Evaluation and Analysis of Graph Navigation query in Edge Streaming Database

Tarek AOUKAR† and Jun MIYAZAKI†

† Department of Computer Science, School of Computing, Tokyo Institute of Technology
Ookayama, Meguro-ku, Tokyo 152-8550, Japan
E-mail: †{aoukar,miyazaki}@lsc.c.titech.ac.jp

Abstract The data generated by users online is forever increasing in both size and format. Various data structures and data layout has been proposed to handle this increase, each with its own advantages, disadvantages, and use scenarios. This paper provides additional evaluation and analysis of the previous work that combined different graph processing approaches (sequential access and edge linkage) to achieve a balanced performance on different workloads simultaneously. We evaluated simple navigation queries, and we refer to work in progress for improving: 1) thread-level parallelism, 2)navigation query using oneapi TBB components, and 3) memory operations with an object pooling technique.

Key words database core technology, graph database, data structures

1 Introduction

The usage of graph data in various fields is forever increasing, from social networks to web graphs, routing algorithms, recommendation and navigation systems, graphs play an essential role in our everyday usage of the internet. Due to the essential role of graph data format, various graph storage and graph processing solutions has been introduced over the years, each with a target workload and trade-offs. Many relational databases (MS SQL, Oracle, etc.) offer graph processing engine to allow processing graph data while maintaining the data storage in relational format. Meanwhile, pure graph databases like Neo4j and PowerGraph store and process their data in graph format.

Graph databases can be categorized based on their system architecture to: a) distributed systems: as a natural solution for the ever growing data size distributed systems such as Pregel [2], PowerGraph [3], Chaos [4], and b) out-of-core systems such as GraphChi [5], X-Stream [10] and others [7]. Distributed systems are a natural solution for the ever growing data size, and have the ability for very large scale graph processing powered by their scale-out architecture. On the other hand, out-of-core systems can avoid the difficulties of distributed systems such as synchronization, load balancing, and clustering. Graph databases can also be categorized based on their focus into one of two categories: 1) Navigation query oriented ones such as Neo4j [9] and DGraph [17], and 2) Global graph processing (edge streaming) oriented ones such as GraphChi [5] & GridGraph [7]. The former focuses on navigation queries such as searching n-hop neighbors, and aims to obtain high performance by optimizing the data structure and data layout used by the data storage and processing engine, while the latter focuses on analysis on the graph as a whole for running algorithms such as PageRank, WCC, SpMV, etc, by optimizing their data structure and data layout for sequential access.

In our previous paper [1] we proposed a system that aims to balance between global graph processing and navigation queries by exploiting the data structures used by navigation query oriented databases to reduce the amount of accessed data, and the data layout from global graph processing databases to maximize the sequential access to a hard disk.

In this paper we focus on the evaluation of our solution and the analysis of the results we obtained. The rest of the paper revisits the previously mentioned related work and proposed solution, followed by the evaluation results of the solution and analysis, then conclude the paper.

2 Related Work

As we previously mentioned, multiple systems have been proposed [6] [8] [11] and are available as both research oriented and commercial products, we briefly introduce some of those databases and their core concepts.

2.1 Distributed Systems

Multiple distributed abstractions for graph-based computation has been implemented, each follows a different computation model. Vertex centric computation model encourages you to think like a vertex, passing messages through edges, and running algorithm in iterations with superstep boundaries for synchronization. Pregel [2], a simple framework for graph computation that is both scalable and fault tolerant hidden behind a simple API, follows this computation model. Gather, Apply, Scatter (GAS) computation model on the other hand solves problems that arise with natural graphs containing highly skew power-law degree distribution. Carry computation over edges instead of vertices increases parallelism and prevents lagging workers at super nodes. PowerGraph [3] follows this computation model. TigerGraph [18] is a native graph database heavily optimized and designed for massive parallel graph analysis. It contains automatic partitioning of graph and a hash index is used to determine which server data belong to. TigerGraph equips a query language, called GSQL, which is friendly to both bulk-synchronous-programming and map-reduce users.

