processors in the system.

2.1 Single-Pass Process

The *single-pass graph loading process* reads the input graph data and stores it in a temporary in-memory data structure from which the final graph is created. Since this temporary structure contains all information about the graph, the input data need only be read once. However, sufficient memory must be available.

In case the input graph contains an explicit list of all vertices, the first step of the loading process is to parse this list and create a dense vertex relabeling from it. Should no vertex list be available, the relabeling is created when the actual graph edges are read. In the process's second step, the graph edges are parsed and stored in thread-local neighbor lists that are partitioned by source vertex. For many of the relabeling approaches described in this paper it is possible to apply an *initial relabeling* at this point to reduce the neighbor lists' memory footprint. For example, if a dataset uses 64-bit identifiers but has less than 4 billion unique vertices, they could be densely mapped to 32-bit identifiers during the initial relabeling. After all input edges are written to the neighbor lists, the lists' sizes are aggregated to count the number of neighbors per vertex. This aggregation is especially efficient when the input edges are known to be *partitioned* such that all neighbors of a vertex appear consecutively. The reason for this is that for partitioned edge lists it can be ensured that all of a vertex's neighbors exist in the same thread-local list. Afterward, if necessary, the vertex relabeling is finalized and used to write the final graph data structure. We depict the single-pass process in the upper half of Figure 1.

2.2 Two-Pass Process

In contrast to the single-pass process, the *two-pass graph loading process* does not require storing all graph edges in a temporary data structure. The general loading process is similar but requires that the input data is read twice. During the first iteration, the two-pass process counts each vertex's neighbors and creates the final relabeling. When the input data is read for the second time, the edges are directly relabeled and written to the final graph data structure. We show the two-pass process in the lower half of Figure 1.

As in the single-pass process, an explicit list of all vertices as well as knowledge about partitioning of the edge list can be used to speed-up the graph loading.

3. PARSING

The graph data parsing step transforms external graph representations, e.g., edge list files, into a system-specific binary format from which the graph data structures can be generated. While graphs can be created from arbitrary data, they are usually specified as pairs of numbers (x, y). The numbers x and y identify vertices in the graph, and the pairs themselves denote edges from x to y in a directed graph, or between the entities in an undirected graph. Other graph representations can easily be translated into this format. The parsing step then comprises finding the tokens that form the graph's edges and loading their identifiers into the system-specific binary format. Finding the identifiers of interest is especially challenging when the input data is split into chunks that are processed in parallel.

In this paper we evaluate two common edge list formats. Vertex lists or additional data, for example properties, can be parsed similarly.

3.1 Binary Edge Lists

Graph generators, like the Kronecker generator used by the Graph500 benchmark [1], frequently write *binary edge lists* that contain fixed-size vertex identifiers. In contrast to proprietary formats, they are simply a binary serialization of the common edge list format. Because all identifiers have a fixed width, this format is *space-efficient*, as identifier commonly fit 32-bit numbers, *efficient to read*, and *easily split into chunks* for parallel processing.

During parsing we directly copy each edge into our system-specific format.[1] As a result, reading from binary edge lists has only negligible parsing cost. We, thus, use it as the baseline in our evaluation.

3.2 Decimal Edge Lists

More common than binary edge lists are character-encoded *decimal edge lists* which can be found in many dataset repositories like SNAP [8] or KONECT [6]. Because decimal-serialized vertex identifiers have varying lengths and must first be transformed to an internal binary representations, human-readable edge lists are harder to parse. In the following we describe three variants of doing so.

3.2.1 General Number Input Operations

Most programming languages that are widely used provide functionality to parse decimal numbers in files or character strings. In our evaluation we chose to measure the performance of parsing numbers from input streams using the C++ standard library. The standard library's number parsing is designed in a very general way and covers many special cases like reading numbers in scientific notation. As we show in our evaluation, its performance characteristics are not desirable for high-performance graph loading.

