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

immediate

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

I. Introduction

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

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

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

- *Storage System Pressure*. Considering a leadership supercomputer, there might be millions of processes running on millions of cores accessing billions of files per second. In this case, recording the rich metadata, like the detailed access history of each process, will place a huge pressure on the storage system. In addition, as storing rich metadata must not affect the application execution speed significantly, the resources (both network and disks bandwidth) dedicated to storing metadata must be limited in most cases.
- *Efficient Processing*. Even if we can collect and store these rich metadata, it is still a big challenge to process them. First, as rich metadata are large and can not be held in one server, a distributed processing framework is necessary in most cases. Second, many use cases require complex analysis rather than simple searching or loading, so flexible processing should be provided for them.
- *Metadata Integration*. As we have described, the rich metadata could be as diverse as the users need. They can contain predefined attributes and relationships of entities, or be extended to any user-defined attributes and relationships. Traditionally, we use different tools (system components or users applications) to store and process different subsets of metadata based on the specific usages. However, this introduces lots of unnecessary redundant metadata storage in different tools or leads to massive inefficient cross-reference operations between applications. For example, the data audit application and data verification application both need to know the file access history. We either need to store this metadata in both two applications, or only store it in one application and issue lots of cross-reference reads from other application later.

In this paper, we proposed unifying all metadata into one property graph that integrates rich metadata from different sources together by providing an unified graph abstraction for all the entities and relationships. Applications can store their

rich metadata using graph storage APIs, and access different categories of metadata using graph query APIs. The benefits are twofold: first, we directly solve the integration issue by using a single representation. All applications will have the same interface to store or process metadata in a single service, where we can apply complex optimizations to improve the performance further. Second, by abstracting metadata into a graph, we are able to utilize rapidly evolving graph techniques to provide better access speed, flexible query languages, and also a high-performance graph-based distributed framework.

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

II. Graph-based Metadata Model

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

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

A. Entity To Vertex

In an HPC platform, there are different entities (e.g. users, processes, and data files) that play different roles. Moreover, users also can define other logical entities, like *user groups* or *work-flow* as they need. So, we define three basic entities, and allow users to extend them as user-defined entities.

- *Data Object*: It represents the smallest data unit in storage systems. Each file in PFS(Parallel File System) indicates one data object. Moreover, the directory is also a data object, which contains multiple other data objects. The applications or users programs are also data objects.
- *Executions* : They represents the execution of applications. There are basically three kinds of executions: the *Job* submitted by the user, parallel *Processes* scheduled from one job, and the possible *Threads* running inside each processes. For simplicity, we name all these entities as *Execution* entity in later discussion.
- *User*: It represents the real users of the cluster.

In addition to these basic entities, users can define their own entities. The only limitation is the new entity must connect with existing entities. The reason is to keep every element in the graph accessible by traveling through the graph.

B. Relation To Edge

We define several basic relationships based on these basic entities in Table I. Each cell shows the relationships from the row identifier to the column identifier. It denotes a directed edge in the metadata graph. For example, *run* indicates that the User runs an Execution, which could be Job or just a Process; *exe* means one Execution was started from an application, which are Data Objects.

TABLE I
Default Relationships Definition.

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

There are *belongs/contains* cells. In the Execution entity case, it means the Job contains Processes and Process belongs to a Job. In the Data Objects case, it can show that one directory may contain multiple files or directories. Users can create their own relationships, which could be metadata that current graph does not record. For example, two users can have a new relationships called *login-together* if they login the system roughly at the same time.

C. Property

To collect rich metadata, we always need to collect annotations on entities or their relationships. In proposed graph model, we can store them using properties. which is in a key-value formation and could be default or user-defined. For example, all entities and relationships both have a default property named 'Type' to distinguish themselves, like User vertice have 'Type' as 'user'. And, there usually are attributes for different entities, like the user name, user privilege, execution parameters, data name, and data permission mode etc. We could abstract them as properties of the graph. Timestamps is another important category of property. For example, the *run* relationship has a $start_{ts}$ and a end_{ts} attribute and the *read* relationship has one ts attribute.

