Managing Rich Metadata in High-Performance Computing Systems Using a Graph Model

Dong Dai, Yong Chen, Philip Carns, John Jenkins, Wei Zhang, and Robert Ross

Abstract—High-performance computing (HPC) systems generate huge amounts of metadata about different entities such as jobs, users, and files. Existing systems can efficiently record and manage part of these metadata, mainly the POSIX metadata of data files (e.g., file size, name, and permissions mode). But another important set of metadata, referred to as "rich" metadata in this study, which record not only wider range of entities (e.g., running processes and jobs) but also more complex relationships between them, are mostly missing in current HPC systems. Yet such rich metadata are critical for supporting many advanced data management functions such as identifying data sources and parameters behind a given result; auditing data usage; or understanding details about how inputs are transformed into outputs. To uniformly and efficiently manage the rich metadata generated in HPC systems, We propose to utilize a graph model in this study. We identify the key challenges of implementing such a graph-based HPC rich metadata management system and present GraphMeta, a graph-based rich metadata management system designed and optimized for HPC platforms, to tackle these challenges. Extensive evaluations on both synthetic and real HPC metadata workloads show its advantages in both performance and scalability compared with existing solutions.

Index Terms—Data models, metadata, high performance computing, graph partitioning

1 Introduction

High-performance computing (HPC) systems generate huge amount of metadata about various entities in the system every second. These metadata include both the traditional *POSIX metadata* which describes the predefined attributes about individual entities such as files and users and the so-called *rich metadata* which describes detailed runtime information about boarder categories of entities and their complex relationships such as running jobs or workflows. A well-known case of rich metadata is data provenance, which describes the relationships among entities such as data files, running jobs, execution context, and their dependencies that contribute to the existence of a data item [1], [2].

With rich metadata, we can effectively enable a variety of data management functionality in HPC environment, such as data auditing, result validation, and reproducible support [3], [4]. For example, the file access history of users can be used to audit users' activities in shared super-computer facilities; the file access history of processes can be used to depict an execution and its configurations; and captured detailed execution history and context can be used to regenerate an environment for reproducing scientific results.

- D. Dai is with the Department of Computer Science, College of Computing and Informatics, University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: dong.dai@uncc.edu.
- Y. Chen and W. Zhang are with the Department of Computer Science, Texas Tech University, Lubbock, TX 79413. E-mail: {yong.chen, x-spirit.zhang}@ttu.edu.
- P. Carns, J. Jenkins, and R. Ross are with the Argonne National Laboratory, Lemont, IL 60439. E-mail: {carns, jenkins, rross}@mcs.anl.gov.

Manuscript received 29 Nov. 2017; revised 27 Oct. 2018; accepted 30 Nov. 2018. Date of publication 18 Dec. 2018; date of current version 12 June 2019. (Corresponding author: Dong Dai.)
Recommended for acceptance by G. Tan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2887380

While rich metadata can support many attractive data management functionality, the support for rich metadata is still very limited in existing HPC systems. Many of recent studies about HPC data management enhancement, such as Spyglass [5] and Magellan [6] are still largely limited to POSIX metadata. The major factor that limits rich metadata management is the lack of necessary facilities to model, store, process, and query the complex rich metadata efficiently in HPC platforms. we summarize three key challenges of managing rich metadata in HPC platforms.

First, rich metadata are heterogeneous. They can contain predefined attributes and relationships such as POSIX metadata of files, as well as user-defined entities, attributes, and relationships, such as the running execution that creates, reads, and writes the data files. This heterogeneity makes modeling rich metadata complicated. Moreover, the rich metadata are typically from different components of HPC systems such as file systems, runtime, and job schedulers. This diversity makes rich metadata management a global service for the entire platform across different system components. These distinct data sources and data formats should be integrated uniformly in order to avoid duplication of functionality across management tools.

Second, the volume of rich metadata is large. In addition to the POSIX metadata, rich metadata contain dynamic runtime information about data files, jobs, users, and environmental variables. A leadership super-computer might include millions of processes and computation cores operating billions of files. They all generate dynamic runtime metadata continuously. This scale can lead to a huge volume of rich metadata generated in a short time, and the metadata must be ingested and managed gracefully. Given the already high pressure on parallel file systems for just maintaining the POSIX metadata, storing the rich metadata is easily a big challenge on the storage engine.

Third, users need highly efficient methods to query the rich metadata in order to accomplish various data management tasks. These queries lead to access patterns such as locating a single entity and relationship (e.g., load attributes of a file or user); iterating over relationships (e.g., find out all users that have accessed a file); and providing conditional traversal across multiple entities (e.g., locate the initial input data sets for an important result following the read/write relationships between executions and data files). These data accesses need to be efficiently supported, given the large volume of rich metadata and the high concurrency of HPC systems.

To tackle these challenges, we have conducted series of studies [7], [8], [9], [10]. In this paper, we summarize our key findings from these studies, and more importantly, extend them with set of new designs and evaluation results. To summarize, in this study, we proposed a graph-based rich metadata management system, called GraphMeta, for HPC platforms. The core idea of GraphMeta is to unify all rich metadata into an heterogeneous property graph, and develop corresponding supportive infrastructure for managing the graph to meet the performance and scale requirements of rich metadata management on HPC platforms. Comparing with previous work, this study has following contributions:

- We extended the property graph-based model with *versioning* to uniformly manage rich metadata. We also discuss the consistency issues relevant with multiple versions;
- We introduced a new *pre-fetching* mechanism in synchronous graph traversal to achieve better performance. We evaluated its efficiency and compared it with previously proposed asynchronous graph traversal.

GraphMeta focuses on supporting advanced data management tasks that utilize rich metadata and is designed to supplement, not to substitute for, the POSIX metadata service of existing parallel file systems. To the best of our knowledge, GraphMeta is the first advanced graph-based rich metadata management solution designed specifically for HPC systems and for supporting sophisticated data management tasks such as data auditing, results validating, and data reproducing.

The rest of this paper is organized as follows. Section 2 introduces the graph-based rich metadata model. Section 3 discusses the challenges of using graph model to manage rich metadata. Section 4 describes the GraphMeta design, covering the data model and APIs. Section 5 describes the implementation details of GraphMeta, including both the storage engine and the new DIDO graph partitioning algorithm. Section 6 reports the evaluation results including both micro-benchmark and end-to-end experiments. Section 7 discusses related work, and Section 8 presents our conclusions and discusses avenues for future work.

2 Graph-Based Rich Metadata Model

Arguably, researchers 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* [11]. This tree is a graph. Following a similar idea, we propose to map rich metadata in HPC systems into an heterogeneous property graph [12], which allows vertices and edges to have arbitrary sets of associated properties, supporting a flexible data schema. We select the graph-based model for two reasons. First, the flexibility of the graph model allows us to model and integrate the heterogeneous rich metadata from different sources efficiently. Thus, different users and applications can manage rich metadata through the same graph-based APIs interacting with a single rich metadata service. This unified model also means that any optimization applied to the rich metadata service can benefit all users and applications. Second, the graph model provides a handy method for processing and reasoning about metadata. Complex queries can be easily expressed as graph traversals and efficiently executed.

To apply property graph model in HPC metadata management, we propose to use vertices to represent entities in HPC system; edges to show relations between entities; properties to annotate both vertices and edges with arbitrary information that users need; and the concept of versioning to track the changes of entities, as detailed in the following sections.

2.1 Entity to Vertex

An HPC platform comprises three basic entities: data files, users, and the running applications. We define these as three basic types of vertices, as follows.

- *Data Object* represents the basic data unit in a storage system. In a parallel file system, a data object would represent a file or a directory.
- *Execution* represents the execution of an application. Multiple levels of executions are possible, such as *Job* submitted by users and running *Processes* of one job. Different rich metadata use cases may require different levels of execution details. We refer to all these entities as *Execution* entities.
- *User* represents an end user of the system.

In addition to these primary entities, users can define their own entities. For example, in a workflow system, users can create *workflow* entities and connect them with *Executions* to trace workflow executions. Also, the system administrators can create *user group* entities to include different *Users* from the same institution for better management. The only limitation is that the user-defined entities must connect with existing entities in order to keep every element in the graph accessible by traveling through the graph.

2.2 Relation to Edge

An example of different relations between entities is shown in Table 1. Each cell shows relations from the row identifier to the column identifier. Each relation will be mapped to a directed edge in the heterogeneous rich metadata graph. In Table 1, *run* indicates that the user starts an execution; *exe* means the execution is based on a corresponding executable file; and *read/write* indicates the I/O operations from executions to data objects. Since all the relations are directed, traveling back from *dest* nodes to *src* nodes will be difficult. Therefore, in the current model, we define corresponding

TABLE 1
Default Relations Definition

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

reversed relations for each relationship, in order to accelerate the reversed traversal (the *wasXXBy* relations).

Table 1 lists several default relations. In the Execution entity case, *has* means that one job has multiple processes. In the Data Object case, the relation *has/belongsTo* means that one directory may contain multiple files or directories. Also, users can create their own relations between any two existing entities. For example, two user entities can have a new relationship called *login-together* if they log into the system roughly at the same time.

2.3 Property

The rich metadata also contains annotations on entities and their relations. In the rich metadata graph, we store these as properties, which are key-value pairs attached on vertices and edges. Users can create their own properties. Examples of properties include user name, privilege, execution parameters, file permission, creation time, data source agent, data quality score, and execution environment variables.