2 2 Single Machine Systems

Single machine systems tend to prefer the GAS computation model. X-Stream [10], a scale-up graph processing system, adopts edge centric implementation that streams unordered edges from storage instead of random access for gaining advantage of sequential access to a hard drive. GridGraph [7] partitions a graph nodes into 1D partitions, then building a logical edges grid in which each cell contains directed edges from a source nodes partition to a destination nodes partition. It scales well with memory and disk bandwidth, outperforming systems like X-Stream [10] and GraphChi [5]

2 3 Flat Data-center

With the developing technologies and low latency high bandwidth network equipment, small clusters in data centers can be viewed as a flat storage, accessing memory locations from machines in the same cluster is much faster than reading from hard drive. This is true under few assumptions like storage bandwidth is slower than network bandwidth, and aggregated network bandwidth from all machines in cluster is lower than switch bandwidth. Chaos [4] exploits those assumptions and uses streaming partitions similar to XStream [10] for sequential access and parallelizes it across machines. Pre-processing of a graph is trivial, splitting vertices, edges, and intermediate data to a randomly selected storage device in the cluster, in a uniform distribution. Work stealing is implemented to balance workload as well. It scales out to 32 machines with slight overhead of $1.6x$ increase in runtime.

3 Proposed method

Some previously mentioned databases sort the edges or index them, which allows locating the related edges with a few random accesses to the index. However, this approach will incur multiple random accesses to load the edges, unless the edges are sorted. This leads to great overhead in sorting edges at first place, along with the complexity of keeping them in order as the graph evolves. On the other hand, using sequential access, streaming large number of edges, and selecting the required edges to process could be a solution for making graph processing faster without much overhead. A key feature that allows navigation oriented graph databases to have a superior performance compared to global processing graph databases is linkage information where edges contain extra data in a pointer like fashion, which allows them to quickly jump between related edges. However, the storage of this information comes at extra I/O cost.

In our database, we aim for acceptable performance among all queries, therefore, linkage information is needed. Since it is not used during streaming, it is stored as a separate record from the edges. The data layout is similar to GridGraph to take advantage of faster initialization and easier maintenance. The used data structures and data layout will be further explained in details.

Our system currently uses key-value storage engine LevelDB [13] to store disk pages (to be discussed in the following subsection)

3 1 Data Structures

The three main data structures are as follows: 1) graph node, 2) graph edge, and 3) edge linkage. The graph node stores a relationship pointer that points to an edge of the graph in which the node is either a source or a destination, and a property block pointer. The graph edge holds a pointer from a node source node to a destination node, as well as a property block pointer. The edge linkage contains useful navigation pointers which are used as a helper for navigation queries to reduce the amount of streaming needed. Figure 1 below illustrates data structures and how they are connected.

ID of each record is the actual offset of the record in the database, and each record holds a validity bit that is used for fast record deletion without overwriting the record immediately. Property block pointers in each record can be used to store 4 bytes of arbitrary data, such as the weight of an edge, if the user data is small. Records are grouped in disk pages (a byte array container), and each record manipulates its own data directly into the associated disk page to facilitate easier data retention mechanism alternative to detecting record data update and copying data over.

3 2 Data Layout

A disk page is organized in the same manner as Partition Attributes Across (PAX) [14] where fields are grouped together, which improves performance in certain operations

