On the Use of Property Graph for Rich Metadata Management in HPC Systems

immediate

***Abstract*—HPC platforms are capable of generating huge amounts of metadata about different entities including jobs, users, and files etc. *Simple metadata*, which describe the attributes of these entities has already been well recorded and used in current systems, like the file size, name and permission mode. However, only a limited amount of *rich metadata*, which record not only the attributes of entities, but also relationships between them, are captured in current HPC systems. The main challenge is that the rich metadata can include huge amounts of data from many sources, including users and applications, and generally must be combined to present a correct view for later query and processing. So, collecting, integrating, processing, and querying such rich metadata place a huge pressure on HPC systems. In this paper, we propose a rich metadata management approach that unifies metadata into one generic and flexible property graph. We argue that this approach both support simple metadata operations, including directory traversal, permission validation, and, more importantly, support rich metadata processing, like provenance storage and query. The benefits of this approach come from the unified way of managing all metadata and also from the rapid evolving graph storage and processing techniques.**

I. Introduction

Metadata, especially rich metadata, contains detailed information about different entities and their relationships. These entities could be users, jobs, processes, data files, or even user-defined entities. Storing and utilizing metadata already provides the basic data management functionalities in existing storage systems, including finding files, controlling file access, and tracing file creation and access time. We categorize this metadata as *simple metadata* since they only contain the predefined attributes about individual entities and very basic relationships (e.g. ownership, POSIX namespace). On the contrary, *rich metadata* include more than individual predefined attributes; they may store user-defined arbitrary attributes of entities and even their relationships. A typical example of rich metadata would be provenance [1] (e.g. lineage).

Provenance is well understood in the context of art or digital libraries, where it respectively refers to the documented history of an art object, or the documentation of processes in a digital object's life cycle [2]. In the computational systems, it indicates a recording of complete history of each data element, including the processes that generated it, the user that started the processes, and even the environment variables, parameters, and configuration files while executing. A complete provenance picture supports a huge amount of data management abilities [3, 4]. For example, the accessing history of users reading/writing data files can help us develop an audit tool to monitor and administrate users in shared supercomputer facilities; the detailed read/write history from processes to data pieces provides a possibility to trace back suspicious executions that generated or were based on wrong datasets; reproducibility also may be possible because we have the complete history of an execution and have a better chance to re-generate the same environment to run it again.

While there are numerous advantages to capture rich metadata like provenance, current HPC platforms still lack basic facilities to collect, store, process, and query rich metadata. The challenge comes from at least three places.

- *Storage System Pressure*. Considering a leadership supercomputer, there might be millions of processes running on millions of cores accessing billions of files per second. In this case, recording the rich metadata, like the detailed access history of each process, will place pressure on the storage system. In addition, as storing rich metadata must not affect the application execution speed significantly, the resources (both network and disks bandwidth) dedicated to storing metadata must be limited in most cases.
- *Efficient Processing*. Even if we can collect and store these rich metadata, it is still a big challenge to process and query them. First, as rich metadata are large and can not be held in one server, a distributed processing framework is necessary in most cases. Second, some use cases require fast searching and reading, however, some require complex queries rather than simple searching, so efficient and flexible processing should be provided for them.
- *Metadata Integration*. As we have described, the rich metadata could be as diverse as the users need. They can contain predefined attributes and relationships of entities, or be extended to any user-defined attributes and relationships. Traditionally, we need different tools to process them. However, this strategy leads to waste of resources as they are not able to reuse the same metadata and processing infrastructure for different use cases.

In this paper, we proposed unifying all metadata into one property graph that integrates rich metadata from different sources together: applications can store their rich metadata using graph storage APIs, and access different categories of metadata using graph query APIs [5]. The benefits are twofold: first, it directly solves the integration issue by using a single representation. All applications will have the same interface to store or process metadata in a single service, where we can apply complex optimizations to improve the performance further. Second, by abstracting metadata into a graph, we are able to utilize rapidly evolving graph techniques to provide

better access speed, flexible query languages, and also a high-performance graph-based distributed framework.