2.4 Versioning

An important need of rich metadata is to track the full history of changes on entities, which motivates the concept of *versioning*. For example, an application may write into the same file multiple times during its execution. If only one vertex in the metadata graph represents the file, these *writes* will turn into multiple edges between the same two vertices (the *Execution* and the *Data Object*) of the graph, which might confuse users when queried on them.

To avoid this confusion, in GraphMeta, we extended the heterogeneous property graph with versioning mechanism, which is essentially an Id implicitly attached on vertices to track and distinguish changing entities. For example, multiple writes between an execution and a file can be mapped to edges toward different versions of the file. In this way, we avoid creating too many vertices for the changing entities and still track their changes efficiently.

Also, versioning helps record rich metadata of an entity after it is removed. For example, if a file is deleted from the file system, its historical versions will still be kept in the graph, in case users want to retrieve details about this deletion or query about other data files that are generated from this deleted file. The versioning concept simplifies the operations on graph entities: all modifications (e.g., deletion) will be converted to creations of corresponding entities with new versions. Versioning does raise potential consistency issues, however. We discuss more details about the consistency model in Section 4.1.

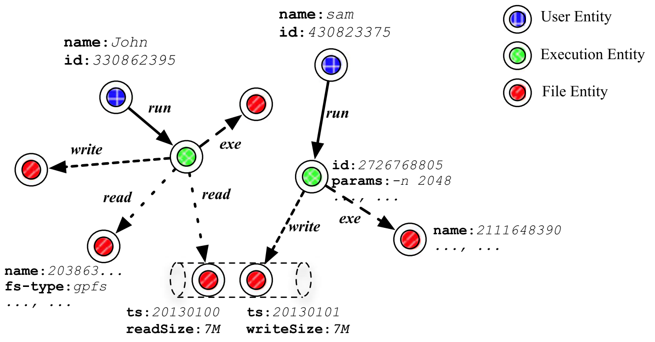


Fig. 1. Example of a rich metadata graph in HPC systems.

2.5 An Example and a Use Case

An example of an HPC rich metadata graph is shown in Fig. 1. In this example, entities including files, directories, users, and groups, as well as jobs and individual processes, can be mapped as vertices. Vertices can obtain multiple versions since the same file could be read/write at different times. relations between any two vertices are defined as directed edges, which have types indicating different relations such as `directory/file` hierarchy, `file/user` ownership, `user/job` runs, and `job read/write` files. Additional attributes of both entities and relations can be added as properties. For example, attributes on `file` vertices can contain user-defined tags in addition to file names or permission modes, and attributes on `user/job` run edges can include environment variables and parameters of the particular run.

With such an HPC rich metadata graph, one can support many critical data management tasks. For example, one can reproduce important results by tracing how the results were generated and how relevant applications were executed. Then, the rich metadata will help rebuild a controlled environment (especially viable in HPC platforms) to repeat computation and validate the results. Another example is auditing changes or damages to shared datasets, which can happen from various sources such as intentional instrumentation, misconfigured applications, or just a buggy library. The detailed rich metadata help trace activities on files and hence identify both the source of the changes or damages and the infected files affected.

2.5.1 Computation Result Analysis

We demonstrated the usefulness of rich metadata using a commonly seen use case: *computation result analysis*. Scientists commonly run many repetitions of the same simulations to generate N result files and `grep/awk` them to calculate the final results for plotting. When one checks the plots, however, unexpected variations showed. Whenever this situation arises, one needs to confirm that whether it is due to a flaw in the model/simulation or to something that was changed accidentally in the system between multiple simulation runs. To do so, one needs to check whether all simulations were using the same environment (environmental variables, configuration files, command lines, etc.). Clearly, a rich metadata graph similar to Fig. 1 helps here, since the execution parameters and system statuses are recorded and queryable. In fact, we can effectively map the query to a graph traversal on the rich metadata graph, such

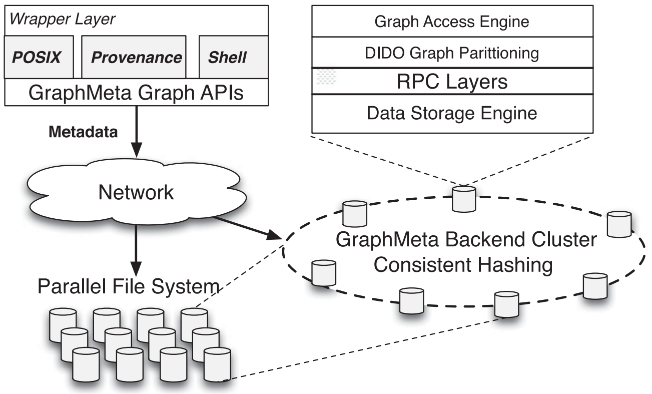


Fig. 2. Overall architecture of GraphMeta.

as *GTravel.v(r).e('wasWrittenBy').v.e('read').repeat().return_fp()*. (please check the APIs described in Section 4.2).

Here, starting from the vertex of the interesting result (`GTravel.v(r)`), we travel back repeatedly (`repeat()`) through relations `wasWrittenBy` and `read` to grab information about the previous execution flows. Instead of returning the destination vertices, this command returns the full path (`return_fp`) to let users fully understand how this result was previously generated.

Such an use case is common in today's data-driven era. Scientists usually spend considerable time analyzing the huge amount of results generated from their applications with different parameters, inputs, and algorithms. The detailed rich metadata bookkeeping the attributes of running jobs, data objects, and users and their relations will be extremely helpful.

3 Features of Rich Metadata Graphs

Although using an heterogeneous graph to model HPC rich metadata is straightforward and promising, building a graph-based metadata management system is nontrivial due to the unique features of HPC rich metadata graphs. In a previous study [9], we examined the rich metadata graphs constructed based on a year's worth of publicly available Darshan logs [13] of the Argonne Intrepid Blue Gene/P supercomputer [14], and we drew the following observations that inform the challenges.

Large Graph. Rich metadata graphs can easily contain hundreds of millions of vertices and edges, and this amount is expected to increase both with the HPC system scale and with more complex use cases. Hence, the ability to support large graphs efficiently in a distributed setting is a primary concern. Note that, the large graph does not necessarily lead to a large graph size (byte-wise). For instance, a billion-edge graph may only take 10 GB space if each vertex takes 10 bytes. However, the graph size (byte-wise) is not the only reason for deploying distributed management. In many cases, the highly concurrent workloads issued from multiple/many clients (e.g., the case for rich metadata management in HPC), demand a distributed deployment to provide quality service to applications, even though the stored graphs are not that large.

High Mutation Rate. Rich metadata are generated concurrently from large-scale applications, which typically involve a large amount of data/metadata activity up front, as well as on regular intervals (e.g., checkpointing). Recording rich metadata at runtime thus requires an ingest rate that can keep up with bursts of activity. This brings challenges to the ingestion speed of rich metadata, which potentially requires a highly write-optimized storage system.

Travel-Based Access Pattern. With the graph model, many rich metadata use cases, from *data audit* to *result validation*, can be implemented in a common pattern: starting from a set of vertices and traveling through relations to reach the interested vertices. Such operations can be implemented as a *graph traversal* in the graph model. Since the graph is normally large and distributed and, at the same time, the queries are commonly issued by multiple data management tasks, the graph traversal must be served efficiently.

It is worth noting that, the graph traversals mapped from HPC rich metadata operations are different from traditional graph traversals. Specifically, due to the heterogeneity of the graph, it is allowed to re-visit the same vertices again in different traversal steps. For example, a file might be written by a job and later be read by another job along the execution of a workflow. Then, to travel back the workflow execution, we need to visit this file twice in different graph traversal steps. Such a change makes traversals on the rich metadata graphs potentially extremely deep. This simply requires much more efficient traversal implementation.

Power-Law Distribution. Similar to POSIX file/directory distributions in HPC systems, where the vast majority of directories contain small file counts and a minority contain huge counts [15], rich metadata graphs follow the power-law distribution [16] on the vertex degrees for various entities, as our previous work observed [9]. Together with the distributed deployment, this makes graph partitioning critical since it significantly affects the performance on both graph mutation and traversal.

Motivated by these observations, we introduce here a distributed graph-based rich metadata management engine, GraphMeta. We describe the details of the GraphMeta solution in next section.

4 GraphMeta Design

The overall architecture of GraphMeta is shown in Fig. 2, which contains two major components. The client-side component includes graph APIs and wrappers. This component is normally linked with applications that run on the computing or login nodes of an HPC cluster to manage rich metadata. The GraphMeta also provides an interactive shell for users to easily manipulate and view the rich metadata instantly.

The server-side component runs on a subset of I/O nodes or computing nodes or a dedicated cluster. Multiple nodes construct the GraphMeta backend cluster in a decentralized way. Each node runs the same set of components, which include a graph partitioning layer, a data storage engine, and a graph access engine. The access engine accepts requests from clients and serves them with the help of other two components. To allow the dynamic growth (or shrink) of GraphMeta cluster based on metadata workloads, we adopt a consistent hashing mechanism to manage the backend servers similarly to Dynamo [17]: the entire hash space is divided into K virtual nodes, with each assigned to one physical server to balance loads. The mapping from virtual nodes to physical servers is kept in the distributed coordinating service *zookeeper* [18].

Note that GraphMeta is designed as a service on top of the existing HPC software stack. Each GraphMeta instance stores its data onto the parallel file system, thus enabling GraphMeta to run on diskless compute nodes as a well-adopted strategy [19]. More important, it simplifies the fault tolerance design by leveraging that of parallel file systems (i.e., through RAID devices).