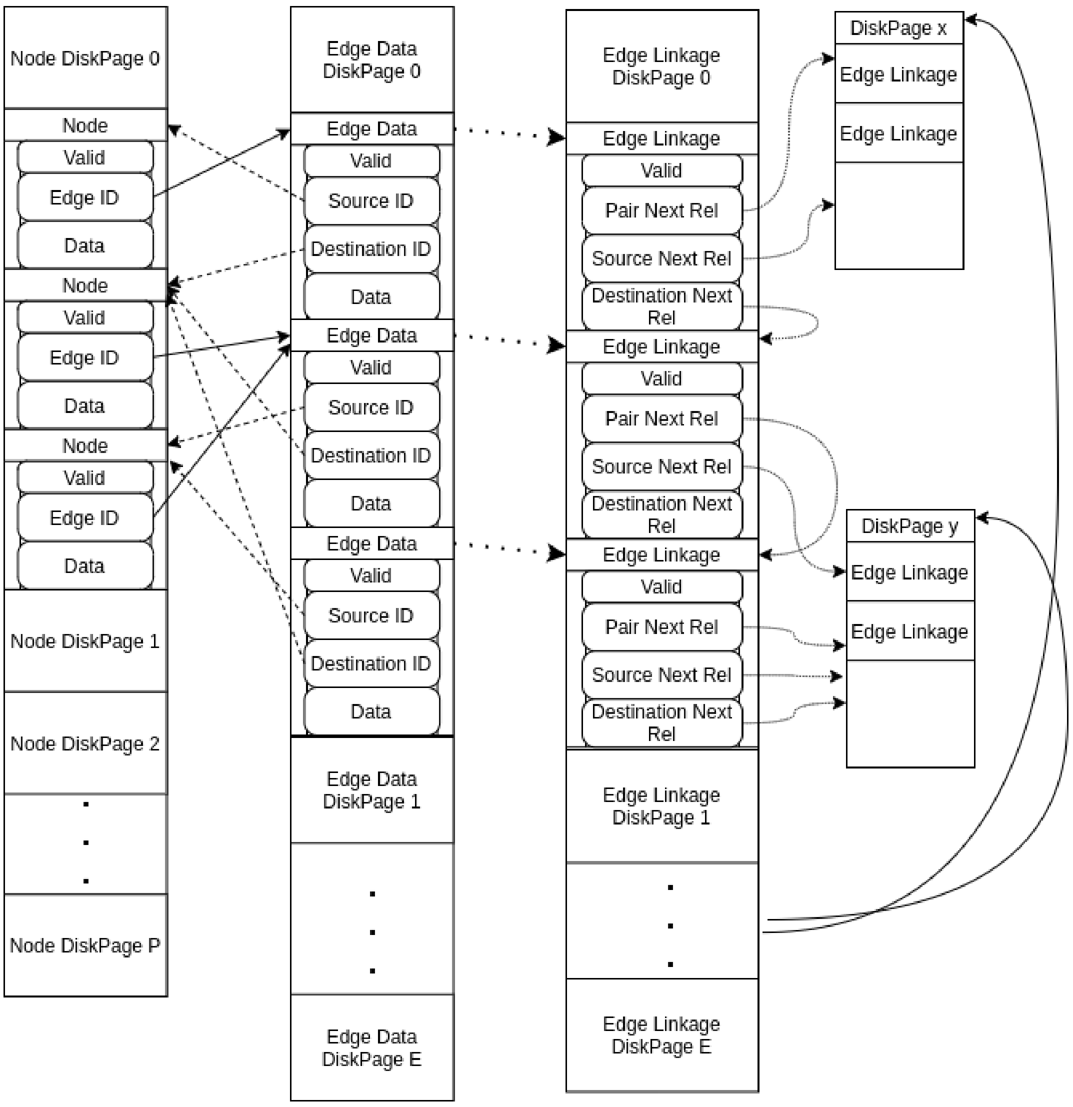

Figure 1 Data Structures Relations

such as finding edges with a certain source or destination in a page due to cache access.

As in GridGraph, the nodes of the graph are split into P partitions, and edges are split into $P \times P$ grid where a cell in position (i, j) holds the edges with source nodes in partition i and destination nodes in partition j.

Edge grid data is stored in a sequential manner according to destination and source partitions allowing to sequentially stream all edges with the same destination partition, and therefore, allowing immediate update of node values. This can help some global graph algorithms to converge faster. In a similar fashion to edge data grid, the edge linkage data is also stored in the grid format and only loaded when required by graph navigation queries to reduce the I/O and memory usage

3 3 Page Pooling

A fast thread-safe page pool is implemented to reuse disk pages at the end of their life-cycle. The pool keeps atomic count for each disk page to track which pages has are not used, and re-initialize them to reduce memory allocation time. The current reuse policy is a simple "reuse all when no free pages exist" policy, while it's a straight forward application, it includes a slight time delay when searching for all unused pages. In the future, we hope to implement a background thread with other policies like LRU or LFU.