3.2.2 Iterative Identifier Parsing

A more optimized and commonly used way of parsing decimal identifiers is by iterating their digits to sum the final value. For a character iterator `iter` and a `delimiter` that ends the identifier, we thus parse decimal identifiers as follows.

```
1 int id = 0;
2 while (*iter != delimiter)
3   id = id * 10 + (*(iter++) - '0');
```

[1]In case the endianness of the input binary does not match the one used by the CPU, an additional but very efficient change of the identifiers' byte order is necessary.

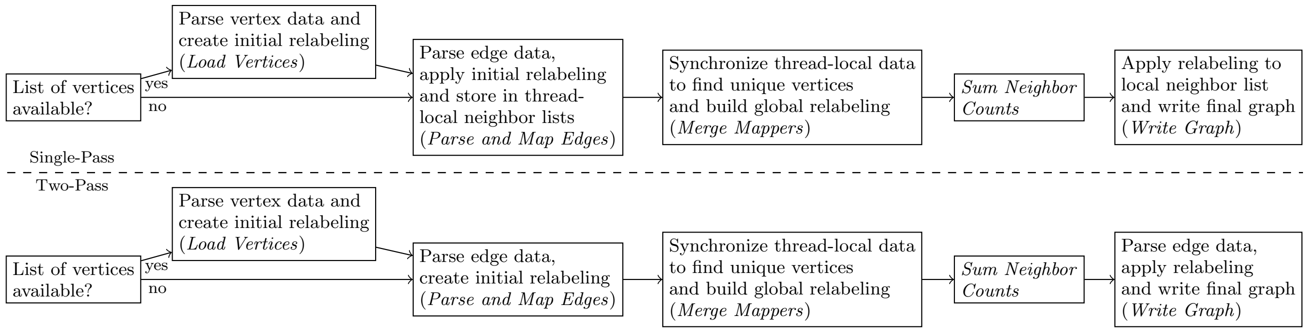

Figure 1: Graph loading process

For each read digit we subtract the '0' character to efficiently translate its ASCII value to the binary representation. Compared to the previously described standard library parsing, such optimized decimal parsing can lead to more than an order of magnitude speedup in this phase.

3.2.3 Vectorized Identifier Parsing

Using the wide vector instructions available in modern CPUs, iterative decimal identifier parsing can be vectorized. We utilize techniques presented in [9, 5] which show slight speedups over iterative parsing for long identifiers.

3.3 Code Generation

For general-purpose graph analytics systems it is useful to let the user specify the format of the external graph data through configuration files. This approach is for example used by PGX [4]. The downside of such flexibility is that the graph loading code is more complex especially in terms of branches, which negatively impacts the graph loading performance. To avoid performance impacts, runtime code generation [11] can be used to construct a specialized loading pipeline for each input format. In our evaluation, each loading pipeline is specifically compiled for the scenario at hand. However, for a lack of space we did not further explore runtime code generation techniques.

4. IDENTIFIER RELABELING

When graph data is loaded from files or from the results of database queries, the contained vertex identifiers $v \in V$ are usually *sparse*, i.e., $V \neq [0, |V|-1]$ where $|V|$ is the number of distinct vertices. *Directly* using these potentially sparse identifiers in the system-internal graph representation avoids overhead from assigning new identifiers to vertices during loading. However, it makes necessary an additional mapping when accessing vertex-related data such as neighbor lists and vertex properties. To avoid these runtime overheads, many graph analytics systems instead relabel the original dataset's vertex identifiers to dense $v' \in [0, |V|-1]$ such that there is a bijective mapping between corresponding v and v'. In the following we discuss relabeling strategies that can be used during graph loading. They differ in what data is stored per worker thread and when the relabeling is created.

4.1 Global Mapping

In the *global mapping* strategy, one global relabeling is shared by all worker threads. When a new, i.e., not yet relabeled, external vertex identifier is found, a new dense identifier is requested from a global atomic counter and inserted into the global mapping. Using this mapping strategy, the temporary data structures of the single-pass graph loading process can directly store relabeled vertex identifiers. When assigning identifiers in parallel, two threads may try to allocate an identifier for the same vertex. This case can be avoided through additional locking, or by a post-processing step that removes the resulting holes, i.e., allocated but unused ids.