4.1 GraphMeta Versioning and Consistency Model

GraphMeta utilizes the graph model to describe the diverse rich metadata. As described in the preceding section, the graph-based rich metadata model supports multiple versions on a vertex to record its history. During implementation, GraphMeta uses server-side timestamps as the version number of the vertex. Specifically, any request on the metadata graph will be implicitly attached with the timestamps of the server when the request was received. Using timestamps as version numbers helps establish a deterministic order of accesses and guarantees that a read operation will not retrieve metadata that are inserted after the read was executed. For concurrent metadata writes from multiple clients on the same entity, the timestamps ensure that the latest write wins; for reads, GraphMeta always returns the data with the latest timestamps that are ahead of the request timestamps. Users are also allowed to manually query data based on a specific timestamp.

However, such a concurrency control depends heavily on the accurate synchronization of server-side timestamps while comparing concurrent operations that have occurred in different servers. Although these timestamps are typically well synchronized in HPC environments—Google's TrueTime, for example, also relies on timestamps to keep its data centers synchronized and to support transactions [20]—a small amount of clock skew is inevitable. Taking this skew into consideration, we have not designed GraphMeta to achieve strong POSIX semantics to replace existing POSIX metadata systems. Instead, the *session semantics*, which guarantees that a single process always reads its latest write, is supported in GraphMeta. Such a relaxed model is acceptable because, unlike POSIX file system metadata, many rich metadata use cases explicitly decouple data collection from subsequent analysis. A delay of a few milliseconds when reading the latest writes is acceptable for most cases, such as in tracing back suspicious executions from broken files, auditing users operations, or counting file reads/writes statistics. Relaxing consistency simplifies our goal of supporting both high ingest rates and efficient traversal with better scalability and availability. Exploring the possibility of supporting stronger consistency models is considered as future work of GraphMeta.

4.2 GraphMeta APIs

GraphMeta provides APIs for operating on metadata graphs, covering vertex/edge ("one-off") access and multistep traversal. Four sets of APIs for graph accessing operations on vertices and edges are provided.

- *Insert vertex/edge*. In order to insert a vertex, the vertex type, Id, and a key-value list containing all the attributes are needed. In order to insert an edge, the edge type, source vertex Id, destination vertex Id, and a key-value list of edge attributes are required. All the vertices and edges are created or updated with a version, which is assigned by the system implicitly. *Updates* on vertices and edges are considered as inserts with a new version.
- *Read vertex/edge*. Read accesses the attributes of specific vertex or edge. Users can specify the attribute name to access only the interested one on the vertex. But, on the edge, users need to manually parse them, since read returns all the attributes as a map. By default, the read operation always returns data of the latest version. Users can also specify the version range, in order to make GraphMeta return all relevant data.
- *Scan edges*. Scan iterates the edges of a given vertex. Users need to give the source vertex Id and the edge type in order to use this function. Optional version information is also supported for retrieving data within a given range. By default, only edges of the latest version are returned.
- *Delete vertex/edge*. Delete can be applied on both vertices and edges. The deleted data are not removed instantly; they are just marked as obsoleted and will be archived during a later compacting phase.

Also, GraphMeta provides graph traversal APIs to support advanced data management use cases. For example, finding the original dataset of a scientific result can be mapped to a multistep traversal from the result files through *readBy* edges back to the original data files. We utilize a query-building language to describe such graph traversals. The primary class is `GTravel`, which contains several core methods listed below (a more detailed definition can be found in our previous work [7]).

- *Vertex/edge selector*. Vertex selector `v()` represents an entry point for graph traversal. It selects a working set of vertices by given Ids. The edge selector `e()` selects specific edges starting from the current working set of vertices by its label argument. Note that versioning is supported in both selectors.
- *Attribute filters*. Attribute filters select needed vertices (`va()`) and edges (`ea()`) during traversal. Different filter types (i.e., *EQ*, *IN*, and *RANGE*) are supported. Multiple property filters can be applied in one step to filter more entities by using the *AND*operation. *OR* is not explicitly supported; users can issue different traversals and combine their results together to form such behavior.
- *Return indicator*. A return method (`rtn()`) tells the traversal engine which working set of vertices should be returned. Normally, graph traversals return the final vertices. GraphMeta extends this behavior by allowing the return of the intermediate set of vertices or the entire path. To return the intermediate vertices, users simply add `rtn()` to the call chain of `GTravel` after the corresponding traversal step. To return the entire path, users simply add `return_fp()` in the traversal request.

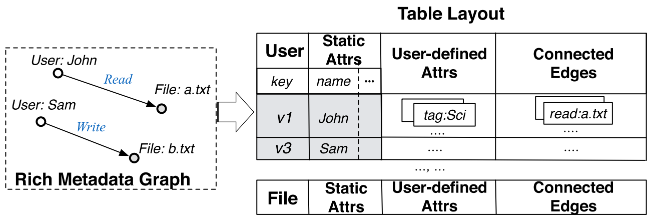


Fig. 3. Logical view of metadata graph.

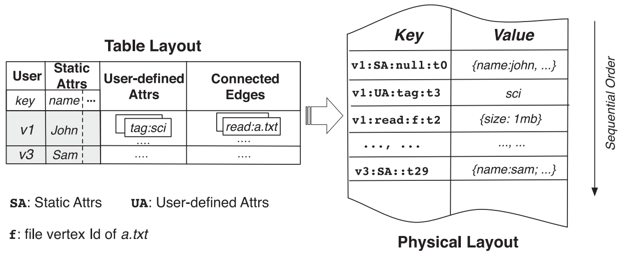


Fig. 4. Data layout of rich metadata graph.

Most rich metadata operations can be readily implemented by using these APIs. For example, file attribute reads can be implemented as file vertex accesses; listing all jobs that one user has executed can be implemented as `scan` with optional timestamps; and validating scientific results can be realized by traversing the graph and returning the entire path to help rebuild an identical previous execution environment.

Currently GraphMeta does not provide indexing for full-text searching. Thus, meaningful Ids should be assigned to vertices for easy access. For example, in the current prototype system, the user vertex Ids are set with the operating system user Ids, so that users can easily learn their own Id to access the metadata graph; distributed file vertex Ids are their full file paths, so that users can easily find a file's rich metadata knowing only its path. In future work we will evaluate the possible use of indexing and searching such as Lucene [21] to simplify locating interesting vertices.

5 GraphMeta Implementation

As discussed in Section 3, it is nontrivial to implement a rich metadata management system in HPC platforms: we need to efficiently ingest the concurrent metadata creations and mutations (i.e., graph insertions) from a large number of compute nodes, while at the same time, partition the graph into multiple storage servers and support graph traversal effectively. In GraphMeta, these are achieved through a write-optimal data store engine, a new graph-partitioning algorithm, and an optimized graph traversal engine, introduced in the next three sections.

5.1 Write-Optimal Local Data Store Engine

GraphMeta implemented a two-layer data layout to locally store the graphs, as shown in Figs. 3 and 4.

Fig. 3 shows the logical layout of a rich metadata graph. The layout can be viewed as a pseudotabular form, where each "row" consists of all relevant data of a vertex, including predefined static attributes, extensible user-defined attributes, and outgoing edges. Static attributes may include *permissions* and *stat* entries for files. User-defined attributes may include annotations or format descriptions for files. Edges are stored inside the *Connected Edges* column. One table is created for one vertex type. Such a tabular-like logical layout is similar to many NoSQL databases (e.g., Cassandra [22] and HBase [23]) and differs from relational databases since it contains "fat columns" that can support more flexible data structures.

Fig. 4 further shows the physical layout of the graph. In the current implementation, we utilize RocksDB [24], a write-optimized database using a log-structured merge-tree (LSM) [25] to persist the physical layout. Unlike other LSM-based storage systems such as LevelDB [26], RocksDB stores key-value pairs into SSTable files in lexicographical order. GraphMeta leverages this feature to maximize sequential access performance. Specifically, all data related with a vertex, including its static attributes, user-defined attributes, and connected edges, are stored together using the same vertex Id as the key prefix. For each vertex, its static attributes have a key formatted as [*vertex-id, marker, attr-key, timestamps*], where the *marker* is a manually selected constant to make static attributes be lexicographically minimal with respect to other entries. Thus, static attribute retrieval can be performed once the vertex is located. The data are possibly already in memory because of the storage system prefetching. The *attr-key* indicates the name of the attribute. The *timestamps* attached at the end of the key construct a reversed order so that the latest results are chosen by default. After the static attributes section, all user-defined attributes are stored together. They have a similar key format except that the *marker* is set to be a larger value in order to guarantee the order. The last section stores all connected edges. They have a different key format, [*vertex-id, edge-type, dest-id, timestamps*]. All edges should be sorted by *edge-type* in order to help both scan and traversal queries that access mostly specific types of edges. Here, *dest-id* refers to the destination vertex Id. The value of these keys is a map of the edge properties.

Since compaction overhead is a common problem of LSM-based storage engines, in our GraphMeta implementation, similar to IndexFS [19], we introduce a second LSM table to store only the keys and pointer to the corresponding record in the full table. This table is smaller and therefore compacts less frequently. Through the local storage engine, GraphMeta benefits write performance significantly. Since LSM is not friendly to read operations, we organize the relevant key-value pairs together to maximize sequential access. In addition, to improve the traversal performance, GraphMeta introduces optimizations on memory caching and prefetching, described in Section 5.3.