3 4 Pre-processing

Pre-processing imports edge list input file into the database in 3 fundamental steps:

A) Creating the nodes in the database.

B) Creating the connection edges.

C) Creating the linkage information has 2 steps:

C.1) Same source linkage.

C.2) Same destination linkage.

The main multi-threaded environment runs a single thread to read the input file then dispatch each entry to worker threads. This allows thread-safe fast sequential reads, and multi thread processing. The worker threads ensure the nodes are created, calculate which cell in the grid the edge belongs to, and place the edge in the cell. Finally, creates linkage information in the linkage grid.

3 5 Query Processing Model

The processing model is different depending on the types of queries. Therefore, we explain how queries are handled for each type of the load. The overview of our system architecture is shown in Figure 2.

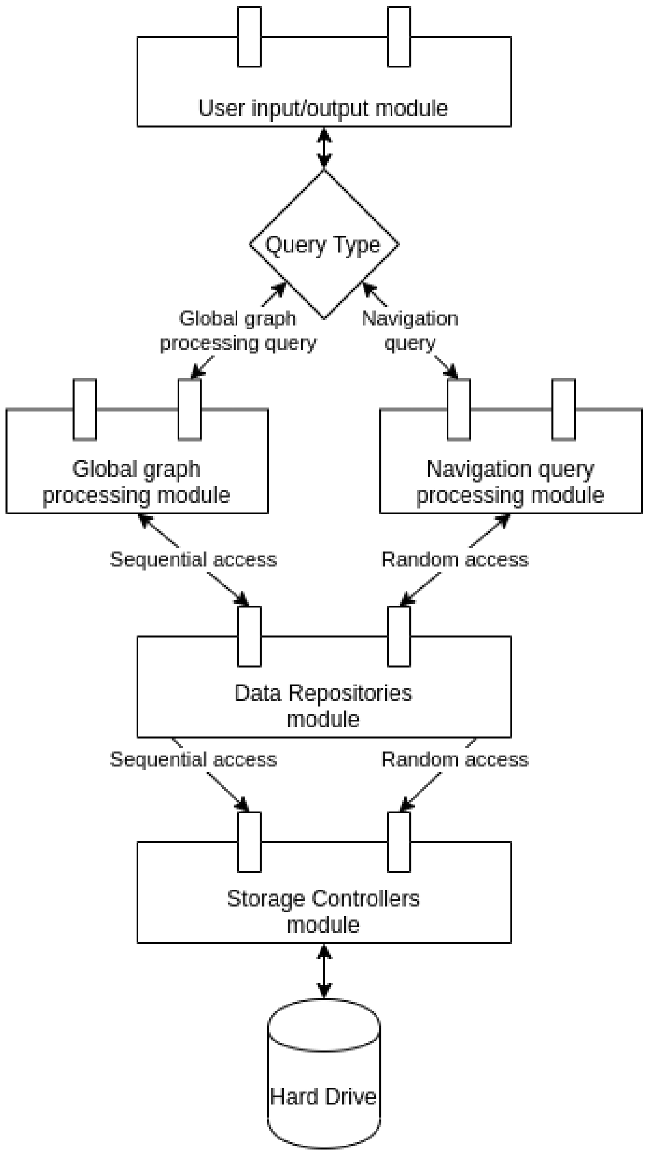

Figure 2 Query Processing Model

3 5. 1 Global Processing Model

When processing global graph algorithms, edge-centric stream apply computation model is used. The read cost of execution the algorithm is single sequential read over the edges data + single sequential write (if writing back to disk) + P^2 random reads of nodes data (can be avoided if memory is large enough to hold nodes data during processing).