In addition, we evaluate a global mapping strategy that uses a local cache to avoid lookups in the frequently modified global mapping.

4.2 Local Mapping

Similar to global mapping, the *local mapping* strategy directly creates a relabeling for vertex identifiers while the input data is read. Hence, it also allows initial identifier relabeling. In contrast, the local mapping strategy creates thread-local mappings. As a result, no data is shared between threads. To avoid collisions among the newly-assigned identifiers, workers request ranges of dense identifiers from a global pool before using them. Depending on the distribution of edges in the input data and the numbers of edges per vertex, an input vertex may be relabeled differently in many local mappings. Once all input data is processed, the thread-local mappings are merged, creating a global identifier mapping that is used in the final relabeling step.

4.3 Global Collection

In contrast to the previously described strategies that directly create mappings, the *global collection* strategy first collects the input data's unique vertex identifiers in a global set. Once all edges have been parsed, a relabeling for the vertex identifiers is created. As the global collection strategy works similarly to a separate vertex list, they should not be used together. Also, the global collection strategy does not allow initial relabeling. Thus, the single-pass loading process must store the potentially larger external vertex identifiers in its internal data structures.

4.4 Local Collection

The *local collection* strategy combines local mapping and global collection. It collects the used external vertex identifiers in thread-local sets, which are merged and used to create the global mapping once all edges have been read. The local collection strategy shares the previously mentioned limitations of global collection.

5. GRAPH DATA STRUCTURES

Depending on the requirements of an algorithm or system, various graph data structures are used in practice. For this

paper we focus on read-optimized in-memory graph structures that are well-suited for analytics. Depending on the decision whether or not vertex identifiers are mapped to dense ranges, commonly used data structures are the compressed sparse row (CSR) and maps of neighbor lists, respectively.

5.1 Compressed Sparse Row

The CSR format comprises two parts: an array containing the concatenated neighbor lists of all vertices in the graph, and an array of offsets at which each neighbor list begins in the first array. Accessing a vertex v's neighbors in the CSR only needs a single indirection: The vertex's neighbor list offset is read in the offset array at position v, and used as indirection into the concatenated neighbor lists. The neighbor count of v is the difference of the offsets of v and the vertex with the subsequent identifier $v + 1$.

From a graph loading perspective, this format is challenging because multiple workers must be able to write into the same array, preferably without synchronization. We solve this challenge differently depending on the loading process.

For single-pass loading, all neighbor information is in memory, partitioned by worker. To write the neighbors array, we then assign disjoint vertex ranges $[u, v]$ to the workers, and let them write all partitions' neighbors for each $n \in [u, v]$. The initial offset of the ranges' first vertex u is easily calculated as $\Sigma_{m<u}|neighbors(m)|$.

In the two-pass loading process, the edge list is iterated a second time, directly writing the CSR's neighbor array. For non-partitioned edge lists in which the neighbors of a vertex may be located anywhere, multiple workers thus even need to be able to write to the same neighbor list concurrently. We allow this in a synchronization-free manner by computing per-vertex offsets in the neighbors array from which each worker can safely write.

5.2 Map of Neighbor Lists

The CSR data structure relies on dense relabeling of the vertex identifiers. It can, thus, not be used when the external vertex identifiers are directly used in the internal data structures. Instead, a map of neighbor lists can be used. The two major disadvantages of this format are that (a) finding a vertex's neighbors involves an additional indirect map lookup, and that (b) neighbor counts must be stored explicitly. For a fair comparison we use the same array of concatenated neighbor lists as in the CSR. Consequently, the data structure creation process is similar.

5.3 Optimization: Sorted Neighbor Lists

For many algorithms it is beneficial if the neighbor lists are sorted by identifier. Hence, we measure how sorting each vertex's neighbor list affects the overall graph loading performance. As an example, consider the triangle counting algorithm which relies on efficient neighbor list intersections. Instead of requiring a generic set intersection, for sorted neighbor lists the algorithm can use an efficient merge join.