5.2 GraphMeta Partition: DIDO Algorithm

The large size, high mutation rate, and unbalanced distribution of the rich metadata necessitate partitioning the metadata graph across multiple servers. Similar needs were also seen in distributed or parallel file systems, where the tree-structure POSIX metadata need to be efficiently partitioned. Various POSIX metadata placement algorithms were already studied. Among them, GIGA+ [15] is a representative one, which provides the state-of-the-art metadata scalability by incrementally splitting an oversize directory into

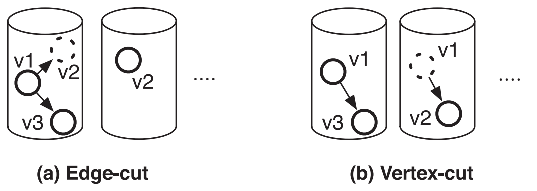


Fig. 5. Edge-cut and vertex-cut graph-partitioning methods. Solid circles/lines represent vertex/edge storage locations.

two equal parts and storing them in two servers. Comparing to metadata partitioning in parallel file systems, partitioning rich metadata is clearly more complicate as the relations (i.e., edges) are no longer only between parents and children: they could be between any two vertices.

Although GraphMeta partitioning is most likely to be a graph partitioning problem, its counterpart in storage system would still be metadata placement algorithms like GIGA+. So, in this study, we extensively studied and compared them to carefully exam the advantages and issues of GraphMeta partitioning.

5.2.1 Graph Partitioning Background

Numerous studies have also been made on general graph partitioning. The formulation that is most relevant to our study is called k-way partitioning. Given a graph G and a number k as inputs, the goal is to cut G into k balanced pieces while minimizing the number of edges cut. Although it is an NP-complete problem [27], many heuristics, such as METIS [28], Multilevel algorithm [29], and SBV-Cut [30], solve this problem with effective solutions. However, those solutions all rely on the global information of the graph, an approach that is not feasible for GraphMeta since the vertices and edges are continuously inserted and have to be partitioned in an online manner.

Research studies also have focused on effective graph partitioning on streaming graphs, such as LDG [31], Fennel [32], and restreaming LDG [33]. However, these methods need at least local information about the graph, for example, knowing all connected edges when inserting a vertex; again, not feasible in GraphMeta. In fact, most of these graph-partitioning algorithms are used in *graph loader*, which loads on-disk graphs into memory in an optimized partition to accelerate later analysis. GraphMeta, on the other hand, is designed to continuously ingest rich metadata without knowing any local or global graph structure, hence, cannot leverage those algorithms.

It is common to use hash strategies to partition graphs online. Typical strategies include *edge-cut* and *vertex-cut* [34], as shown in Fig. 5. Edge-cuts distribute vertices, together with their outgoing edges by hashing the vertex Id. Many distributed graph databases, including Titan [35] and OrientDB [36], use this strategy as the default partitioner. Edge-cuts are impractical for GraphMeta, however, because vertices in metadata graphs can have millions of outgoing edges, causing significant load imbalance. Vertex-cut distributes edges instead of vertices by hashing the edge Id, as Fig. 5b shows. It is used in graph databases and graph-processing frameworks such as GraphX [37] and PowerGraph [34]. It achieves balance for high-degree vertices but performs poorly for low-degree vertices.

GraphMeta requires retaining fast point access (individual vertices), efficient edge scan, and optimized multistep traversals without excessive server communications. These requirements cannot be well met through existing graph partitioner, thus motivating the introduction of a new graph-partitioning algorithm.

5.2.2 DIDO Graph Partitioning

In GraphMeta, we proposed a *destination-dependent optimized* (DIDO) partitioning algorithm to orchestrate parallelism and locality of graph traversal. The core idea of DIDO involves two aspects: first, it incrementally partitions vertices based on their out-degrees for better parallelism; and second, it considers edge placement with the location of the respective destination vertices during partitioning in order to improve locality.

Incremental Partitioning and Splitting. DIDO begins with placing vertex and all its out-edges and associated attributes together on a single server, similar to edge-cut. Specifically, vertices are mapped to servers by applying a hash function on vertex Ids, which enables single-hop vertex lookups to guarantee a fast point access on the vertex. Note that the hash function returns a virtual node Id, which is mapped to one physical node through consistent hashing. For simplicity, we refer to virtual nodes as servers in the following text.

Any new edge will be inserted into the server that stores its source vertex. When more edges are inserted that cause the out-degree of their source vertex to surpass a configurable threshold (*split threshold*), DIDO starts to partition that vertex into multiple parts in order to increase the parallelism (i.e., reducing the time cost of scanning its edges in one single server). Specifically, DIDO will partition all out-edges of the vertex into two partitions: one stays on the original server, and one moves to another server in order to balance the loads.

In this way, the vertices are incrementally partitioned based on the number of its out-edges until all its edges are spread across all the storage servers. This incremental splitting strategy differentiates vertices based on their out-edge degrees and achieves much better parallelism for high-degree vertices without sacrificing the performance on low-degree vertices.

Destination-Dependent Splitting. DIDO further considers the locality with respect to accessing the destination vertices during splitting, as other wise, the traversal performance will be affected. For example, if an edge ($v \rightarrow u$) is spit to a server (s_1) which u was not actually stored in, the performance of traveling from v to u will be worse as more network communications are needed.

To maximize the locality, we propose the *destination-dependent optimization* with the help of *partition tree* data structure, during splitting. An example of the partition tree is shown as Fig. 6. It is a per-vertex data structure and is built using the following strategy. First, the root of the vertex will be set to S_v, which actually stores the source vertex v. Then, each node of the tree has two children. The left child corresponds to the same server as the parent; the right one indicates the next server that has not been shown in the tree yet. We choose the next server in a round-robin way, starting from S_v. Repeating this, all k servers will assigned into such a tree with a maximum of $\log(k) + 1$ levels. This

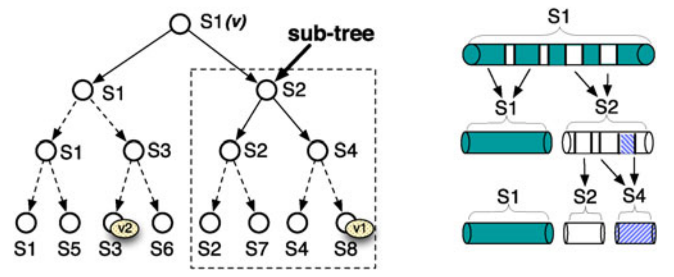


Fig. 6. DIDO tree structure for server assignment (left). The right figure demonstrates how data is moved during partitioning. Labels on tree nodes correspond to virtual nodes, referred to as “server” in the text for simplicity.

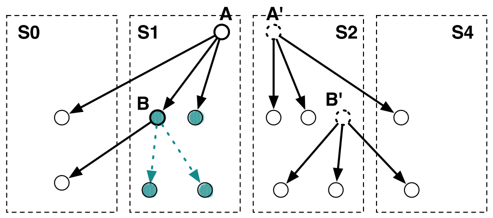


Fig. 7. Prefetching scenario while a traversal happens from A. Solid and dashed circles show that edges of vertex A and B are partitioned into two servers. Colored and dashed vertices and edges are prefetched in S_1.

tree organization is fixed for each v and is easily calculable before any splitting happens.

Then, during the incremental partitioning, when we split the out-edges of a vertex (v) on S_v into two servers (S_v and S_{new}), we always select the new server (S_{new}) based on the tree structure (i.e., the right child of S_v in the tree). For example, as Fig. 6 shows, the first time S_2 is extended, S_4 will be chosen to share its edges. If S_2 is extended again, another server S_7 will be chosen.

After deciding which server (S_{new}) will be used, we further decide which parts of the out-edges of v need be moved to S_{new}. DIDO does not naively split all edges into two equal parts. Instead, it calculates the locations of the destination vertices of the edge and tries to *move edge into the server that could eventually be split to where the destination vertex is stored*, based on the partition tree. For example, in Fig. 6, when S_1 (tree root) is partitioned for the first time, the edge $e_1(v \rightarrow v_1)$ will be put into the new server S_2 because v_1 is stored in S_8, which is a grandchild of S_2 in that subtree. In this way, after several rounds of splitting, it is guaranteed that any moved edge either has been co-located with its destination vertex or will be co-located upon further partitioning. This approach indicates a strong locality. Our evaluations also showed significantly improved scan/scatter and traversal performance.

5.2.3 Comparison and Discussion

The idea of using an incremental strategy to partition power-law distributed entities was also investigated in graph processing frameworks [38]. But, it mostly focuses on improving the throughput of graph processing like running PageRank, hence largely ignores the connectivity of split edges. On the POSIX metadata management, GIGA+ [15] is one example of incrementally partitioning storage of directory entry metadata, scaling to millions of files in a single directory. Comparing with these existing studies, DIDO significantly improves them through the new partition tree-based placement strategy, which guarantees edge/vertex locality to largely improve scan and traversal performance. Detailed evaluation results are presented in Section 6.

5.3 Synchronous Graph Traversal Engine

Since many rich metadata use cases can be straightforward mapped to graph traversal requests, the performance of graph traversal is critical to GraphMeta. In our previous work [7], we have proposed an asynchronous graph traversal engine (GraphTrek) to better support the deep graph traversals by avoiding synchronously waiting for possible stragglers in the distributed systems.

In GraphMeta, we re-visited the design of graph traversal engine. We tested the level-synchronous breadth-first traversal engine again and argued that with proper optimizations (*pre-prefetching* as described later) and proper graph partitioning algorithm (*DIDO* which leads to better locality), the synchronous graph traversal engine can achieve very similar performance comparing with highly optimized asynchronous engine. The key rational is when there are stragglers during synchronous graph traversal, the fast-moving processes can pre-fetch the locally stored edges and destination vertices.