The user needs to define 3 functions: 1)$shouldIterate()$ returning a Boolean that indicates if the algorithm needs to run for another iteration, the user is responsible for calculating and maintaining the state of the algorithm internally. 2)$processEdge(src, dst, acc)$ which carries processing over the edge connection node src to node dst, the accumulator acc can be used to accumulate values for the same destination node before calculating final value. 3)$updateNode(node, acc)$ updates the node final value -if needed- from the accumulated calculations over edges.

3 5. 2 Navigation Processing Model

The navigation processing is built with TBB parallel processing functionality. Each query is built with parallel functions, starting by streaming nodes and filtering them, this will reduce the number of edge cells that need to be streamed from hard storage for reducing the total I/O required for the query. A query language is currently not in consideration due to complexity and being out of scope of this paper.

4 Evaluation

To execute queries on the proposed database, a pre-processing step is required to import data from input format to our database format. The input datasets are in edge list format, each line of the input file represents a directed edge, and contains the IDs of the source and destination nodes. This pre-processing is required only once. After pre-processing is done, both navigation and global processing queries can be processed.

Evaluation was carried on commodity PC to show the viability of processing real world social graphs. The implementation is in C++ and uses LevelDB as storage engine. When our previous paper presented implementation based on customized threads and boost lock free structures, the performance was acceptable. In order to improve the efficiency of parallel processing, we decided to try other implementation using oneapi TBB library to manage parallel processing, along with improving caching and page pooling.

4 1 Datasets

- Twitter 2010 [15]: The Twitter2010 dataset is a publicly available network dataset of twitter followers.

It contains over 41.7 million users and over 1.5 billion relations. The dataset comes in edge list format with size over 26.2GB.

- LUBM-5000 [16]: Using the LUBM as a reference, a simplified version was coded to generate similar data in the edge list format. We generated 5000 universities with the final list size reaching 5.2GB.

4 2 System Environment

The experiments were conducted on a PC running 64-bit Ubuntu 18.04.3 LTS equipped with an Intel® Core™ i7-9700 CPU running at 3GHz with 12288 KB cache memory, 64GB of main memory and 2TB of an HDD running at

7200 RPM. Note that most of our experiments memory consumption maintained up to maximum of 6GB during each experiment. The main memory consumption is during preprocessing, specifically when generating edge linkage data.

4 3 Experiments

- **Pre-processing**: Carried over Twitter2010 dataset to evaluate the speed of pre-processing for large data.
- **PageRank**: Carried over Twitter2010 dataset after pre-processing and intilizing all weights and ranks to 1.
- **Navigation Queries**: Carried over LUBM-5000 data set because Twitter2010 has only one type of nodes (users) and one type of edges (follow relation). We evaluate against 3 types of queries:
  - Filtering based on node type.

$$n(UndergraduateStudent) \xrightarrow{e} n(Department)$$

  - Filtering based on Edge Type.

$$n_1 \xrightarrow{e(teacherOf)} n_2 \xleftarrow{e(takesCourse)} n_3$$

  - Filtering based on Node and Edge Types.

$$n(Professor) \xrightarrow{e(worksFor)} n(Department)$$

$$n(Department) \xleftarrow{e(memberOf)} n(UnderGradStudent)$$

4 4 Custom threads & Boost structures implementation

- **Pre-processing:** The previously submitted results, the pre-processing did not include edge linkage generation (pre-process until step B), it took 3665 sec to finish. We later complete the pre-processor and achieve a full process until step C.2 in 41978 sec.
- **PageRank** Initial implementation was able to finish a single iteration of PageRank algorithm in 770 sec, while the final implementation finished in 454 sec.
- **Navigation Queries** After implementing the linkage information, we carried 3 types of navigational queries on LUBM-5000 dataset:
  - Filtering based on node type: finished in 107 sec.
  - Filtering based on Edge Type: finished in 343 sec.
  - Filtering based on Node and Edge Types: finished in 113 sec.