6. EVALUATION

In this section we evaluate how the the presented building blocks influence graph loading times. There are hundreds of possible permutations in which the building blocks can be combined to a graph loading process. Hence, we limit our evaluation to the most significant results.

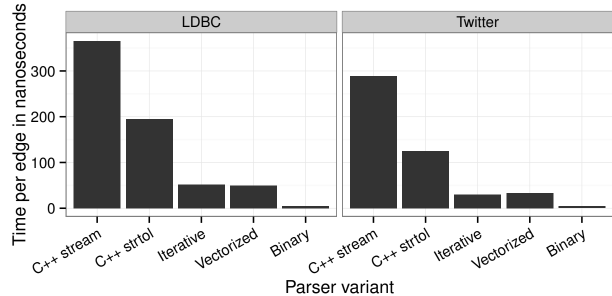

Figure 2: Parsing time per edge, varying parsers and graphs.

6.1 Experiment Setup

We implemented the presented graph loading techniques in C++14 and compiled them using GCC 5.2.1. All measurements were made on a dual-socket machine with two Intel Xeon E5-2660 v2 CPUs (2×20 logical threads at 2.2GHz) and 256GB of memory. As operating system we used Ubuntu 15.10 with kernel 4.2.0.

We evaluated the graph loading process using two publicly available small-world networks: the LDBC SNB graph [2] at scale factor 1000 and the Twitter graph [7]. The LDBC graph contains 3.6M vertices and 447M edges, and has a file size of 12GB. The Twitter graph has over ten times more vertices but only three times as many edges; it contains 41.6M vertices and 1.5B edges in a 25GB dataset.

The datasets exist in two variants: with their edges *partitioned* by source, and with randomized edge order. For our experiments we stored them in a ramdisk to avoid disk access and caching overheads.

6.2 Parsers

In Section 3 we introduced several parsing techniques: general parsing operations, specialized iterative and vectorized identifier parsing, and reading binary data. We evaluate their performance for parsing the edges list of each dataset. All parser measurements are done single-threaded to avoid effects from insufficient memory bandwidth. Figure 2 shows our results. We normalized the runtimes for the datasets by dividing them by the number of parsed edges; hence, the parsing time is given in nanoseconds per edge.

Using the C++ Standard Library's optimized number parsing `strtol` exhibits a 2-3x speedup over general stream input parsing. The optimized identifier parsing functions exhibit an additional 8-11x speedup. For our scenario, we did not find a significant difference between iterative and vectorized parsing. Also, when only parsing of partitioned edge lists is considered, the overhead of caching the parsed source vertex, did not pay off in our experiments. We use the iterative parser for all subsequent measurements.

We found a significant difference in the parsing time per edge between the datasets. On average, it was 50% lower for the Twitter dataset. This is explained by the datasets' difference in average identifier length of 13 and 8 characters per identifier for LDBC and Twitter, respectively.

When binary files are used, no parsing is necessary. This variant is additional 8x faster than iterative parsing.

6.3 Relabeling Strategies

We evaluated how the identifier relabeling strategies introduced in Section 4 influence the runtime of the parallel graph

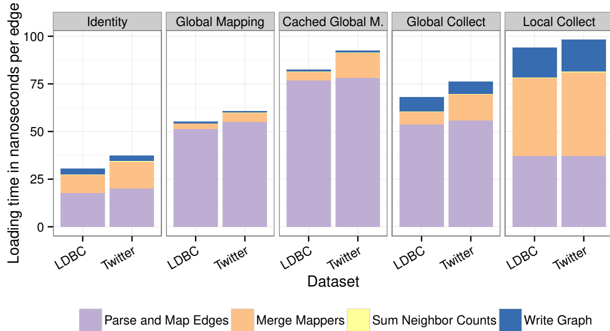

Figure 3: Loading time for varying mappers and stages.

loading process. Figure 3 depicts the—again normalized—runtimes for creating the final in-memory graph representation from the input dataset's non-partitioned edge list. It further shows how the various stages contribute to the loading time. The stages' names follow Figure 1. We only show results for the single-pass loading process as the two-pass results exhibit the same trends. Note that the runtime per edge can be lower than the parsing times in the previous section because we use all 40 CPU cores in this experiment.