More specifically, the pre-fetching during traversal can happen in two places. First, whenever visiting a vertex (v_1) during traversal, its edges can be prefetched based on the `edge-type` parameter. This prefetching is highly efficient and mostly done implicitly since edges are stored right after the vertex in local storage layout of GraphMeta.

Second, in multistep synchronous graph traversal, in order to serve traversal request on step k, a server needs to read all vertices belonging to this working set of step k, filter out those that do not satisfy the traversal request, read their edges and the destination vertices to build the working set for next step, and wait for step $k+1$ to start. With prefetching, while waiting for the next step to start, a helper thread can be started to prefetch all vertices belong to $k+1$, their edges, and destination vertices into memory as long as they are stored locally. Fig. 7 shows an example of this prefetching strategy. In server S_1, after reading vertex A and its edges, GraphMeta fetches all colored vertices, their connected local edges, and local destination vertices. Remote vertices and edges will not be fetched. In the next step, $k+1$, the cost of loading vertices from disks can be saved since they are already in memory. If after prefetching step $k+1$, the server still needs to wait, it will further carry out prefetching more steps. Thus, the longer the waiting is, the more chances for prefetching. Since only vertices or edges stored locally are prefetched, this strategy can largely overlap network communication and hence significantly reduce the traversal time. Overall, prefetching is most effective for deep traversals that contain long steps, a situation that is common in rich metadata use cases.

Fig. 7 also shows that DIDO partitioning is critical for improving deep traversal performance by coordinating with the prefetching mechanism. As we have described, the number of vertices and edges that can be prefetched is based on their locality. DIDO provides better locality

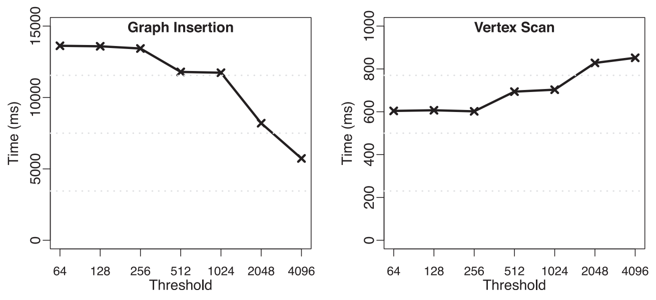


Fig. 8. Insert and scan performance versus split threshold.

compared with naive incremental partition strategies such as GIGA+ and hence increases the possibility of prefetching.

In GraphMeta, therefore, we instead selected synchronous traversal as the default engine and left the asynchronous engine as an option. The reasons are mainly threefold: (1) DIDO's graph-partitioning algorithm provides a more balanced graph distribution, which is less likely affected by stragglers; (2) the prefetching mechanism in synchronous traversal also reduces unnecessary waits on global synchronization similar to what asynchronous traversal does; and (3) progress tracking and fault tolerance of synchronous traversal are much simpler and more efficient. We will show the performance comparison between these two engines in evaluation section.

6 Evaluation

In this section, we present evaluation results with GraphMeta. We conducted detailed experiments on the graph-partitioning algorithms, prefetching, and the entire system in different scenarios. We compared GraphMeta with representative graph databases (i.e., Titan) which can also potentially serve the graph-based rich metadata model. We also evaluated on POSIX workloads to show the performance advantage of GraphMeta. All results indicate that GraphMeta is an efficient graph-based engine for managing large-scale rich metadata in HPC platforms.

All evaluations were conducted on the Fusion cluster at Argonne National Laboratory [39]. Fusion contains 320 nodes, and we used 2 to 32 nodes as backend servers in these evaluations. Each node has a dual-socket, quad-core 2.53 GHz Intel Xeon CPU with 36 GB of memory and 250 GB of local hard disk. All nodes are connected with a high-speed network interconnection (InfiniBand QDR 4 GB/s per link, per direction). The global parallel file system includes a 90 TB GPFS file system, which contains 8 metadata servers.

6.1 Evaluation Datasets

We used two sets of datasets in the evaluations. The first is generated from a whole year's Darshan logs from the Intrepid supercomputer [14]. This dataset represents the metadata generated in HPC platforms. The entire graph contains around 70 million vertices and edges. The distribution of vertex degrees follows the power-law distribution, where the highest degree vertex has around 30 K connected edges and most of the vertices have fewer than 10 connected edges.

The second set of datasets includes two graph datasets, which were used for evaluating the performance of graph partitioning and traversal performance. One of the tested graph is the amazon0302 data from SNAP dataset [40]. It contains 2,469,754 edges and 262,111 vertices. Another graph is a synthetic graph generated by the RMAT graph generator [41] using a "recursive matrix" model simulating a real-world power-law graph. The generated directed graph contains 1 million vertices and 16 million edges with an average out-degree of 16. We used the following parameters for all generated RMAT graphs with moderate out-degree skewness: $a = 0.45, b = 0.15, c = 0.15, d = 0.25$.

Note that, the vertices and edges in the all these graphs contain randomly generated attributes (the attribute size is 128 bytes).

6.2 DIDO Parameters: Split Threshold

As an incremental graph-partitioning algorithm, DIDO contains a critical parameter, *split threshold*, to determine when a vertex should start to split. A threshold that is too small causes frequent splits that may degrade write performance; a threshold that is too large may create a performance bottleneck and affect the scan performance. In this evaluation, we issued insert and scan on a single vertex with 8,192 edges on a 32-node cluster from a single client. We changed the split threshold from 128 to 4,096, corresponding to 16 to 512 K physical storage with a 128-byte vertex size, to find out the rule for determining the split threshold.

The results are shown in Fig. 8, where the y-axis shows execution time (ms) and the x-axis corresponds to different threshold values. In general, graph insertion is faster with larger thresholds because it reduces the split frequency. However, it degrades the scan operation as more edges are stored in a single server. Determining an optimal threshold is difficult because of the influence of many factors including disk speed, network bandwidth, and number of servers. We recommend that users set the best parameters according to their use cases and the cluster. In this series of tests, between 128 and 256, we chose 128 as the default threshold for two reasons. First, it is small enough for most of our test graphs to use up to all 32 servers. Second, it achieves a good balance between scanning and insertion performance.

6.3 Traversal Prefetching Performance

6.3.1 Prefetching Breakdown Analyais

GraphMeta introduced a new prefetching mechanism to improve the performance of level-synchronous breadth-first traversal. We evaluated the efficiency of prefetching by conducting a breakdown performance analysis of an example multistep traversal. Specifically, we ran a 6-step traversal on an 8-server GraphMeta cluster, starting from a randomly selected vertex. We chose 8 servers to save space while plotting the results (as shown in Fig. 9). A 32-server cluster will give similar results.

In Fig. 9, we tracked the time cost of each step in each server and plotted the detailed duration time of each phase of a step. Specifically, we measured the time cost of loading local vertices, the time cost of loading local edges, and the time cost of prefetching. The figure clearly shows the benefit of introducing prefetching in GraphMeta. The performance advantage also increases while more steps are needed in the traversal. We observe similar results in later evaluations.

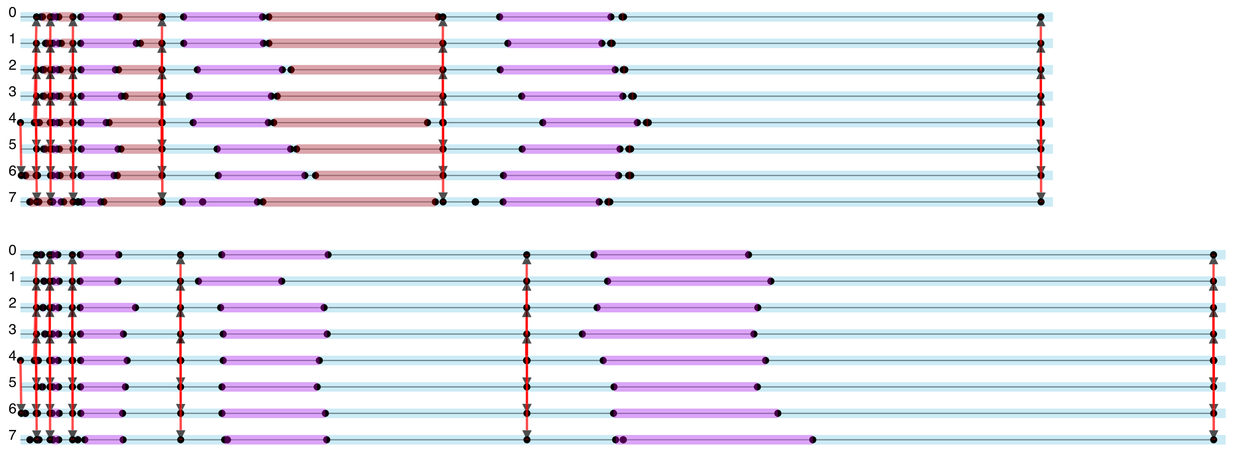


Fig. 9. Detailed demonstration of 6-step graph traversal on 8 servers. The upper one shows the result with prefetching, and the lower one shows the result without prefetching. The red arrows show a request of starting the kth step of the traversal. The blue bars show the time cost of reading local vertices. The purple bars show the time cost of reading local edges. The brown bars in the upper figure show the time cost of prefetching in each step. The prefetching stops immediately when the next step starts.

6.3.2 Performance Comparison

In addition to the breakdown analysis, we further evaluated the performance benefit of prefetching in synchronous graph traversal. In this evaluation, we ran 2-step, 4-step, and 8-step graph traversal using plain synchronous engine (Plain-Sync), prefetch-enabled synchronous engine (Pre-Sync), and asynchronous engine (Async) with four different partitioning algorithms, on the 8-node cluster. In these evaluations, we used amazon0302 graph and started the traversal from vertex 0. Similar results can be seen in other graphs.