4 5 Oneapi TBB implementation

- **Pre-processing** We carried the pre-processing until step B, then again until step C.2 in order to understand the time consumption on the workload, and to be able to compare to previous implementation. When stopping at step B (creating connection edges) the Twitter-2010 dataset was pre-processed in 2693 sec.
- **PageRank** We achieved a massive performance increase by using TBB parallelism along with caching intermediate results. Our implementation has finished a single iteration of PageRank in 140 sec.
- **Navigation Queries** TBB implementation is functionally similar for easier comparison and to emphasis the performance gained from the underlying storage rather than custom performance engine.
  - Filtering based on node type: finished in 94 sec.
  - Filtering based on Edge Type: finished in 88 sec.
  - Filtering based on Node and Edge Types: finished in 42 sec.

Table 1 Results comparison (in sec)

Implementation	Boost	TBB
Pre-processing: Step B	3665	2693
Pre-processing: Full	41978	-
PageRank	454	140
Filter on node	107	96
Filter on edge	343	88
Filter on node & edge	113	42

5 Discussion of results

The analysis of the results is built on comparing between the old and new implementation, some analysis was carried by profiling code under the same evaluation environment with a sample of 10% of the original dataset for twitter-2010.

5 1 PreProcessing

Comparing the old and new implementation, we see roughly 18% improvement up until stage 2. We can clearly see increased time consumption when generating the linkage information. The biggest factor is I/O boundaries, leveldb has to merge files frequently, specially in case of linkage information, because the entry size is larger and the access is quite random. More about leveldb behaviour analysis is discussed later. During our new implementations, we tried multiple ordering for the steps, and implemented them in various ways. The fastest implementation we had (which the results are based on), performed preprocessing in 3 cycles, the first cycle created the nodes, edges using oneapi TBB pipeline, while the second cycle generates same source linkage information, and the final cycle generates same destination linkage information. This order of steps was the faster because of maximum locality, and less blocking I/O from various sources.

5 2 PageRank

One major delay in processing PageRank algorithm was due to page caching mechanism and writing intermediate results to a hard drive. Our new implementation addresses these problems: 1) page caching was improved with a faster implementation, and 2) intermediate results were cached in memory during processing. We noticed even for the large graph like Twitter2010 that the memory constrains still allow to cache all intermediate results.

5 3 Navigation Queries

The functional similarity in processing allowed us to get insights on the performance gain from the underlying storage and caching mechanisms. Without proper navigation query engine, writing complex efficient queries can be challenging for the users. Therefore, we plan to address this issue by using TBB flow graph to represent navigational queries as flow graphs with functional nodes.

5 4 Page Pooling

Since disk pages are considerably large objects that will be created and delete frequently at run-time, the allocation and deallocation of memory consumes a considerable amount of time due to the fact that *new* and *delete* operations often require a system call, which causes a context switch from user mode to kernel mode, as well as memory management and paging done by the kernel. Through a simple page pooling and user-mode memory management mechanism, the time can be saved by moving deleted disk pages to a pool and partially resetting their internal state by invalidating the entries in the disk page (the first n bytes of the page only); so they can be used later without the need to reallocate the memory space over and over.

5 5 Leveldb file merging

While the use of LevelDB allowed a faster development and easier implementation, it takes considerably long amount of time and I/O when merging files on the hard drive, during that time the database can not carry any write operations which halts most of the processing. Moreover, the larger the database grows the longer time merging process consumes, this can be observed well in pre-processing, specifically in linkage generation where writing size is large and random. Additionally, LevelDB read/write operations consume an extra buffer space, which can be viewed as additional memory consumption and processing time when copying those buffers to application code, while this is a good practice, it falls back a bit on performance, specially when those operations are very frequent. Therefore, we are considering to develop our own customized storage engine with less overhead to allow faster processing.