Users can create their own properties except the keys need to be unique in each users' namespace. We isolate properties by users, so that users are free to create any metadata without introducing global contention.

III. Metadata Graph Prototype

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

A. Mapping Strategy

Darshan utility is a MPI library that can be linked to users' applications and generates I/O behaviors logs during they are executing. Each Darshan log file represents a distinct job. The log entries of a job contain the user id who started this job, the executable file that the job was based on, the parameters of this execution, some environmental variables, and most importantly, the file access history of each process (ranks) inside this job (MPI program). Note that, the collected Darshan traces have been anonymized, only storing the hashed values of file names, path, user names, and job names.

We map Darshan logs to the metadata graph defined in Section II. Basically, each unique user id indicates a User entity, each Darshan log file represents a Job, all the ranks inside a job correspond to the Processes, and both the executables and data files are abstracted as different Data Object entities. Currently, Darshan does not capture directory structure as it only stores the hashed value of file paths, so we synthetically create the simplest directory structures: data files visited by each execution are considered under the same directory, and all these directories accessed by one user are placed under one directory for each user. We collect basic metadata in Darshan logs as the properties too: the *Type* property is set to each entity and relationship, the *start_time* and *end_time* of a job is mapped to the $start_{ts}$ and end_{ts} properties of *run/exe* and *read/write* relationships. Based on this mapping, we were able to import a whole year' Darshan trace (*2013*) on Intrepid machine into an example graph.

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

B. Graph Size

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

TABLE II
Darshan Graph Size and Some Comparisons.

Number	Basic	With I/O Ranks	With Full Ranks	With Directory
Vertice	34,656 K	41,729 K	147,886 K	147,934 K
Edges	126,488 K	133,561 K	239,766 K	366,253 K
Properties	448,775 K	484,141 K	1,015,070 K	1,394,628 K
Total Size	609,918 K	659,431 K	1,402,722 K	1,908,815 K

	Twitter	Road Graph USA	Web Page Graph	Human Brain
Vertice	645,750 K	24,000 K	2.1 billion	100 billion
Edges	81,364,500 K	29,100 K	15 billion	1,000 trillion

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

In the button half of Table II, we show several typical large-scale graphs from different fields, including the social network (e.g. Twitter), the road map graph (e.g. USA map), the Internet web pages, and the largest and most complex graph, human brain. By comparing with them, we can notice that, although the metadata graph is large and could easily be much larger, they are still manageable using current techniques.

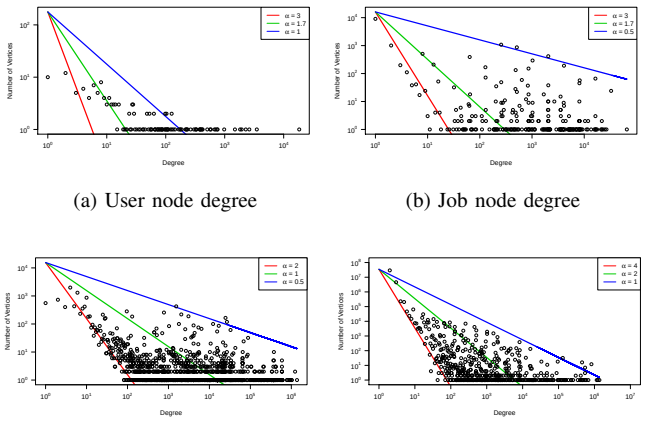
C. Graph Structure

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

TABLE III
Statistics of Metadata Graph.

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

This table shows different entities in metadata graph have totally different size and structures. The degree of a user node indicates how many jobs the user submitted in total; the degree of a job node shows how many processes it contains; the degree of each process node shows how many files are read or written by it. Data Objects both have *contains/belongs* edges to other Data Objects and also have *exe, read, write* edges to the Execution nodes.



(a) User node degree

(b) Job node degree

(c) Process node degree

(d) File node degree

Fig. 1. Node degree for different entities.

