GraphMeta: A Graph-Based Engine for Managing Large-Scale HPC Rich Metadata

Dong Dai[1], Yong Chen[1], Philip Carns[2], John Jenkins[2], Wei Zhang[1], and Robert Ross[2]

[1]Computer Science Department, Texas Tech University, USA, {dong.dai, yong.chen, x-spirit.zhang}@ttu.edu
[2]Mathematics and Computer Science Division, Argonne National Laboratory, USA, {carns, jenkins, rross}@mcs.anl.gov

***Abstract*—High-performance computing (HPC) systems face increasingly critical metadata management challenges, especially in the approaching exascale era. These challenges arise not only from exploding metadata volumes but also from increasingly diverse metadata, which contains data provenance and user-defined attributes in addition to traditional POSIX metadata. This "rich" metadata is critical to support many advanced data management functionality such as data auditing and validation. In our prior work, we presented a graph-based model that could be a promising solution to uniformly manage such rich metadata because of its flexibility and generality. At the same time, however, graph-based rich metadata management introduces significant challenges. In this study, we first identify the challenges presented by the underlying infrastructure in supporting scalable, high-performance rich metadata management. To tackle these challenges, we then present GraphMeta, a graph-based engine designed for managing large-scale rich metadata. We also utilize a series of optimizations designed for rich metadata graphs. We evaluate GraphMeta with both synthetic and real HPC metadata workloads and compare it with other approaches. The results show that its advantages in terms of rich metadata management in HPC systems, including better performance and scalability compared with existing solutions.**

I. Introduction

High-performance computing (HPC) systems can generate a large amount of metadata about different entities including jobs, users, and files. Traditional POSIX metadata describes simple and predefined attributes of these entities, such as the file size, name, and permissions, and is well captured in current file systems and widely used. Rich metadata can describe detailed information about different entities and their relationships and extend simple metadata to an in-depth level. A well-known case of such rich metadata is provenance, which describes the relationships among entities such as data sources, processing steps, processes, context, and dependencies that contribute to the existence of a data item [11]. This detailed level of rich metadata enables a variety of data management capabilities, such as data auditing, validation, and reproducibility support, which are increasingly important in HPC environments [41, 20]. For example, the file access history of users can be used to audit users' activities in shared supercomputer facilities; the file access history of processes can be used to identify executions and configurations; and captured detailed provenance can be used to regenerate an environment for reproducing scientific results. Rich metadata support is still largely missing in HPC systems, however, primarily because of the lack of a model to abstract them uniformly and an underlying infrastructure to manage them highly efficiently.

We are developing a rich metadata solution for HPC systems based on the new concept of unifying metadata into one generic property graph (initial research findings have been published in [22] [19] and [18]). The motivation for using a graph-based model [5] is twofold. First, the graph is a natural way to describe heterogeneous metadata entities and their relationships. Unlike relational databases, graphs allow flexible and dynamic schema, making them more extensible for diverse rich metadata. Second, operations such as identifying the relevant input of a result need traversal operations, which have been shown to be inefficient in relational databases [45]. Other schema-free storage solutions, such as key-value stores [39, 21] and NoSQL databases [33, 31], lack the semantics to natively express these types of operations.

Although a graph-based rich metadata model is promising, efficiently managing the metadata graphs in an HPC environment is challenging for two major reasons. First, rich metadata contains dynamic runtime information including jobs, processes, users, environment variables, parameters, and configuration files. This diverse and dynamic information can lead to a huge volume of metadata in a short time, and it must be stored and ingested gracefully. Second, after storing the metadata, highly efficient methods are needed to access them. Data management based on rich metadata leads to access patterns such as locating a single entity or relationship, iterating over relationships, and providing conditional traversal across multiple relationships. These new types of metadata accesses are critical but challenging to efficiently support.

To tackle these challenges, we introduce *GraphMeta* in this study, *a distributed graph-based rich metadata management engine* for HPC systems. A prototype system has been provided in [2]. We note that 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 capabilities such as data auditing, validation, and

reproducibility.

The main contributions of this work are fourfold.

- We introduce a distributed graph-based rich metadata engine to uniformly manage metadata graph accesses.
- We design and implement an advanced metadata graph storage solution together with a write-optimal storage engine to provide both high ingestion speed and high traversal speed of rich metadata.
- We design an innovative graph-partitioning algorithm, namely, DIDO (destination-dependent optimized), to efficiently distribute large-scale metadata graph across storage clusters with better load balance and data locality in an online manner.
- We conduct comprehensive evaluations of GraphMeta to validate the design goals of GraphMeta and to compare with other possible solutions.

The rest of this paper is organized as follows. Section II presents our motivation for using a graph model to manage rich metadata and summarizes the challenges in an HPC environment. Section III presents the GraphMeta system and solution. Section III-A discusses the GraphMeta data model. Sections III-B and III-C describe the graph data storage engine and the DIDO partitioning algorithm, respectively. Section IV reports the evaluation results including both microbenchmark and end-to-end experiments. Section V discusses related work, and Section VI briefly summarizes our conclusions and discusses avenues for future work.

II. Motivation and Challenges

In previous work we have explored a graph-based model for rich metadata management and use cases supported by such a model. A brief summary is given in this section to provide necessary context for GraphMeta; please refer to our previous paper for a more complete treatment [22].