Fig. 10 shows the results. From these plots, we can conclude: 1) pre-fetching improves the performance of synchronous traversal for most of the partitioning algorithms; 2) For

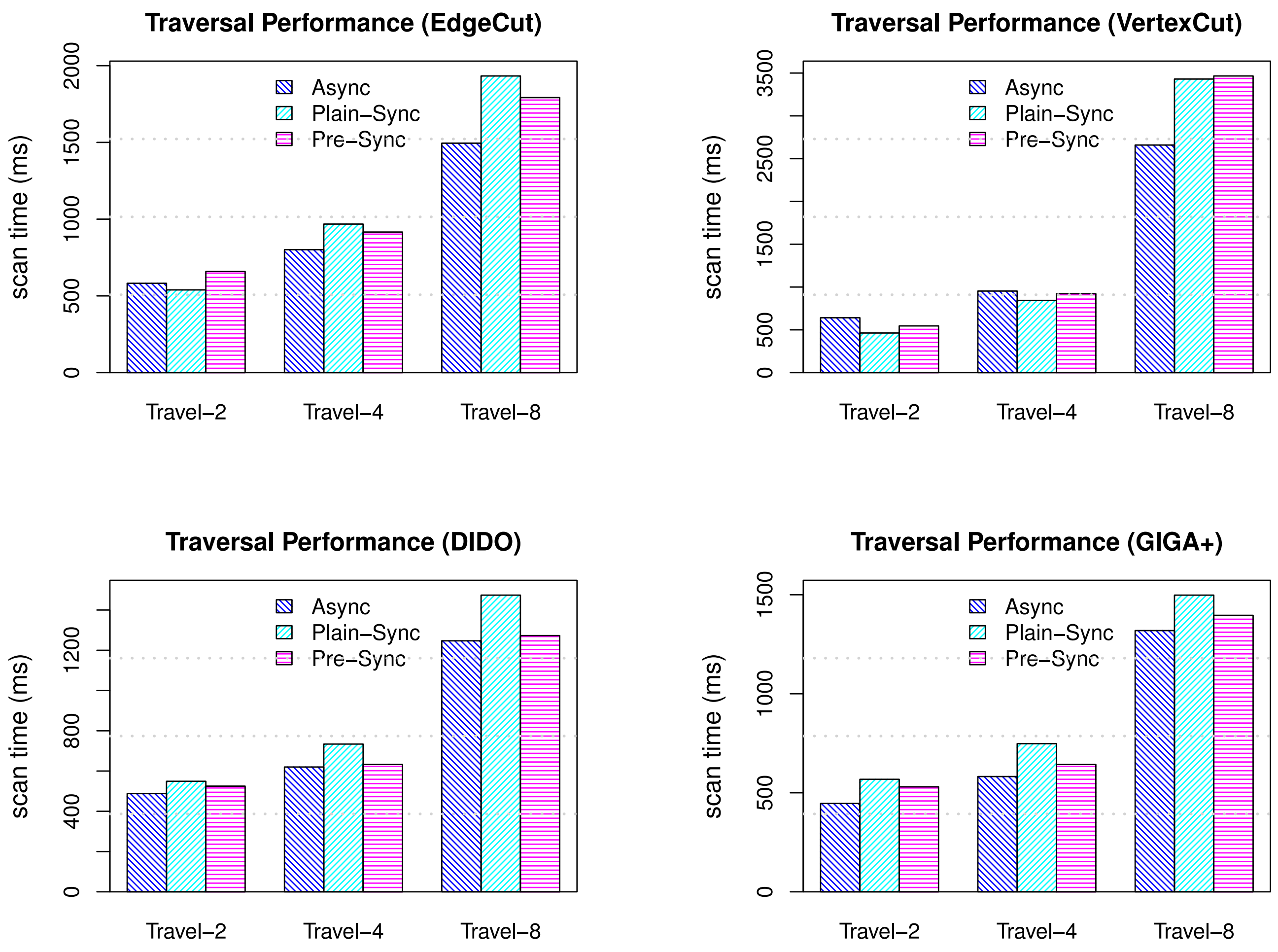


Fig. 10. Traversal performance comparison using three different traversal engines with four partitioning algorithms.

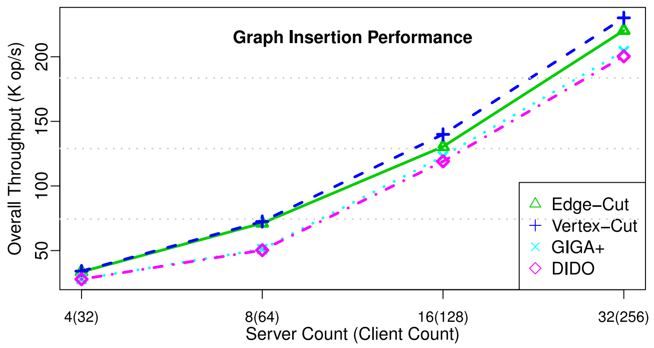


Fig. 11. Insertion performance with different partition strategies.

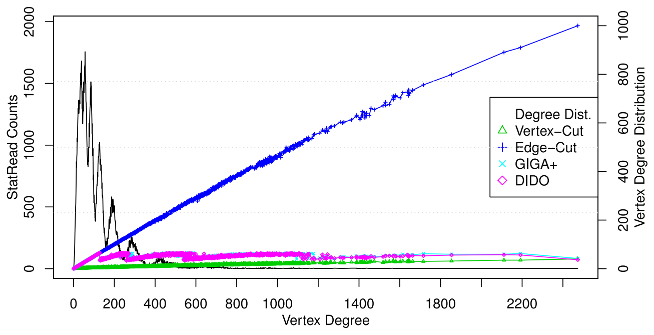


Fig. 13. *StatReads* of scan/scatter.

partitioning algorithms like DIDO or GIGA+, the performance of pre-fetched synchronous traversal matches the performance of the optimized asynchronous traversal; 3) the pre-fetching does not improve the traversal performance for vertex-cut, mostly because the performance bottleneck for vertex-cut is the network (also shown in later evaluation result Fig. 14) instead of accessing the graph from storage.

6.4 Evaluating Graph Partitioning

We compared different graph-partitioning strategies, including edge-cut, vertex-cut, GIGA+, and DIDO, with different graph operations in this section. The GIGA+ implementation was imported from the IndexFS project [19], and we mapped directories and files as vertices. For vertex-cut, we used a combination of source vertex Id and destination vertex Ids as edge Ids in order to distribute the edges. For edge-cut, we simply hashed the source vertex Id to assign vertices.

6.4.1 *Metadata Ingestion Performance*

We first show the performance of inserting metadata into the graph using different graph-partitioning algorithms. We used the real-world Darshan logs as evaluation datasets. We used n servers and $8 * n$ clients to perform these evaluations ($n = 4 \rightarrow 32$). Each client loaded part of the Darshan logs and issued graph insertions in parallel.

Fig. 11 shows the insertion performance of four partitioning algorithms. They all scaled well with more servers: for a 32-node setting, we were able to achieve around 200 K ops/s graph insertions. Among all the partitioning algorithms, vertex-cut achieves the best insertion performance because it hashes all edges uniformly across the cluster without

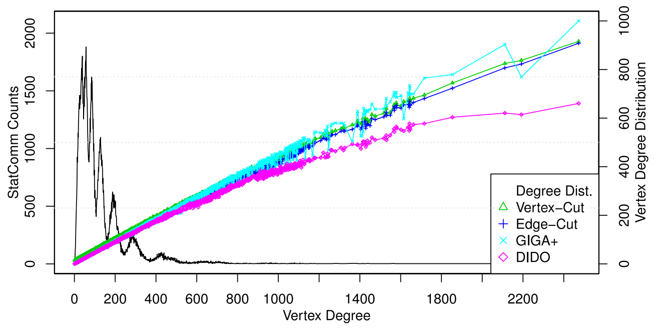


Fig. 12. *StatComm* of scan/scatter.

generating any performance bottleneck. But as we have discussed, this simple strategy also leads to problems for scanning low-degree vertices. Edge-cut is worse than vertex-cut, mainly because of the existence of high-degree vertices. But since the number of high-degree vertices is small, the overall performance degradation is not too large. GIGA+ and DIDO both have a little worse performance than does vertex-cut, mainly because of the splitting phases for high-degree vertices. The small performance difference between DIDO and GIGA+ is due mostly to the extra computation of edge placement of DIDO while splitting.

6.4.2 *Graph Scan/Scatter and Traversal Performance*

To effectively compare different partition algorithm with the scan/scatter and traversal workload, we conducted both statistical and real performance evaluations. For statistical comparison, we devised two metrics to compare the graph-partitioning algorithms:

1. *StatComm* measures the cross-server communication caused by the partitioning. Specifically, if the vertex and edges are not stored together, StatComm is incremented. By calculating the total StatComm value for the entire operation, we were able to infer the total communication cost.
2. *StatReads* measures the I/O costs across different servers due to partitioning. For each traversal step, it counts the number of requests falling into each storage server and chooses the maximal one as the I/O cost for that step. Adding together all StatReads generated from each step reflects the total cost of I/O operations.