Figure 1 shows the node degree distributions of those four different entities. The x-axis denotes the degree, the y-axis shows the number of vertices which have that number of degree. Both x-axis and y-axis are 'log' values.

To describe the graph structure, we compared these distributions with a well-known property of nature graphs: the *skewed* power-law degree distribution, where most vertices have relatively few neighbors while a few vertices have many neighbors. Under power-law degree distribution, the probability that a vertex has degree d is given by: $\mathbf{P}(d) \propto d^{-\alpha}$. Here, α is a positive constant that control the "skewness" of the degree distribution: higher α indicates lower density, and vast majority of vertices are low degree. Lower α shows higher density and more high degree vertices.

In Fig. 1, to show whether the distribution fits the power-law attributes, we plot three lines to describe the power-law distribution with different α values. Intuitively, we can notice that the *File* nodes fit the power-law degree distribution best as Fig. 1(d) shows. There are two things we can learn from this. First, the files access fit the power-lay degree distribution, which means that, in real systems, most files are seldom accessed, but there are a small part of files are really visited highly frequently. Second, as the figure shows, the larger α value (4) actually fits the distribution better. This indicates the *File* nodes have lower density, which means the majority of files have low degrees.

Other three distributions, from *Process* nodes, to *Job* nodes, and to the *User* nodes, are becoming more and more unlike power-law distribution. For the *Process* graph, vertices with the lowest degrees is even fewer than vertices with larger degree. This means the processes in HPC systems tend to visit at least several files instead of just visiting one file. For *Job* graph, the node degree distribution scatters randomly between the $\alpha = 3$ line and $\aleph = 0.5$ line. This is reasonable because most jobs in HPC tend to have more processes. As there are only 117 users, we do not consider they fit any distribution. But from Fig 1(a), we still can see most users only issue several jobs, a very small number of users will issue the most jobs.

These graph structures direct the way of graph storage and processing. For example, the way to split a power-law graph has been well studied. Besides, we could also use this graph structures as a basis to create synthetic graphs to test the underlying performance of underlay graph infrastructure as currently real rich metadata are hard to get in a running HPC system.

IV. Use Cases On Using Metadata Graph

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

A. User Audit

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

B. Hierarchical Data Traversal

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

An appealing advantage of using graph to store the directory structure of file system is the scalability. Traditional POSIX directory structure limits the number of files inside one directory. So, HPC system that may have millions of files under one directory, needs to deploy specific system like Giga+ to distribute the metadata into multiple servers for better

performance. However, for proposed graph model, this is a genetic graph slicing problem, which we have several standard ways to tackle.

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

C. Provenance Support

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

In this challenge, a simple example work-flow was provided as the basis, a workable provenance system should be able to represent the work-flow and all the relevant provenance for the example work-flow, and, most importantly, be able to answer nine predefined queries. Based on proposed graph model, we can easily abstract the work-flow as serial of Executions run by one User. Each execution reads several Data Objects and generates outputs for applications in next phase. Based on this work-flow, the provenance system needs to answer nine queries. Here, we use the 8th query as an example:

```
1 //Query 8
  wf{*}: upstream(x) union x
3        where x.module=AlignWarp
         and
5        y in input(x)
         and
7        y.annotation('center')='UChicago'
```

This query mean we need to return all applications whose module is 'AlignWarp' and all their inputs are annotated with key-value pair: ['center':'UChicago']. This can be expressed easily in the metadata graph: 1) query all the nodes that represent Execution entity and have an *exe* out-edge that points to a Data Object named 'AlignWarp'; 2) start from all those nodes and filter out those executions whose property 'center' does not equal to 'UChicago'; 3) return all these executions.

A notable advantage of our metadata graph comparing with pure provenance system is that we can cross-reference different category of metadata in an unified way. If the provenance query needs the help of other metadata, like the file size, permission mode, or user group information etc., processing them in a unified graph will be more efficient and straightforwards.

