gRMM: A Unified Graph-Based Rich Metadata Model for HPC Platform

immediate

***Abstract*—HPC platforms are capable of generating huge amounts of metadata about different entities including jobs, processes, users, and files etc. *Simple metadata*, which describes the attributes of these entities has already been widely used, like the file size, name and permission mode. On the other hand, the *rich metadata*, which records not only the attributes of entities but also relationships among them, is still missing in most HPC systems. For example, the rich metadata *provenance*, which records the entire life cycle of data objects, is not well supported, even given the fact that provenance provides many appealing data management functionalities, such as determining the quality of the data set, finding the source of data corruption, and tracing all data dependency etc. The main challenge is that the rich metadata can be too flexible to define and collect in general: they may contain arbitrary information and come from different system components or even users applications. Collecting, storing, and processing such rich metadata will place huge pressure on the HPC platform.**

In this paper, we propose *gRMM*, a rich metadata management solution that abstracts all the metadata (both the *simple and rich metadata*) into one generic and flexible property graph-based model. We show that *gRMM* can be leveraged to efficiently support simple metadata operations, including directory namespace, permission validation etc., and also support rich metadata processing, especially the provenance storage and query. We also present the detailed design for integrating *gRMM* into current HPC platform efficiently.

I. Introduction

Metadata, especially rich metadata, contains the detailed information of different entities and their relationships in HPC. These entities could be users, jobs, processes, data files, and even user-defined entities. Storing and utilizing the metadata has already provided the basic data management functionalities in most existing storage systems, including finding files, controlling file accessing, and tracing file creation and accessing time etc. We category this part of metadata as the *simple metadata* since they only contain the predefined attributes about individual entity. As a contrary, *rich metadata* cares more than individual predefined attributes; it may store the 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. In the computational systems, it indicates a recording of complete history of each data piece, including the processes that generated it, the users that started the processes, and even the environment variables, parameters, and configuration files while executing. A complete provenance information supports huge amount of data management abilities. For example, the accessing history of users reading/writing data files can help us develop a 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 the 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.

With such greatness of rich metadata like provenance, current HPC platforms still lack of providing basic facility to collect, store and process the rich metadata. The challenge comes from at least three places.

- *Storage 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 detailed accessing history of each process will place a huge pressure on the storage systems. In addition, as storing rich metadata should not affect the application execution speed significantly, the resources (both network and disks bandwidth) dedicated to storing metadata are limited to use in most cases.
- *Efficient Processing*. Even we already have the perfect collected and stored rich metadata, it is still a big challenge to process them. First, as the rich metadata is large and can not be hold in one server, the distributed processing framework is necessary in most cases. Second, many use cases require complex analysis instead of simple searching or reading, so flexible processing should be provided for them.
- *Metadata diversity*. As we have described, the rich metadata could be as diverse as the usages need. They can only contain predefined attributes and relationships of entities, or be extended to any user-defined attributes and relationships. Traditionally, we used different tools (system components or users applications) to store and process part of metadata based on the specific usages. However, this introduces lots of unnecessary redundant metadata storage in different tools. For example, both the data audit tools and data verification tools need to store the file access history of users, so this metadata usually was stored twice in two applications. If we only store each metadata once in one application or component,

then there will be massive cross-reference operations between them during later processing or analysis, which is inefficient.

In this paper, we proposed a idea of unifying all metadata (*simple* and *rich*) into a general graph-based model for HPC platform. By exploring the collection, storage, and processing of metadata, we form a practical solution named *gRMM* introduced in this paper to provide such unified metadata service.

gRMM integrates itself closely into HPC storage systems to store rich metadata in an efficient and consistent way. In our case, the underlying storage system is Triton. The data model for storing is based on graph abstraction: all metadata for entities, attributes and relationships was abstracted as elements in a unified graph model and stored by calling the graph-based storage APIs. Moreover, by closely integrating into storage systems, *gRMM* is able to trace all reads/writes on data pieces in storage system and provides detailed metadata for these data pieces automatically. Users can easily access these metadata later, combine them with information from other components to build rich metadata, and store them.

More than this, *gRMM* also contains a runtime library to help users query and process existing metadata. All these queries and processing are based on mature graph algorithms, which is expressive and efficient. In general, *gRMM* aims at providing a genetic layer for easing the burden of collecting, storing and managing metadata in a modern HPC system. Working with a fault-tolerant object storage system like Triton, we could form a fully functional parallel file systems using *gRMM* easily.

This paper was organized as follow: in Section II, we will first introduced the rich metadata graph model. In Section III, we introduce different metadata use cases with incremental complexity, showing how to map different use cases into proposed graph model, and solve them using facilities provided by *gRMM*. In Section IV, we introduce system components and discuss the design considerations and challenges in different components. In section V, we conclude the study and list future work.