This paper is organized as follows. We first introduce the definition of proposed graph model for rich metadata in Section II. In Section III, we explore the basic attributes of such graph by building an example graph using the metadata collected from the Darshan trace of a leadership supercomputer (Intrepid). In Section IV, we show the strategy to implement several critical data management functionalities based on the graph model. In Section V, we briefly introduce the relevant techniques from the graph storage and processing community, and discuss the challenges on current graph infrastructure. The last section (Section VI) concludes this paper and proposes the future work.

II. Graph-based Metadata Model

In fact, we already consider metadata as a graph. The traditional directory-based file management constructs a tree structure to manage files with additional metadata stored in *inodes* at leaves in the tree [6]. This tree is a graph. The provenance standard (*Open Provenance Model* [2]) considers the provenance of objects is represented by an annotated causality graph, which is a directed acyclic graph enriched with annotations capturing further information.

We generalize these graphs in HPC scenarios and propose the metadata graph model. The metadata graph is derived from the *property graph model* [8], which includes vertice that represent entities in the system, edges that show their relationships, and properties that annotate both vertice and edges and can store arbitrary information users want. Based on the entities in HPC environment, we introduce the strategy to map the possibly arbitrary rich metadata into this property graph model.

A. Entity To Vertex

In an HPC platform, there are three basic entities: users, applications, and data files. Moreover, users also can define other logical entities, like *user groups* or *work-flow* as they need. So, in our strategy, we define three basic entities and allow users to extend them to build more entities.

- *Data Object*: It represents the smallest data unit in storage systems. Each file in PFS (Parallel File System) indicates one data object. Moreover, the directory is also a data object, which contains multiple files.
- *Executions*: They represents the execution of applications. There are three levels of executions: *Job* submitted by the user; *Processes* scheduled from one job; and *Threads* running inside one process. Different use cases require certain level of execution details, and generate graphs with different size. For simplicity, we name all these entities as *Execution* entity in later discussion.
- *User*: It simply means the real users of the cluster.

In addition to these basic entities, users usually define their own entities. The user-defined entities must connect with existing entities to keep every element in the graph accessible by traveling through the graph.

B. Relation To Edge

There are several basic relationships between the basic entities in Table I. Each cell shows the basic relationships from the row identifier to the column identifier. Each relationship denotes a directed edge in the metadata graph. For example, *run* indicates that the user starts an execution; *exe* means one execution is based on the data objects as the executable files; *read/write* indicates the I/O operations from executions to data objects. For all those relationships, we also define the reversed ones to accelerate the reversed traversal.

TABLE I
Default Relationships Definition.

	User	**Execution**	**Data Object**
User		*run*	
Execution	*wasRunBy*	*belongs, contains*	*exe, read, write*
Data Object		*exedBy, wasReadBy, wasWrittenBy*	*belongs, contains*

There are several *belongs/contains* relationships. In the Execution entity case, it means one job contains multiple processes, which in turns belong to this job. In the Data Objects case, it can show that one directory may contain multiple files or directories. Users can create their own relationships from two existing entities. For example, two users can have a new relationships called *login-together* if they login the system roughly at the same time.

C. Property

Rich metadata also contain annotations on entities and their relationships. In graph model, we store them as properties, which are key-value pairs attached on vertices and edges. Users can create their own properties on existed vertices and edges except their keys need to be unique in each user's namespace. By isolating properties by users, we avoid global contention among different users. The properties could be very flexible and diverse. For example, there are properties like user name, privilege, execution parameters, file permission, and file creation and access time etc.

III. Metadata Graph Prototype

Collecting rich metadata usually requires modifications on HPC runtime as many rich metadata were generated from the runtime systems, like the jobs, processes, and read/write operations [9, 10, 11]. To help understand the attributes of a rich metadata graph in an HPC context, we exploit Darshan trace logs as a source of rich metadata in current prototyping [12].

A. Mapping Strategy

Darshan utility is a MPI library that can be linked to users' applications and generates I/O behaviors logs during executing [13]. Each Darshan log file represents a distinct job. The log entries of a job contain the user id who started this job, the executable file that the job was based on, the parameters of this execution, some environmental variables,

and most importantly, the file access statistics of each process (ranks) inside this job (MPI program). Note that, the collected Darshan traces have been anonymized, only storing the hashed values of file names, path, user names, and job names [14].