A. Motivation

Rich metadata in HPC systems can be naturally mapped into a property graph model [5], which enables vertices and edges to have arbitrary sets of associated properties, allowing flexible data schema. An example of representing HPC entities as property graph entities 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 normally have attributes to denote their types, such as *file* vertex or *job* vertex. Relationships between any two vertices are defined as directed edges, which have types indicating different relationships such as directory/file hierarchy, file/user ownership, user/job runs, and job read/write files. Additional attributes of both entities and relationships 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 a graph model, we can support a wide range of advanced data management tasks through graph operations.

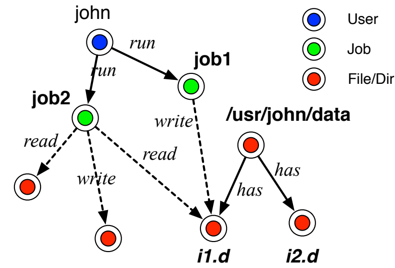

Fig. 1: Exemplar metadata graph in HPC systems.

We take *scientific result validation* as an example. In order to validate a scientific result, a generic way is to execute the complete workflow or parallel job again under the same environment with the same parameters, input datasets, and configurations. With a rich metadata graph, we can track back through edges from the validating result vertex to the original dataset's vertices and validate all elements (executable files, processing steps, parameters, environment variables, and configuration files of each execution) that have contributed to the existence of such a result. Because all the rich metadata is kept in the graph, tracking back executions becomes as simple as graph traversal.

B. Challenges

Although using a graph-based rich metadata model is straightforward, the challenges of building such a graph-based metadata management solution are significant. We have examined the metadata graphs generated from a year's worth of publicly available Darshan logs [12, 13] on the Argonne Intrepid Blue Gene/P supercomputer [8]. We are able to draw the following observations that inform the challenges.

Large graph size. The rich metadata graphs can have hundreds of millions of vertices and edges [22], which are expected to increase both with the HPC system scale and with more complex use cases. Hence, the ability to support large-size graphs is a primary concern.

Large mutation rate. Rich metadata is 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.

Scan-based access pattern. Rich metadata use cases, from *data audit* to *result validation*, are implemented in a common pattern: start from a set of vertices and travel through relationships to reach more vertices. A single iteration is called *scan/scatter*, and multiple iterations are *graph traversal*. Since the graph is normally large and distributed, such access patterns must be serviced efficiently.

Power-law edge distribution. Similar to POSIX file/directory distributions in HPC systems [38], rich metadata graphs follow the power-law distribution [24] on the vertex degrees for various entities, as our previous work observed [22]. This makes graph partitioning critical because it significantly affects the performance on both graph mutation and queries.

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

III. GraphMeta Design and Implementation

Fig. 2 shows the overall architecture of GraphMeta, which contains two major components. The client-side component includes graph APIs and wrappers for efficiently managing specific types of rich metadata such as provenance. It is normally linked with applications that run on the compute or login nodes of an HPC cluster to manage rich metadata. GraphMeta also provides an interactive shell for users to easily manipulate and view the rich metadata.

The server-side component runs on a subset of I/O nodes or compute nodes or even 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 utilizing the other two components. In order to allow the dynamic growth (or shrink) of the GraphMeta backend cluster based on metadata workloads, a consistent hashing mechanism is adopted to manage the backend servers in a similar manner to Dynamo [23]: 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* [27].

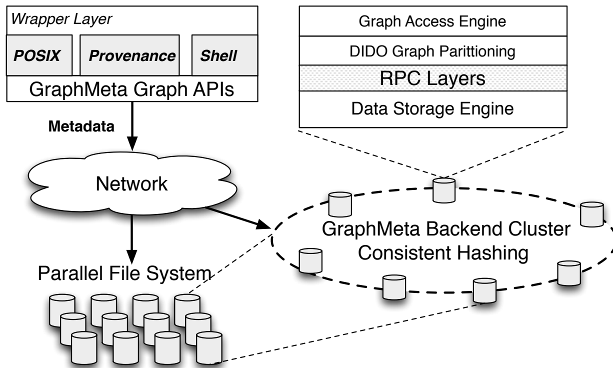

Fig. 2: Overall architecture of GraphMeta.

GraphMeta is designed as a service on top of an existing HPC software stack. Each GraphMeta instance stores its data into a parallel file system, which enables GraphMeta to run on diskless compute nodes as a well-adopted strategy [40]. More important, it simplifies the fault tolerance design by leveraging that of parallel file systems (e.g., through RAID devices).

A. GraphMeta Data Model

GraphMeta adopts a rich metadata-oriented data model. It allows users to define vertex and edge types before using them. A vertex type contains a name and its mandatory attributes. For example, a user vertex can include user id, name, group info, and permissions. An edge type can be defined by its name and its source and destination vertices types. These types are used to differentiate entities, locate entities quickly, constrain graph operations, and prevent certain types of corruption (e.g., invalid edges between vertices).

An important need of rich metadata is to keep full history. For example, a user may run the same application multiple times, indicating the creation of multiple edges between the same two vertices in the graph. In GraphMeta, all these edges are kept, in order to satisfy queries about past runs. In addition, GraphMeta records rich metadata about an entity even if it is removed. This feature is advantageous, for example, if a user wants to retrieve details about a deleted file or query about other data files that are generated from this deleted file. In the GraphMeta data model, we introduce a version concept to the generic property graph, which is implicitly attached on all vertices, edges, and their attributes, to maintain and distinguish metadata. The version also indicates that all modifications (e.g., deletion) in GraphMeta are converted to creation of corresponding entities with new versions.