In this test, we created a RMAT graph with 100 K vertices and 12.8 M edges to simulate the metadata graph generated from typical HPC systems such as Intrepid [14]. We applied four graph-partitioning algorithms on them in both the scan and graph traversal cases. The split threshold for DIDO and GIGA+ was configured as 128. To show the effects of vertex degree, we sampled one vertex from each degree to run the tests. In all the evaluations, we used 32 physical servers. Figs. 12, 13, 14, and 15 show the results. Each figure shows the performance of one metric for a case. The x-axis denotes all possible degrees of the generated RMAT graph (from 1 to ≈ 2,500). The black *Degree Dist.* lines corresponding to the right y-axis show the number of vertices that have a certain number of degrees. The other four lines corresponding to the left y-axis denote the metric values of vertex-cut,

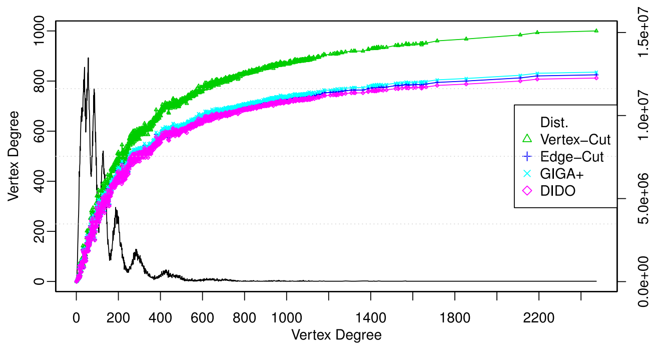


Fig. 14. *StatComm* of 3-step traversal.

edge-cut, GIGA+, and DIDO, respectively. For all of them, smaller is better.

From these figures, we see that both *StatComm* and *StatReads* increase with the vertex degree, as expected since high-degree vertices touch more vertices while traveling. In Figs. 12 and 14, we compare *StatComm* among the four partitioning algorithms. We can see that the DIDO algorithm constantly exhibits the least cross-server communication in all cases, especially compared with the naive incremental partitioner GIGA+. This is due mostly to the tree-based edge placement optimization in DIDO. For scan/scatter, GIGA+, vertex-cut, and edge-cut share similar cross-server communication except that GIGA+ contains more variations. For multistep traversal, vertex-cut becomes significantly worse. Looking closely In Fig. 12, we see that DIDO actually starts to have smaller *StatComm* from vertices over 128 degree as they start to split.

In Figs. 13 and 15, we check the *StatReads*. Overall, vertex-cut obtains the best I/O balance since it fully utilizes all storage servers. This also explains why it is not sensitive to optimization like pre-fetching mechanism. But DIDO and GIGA+ still can keep a very small difference as they incrementally partition the large vertices. In both test cases, edge-cut is significantly worse because of the imbalanced graphs. In Fig. 13, we see that for small degree servers (less than 128), DIDO is actually worse than vertex-cut. But once the vertices start to split, better I/O balance is achieved by DIDO. We also note that GIGA+ performs worse in terms of the *StatComm* metric. The main reason is that in GIGA+, the edge is distributed away from its source vertex and is assigned to a random server, thus losing the locality with its destination vertex.

We further evaluated the performance of the different partitioning algorithms using the amazon0302 graph on

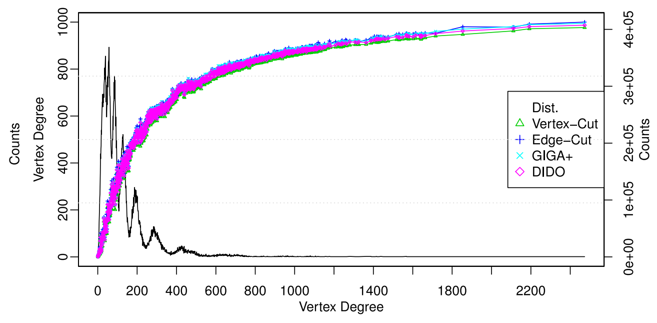


Fig. 15. *StatReads* of 3-step traversal.

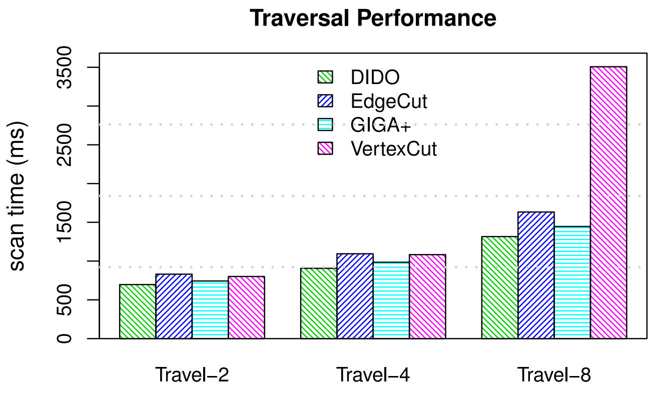


Fig. 16. Traversal performance on sampled vertices.

the 32-node cluster to confirm the statistical results in Fig. 16. All the traversals started from vertex 0 and lasted 2, 4, or 8 steps in this evaluation. From these figures, we can identify results to as the statistic ones. First, across three different traversal cases, vertex-cut performs the worst because of the extra network communication, which dominates the traversal time. DIDO is always the best among all the partitioning algorithms due to the incremental partitioning and also the locality optimization. In addition, the performance benefit of using DIDO increases when the traversal has longer steps. For example, the performance benefit of DIDO comparing to edge-cut increases from 15 percent for 2-step traversal to 19 percent for 8-step traversal. Long-step traversal is common in rich metadata use cases. For example, to validate a result, we need to track back the rich metadata until reaching the original data set, which needs a very long-step traversal and can benefit significantly from the DIDO algorithm.

6.5 GraphMeta versus Graph Databases

A number of distributed graph databases, such as Titan [35] and OrientDB [36], can be used for storing and processing rich metadata graphs. The major challenge is the performance and scalability on large-scale power-law rich metadata graphs. Many existing graph databases require users to manually partition their graphs, a task that is not easy to perform from the client application side. In contrast, GraphMeta leverages server-side information about vertices in order to achieve an efficient partition strategy, potentially providing much better scalability.

In our next evaluation, we used a simple graph insertion workload to show the performance difference of GraphMeta and a representative graph database, Titan (over Cassandra). We chose Titan mainly because of its scalability and performance advantages over existing databases [42]. The evaluation was conducted by running n $(4 \rightarrow 32)$ servers and 256 clients; each issues the same number (10,240) of graph insertions on the same vertex v_0 in order to simulate a strong-scaling experiment. We show the results in Fig. 17. Here, we can easily identify the performance advantage of GraphMeta and its better scalability, features that are critical in HPC environments.

6.6 GraphMeta on POSIX Workloads

GraphMeta is not designed to substitute for the POSIX metadata service, but it still needs to keep a valid copy of

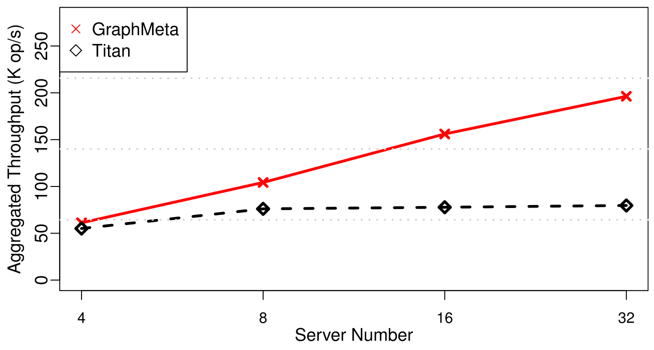


Fig. 17. Graph insertion performance.

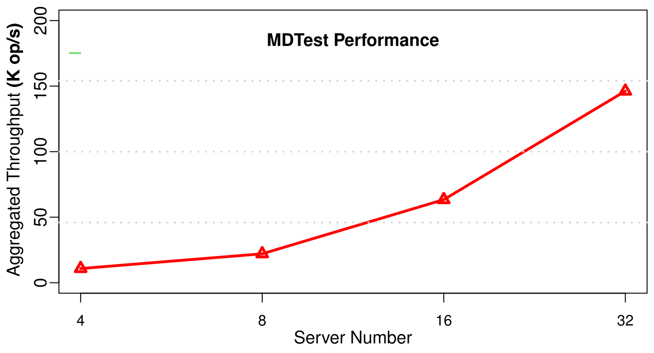


Fig. 18. Aggregated performance on mdtest.

POSIX metadata for many queries. We ported the synthetic *mdtest* benchmark (metadata performance benchmark) with the GraphMeta interface to evaluate the performance of creating a large number of files into a single directory [43]. For n servers, $8 * n$ clients issue file creations concurrently. Each client created the same number (4,000) of files. Fig. 18 plots the aggregated operation throughput, in the number of file creations per second, as a function of the number of servers (from 4 to 32).

The results show the capability of GraphMeta to gracefully absorb POSIX metadata. The installed GPFS running on Fusion is far behind GraphMeta's performance (around 150 K ops/s on 32 servers). We also considered IndexFS [19], which provides state-of-the-art scalability on metadata performance. Because the current IndexFS is not stable on the GPFS file system on Fusion, we could not conduct a direct comparison. But by looking at the original performance numbers from the IndexFS paper [19] under the same workload (i.e., creating empty files), we can easily see that GraphMeta exhibits a performance scalability pattern similar to that of IndexFS. Note that the GraphMeta numbers are generated without optimizations such as client-side caching and bulk operations that IndexFS used. We will evaluate them in future work to further improve the performance.

7 Related Work

A wide range of research efforts are related to this work. We classify them into three categories: scalable metadata management systems, graph systems, and graph-partitioning algorithms.