We map Darshan logs to the metadata graph defined in Section II. Basically, each unique user id indicates a User entity, each Darshan log file represents a Job, all the ranks inside a job correspond to the Processes, and both the executables and data files are abstracted as different Data Object entities. Currently, Darshan does not capture directory structure as it only stores the hashed value of file paths, so we synthetically create the simplest directory structures: data files visited by each execution are considered under the same directory, and all these directories accessed by one user are placed under one directory for each user. Based on this mapping, we were able to import a whole year' Darshan trace (*2013*) on Intrepid machine into an example graph [14].

In fact, current mapping of Darshan logs emitted many common metadata due to the limitation of the data sources. For example, the logs do not have the metadata about the users; do not contain the directory structure or file permissions; and each job is based on a single executable file without configuration files and parameters. However, the generated graph still show many interesting properties of such metadata graph and offer a great potential in data management.

B. Graph Size

The first property of metadata graph is the their potential size. The graph size described here is in terms of the number of vertice, edges, and properties[1]. The real storage size is based on those numbers but may vary under different data structures and storage layouts.

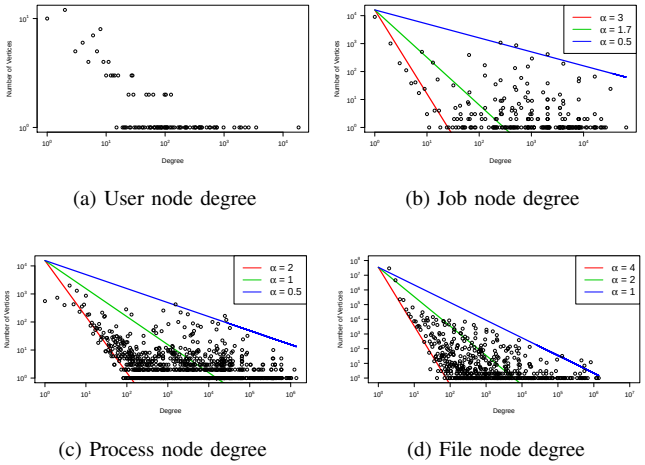
TABLE II
DARSHAN GRAPH SIZE AND SOME COMPARISONS.

Number	Basic	With I/O Ranks	With Full Ranks	With Directory
Vertice	34,656 K	41,729 K	147,886 K	147,934 K
Edges	126,488 K	133,561 K	239,766 K	366,253 K

	Road Graph USA [15]	Twitter	Orkut [16]	Web Page Graph
Vertice	24 M	645 M [17]	3 M	2.1 B [18]
Edges	29 M	81,364 M [19]	117 M	15 B [18]

The top half of Table II shows the graph size of example metadata graph for different levels of detail. The first column considers the job as Execution entity and eliminates the all the processes (ranks) inside this job. All the I/O behaviors from different processes inside a job are considered as from the Job entity. The second column (*With I/O Ranks*) records part of the ranks which have I/O operations. In many cases, this indicates the rank 0 process or the aggregators in two-phase I/O. The third column (*With Full Ranks*) records all the ranks as Processes entities no matter they performed I/O or not. The last column shows the graph size with the synthetic directory structures and full ranks. This table clearly shows that increasing the level of detailed during collecting metadata will dramatically increase the graph size. So, administrators should wisely choose their metadata according to the usage. And, the user-defined relationships should be created with caution to avoid huge increasing in graph size too.

[1]K = thousand; M = million; B = billion; T = trillion



(a) User node degree (b) Job node degree

(c) Process node degree (d) File node degree

Fig. 1. Node degree for different entities.

In the button half of Table II, we show several typical large-scale graphs from different fields, including the social network (e.g. Twitter, Orkut), the road map graph (e.g. USA map), and the Internet web pages. By comparing with them, we can notice that, although the metadata graph is large and could easily be much larger, this kind of size has already been managed well in many existing systems.

C. Graph Structure

In addition to the graph size, the graph structure also matters in future storage and processing. Before discussing the graph structure, we first list a brief view of different entities of the example metadata graph in Table III.

TABLE III
STATISTICS OF METADATA GRAPH.