GraphMeta uses server-side timestamps as the version number. Versions can be used to order concurrent read/write accesses on the metadata. In these cases, the timestamps establish a deterministic order of accesses. It guarantees that a scan operation will not retrieve edges that are inserted after it is issued. For concurrent writes on the same entity, 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.

Timestamps are typically well synchronized in HPC environments. For example, Google's TrueTime relies on timestamps to keep its data centers synchronized and to support transactions [17]. However, a small amount of clock skew is inevitable. Taking the clock skew into consideration, GraphMeta is not designed 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. This relaxed consistency model is acceptable because, unlike POSIX file system metadata, many rich metadata use cases explicitly decouple data collection from subsequent analysis. A few milliseconds delay when reading the latest writes is acceptable in most cases, such as when tracing back suspicious executions from broken files, or auditing user 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 with GraphMeta.

GraphMeta provides APIs to operate on metadata graphs, including vertex/edge ("one-off") access, scan/scatter vertex, and multistep traversal. Because of the page limit, the details are omitted in this paper but can be found in our previous work [18]. 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 to rebuild an identical previous execution environment.

B. GraphMeta Data Storage Engine

GraphMeta has two main goals: high-speed rich metadata ingestion and highly efficient processing of scan and traversal requests. To achieve these goals, GraphMeta is designed with a two-layer data layouts—a logical layout and a physical layout—with a write-optimal storage engine, as shown in Fig. 3.

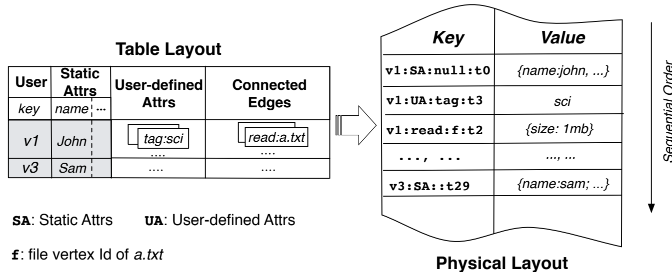

Fig. 3: Data layout of rich metadata graph.

The left part of Fig. 3 shows the logical layout of graph entities in 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 include permissions and `stat` entries for files, or executable names for jobs. User-defined attributes include annotations and format descriptors for data 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 [33] and HBase [31]) but differs from relational databases because it contains "fat columns" that can support more flexible data structures.

The right part of Fig. 3 shows the physical layout in the storage system. In the current implementation, we utilize RocksDB [6], a write-optimized database using a log-structured merge-tree (LSM) system [37]. Unlike other similar LSM-based storage systems such as LevelDB [25], RocksDB stores key-value pairs in SSTable files in lexicographical order. GraphMeta leverages this feature to maximize sequential access while generating the physical layout.

All data related with a vertex, including its static attributes, user-defined attributes, and connected edges, is stored together using the same vertex Id as key prefix. For each vertex, the 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. Therefore, once the vertex is located, static attribute retrieval can be performed. The data is possibly already in memory as a result of the prefetching mechanism of the storage system. 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, namely, "[*vertex-id, edge-type, dest-id, timestamps*]." Making all edges sort by *edge-type* is important because it aids both scan and traversal queries that mostly access specific types of edges. Here, *dest-id* refers to the destination vertex Id. Multiple edges may exist between the same two vertices and can be distinguished by timestamps. The value of these keys is a map of edge properties.

C. GraphMeta Partition: DIDO Algorithm

The large size, large mutation rate, and unbalanced distribution of the metadata graphs necessitate partitioning the graph across multiple servers. The graph partitioner is a key component of GraphMeta and differentiates it from other solutions.

Graph partitioning already has been well studied. The formulation most relevant to GraphMeta is called k-partitioning. The goal is as follows: Given a graph G and a number k as inputs, cut G into k balanced pieces while minimizing the number of edges cut. Although the problem is NP-complete [14], many heuristics exist that solve this problem with practical and effective solutions, such as METIS [30], mMultilevel algorithms [9], and SBV-Cut [32]. However, those solutions all rely on the global information of the graph (e.g., the connectivity), which is not feasible for GraphMeta since the vertices and edges have to be continuously inserted and partitioned in an online manner. A number of research studies also have focused on graph partitioning on streaming graphs, such as LDG [42], Fennel [43], and "restreaming" LDG [36]. However, these methods need at least local information about the graph, for example, knowing all connected edges when inserting a vertex, information that is 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 in order 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, GraphMeta cannot leverage those algorithms.

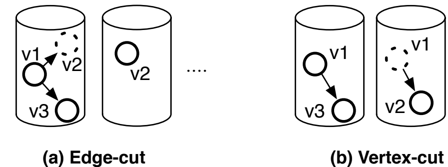

Fig. 4: Edge-cut and vertex-cut graph-partitioning methods. Solid circles/lines represent vertex/edge storage locations.

Graph databases, which play a role similar to that of GraphMeta, usually take hash-based graph-partitioning strategies and do not require extra information about graphs. Typical strategies include *edge-cut* and *vertex-cut* [26], as shown in Fig. 4. Edge-cuts distribute vertices together with their outgoing edges

by hashing the vertex Id. Many distributed graph databases, including Titan [7] and OrientDB [4], use this strategy as the default partitioner. However, edge-cuts are impractical for GraphMeta because vertices in metadata graphs can have millions of outgoing edges, causing significant load imbalance. Vertex-cuts distribute edges instead of vertices by the hashing edge Id, as Fig. 4(b) shows. This strategy is used not only in graph databases but also in graph-processing frameworks such as GraphX [48] and PowerGraph [26]. Vertex-cuts achieve balance for high-degree vertices but perform 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 partitioners, thus motivating the new graph-partitioning algorithm introduced in this research.