It can be seen that the *identity* relabeling strategy which does not perform any relabeling performs best. Its main cost factor is determining all unique vertex identifiers in the *Merge Mappers* phase, e.g., to get the number of vertices in the graph. The *global mapping* strategy is less than two times slower and ensures that the internal graph data structures use dense identifiers, which greatly improves the performance of analytics algorithms on this data, as we show in Section 6.5. The *cached global mapping* strategy which adds a thread-local cache for already assigned identifiers performs worse because of the additional pollution of the CPU's shared L3 caches. Also using a shared data structure, the *global collection* strategy exhibits similar performance as global mapping in the *Parse and Map Edges* phase. It has, however, additional overhead from creating the relabeling and applying it in the later phases. While the *local collection* strategy provides performance benefits in the initial edge parsing phase, as no synchronization between the threads is necessary, it performs worst overall because of the expensive merging of all locally stored external identifiers step. Both the local and global collection strategy involve expensive relabeling when the final graph is written because all vertex identifiers are relabeled to dense identifiers at this point. The *local mapping* strategy has similar disadvantages as the local collection strategy; we omit its detailed evaluation because of a lack of space.

6.4 Process and Properties of Input Data

In Section 2 we introduced the single-pass graph loading process which builds a temporary representation of the graph in memory and then creates the final graph from it, and the two-pass process that iterates over the input data twice, first preparing statistics about the graph and then writing it directly to the final data structure. Now, we evaluate how these two processes perform for graph datasets that (a) have edge lists partitioned by source vertex, or are non-partitioned, and (b) have explicit lists of all vertices in the dataset, or not. We performed each measurement using the two best-performing relabeling strategies from the previous section: identity and global mapping.

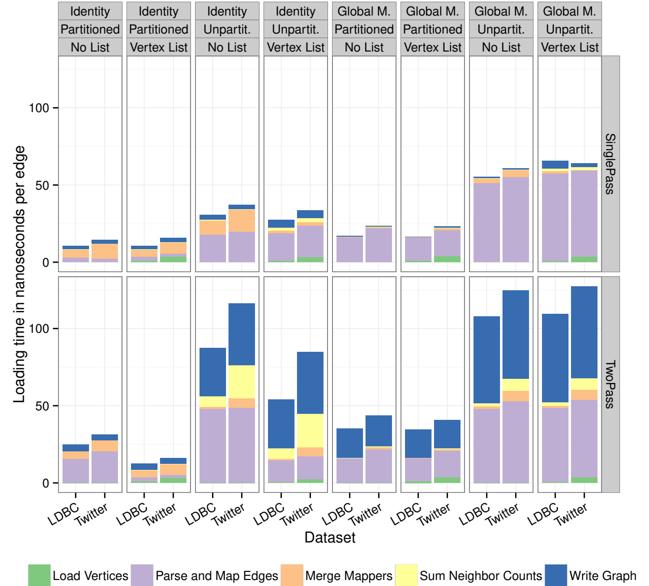

Figure 4: Graph loading times when having vertex files and partitioned edge lists.

We show our results in Figure 4. As is to be expected, the more beneficial properties a graph dataset has, the faster it can be loaded. Thus, when there is an explicit list of vertices in the graph and the edge list is partitioned, the graph can be loaded fastest, independent of the used process or relabeling strategy. For edge lists that are non-partitioned, the presented results are equal to the ones presented in the previous section.

The single-pass process exhibits lower runtimes than the two-pass loading process because the latter must parse the input data a second time to write the final graph. This additional parse iteration is especially disadvantageous when the edge list is not partitioned, as indicated by the blue bars. However, we found that the two-phase approach consumes around 50% less memory during graph loading. Hence, it allows loading larger datasets given a fixed amount of memory.