	User	Job	Proc.	Rank	File
Num	117	47,592	10,085,931	113,278,038	34,608,033

This table shows different entities in metadata graph have totally different size, so in this section, we will consider them separately. Figure 1 shows the node degree distributions of those four different entities. The x-axis denotes the degree, the y-axis shows the number of vertices which have that number of degree. Both x-axis and y-axis are 'log' values.

In nature, many graphs obey the *skewed* power-law degree distribution [20]. It means most vertices have relatively few neighbors while a few vertices have many neighbors. We can use function $\mathbf{P}(d) \propto d^{-\alpha}$ to describe the probability that a vertex has degree d in such graphs. Here, α is a positive constant that control the "skewness" of the degree distribution:

higher α indicates lower density, which means vast majority of vertices are low degree. Lower α shows higher density and more high degree vertices.

To compare the actual distribution with the power-law attributes, we plot three lines of the power-law distribution with different α values. Intuitively, we can notice that the *File* nodes fit the power-law degree distribution best as Fig. 1(d) shows. The degree of a file node indicates the number of processes that ever read/write it. So, the distribution indicates that most files are seldom accessed, and a very small part of files are visited highly frequently. Second, as the larger α value ($2 \rightarrow 4$) actually fits the distribution better, which indicates the *File* nodes have lower density, saying that the majority of files have low degrees.

Other three distributions of *Process*, *Job* and *User*, are more and more unlike the power-law distribution. For the *Process* graph (Fig. 1(c)) , there are not enough processes with low degree. This means most processes in HPC systems tend to visit at least several files. For *Job* graph (Fig. 1(b)), the distribution scatters randomly between the $\alpha = 3$ line and $\alpha = 0.5$ line. As there are only 117 users, we do not consider they fit any distribution. But from Fig 1(a), we still can see most users only issue several jobs, a very small number of users will issue the most jobs.

These graph structures direct the way of graph storage and processing including graph partition and storage layout. Moreover, they also lay as a basis to create synthetic graphs for evaluation test as collecting enough rich metadata in running HPC systems are still hard.

IV. Use Cases On Using Metadata Graph

Unifying rich metadata into one graph turns many appealing data management functionalities into graph traversal operations or graph queries. In this section, we will show how to map the use cases from real-world scenarios to graph operations.

A. User Audit

Data auditing is critical in large computing facilities where different users share the same cluster. It at least requires the detailed view of users file access for future security check. In metadata graph, we already collect the *run* relationships between Users and Executions, and the *read/write* relationships between Executions and Data Objects. And, all those relationships contain properties like timestamps. In such graph, the need to find all the files that were read by a specific user during given time frame [t_s, t_e] will become graph operations like this: 1) query the metadata graph from the given user; 2) travel through *run* edges to Execution nodes; 3) filter executions based on the given time frame; 4) and travel through the *read* edges to the final files. Similarly, if we want to get all the users who used to read to a sensitive file, we can do the similar graph query from the give file node.

B. Hierarchical Data Traversal

Hierarchical data organization is used to present a logical layout of data sets to users. The simplest example of hierarchical data traversal is traditional directory namespace traveling. In metadata graph model, we already abstract both the directories and files as Data Object entities. The *belongs* and *contains* relationships between different Data Objects represent the relationships between files and directories. So, given an absolute path, locating the file becomes going through a bunch of *contains* edges from a Data Object node. Each time, we filter the edges according to the given names. Moreover, the access control metadata attached in users, files, and directories also could be verified while traversing.

An appealing advantage of using graph to store the directory structure of file system is the scalability. Traditional POSIX directory structure limits the number of files inside one directory. So, HPC system that may have millions of files under one directory, needs to deploy specific system like Giga+ to distribute the metadata into multiple servers for better performance [21]. However, for proposed graph model, this turns into a graph partition problem, which we have been well studied [22, 23, 24].

In addition to traditional POSIX-style files and directories, semantic data management would be another hierarchical traversal use case. Scientists usually need to manage their data in a semantic way, like arranging all of the inputs and outputs of a single simulation execution together. Traditionally, this needs careful file naming and directories placement. But in metadata graph, we can simply create new entity named Simulation and connect it with the Data Object with *input/output* relationships. This will intelligently help users organize data in multiple dimensions.

C. Provenance Support

Provenance has a wide range of use cases including data reproducibility, work-flow management etc. As a superset of provenance, the metadata graph model is able to support these usages. In this subsection, we borrow the problem from the first Provenance Challenge as an example [25].