1) Rich Metadata Graph Partitioning Analysis: Graph partitioning influences the cost of metadata operations. Among them, point accesses on single vertex/edge are less affected by graph partitioning since as they normally are stored in a single server (unbalanced partitioning still may make the server itself overloaded and slow). But, scan/scatter vertices and graph traversal will perform significantly different because of the partitioning strategies. We take scan operations as an example to analyze because traversal is essentially a multistep scan. The total cost of a scan can be defined as follows.

$$T_{scan(v_1)} = T_{v_1} + T_{e_i|e_i \in out_e(v_1)} + T_{dst(e_i)} \quad (1)$$

Here, T_{v_1} means the cost of reading v_1, and $T_{e_i|e_i \in out_e(v_1)}$ indicates the cost of iterating all out-edges of v_1. Since $|e_i|$ can be huge in a metadata graph, partitioning them across multiple servers can improve the performance by increasing the parallelism. At the same time, however, increasing parallelism may introduce extra network communication. The last field, $T_{dst(e_i)}$, includes the cost of reading all destination vertices of edges. If e_i and $dst(e_i)$ are partitioned into different servers, extra network communication is needed in order to transmit the requests. Therefore, the locality is critical in partitioning metadata graphs.

Based on this analysis, we observe that high-degree parallelism is desired for high-degree vertices; but the locality between source vertices and edges and edges and destination vertices is also critical for delivering the best performance.

2) DIDO Graph Partitioning: We propose a DIDO partitioning algorithm to orchestrate parallelism and locality. The core idea of DIDO contains two aspects: first, for better parallelism, DIDO incrementally partitions vertices based on their out-degrees; second, for better locality, DIDO considers edge placement with the location of respective destination vertices during partitioning.

Initially, DIDO begins by placing a 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 the vertex Ids, which enables single-hop vertex lookups to guarantee a fast point access on the vertex. Note that the hash function actually 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.

New edges are inserted into the same server as its source vertices. When more edges are inserted that cause the out-degree of a vertex to surpass a configurable threshold (*split threshold*), DIDO partitions all out-edges of that vertex into two partitions: one stays on the original server, and one moves to another server. In this way, the vertices are incrementally partitioned based on the number of out-edges. This incremental strategy differentiates vertices based on their out-edge degrees and achieves a better balance for $T_{e_i|e_i \in out_e(v_1)}$ in Equation 1. However, it does not account for locality with respect to reading destination vertices ($T_{dst(e_i)}$).

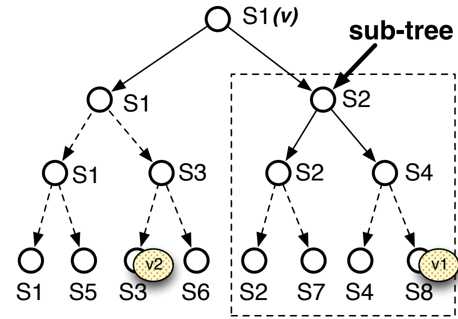

Fig. 5: DIDO's partition tree structure for vertex v_1.

To maximize locality, we propose a significant improvement over the naive incremental partitioning approach. The core data structure, called *partition tree*, is shown in Fig. 5. It is built by using the following strategy. First, the root is set to S_v, the server that stores the source vertex v. 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 used in the tree yet. We choose it in a round-robin way. It can be calculated as $S_l + 1 \bmod k$, where S_l indicates the last extended server Id and k is the total number of servers (virtual nodes), which is a configurable constant given by the user. Repeating this, all k servers are assigned into such a tree with a maximum of $log(k) + 1$ levels. This tree organization is fixed for each v and is easily calculable before any splitting happens.

During partitioning, after splitting all out-edges of a vertex into two partitions: one stays in the original server, and another one is placed to an extended server by DIDO. Which server to choose and which part of the out-edges should be moved are decided based on this tree.

Extended servers are always selected as the right child of the current server. For example, as Fig. 5 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.

To decide which parts of the out-edges need be moved, DIDO does not naively split all edges into two equal parts. Instead, it calculates the locations of the destination vertices of each edge and always puts edge into the child that leads the

path to where the destination vertex is stored. For example, in Fig. 5, 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. Edge $e_2(v \rightarrow v_2)$ will be kept in S_1 because v_2 is stored in S_3 in the left subtree. In this way, after several rounds of splitting, any partitioned edge either has been colocated with its destination vertex or will be colocated upon further partitioning. This strong locality significantly improves scan/scatter and traversal performance, as evaluations show.

Comparison and Discussion. The idea of using an incremental strategy to partition power-law distributed entities was investigated in [16, 38]. GIGA+ [38] is one example of its use in file system metadata management. DIDO significantly improves them, however, through its new partition tree-based placement strategy, which guarantees edge/vertex locality to largely improve scan and traversal performance.