II. Graph-based Metadata Model

Many times, we have already consider metadata as graph or part of graph. The traditional directory-based file management constructs a tree structure to manage files. This tree is a subgraph. The provenance standard considers the provenance of objects is represented by an annotated causality graph, which is a directed acyclic graph, enriched with annotations capturing further information. This is already a graph, but contains restrictions on causality.

We generalize these graphs in HPC scenarios and propose the metadata graph model in this paper. The metadata graph is derived from the property graph model instead of traditional weighted graph. It includes vertice that represent entities in the system, edges that show their relationships, and properties, which is the main difference from traditional graph. The properties annotate both vertice and edges, and can store arbitrary information users need. In the following subsections, we will introduce the details of metadata graph model.

A. Entity/Vertex

In a HPC platform, there are different entities, like 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 needed. In *gRMM*, 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. It could be files or even data objects.
- *Thread/Process/Job*: It represents the execution of application. There are basically three kinds of executions: the *job* submitted by the user, parallel *processes* scheduled based on the submitted job, and the *threads* running in each processes. For simplicity, we name these entities as *Execution* entity in later discussion.
- *User*: It represents the users of the cluster. They submit jobs, run applications, and start or stop jobs.

In addition to these basic entities, *gRMM* also allows users to create their own entities. However, it is not allowed to create an entity without connecting with existing entities. The main reason is to keep every element in the graph accessible by traveling through the graph.

B. Relation/Edge

Based on the basic entities, we define the relationships between them as the basic relationships as Table I shows.

TABLE I
Default Relationships Definition.

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

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 the Execution starts from an executable, which is a Data Object. The *belongs/contains* cell shows a general relationship between two entities. In the Execution entity case, it means the Job contains Processes and Process belongs to a Job. In the Data Objects case, it describes that one directory contain multiple files as the directory is also considered as a file in most directory-based storage systems. *belongs/contains* are one of the most common pair relationships and can be used to form new user-defined entities.

gRMM allows users to create their own relationships. The new relationships can be used to support more complex semantics 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

Both entities and relationships may be annotated with arbitrary properties. A property is a key-value pair, which could be default or user-defined. We list some default properties here.

- *Type*. Entities and relationships both have a default property named 'Type' to distinguish themselves. For example, the User entity has type 'User' and *Run* relationship has type 'run'.
- *Attributes of Entities*. There are all different kinds of attributes for User, Execution, and Data Object entities. The complete list can be too long, but some significant examples are listed here, including the user name, user privilege, execution parameters, data name, and data permission mode etc.
- *Attributes of Relationships*. Most of the edges have timestamps as attributes. For example, the *run* relationship has a $start_{ts}$ and a end_{ts} attribute; the *read* relationship has one ts attribute.

gRMM does not place any limitation on the user-defined properties except their keys need to unique in each users' namespace. The namespace is a concept to divide isolate these user-defined entities, relationships and attributes. Each user has its own namespace, so that they are free to create any metadata they need without introducing global confusion.

III. Use Cases

Metadata graph model introduced in previous section is expressive and provides many new functionalities. We will show this based on several use cases from simple to complex in this section.

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 both for daily maintenance and security. In *gRMM* model, we collect the *run* relationships between Users and Executions. Each Execution will read or write certain set of files and generate relationships named *read/write*. If administrators want to find all the files that were read by a specific user during given time frame $[t_s, t_e]$, they can query the graph from the given user, travel through *run* edges to all the executions. Then, they can filter them and travel through the *read* edges to get the file collection. We show the query script using the Gremlin graph traversal language for this user-centric data audition.

```
files = graph.V(userA).out('run')
          .filter{it.start_ts > t_s
                  && it.end_ts < t_e}
          .out('read')
```

Another typical use case will be to find all the users that used to write to a file which was found broken later from time t_s. The Gremlin script for this query looks like following:

```
users = graph.V(fileA).out('wasWrittenBy')
          .filter{it.start_ts > t_s}
          .out('wasRunBy')
```

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. The tree structure, which is used to organize all the files, mainly includes two different kinds of nodes: directories and files. The directories can be viewed as a special kind of file which only stores indexes of other files or directories.

In *gRMM*, we abstract both the directories and files as Data Object entity. Directory *contains* multiple files or directories; both files and directories *belongs* to a certain directory. Given an absolute path, locating the file becomes going through a bunch of *contains* edges; listing all files inside a directory becomes getting the destiny nodes of all the *contains* edges. Access control metadata attached in users, files, and directories also should be stored and verified while traversing directories.

```
// locate: /rootFS/dir/file.data
graph.V(rootFS).out('contains')
     .filter{it.name = dir}
     .out('contains')
     .filter{it.name = file.data}
```

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 use deploy specific system like Giga+ to distribute the metadata into multiple servers to improve the performance. However, for *gRMM*, directories with millions of files indicate a node with millions of out-edges in the graph. There are already bunch of standard ways to slice such a 'big' node into different servers for load balance in distributed graph storage systems.