Scalable metadata management systems have been extensively investigated in HPC environments. Most focus on traditional POSIX metadata, designed and used for parallel file systems [44]. Specific systems such as Spyglass [5] and Magellan [6] are designed for storing and utilizing POSIX metadata, too. IndexFS [19] and its precedent GIGA+ [15] provide high-performance metadata operations. In this research, instead of the limited POSIX metadata, we propose a graph model to support general rich metadata, which are increasingly important in HPC environments. Systems such as QMDS [45] also leverage a similar graph model to support extended file attributes and relationships, but only with limited scalability compared with GraphMeta.

Graph databases are designed for storing graph-based data. Numerous graph databases have been developed in recent years, including Neo4j [46], OrientDB [36], and Titan [35]. Systems such as Neo4j, however, do not provide a scalable design. Because the distributed graph databases require users to manually shard their graphs, performance problems still arise since the servers do not participate in such a partitioning phase. For these reasons, their performance is limited to supporting the desired rich metadata management.

Graph partitioning is a key factor that affects the scalability and performance of graph-based distributed systems. Research efforts on this topic include heuristics strategies such as the METIS family [28], Multilevel algorithm [29], label propagation [47], and SBV-Cut [30], which need a global graph structure to conduct partitioning. There are also heuristics strategies on streaming graphs through one-pass or multipass on the streams, including LDG [31], Fennel [32], and restreaming LDG [33], which need local graph structure information. Moreover, there are hash-based edge-cut and vertex-cut strategies [34] and their variations including grid-based [48]. We propose the DIDO graph-partitioning algorithm in this study, which considers both the I/O balance and locality, achieving good performance on graph-based operations.

8 Conclusion

Motivated by the needs of HPC data management tasks, we investigated the idea of utilizing and unifying HPC rich metadata through a graph-based model based on prior work. After identifying the challenges on the underlying infrastructure for efficiently supporting such graph-based rich metadata management, we designed and built GraphMeta, a graph-based engine for managing rich metadata in HPC systems. It supports different types of metadata management tasks and delivers scalable performance on both metadata ingestion and traversal. The evaluation and comparisons show the performance advantages of GraphMeta and support the possibility of using graph database techniques in an HPC rich metadata management system.

We plan to investigate fault tolerance and recovery capability for both graph persistence and complex graph traversal. In addition, we will explore the implementation of a stronger consistency model or even transaction support.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract

No. DE-AC02-06CH11357; and by the US National Science Foundation under grant CNS-1852815, CNS-1162488, CNS-1338078, IIP-1362134, and CCF-1409946.

REFERENCES

[1] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *Proc. Int. Conf. Database Theory*, 2001, pp. 316–330.

[2] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems," in *Proc. USENIX Annu. Tech. Conf. Gen. Track*, 2006, pp. 43–56.

[3] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-Science," *ACM SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, 2005.

[4] C. T. Silva, J. Freire, and S. P. Callahan, "Provenance for visualizations: Reproducibility and beyond," *Comput. Sci. Eng.*, vol. 9, no. 5, pp. 82–89, 2007.

[5] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proc. USENIX Conf. File Storage Technol.*, 2009, pp. 153–166.

[6] A. Leung, I. Adams, and E. L. Miller, "Magellan: A searchable metadata architecture for large-scale file systems," Univ. California, Santa Cruz, CA, USA, Tech. Rep. UCSC-SSRC-09–07, 2009.

[7] D. Dai, P. Carns, B. R. Ross, J. Jenkins, K. Blauer, and Y. Chen, "GraphTrek: Asynchronous graph traversal for property graph based metadata management," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 284–293.

[8] D. Dai, "GraphMeta prototype," 2015. [Online]. Available: http://discl.cs.ttu.edu/gitlab/dongdai/graphfs

[9] D. Dai, R. B. Ross, P. Carns, D. Kimpe, and Y. Chen, "Using property graphs for rich metadata management in HPC systems," in *Proc. 9th Parallel Data Storage Workshop*, 2014, pp. 7–12.

[10] D. Dai, P. Carns, R. B. Ross, J. Jenkins, N. Muirhead, and Y. Chen, "An asynchronous traversal engine for graph-based rich metadata management," *Parallel Comput.*, vol. 58, pp. 140–156, 2016.

[11] A. S. Tanenbaum and A. Tannenbaum, *Modern Operating Systems*, vol. 2. Englewood Cliffs, NJ, USA: Prentice Hall, 1992.

[12] Property graph. [Online]. Available: www.w3.org/community/propertygraphs/, last accessed 2018.

[13] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of Petascale I/O Workloads," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–10.

[14] FTP site: Darshan data, 2013. [Online]. Available: ftp://ftp.mcs.anl.gov/pub/darshan/data/

[15] S. Patil and G. A. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proc. USENIX Conf. File Storage Technol.*, 2011, pp. 13–13.

[16] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 251–262, 1999.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, 2007, pp. 205–220.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, Art. no. 9.

[19] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al., "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013, Art. no. 8.

[21] B. Goetz, "The Lucene search engine: Powerful, flexible, and free," *JavaWorld*. 2000. [Online]. Available: http://www. javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html

[22] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[23] A. Khetrapal and V. Ganesh, "Hbase and Hypertable for Large scale distributed storage systems," Tech. Rep., Purdue University, 2006.

[24] Rocksdb. [Online]. Available: http://rocksdb.org/, last accessed 2018.

[25] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[26] S. Ghemawat and J. Dean, "LevelDB, A fast and lightweight key/value database library by Google," 2014, Available: https://github.com/google/leveldb, last accessed 2018.

[27] U. I. T. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM J. Sci. Comput.*, vol. 32, no. 2, pp. 656–683, 2010.

[28] G. Karypis and V. Kumar, "Metis—Unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995, Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.376, accessed 2018.

[29] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proc. 20th Int. Parallel Distrib. Process. Symp.*, 2006, pp. 124–124.

[30] M. Kim and K. S. Candan, "SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices," *Data Knowl. Eng.*, vol. 72, pp. 285–303, 2012.

[31] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1222–1230.

[32] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342.

[33] J. Nishimura and J. Ugander, "Restreaming graph partitioning: Simple versatile algorithms for advanced balancing," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 1106–1114.

[34] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, Art. no. 2.

[35] Titan. [Online]. Available: http://thinkaurelius.github.io/titan/, last accessed 2017.

[36] OrientDB. [Online]. Available: http://www.orientechnologies.com/orient-db.htm, last accessed 2018.

[37] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, Art. no. 2.

[38] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 1.

[39] Fusion. [Online]. Available: http://www.lcrc.anl.gov/fusion/, last accessed 2016.

[40] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

[41] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.

[42] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in *Proc. Int. Conf. Social Comput.*, 2013, pp. 708–715.

[43] mdtest: HPC benchmark for metadata performance. [Online]. Available: http://sourceforge. net/projects/mdtest/, last accessed 2018.

[44] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.

[45] S. Ames, M. Gokhale, and C. Maltzahn, "QMDS: A file system metadata management service supporting a graph data model-based query language," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 28, no. 2, pp. 159–183, 2013.

[46] J. Webber, "A programmatic introduction to Neo4j," in *Proc. 3rd Annu. Conf. Syst. Program. Appl.: Softw. Humanity*, 2012, pp. 217–218.

[47] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. 6th ACM Int. Conf. Web Search Data Mining*, 2013, pp. 507–516.

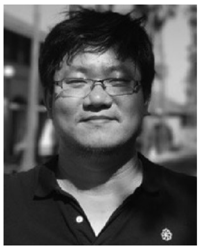
[48] N. Jain, G. Liao, and T. L. Willke, "GraphBuilder: Scalable graph ETL framework," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, Art. no. 4.

Dong Dai is an assistant professor with the Computer Science Department, University of North Carolina at Charlotte. He received his BS and PhD degrees in computer science from the University of Science and Technology of China. His major research interests are building high-performance storage systems, such as parallel file systems, metadata management systems, and graph storage to facilitate data-intensive applications.

Yong Chen is an associate professor and the director of the Data-Intensive Scalable Computing Laboratory, Computer Science Department, Texas Tech University. He is also the site director of the Cloud and Autonomic Computing Site, Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and cloud computing.

Philip Carns received the PhD degree in computer engineering from Clemson University, in 2005. He is a principal software development specialist with the Mathematics and Computer Science Division, Argonne National Laboratory. He is also an adjunct associate professor of electrical and computer engineering with Clemson University and a fellow of the Northwestern-Argonne Institute for Science and Engineering. His research interests include characterization of I/O patterns, simulation of large-scale storage systems, and design of high-performance system software.

John Jenkins received the PhD degree from North Carolina State University, in 2013 and served as a postdoctoral appointee with Argonne National Laboratory through 2016, during which this research was performed, specializing in distributed storage systems and I/O middleware in the context of leadership-class scientific computing facilities. He is currently a software engineer in the finance industry.



Wei Zhang is working toward the PhD degree in the Department of Computer Science, Texas Tech University. His research interests include metadata indexing, graph data management, distributed data management, storage systems, high performance computing. He is now working on a project related to metadata search over scientific datasets.

Robert Ross received the PhD degree in computer engineering from Clemson University, in 2000. He is a senior computer scientist with Argonne National Laboratory and a senior fellow with the Northwestern-Argonne Institute for Science and Engineering. He is the director of the DOE SciDAC RAPIDS Institute for Computer Science and Data. His research interests include system software for high performance computing systems, in particular distributed storage systems and libraries for I/O and message passing. He was a recipient of the 2004 Presidential Early Career Award for scientists and engineers.