D. GraphMeta Graph Access Engine

We implemented the level-synchronous breadth-first traversal engine as the default traversal engine in GraphMeta. Implementation details can be found in our previous research [18]. The main difference is that in GraphMeta we elected to leverage synchronous traversal for two reasons: (1) the DIDO partitioning algorithm generates a more balanced graph distribution, which is less likely to be affected by stragglers; and (2) progress tracking and fault tolerance of sync traversal are much simpler and more efficient to implement.

IV. Evaluation

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

The evaluations were conducted on the Fusion cluster [1] at Argonne National Laboratory. 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 memory and 250 GB 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.

A. Evaluation Datasets

We used two datasets in the evaluations. The first one is a Darshan log generated from a whole year's trace (2013) from the Intrepid supercomputer [8]. 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 30K connected edges and most of the vertices have less than 10 connected edges.

The second dataset is a synthetic graph, which was generated by the RMAT graph generator [15] using a "recursive matrix" model as a real-world power-law graph. The vertices and edges in the synthetic graph contain randomly generated attributes (the attribute size is 128 bytes). We used the following parameters for all generated RMAT graphs with moderate out-degree skewness: $a = 0.45, b = 0.15, c = 0.15, d = 0.25$.

B. DIDO Parameters: Split Threshold

Incremental graph-partitioning algorithms such as DIDO contain a critical parameter, the *split threshold*, to determine when a vertex should start to split. 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 16K to 512K physical storage with a 128-byte vertex size.

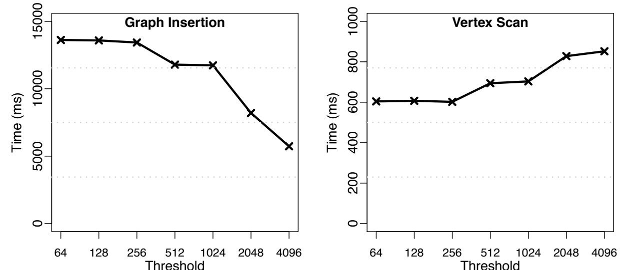

Fig. 6: Insert and scan performance vs. split threshold.

The results are shown in Fig. 6, 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 one single server. An optimal threshold is difficult to determine because of the influence from many factors including disk speed, network bandwidth, and the number of servers. We recommend that users set the best parameters according to their use cases and the cluster. In this series of tests, we chose 128 as the default threshold. The reason for choosing 128 was twofold. 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.

C. Evaluating Graph Partitioning

In this subsection, we compare four graph-partitioning strategies—edge-cut, vertex-cut, GIGA+, and DIDO—for different graph operations. The GIGA+ implementation was imported from the IndexFS project [40], and we mapped directories and files as vertices. For vertex-cut, we used the combination of source vertex Id and destination vertex Ids as the edge Id to distribute the edges. For edge-cut, we used the source vertex Id to assign vertices.

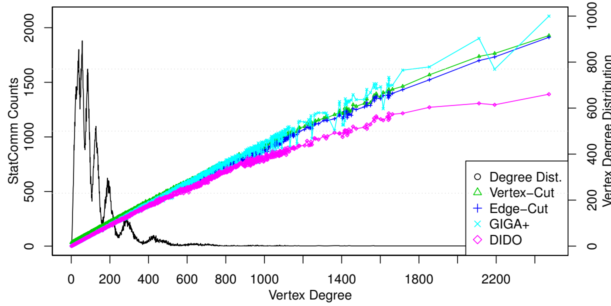

Fig. 7: *StatComm* of scan/scatter.

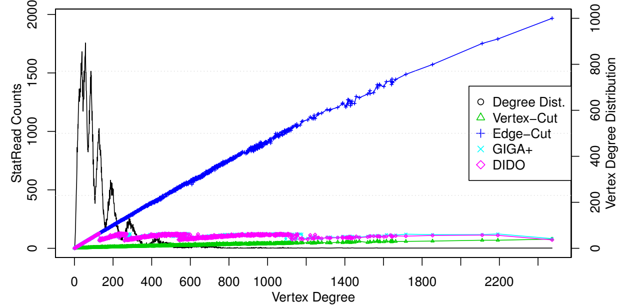

Fig. 8: *StatReads* of scan/scatter.

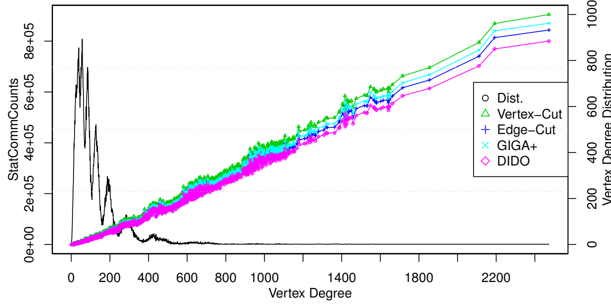

Fig. 9: *StatComm* of 2-step traversal.

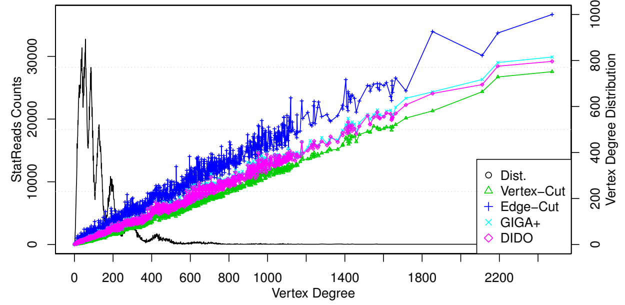

Fig. 10: *StatReads* of 2-step traversal.