In addition to traditional POSIX-style files and directories, several other hierarchical data traversal examples are possible in *gRMM* as well. For example, the semantic data management and traversal. 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. Although careful file naming and directories placement helps here, it is still too rigid for use in complex scenarios as when a file is a common input of multiple simulations. However, storing enough metadata between simulation executions and their inputs/outputs can intelligently help users organize data in multiple dimensions.

C. Provenance Support

Provenance has a wide range of use cases, *gRMM* as a superset of provenance, is able to support most of these usages. To show how metadata graph supports provenance, we use the problem in 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:

```
//Query 8
wf{*}: upstream(x) union x
       where x.module=AlignWarp
       and
       y in input(x)
       and
       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 easily expressed in Gremlin script running on the metadata graph:

```
//Query 8
exes = graph.V(Executions).out('exe')
      .filter(it.name = AlignWarp)
exes.out('read').filter(it.center = 'UChicago')
```

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

IV. Prototype and Design Choices

In this section, we introduce a prototype system showing how to build an example metadata graph from Darshan trace. This prototype gives a general idea of the complexity and challenges of proposed graph model. Based on that, we will discuss several important design choices for *gRMM* including the storage choice, the different detailed levels of metadata choice, and the interface design etc.

A. Darshan Metadata Graph Prototype

The most unique feature of rich metadata is that it contains the relationships between different entities like users, processes, and data files. Collecting such relationships usually requires modifications on the runtime systems. In this prototype, instead of changing the runtime system of HPC platform, we exploit the Darshan trace logs as the source of rich metadata.

Darshan utility is a MPI library that can be linked to users' applications and generates I/O behaviors logs during applications 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 current execution was based on, the parameters of this execution, and most importantly, the file access history of each process inside this job. To reduce log size, Darshan only stores the hashed information instead of original value.

We can abstract this information in Darshan log file to different entities in *gRMM*. First, each unique user id indicates a User entity; second, each Darshan log file represents a Job entity, and all the ranks (processes) inside it are corresponding to the Processes entities; and, both the executables and data files are abstracted as Data Object entities. There is not any directory structure as Darshan only use the hashed value to distinguish different files. All the parameters are considered as the properties of Job entities. Based on this abstraction, we were able to get three-year' Darshan trace (*2010, 2012, 2013*) on Intrepid machine in Argonne National Lab. into an example graph.

TABLE II
Graph Size with different abstraction levels.

	Basic Abstraction	Abstraction with Ranks	Abstraction with Directory
Vertices			
Edges			
Properties			

Table II shows some statistics of such example graph. The first column named basic abstraction means that for the Execution entities, we only consider the Job. All the I/O behaviors from the processes inside a job will be considered to be from the Job entity. In this case, we got a graph containing x vertice, y edges, and z properties. For some use cases, the processes information (ranks in MPI applications) is also needed, so, in the second column of Table II, we show the graph size with Processes entities. There are *belongs/contains* relationships between Job and Processes entities, and also all the *read/write* relationships are between Processes and Data Object now. The graph size increases sharply as you may see.

As we have described, the directory structure is missing in current Darshan trace. However, this part of metadata is critical in most of storage systems, so we create synthetic *contains* and *belongs* relationships between all Data Object entities according to the files and directories size in the real file systems of Intrepid at that time. The third column of Table II shows the graph size with such directory structure.

B. Graph Storage Choice

The proposed metadata graph follows a property graph model, which is different from traditional graph that only contains simple weight value on each edge. The property graph needs to store both the graph structure, which indicates the connections between nodes, and the rich data associated with the graph including properties on both nodes and edges. So, traveling through the property graph is different from traditional graph: we should only travel through limited number of edges which satisfy certain conditions, for example the edges whose type is *run* instead of going through all the out-edges.

There are several choices to store such graphs. The first choice will be a relation database, which was widely used

to store provenance graph. Relation database is mature, supports huge data size with proper distributed deployment, and provides a flexible query language (SQL) to execute complex queries. According to the normal forms, storing such a graph in relation databases requires a complex scheme with multiple tables and foreign keys to refer each other. This table scheme causes poor performance while traveling the graph through edges as we need to join different tables to build a complete path. To avoid such time-consuming 'join' operations,

Comparing with relation databases, graph databases are more suitable for such property graphs. Graph traversal languages like Gremlin, Cypher are also supported in these graph databases for querying.

Another storage choice is Key-Value storage systems. Comparing with relation databases and graph databases, they do not have any scheme. Simply mapping the nodes and edges as different objects and store their properties as the value of these objects provides a plain strategy to store the metadata graph.

@TODO: In this section, we will show the performance of different storage systems (RDBMS, GraphDB, KeyValue) in different scenarios (Insert, Locate, Traversal).

C. Metadata Detailed Level

D. Metadata Graph Interface

V. Conclusion & Future Work

References