6.5 Graph Data Structure and Algorithms

In this section we evaluate how the generated graph data structure and its internal ordering influence loading and analytics runtimes. To that end we loaded the Twitter dataset and ran two well-known algorithms: PageRank (PR, 20 iterations) [12], and triangle counting (TC) [13]. We stored the graph in a densely relabeled CSR, as well as in a map of neighbor lists that uses identity mapping. Furthermore, we created each data structure twice; once with sorted and once with unsorted neighbor lists. Because set intersections are inefficient on unsorted neighbor lists, no TC results are given for this case. In Table 1, we report the graph loading time from a non-partitioned edge list, the algorithm runtime, and the resulting overall workload runtime.

Our results show that while identity mapping exhibits the best runtimes for loading alone, its benefits are quickly outweighed by the additional mapping overhead at algorithm runtime. Furthermore, we show that sorting the neighbor lists in the graph is a relatively cheap operation. It is a pre-

Table 1: Twitter loading time and algorithm runtimes.

	Alg.	CSR			Map		
		Load +	Run =	Total	Load +	Run =	Total
Sorted	PR	$37s$	$33s$	**70s**	$25s$	$194s$	**219s**
	TC	$37s$	$49s$	**86s**	$25s$	$66s$	**91s**
Unsorted	PR	$34s$	$33s$	**67s**	$21s$	$193s$	**214s**

requisite for our TC implementation, but does not provide a performance significant benefit for PR.

Even using our highly-tuned loading process, the initial graph loading can take as long as, or even longer than the actual algorithm. Thus, optimizing graph loading times can significantly improve the overall runtime of exploratory workloads.

6.6 Overall Loading Times

This paper aims at inspiring graph analytics systems designers to improve their systems' loading times. To do so, we compared the edge list loading times of the approaches presented in this paper with two publicly available state-of-the-art competitors: the in-memory graph analytics system PGX 1.2.1 [3, 4], and the benchmarking framework GraphBig 3.2 [10]. Table 2 shows our measurements.

Table 2: System graph loading times.

System	Twitter	LDBC
PGX	2153s	632s
GraphBig	*out of memory*	1682s
Ours *Non-partitioned*	89s	24s
Ours *Partitioned*	34s	7s

PGX can load both the non-partitioned LDBC and Twitter dataset within the memory limits of our evaluation machine. Its loading times are about 25x slower than our non-partitioned results. However, it must be noted that the system created two CSRs that store in-edges a out-edges, respectively. In contrast, *GraphBIG* only utilized a single CPU core during loading, and was only able to load the LDBC graph. For this dataset it was about 70x slower than our parallel approach. The system was not able to load the Twitter dataset with the 180GB of main memory we allowed it to consume.

7. CONCLUSIONS

In this paper we gave an overview of various graph loading strategies and evaluated them experimentally. We demonstrated that there is potential for existing graph analytics systems to dramatically reduce loading times by leveraging the input datasets' properties and fully utilizing the parallel compute resources of modern machines.

8. ACKNOWLEDGMENTS

Manuel Then is a recipient of the Oracle External Research Fellowship.

9. REFERENCES

[1] The graph 500 benchmark. http://www.graph500.org/specifications. Accessed: 2016-01-10.

[2] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[3] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[4] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. ACM, 2015.

[5] M. Kaufmann, T. Mühlbauer, M. Then, A. Gubichev, A. Kemper, and T. Neumann. Hochperformante analyse von graph-datenbanken. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 311–330, 2015.

[6] J. Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 1343–1350. International World Wide Web Conferences Steering Committee, 2013.

[7] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[8] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[9] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, 2013.

[10] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. Graphbig: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. ACM, 2015.

[11] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[13] M. Sevenich, S. Hong, A. Welc, and H. Chafi. Fast in-memory triangle listing for large real-world graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis*, page 2. ACM, 2014.

[14] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.