1) Metadata Ingestion Performance: In this test, we first show the performance of metadata insertion using different graph-partitioning algorithms based on the real-world Darshan logs as evaluation datasets. We had n servers and $8*n$ clients perform this evaluation, where $n = 4 \rightarrow 32$. Each client loaded part of Darshan logs and issued graph insertions in parallel.

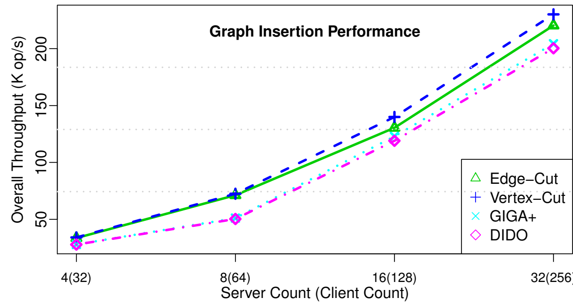

Fig. 11: Insertion performance with different graph partition strategies.

The insertion performance of different partitioning strategies is reported in Fig. 11. The insertion performance of four partitioning algorithms scale well when more servers are used: for a 32-node setting, we were able to achieve around 200K ops/s graph insertions. Vertex-cut achieves the best insertion performance because it distributes all edges across the cluster without generating a performance bottleneck. But as we have discussed, this simple strategy faces serious problem for scanning or traversal on low-degree vertices. Edge-cut has worse performance mainly because of the existence of high-degree vertices. Since the total number of high-degree vertices is small, however, the overall performance degradation is not too large. GIGA+ and DIDO both have a little worse performance than does the vertex-cut, because of the extra incremental splitting phases for high-degree vertices. The small performance difference between DIDO and GIGA+ is due mostly to the extra computation of edge placement while splitting.

2) Graph Scan/Scatter and Traversal Performance: To effectively compare different partitioning algorithms with the scan/scatter and traversal workload, we conducted both statistical and real performance evaluations.

For statistical comparison, we devised two metrics: **StatComm** and **StatReads**. 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. 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 all StatReads generated from each step together reflects the total cost of I/O operations.

In this test, we created an RMAT graph with 100k vertices and 12.8 million edges to simulate the metadata graph generated from typical HPC system like Intrepid [8]. We applied four graph partition algorithms on them in both scan and 2-step 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. Figures 7 to 10 show the results. Each figure shows the performance of one metric for a traversal case. The x-axis

on these figures 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, edge-cut, GIGA+, and DIDO, respectively. For all of them, smaller is better. From these figures, we can see that both StatComm and StatReads increase with the vertex degree, which is expected because high-degree vertices touch more vertices while traveling.

From the first column, we compare StatComm among the four partitioning algorithms and two requests. We can see that DIDO exhibits the least cross-server communication in all cases, especially compared with the native incremental partitioner GIGA+. This is due mostly to the tree-based edge placement optimization in DIDO. On the right column, we compare StatReads. Overall, vertex-cut obtains the best I/O balance since it fully utilizes all storage servers. But, DIDO and GIGA+ still are able to keep a very small difference as they incrementally partition the large vertices. Among both test cases, edge-cut is significantly worse because of the imbalanced graphs.

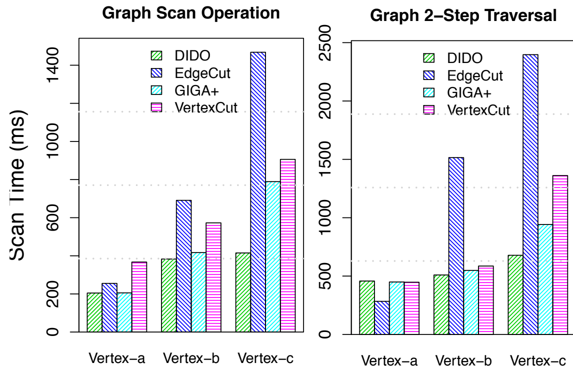

Fig. 12: Scan and 2-step traversal performance on sampled vertices.

To confirm the statistical results, we further evaluate the real performance of different partition algorithms based on the Darshan graph. To save space, we sample only three different vertices to run scan and 2-step graph traversal. These three vertices are manually selected based on their degrees: $vertex_a$ with only 1 edges connected, $vertex_b$ with a medium number (572) of degrees, and $vertex_c$ with around 10K connected edges. Similarly, we conducted evaluations on 32 servers. We report the results in Fig. 12. There are three groups of results for each case, corresponding to $vertex_a$, $vertex_b$, and $vertex_c$, respectively. The figure shows results similar to the statistic ones. For low-degree vertices such as $vertex_a$, vertex-cut performs the worst on both the scan and traversal operations because of the extra network communications. GIGA+ and DIDO are as good as edge-cut for the scan operation because they do not split the vertex; but they perform worse for 2-step traversal, mainly because of the extra cost from splitting $vertex_a$'s neighbor. For medium-degree and high-degree vertices, the edge-cut always performs the worst for both scan and multistep traversal. The reason is mainly the imbalanced disk accesses as Figures 8 and 10 show. For most test cases, we can observe a performance benefit with DIDO compared with other algorithms, especially for the high-degree vertex. This mainly comes from the better data locality achieved in the DIDO design.