V. Graph Facilities & Design Choices

The limitation of traditional databases especially the relational databases has lead the development of new categories of system called *graph databases* to cover the requirements of complex graph-based relationships. In the last couple of years, there have been an increasing work about graph databases. Some popular graph databases include Sones, AllegroGraph, DEX, G-Store, HyperGraphDB, InfinitGraphDB, Neo4j, Titan, and OrientDB etc.

A. Graph Databases

These graph databases can be categorized based on different metrics. Based on data storage, there are in-memory databases (e.g. Sones) and disk-based databases (e.g. AllegroGraph, DEX, Neo4j, Titan). Based on the graph data structure, there are *simple graphs* databases, *hypergraphs* databases, and *property graphs* databases. Here, the simple graph indicates graph defined as a set of nodes connected by weighted edges. AllegroGraph and G-Store only support this simple graph data structure. Hypergraphs extends the simple graphs by allowing an edge to relate an arbitrary number of nodes. Databases like HyperGraphDB and Sones support these hypergraphs. Property graph indicates graph where nodes and edges contain properties. This property graph is the very basic of our proposed metadata graph model. Many databases like Neo4j, Titan, InfinitGraph, and DEX etc. support such graphs.

In addition to this, some graph databases support query languages like SQL in relational databases. For example, Neo4j supports Cypher graph traversal language, Titan supports Gremlin language etc. Although most graph databases provide basic APIs to travel graphs, it is still more easy and efficient to use a query language.

Whether support distributed deployment also classify all these databases. Only part of those databases support distributed deployment due to the fact that distributing graph into different servers with ACID and enough performance is challenge. For example, Neo4j only supports high availability deployment, which means multiple copies of each data across the cluster rather than a distributing. There are still lots of research work going on this problem.

B. Storing and Indexing Requirement

Actually, the graph database choices in our case are limited by the attributes of the metadata graphs. First, the metadata graphs are too large to fit into memory, so, the disk-based databases are better. Moreover, as the metadata graph is based on property graph model, which may store a large amount of properties, the graph could even too large to fit in one server. The distributed supports will be another necessary component. About the query language, it is not the first concern yet, but still, graph databases with a query language and some query optimizer (e.g. indexing) will be much better.

C. Traversal Requirement

From the example use cases in previous section, we can see that the key to use the metadata graph to support different use cases is effective graph traversal. In fact, the graph size, graph data structure, and storage layouts, all decided the performance of traveling a graph. The simplest case is to deal

with moderate-sized simple graph. In this cases, traveling from one or several , or even all vertices and explore their k-hop neighborhood can be possible to perform in memory as the simple graph is smaller and possible be cached into memory.

However, traveling through a property graph like our metadata graph will be much complex. The main reason is that, during traveling, we usually need to apply filter or computations on some properties like previous provenance example shows. As these properties are too big to be cached in memory, each time, we need to load them from persistent devices. This will introduce lots of random seeking leading to a poor performance. This is actually still a big challenge for property graph databases waiting for solving.

D. Graph Processing

In addition to the graph databases, there are also graph processing frameworks which can be used to perform computation or queries on graphs in a distributed way. Typical examples of these frameworks include Gigraph, which was designed and implemented based on Pregel computing model; GraphX, which was based on Spark computing framework; GraphLab, X-Stream, and GiGraph+ etc.

However, most these distributed graph processing frameworks work on the unstructured graphs which are usually simply stored as a plain file in CSR or CSC formats in a general storage back-end , like local disk, HDFS, S3, or RDD etc. So, there is a big gap from deploying graph algorithms on these plain graph formats to running these algorithms on a graph database. However, the ability to run complex graph algorithms is necessary for metadata management, and this ability can not be easily satisfied using the querying or searching facilities provided by graph databases. For example, in HPC system, we can run *community discovery* algorithms on metadata graph to find the 'closely' data files, which can be used to optimize their physical placements for better future I/O performance. These algorithms get the whole graph involved in the computation, contain multiple iterations, and last a long time. A distributed fault-tolerant graph processing model is a much better choice than writing applications to manage all these complexity.

VI. Conclusion & Future Work

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

References