6 Conclusion

Through iterative cycles of small improvements in preprocessing, caching, memory management, threading, and various code areas, we started to see improvements that gets us more acceptable performance on all workloads. The collection of all improvements has reduced the time by: Preprocessing 18%, PageRank 70%, and Navigation queries $10 \sim 75\%$. In the initial implementation, we validated the soundness of our solution. However, the performance was falling behind due to poor implementation techniques. After solving the processing bottlenecks, we start addressing I/O bottleneck in caching implementation. We further start to feel the limitation using LevelDB as a storage engine, driving us to start planning our own implementation.

6 1 Future Work

Implementing navigation query engine using TBB flow graph, a user defined graph can be used to process navigation query. Optimizing processing engine and storage engine are our top priorities. Taking into consideration the increasing write time as data grows, the storage engine will become a further bottleneck in the process. Therefore we would like to address that issue in our next development cycle. Later, we would optimize the processing engines to take full advantages of the storage engine to increase throughput and reduce I/O waiting time. Finally, until this moment, because we were running the database with a single query at a time, I/O scheduling was required. However, in real life scenario, multiple queries of different types can be running simultaneously. We would like to add time and priority aware scheduling system.

Acknowledgement

This work was partly supported by JSPS KAKENHI Grant Numbers 18H03242, 18H03342, 19H01138A.

References

[1] Tarek Aoukar, Jun Miyazaki. Graph Navigation Query in an Edge Streaming Graph Database, The 12th Forum on Data Engineering and Information Management (DEIM 2020), Proc. of DEIM 2020, Mar. 2020. H7-1

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). Association for Computing Machinery, New York, NY, USA,135– 146.DOI:https://doi.org/10.1145/1807167.1807184

[3] Joseph E. Gonzalez and Yucheng Low and Haijie Gu and Danny Bickson and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12) USENIX. 17–30.

[4] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: scale-out graph processing from secondary storage. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). Association for Computing Machinery, New York, NY, USA, 410–424. DOI:https://doi.org/10.1145/2815400.2815408

[5] Aapo Kyrola and Guy Blelloch and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC.Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12) USENIX. 31–46.

[6] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In Proceedings of the Conference on Operating Systems Design and Implementation (2014), USENIX Association, pp. 599–613.

[7] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In Proceedings of the Usenix Annual Technical Conference (2015), USENIX Association, pp. 375–386.

[8] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In Proceedings of the European Conference on Computer Systems (2015), ACM, pp. 1:1–1:15.

[9] H. Huang and Z. Dong, "Research on architecture and query performance based on distributed graph database neo4j, " in Consumer Electronics, Communications and Networks (CECNet), 2013 3rd International Conference on, Nov 2013, pp. 533–536.

[10] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In Proceedings of the Twenty- 6 Fourth ACM Symposium on Operating Systems Principles (SOSP ' 13). Association for Computing Machinery, New York, NY, USA, 472–488. DOI:https://doi.org/10.1145/2517349.2522740

[11] Zhu, X., Chen, W., Zheng, W., & Ma, X. (2016). Gemini: A computation-centric distributed graph processing system. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (pp. 301-316).

[12] Maass, S., Min, C., Kashyap, S., Kang, W., Kumar, M., & Kim, T. (2017, April). Mosaic: Processing a trillion-edge graph on a single machine. In Proceedings of the Twelfth European Conference on Computer Systems (pp. 527- 543). ACM.

[13] S. Ghemawat et al. LevelDB. https://code.google.com/p/leveldb/.

[14] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. The VLDB Journal 11, 3 (November 2002), 198–215. DOI:https://doi.org/10.1007/s00778-002-0074-9

[15] Kwak, H., Lee, C., Park, H., & Moon, S. (2010, April). What is Twitter, a social network or a news media?. In Proceedings of the 19th international conference on World wide web (pp. 591-600). AcM.

[16] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowl-edge base systems.Journal of Web Semantics, 3(2-3):158–182, 2005.

[17] Manish R Jain et al. DGraph https://github.com/dgraph-io/dgraph

[18] Deutsch, A., Xu, Y., Wu, M. and Lee, V., 2019. TigerGraph: A Native MPP Graph Database. arXiv preprint arXiv:1901.08248.