We have showed the performance benefit of DIDO compared with the native incremental graph-partitioning algorithm. Those results are generated for scan and 2-step traversal, however, which have only a small number of steps. In fact, the performance advantage of DIDO can be further confirmed for a longer step traversal due to better locality between edges and their destination vertices.

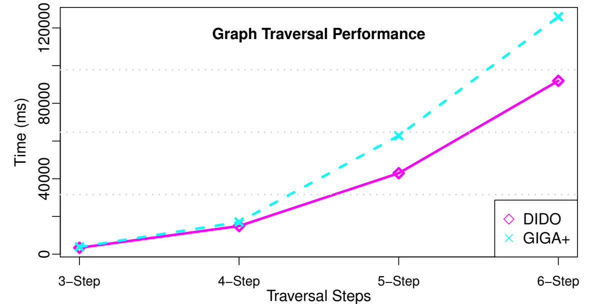

Fig. 13: Deep traversal performance on sampled vertices.

In Fig. 13, we show the traversal performance of GIGA+ and DIDO from $vertex_c$ in the same Darshan graph used in the previous evaluations. From the figure we can clearly observe that as the step gets longer, the performance difference significantly increases. 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. The process involves a very long step traversal and can benefit significantly from the DIDO algorithm.

D. GraphMeta vs. Graph Databases

A number of distributed graph databases, such as Titan [7] and OrientDB [4], can be used for storing and processing rich metadata graphs. The major disadvantage for rich metadata management is their limited scalability on large-scale power-law graphs. Existing graph databases require users to manually partition their graphs, not an easy task to perform from the client application side. In contrast, GraphMeta leverages server-side information about vertices to achieve an efficient partition strategy and much better scalability. In this 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 among existing databases [29]. 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 to simulate a strong-scaling experiment. We show the results in Fig. 14. Here, we can easily identify the performance

advantage of GraphMeta and its better scalability, which are critical for HPC environment.

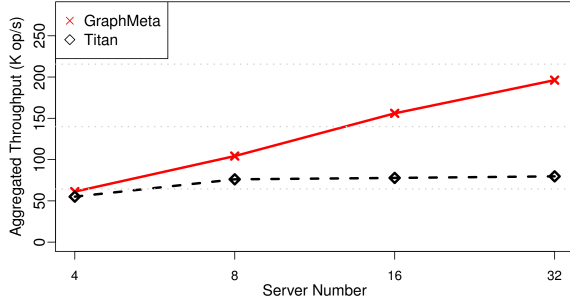

Fig. 14: Graph insertion performance.

E. GraphMeta on POSIX Workloads

GraphMeta is not designed to substitute for the POSIX metadata service. It still needs to keep a valid copy of POSIX metadata for many queries. In this evaluation, we ported the synthetic *mdtest* benchmark [3] with the GraphMeta interface to evaluate the performance of creating large number of files into a single directory. For n servers, $8 * n$ clients issued file creations concurrently. Each client created the same number (4,000) of files. Fig. 15 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).

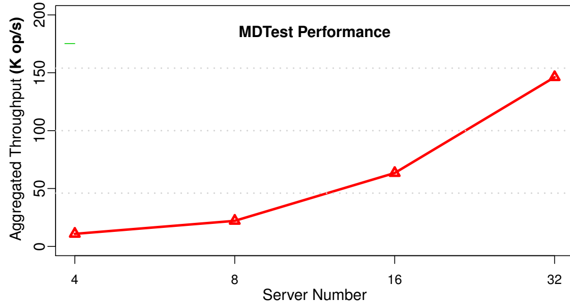

Fig. 15: Aggregated performance on mdtest.

These results show the capability of GraphMeta to gracefully absorb POSIX metadata. The GPFS running on Fusion is far behind GraphMeta performance (around 150 K ops/s on 32 servers). We also considered IndexFS [40], 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. However, by looking at the original performance numbers from the IndexFS paper [40] under the same workload (i.e., creating empty files), we can easily identify 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 these optimizations in future work to further improve the performance.

V. Related Work

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

Scalable metadata management systems have been extensively investigated in HPC environments. Most of them focus on traditional POSIX metadata, designed and used for parallel file systems [47, 49]. Specific systems such as Spyglass [35] and Magellan [34] are designed for storing and utilizing POSIX metadata, too. IndexFS [40] and GIGA+ [38] provided high-performance metadata operations. In this research, instead of the limited POSIX metadata, we propose a graph model to support general rich metadatam which is increasingly important in HPC environments. Systems such as QMDS [10] 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 [4], and Titan [7]. Systems such as Neo4j do not provide a scalable design. Distributed graph databases require users to manually share their graphs, leading to performance problems because the servers do not participate in the partition phase. Therefor, their performance in supporting the desired rich metadata management is limited.

Graph partitioning is a key factor that affects the scalability and performance of graph-based systems. Research efforts on this topic include heuristics strategies such as the METIS family [30], multilevel algorithms [9], label propagation [44], and SBV-Cut [32], which need global graph structure to conduct partitioning. There are also heuristics strategies on streaming graphs through one-pass or multipass on the streams; examples include LDG [42], Fennel [43], and “restreaming” LDG [36], which need local graph structure information. Researchers have also developed hash-based edge-cut and vertex-cut strategies [26] and their variations including grid-based [28]. We propose the DIDO graph-partitioning algorithm in this study, which considers both the I/O balance and the locality, achieving good performance on graph-based operations.

VI. Conclusion

Motivated by the needs of HPC rich metadata management, we investigated the idea of unifying rich metadata into a graph-based model based on prior work. After identifying the challenges of 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. GraphMeta 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 indicate 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, perhaps, transaction support.