In this challenge, a simple example work-flow was provided as the basis, a workable provenance system should be able to represent the work-flow and all the relevant provenance for the example work-flow, and, most importantly, be able to answer predefined queries. Based on proposed graph model, we can easily abstract the work-flow as serial of executions run by the same user. Each execution reads several Data Objects and generates outputs for applications in next phase. Based on this work-flow, the provenance system needs to answer queries like: *find the execution whose model is AlignWarp and inputs have annotation* ['center':'UChicago']. This can be expressed easily in the metadata graph: 1) query all the Execution vertices that have *exe* out-edge pointing to a Data Object named 'AlignWarp'; 2) start from all those Execution vertices and get those executions whose property 'center' equal to 'UChicago'.

A notable advantage of the metadata graph comparing with pure provenance system is that we can cross-reference different category of metadata in an unified way. If the provenance query needs the help of other metadata, like the

file size, permission mode, or user group information etc., processing them in a unified graph will be more efficient and straightforwards.

V. Graph Facilities & Challenges

Graph databases and distributed processing frameworks are two basic facilities to support our metadata graph model. Although there are already a large number of them, the challenges are still existing due to the specific requirements of storing, processing, and querying the large-scale metadata graphs.

A. Graph Databases

The graph databases are designed to cover the requirements of complex graph-based relationships, which embarrass the traditional relational databases. In the last couple of years, there have been an increasing number of graph databases implementation, including AllegroGraph, DEX, G-Store, HyperGraphDB, InfinitGraphDB, Neo4j, and Titan etc [26, 27, 28, 29, 30, 31, 32]. They can be categorized based on different metrics. Based on storage device, there are in-memory databases and disk-based databases. Based on the supported graph data structure, there are simple graphs databases, hypergraphs databases, and property graphs databases[2]. Based on distributed deployment, there are single server databases, high availability databases, and distributed databases.[3]

Among those graph databases, metadata graphs first requires a disk-based solution as they are too large to fit into memory; second, the supported graph structures should be property graph model. Also, the distribution supports will also be necessary since the graph size may overflow the disks of a single sever. There are several implementations that satisfy such requirements, like Titan, DEX, Neo4j etc. But, the challenge is the performance. Updating edges across different servers will significantly reduce the performance, so the graph databases should consider the structure of different entities and provide intelligent storage layout. Another key performance challenge is graph traversal. In fact, traveling through a property graph like our metadata graph will be even more complex. The main reason is that, during traveling, we usually need to apply filter or computations on some properties like the provenance example shows. As these properties are too big to be fully cached in memory and loading them from persistent devices will introduce too many random seeking, we will need a intelligent cache strategy in our design too.

B. Graph Processing

In addition to the graph databases, there are also graph processing frameworks which can be used to perform computation or queries on graphs in a distributed way. Typical examples of these frameworks include Gigraph [34], which was designed and implemented based on Pregel computing model [35]; GraphX [36], which was based on Spark computing framework [37]; GraphLab [38], and X-Stream [39] etc. Those processing frameworks are a complement of the querying and searching from graph databases. For example, we can run *community discovery* algorithms on metadata graph to find the 'closely' data files. The results can be used to optimize physical placements for better I/O performance. These algorithms usually get the whole graph involved in the iterative computation and last a long time.

However, most these distributed graph processing frameworks work on the unstructured graphs which are usually simply stored as a plain file in adjacency list formats in a general storage back-end. So, there is a big gap from deploying graph algorithms on these plain graph formats to running these algorithms on a graph database, which has an optimized storage layout for querying and searching.

[2]Here, the simple graph indicates graph defined as a set of nodes connected by weighted edges. Hypergraphs extends the simple graphs by allowing an edge to relate an arbitrary number of nodes [33]. Property graph indicates graph where nodes and edges contain properties. This property graph is the very basis of our proposed metadata graph model.

[3]Here, the difference between high availability (HA) and distribution is HA only provides distributed reads on identical copies of the same dataset.