ACKNOWLEDGMENT

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 National Science Foundation under grant CCF-1409946 and CNS-1338078.

REFERENCES

[1] Fusion. In *http://www.lcrc.anl.gov/fusion/*.
[2] GraphMeta Repo. http://discl.cs.ttu.edu/gitlab/dongdai/graphfs.
[3] mdtest. http://sourceforge.net/projects/mdtest/.
[4] Orientdb. http://www.orientechnologies.com/orient-db.htm.
[5] Property graph. www.w3.org/community/propertygraphs/.
[6] Rocksdb. http://rocksdb.org/.
[7] Titan. http://thinkaurelius.github.io/titan/.
[8] FTP Site: Darshan Data. ftp://ftp.mcs.anl.gov/pub/darshan/data/, 2013.
[9] A. Abou-Rjeili and G. Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.
[10] S. Ames, M. Gokhale, and C. Maltzahn. QMDS: A File System Metadata Management Service Supporting A Graph Data Model-Based Query Language. *International Journal of Parallel, Emergent and Distributed Systems*, 28(2):159–183, 2013.
[11] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and Where: A Characterization of Data Provenance. In *Database Theory ICDT 2001*, pages 316–330. Springer, 2001.
[12] 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)*, 7(3):8, 2011.
[13] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *IEEE International Conference on Cluster Computing and Workshops, 2009 (CLUSTER'09)*, pages 1–10. IEEE, 2009.
[14] Ü. i. t. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal on Scientific Computing*, 32(2):656–683, 2010.
[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 200 SIAM International Conference on Data Mining*, volume 4, pages 442–446. SIAM, 2004.
[16] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.
[17] 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 Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.
[18] 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 *IEEE International Conference on Cluster Computing, IEEE CLUSTER*. IEEE, 2015.
[19] 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 Computing*, 2016.
[20] D. Dai, Y. Chen, D. Kimpe, and R. Ross. Provenance-based Object Storage Prediction Scheme for Scientific Big Data Applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, 2014.
[21] D. Dai, X. Li, C. Wang, M. Sun, and X. Zhou. Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud. In *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pages 48–56. IEEE, 2012.
[22] D. Dai, R. B. Ross, P. Carns, D. Kimpe, and Y. Chen. Using Property Graphs for Rich Metadata Management in HPC Systems. In *9th Parallel Data Storage Workshop (PDSW), 2014*, pages 7–12. IEEE, 2014.
[23] 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 *SOSP07, Proceedings of the 21st ACM SIGOPS Symposium n Operatin Systems Principles*, volume 41, pages 205–220, 2007.
[24] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.
[25] S. Ghemawat and J. Dean. Leveldb, 2014.
[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.
[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-Scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.
[28] N. Jain, G. Liao, and T. L. Willke. GraphGuilder: Scalable Graph ETL Framework. In *First International Workshop on Graph Data Management Experiences and Systems*, page 4. ACM, 2013.
[29] S. Jouili and V. Vansteenberghe. An Empirical Comparison of Graph Databases. In *Social Computing (SocialCom), 2013 International Conference on*, pages 708–715. IEEE, 2013.
[30] G. Karypis and V. Kumar. Metis-Unstructured Graph Partitioning and Sparse Matrix Ordering System, version 2.0. 1995.
[31] A. Khetrapal and V. Ganesh. HBase and Hypertable for Large-Scale Distributed Storage Systems. *Purdue University*, 2006.
[32] M. Kim and K. S. Candan. SBV-Cut: Vertex-Cut Based Graph Partitioning Using Structural Balance Vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.
[33] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating System Review*, 2010.
[34] A. Leung, I. Adams, and E. L. Miller. Magellan: A Searchable Metadata Architecture for Large-Scale File Systems. *University of California, Santa Cruz, Tech. Rep. UCSC-SSRC-09-07*, 2009.
[35] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *FAST*, volume 9, pages 153–166, 2009.
[36] J. Nishimura and J. Ugander. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pages 1106–1114. ACM, 2013.
[37] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
[38] S. Patil and G. A. Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *FAST*, 2011.
[39] Redis. http://redis.io/.
[40] K. Ren, Q. Zheng, S. Patil, and G. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC14*, pages 237–248. IEEE, 2014.
[41] Y. L. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in E-Science. *ACM Sigmod Record*, 34(3):31–36, 2005.
[42] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM international conference on Knowledge Discovery and Data Mining*. ACM, 2012.
[43] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM international conference on Web Search and Data Mining*, pages 333–342. ACM, 2014.
[44] J. Ugander and L. Backstrom. Balanced Label Propagation for Partitioning Massive Graphs. In *Proceedings of the sixth ACM international conference on Web Search and Data Mining*. ACM, 2013.
[45] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th annual Southeast Regional Conference*, page 42. ACM, 2010.
[46] J. Webber. A Programmatic Introduction to Neo4j. In *Proceedings of the 3rd annual conference on Systems, Programming, and Applications: Software for Humanity*, pages 217–218. ACM, 2012.
[47] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 307–320. USENIX Association, 2006.
[48] 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*, page 2. ACM, 2013.
[49] J. Zhou, Y. Chen, W. Wang, and D. Meng. Mams: A highly reliable policy for metadata service. In *the 44th International Conference on Parallel Processing (ICPP'15)*, 2015.