VI. Conclusion & Future Work

In this paper, we proposed an idea of unifying rich metadata in a HPC platform into a property graph model, which supports a wide range of metadata management requirements in a simple and efficient way. By prototyping such a metadata graph from Darshan I/O traces of a real world leading supercomputer, we explorer the attributes of such graphs and compare it with some popular big graphs. After that, we introduce the existed graph facilities and discuss their feasibilities and limitations in our scenario. In general, we argue the benefits of unifying HPC metadata into a graph and also present the feasibility of implementing such a graph in current HPC platform. In future, the work will be implementing such a platform with optimized or tweaked graph facilities and provide a practical metadata solution for Exscale data management challenge.

References

[1] "Provenance," in *http://en.wikipedia.org/wiki/Provenance*.

[2] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers *et al.*, "The Open Provenance Model Core Specification (v1. 1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011.

[3] Y. L. Simmhan, B. Plale, and D. Gannon, "A Survey of Data Provenance in e-science," *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, 2005.

[4] C. T. Silva, J. Freire, and S. P. Callahan, "Provenance for Visualizations: Reproducibility and Beyond," *Computing in Science & Engineering*, vol. 9, no. 5, pp. 82–89, 2007.

[5] S. Jouili and V. Vansteenberghe, "An Empirical Comparison of Graph Databases," in *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 2013, pp. 708–715.

[6] A. S. Tanenbaum and A. Tannenbaum, *Modern Operating Systems*. Prentice hall Englewood Cliffs, 1992, vol. 2.

[7] D. M. Ritchie, "UNIX Time-sharing System: A Retrospective," *Bell System Technical Journal, The*, vol. 57, no. 6, pp. 1947–1969, 1978.

[8] "Property Graph," http://www.w3.org/community/propertygraphs/.

[9] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-Aware Storage Systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 43–56.

[10] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in Provenance Systems," in *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009.

[11] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in Automatic Provenance Collection," in *Provenance and annotation of data*. Springer, 2006, pp. 171–183.

[12] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of Petascale I/O Workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[14] "FTP site: Darshan data." ftp://ftp.mcs.anl.gov/pub/darshan/data/.

[15] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. American Mathematical Soc., 2009, vol. 74.

[16] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities based on Ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 2012, p. 3.

[17] "Twitter Statistics," http://www.statisticbrain.com/twitter-statistics/.

[18] J.-L. Guillaume, M. Latapy *et al.*, "The Web Graph: an Overview," in *Actes d'ALGOTEL'02 (Quatrièmes Rencontres Francophones sur les aspects Algorithmiques des Télécommunications)*, 2002.

[19] "Twitter Edge Statistics," http://www.theguardian.com/technology/blog/2009/jun/29/twitter-users-average-api-traffic.

[20] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet Topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[21] S. Patil and G. A. Gibson, "Scale and Concurrency of GIGA+: File System Directories with Millions of Files." in *FAST*, vol. 11, 2011, pp. 13–13.

[22] M. Kim and K. S. Candan, "SBV-Cut: Vertex-cut based Graph Partitioning Using Structural Balance Vertices," *Data & Knowledge Engineering*, vol. 72, pp. 285–303, 2012.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[24] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-Law Graphs," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.

[25] "Provenance Challenges," http://twiki.ipaw.info/bin/view/Challenge/.

[26] "AllegroGraph," http://franz.com/agraph/allegrograph/.

[27] "DEX," http://www.sparsity-technologies.com/.

[28] R. Steinhaus, D. Olteanu, and T. Furche, "G-Store: A Storage Manager for Graph Data," Ph.D. dissertation, Citeseer, 2010.

[29] B. Iordanov, "HyperGraphDB: a Generalized Graph Database," in *Web-Age Information Management*. Springer, 2010, pp. 25–36.

[30] "InfiniteGraph," http://www.objectivity.com/infinitegraph.

[31] J. Webber, "A Programmatic Introduction to Neo4j," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012, pp. 217–218.

[32] "Titan," http://thinkaurelius.github.io/titan/.

[33] C. Berge and E. Minieka, *Graphs and Hypergraphs*. North-Holland publishing company Amsterdam, 1973, vol. 7.

[34] "Gigraph," http://giraph.apache.org/.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[36] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[38] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A New Framework for Parallel Machine Learning," *arXiv preprint arXiv:1006.4990*, 2010.

[39] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-Centric Graph Processing using Streaming Partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.