Grid-Based Metadata Services

Ewa Deelman[1], Gurmeet Singh[1], Malcolm P. Atkinson[2], Ann Chervenak[1], Neil P Chue Hong[3], Carl Kesselman[1], Sonal Patil[1], Laura Pearlman[1], Mei-Hui Su[1]

[1]Information Sciences Institute, University of Southern California
[2]Department of Computing Science, University of Glasgow & Division of Informatics, University of Edinburgh
[3]Edinburgh Parallel Computing Centre, UK

Contact Author: Ewa Deelman (deelman@isi.edu)

Abstract—Data sets being managed in Grid environments today are growing at a rapid rate, expected to reach 100s of Petabytes in the near future. Managing such large data sets poses challenges for efficient data access, data publication and data discovery. In this paper we focus on the data publication and discovery process through the use of descriptive metadata. This metadata describe the properties of individual data items and collections. We discuss issues of metadata services in service rich environments, such as the Grid. We describe the requirements and the architecture for such services in the context of Grid and the available Grid services. We present a data model that can capture the complexity of the data publication and discovery process. Based on that model we identify a set of interfaces and operations that need to be provided to support metadata management. We present a particular implementation of a Grid metadata service, basing it on existing Grid services technologies. Finally we examine alternative implementations of that service.

I. INTRODUCTION

Today, advances in science are made possible largely through the collaborative efforts of many researchers in a particular domain. We see collaborations of hundreds of scientists in areas such as gravitational-wave physics [1], high-energy physics [2], astronomy [3] and many others coming together and sharing a variety of resources within a collaboration in pursuit of common goals. These resources are distributed and can encompass people, scientific instruments, compute and network resources, applications, and data. Although the scale of the resources pooled within collaborations is increasing, there is a particular explosion in the size of the data that is made available. It is common to see datasets on the order of terabytes today with petabyte-scale sets coming online in the near future. Grid environments [4] enable efficient resource sharing in collaborative distributed environments, and in this paper we focus on the area of data management, with a particular emphasis on metadata management issues. We address the problem of discovery and manipulation of data objects.

One of the challenges of these shared environments is to identify and locate the subset of the data objects that are of interest to any particular data analysis activity. The standard solution to this problem is to describe the characteristics of each data object with one or more attributes, or “metadata” and to use this metadata as the means for identifying what data objects will be of use. In this paper, we explore from a system perspective how to manage metadata in such a way as to make it accessible for discovery and access of data in a distributed, collaborative environment.

The boundary between data and metadata is to some extent arbitrary and may vary during a data object’s life time. For example, for some users a table of astronomic objects derived from images is primary data and for others it is metadata that indexes the primary or calibrated data. Extensive use of human annotation is common in the large numbers of curated biomedical databases. To some the annotation is metadata, but we require metadata to describe those annotations to gauge their quality. A typical modern hospital will produce about 1 petabyte of digital data per year – improved practices by instrument manufacturers have automated the association of metadata, such as instrument settings, with the primary data. For the maintainer of the instruments these may be primary data. In any given context this ambivalence is resolved. Therefore, for the remainder of the paper we refer to the data currently being interpreted to enable the interpretation of other data as “metadata”.

Metadata allows collaborations to publish data with enough information for scientists to be able to identify the desired data products. Metadata attributes can encompass a variety of information. Some metadata is application independent, such as the creation time, author, etc. described in Dublin Core [5], while other metadata is application dependent and may include attributes such as duration of an experiment, temperature, etc. Metadata may refer to raw, experimental data that has been collected by an instrument or to data that has been processed in some fashion, for example calibrated. Metadata adds value to scientific data. Without metadata, the researcher is unable to evaluate the quality of the data. For example, it is impossible to conduct a correct analysis of a data set without knowing how the data was cleaned, calibrated, what parameters were used in the process, etc. Traditionally, metadata was recorded in laboratory notebook. More recently scientists have devised data formats to encode the metadata in the data files themselves (as in the case of the FITS-formatted files used to contain astronomy images). However, these mechanisms do not scale to large collaborative environment where there are many files. In this paper we focus on services that can provide sufficient functionality to efficiently publish and query metadata about

large-scale data sets. Collaborations have significant amount of data in multiple data bases or files. Grid technologies [4] are being increasingly used to provide data management infrastructure to access distributed resources within a collaboration. However, the issue of data discovery still remains. From the point of view of ease of use and scalability, the discovery need to be done based on annotations, or metadata. This paper addresses the question of what services are needed in Grid environments to facilitate data discovery. We argue that although standardized database services can be used, specialized metadata services can greatly simplify metadata management.

The contributions of this paper are:

- A description of metadata services requirements.
- An overview of an architecture for metadata services in service rich environments such as the Grid.
- A description of the data model supported by the service and the interfaces it exposes.
- An evaluation of the alternative implementations of the service, MCS (Metadata Catalog Service).

II. REQUIREMENTS FOR METADATA MANAGEMENT ON THE GRID

The astronomy community provides a good example of how metadata services are used in collaborative environments. Initially when the data, in the form of images, is collected by an instrument, such as a telescope, it is pre-processed and calibrated and stored in an archive. Metadata about the images describing the location in the sky, the calibration parameters, etc., is stored as well. Additional processing may occur to extract interesting features of particular regions of the sky or to produce images focusing on particular celestial objects. The information about the processing is captured in metadata attributes and stored as well. Once the metadata and data are prepared in this fashion, it is released to the group of scientists within the collaboration. Researchers can then pose queries on the metadata to discovery data of relevance to their work. Based on the results of the searches conducted on the metadata scientists may want to organize the metadata in a way that is most appropriate for their research. During this phase of the data publication process, the data may be further annotated by the collaborators. After a certain period of time, usually on the order of two years, the data and the metadata are released to the general public. At this time, users outside of the initial collaboration search the metadata based on attributes that are important to them.

There are several requirements for managing metadata on the Grid. The requirements can be broadly grouped in four main categories:

1. The need to store and share the metadata.
2. The need to organize the metadata in a logical fashion for ease of publication and discovery.
3. The need to customize the view of the data by individuals.
4. The need to support metadata about large-scale data sets.

We can refine each of these requirements further. Sharing metadata necessitates having well defined interfaces for storing and querying metadata attributes and for adding new attributes. In general, metadata is domain-specific, however, some metadata crosses many domains, such as creator and creation time. Metadata services thus need to be able to support generic as well as domain-dependent attributes. In the scenario above, the metadata evolved over time; thus metadata services need to be able to evolve as well, providing a flexible way to add and delete metadata attributes. Queries need to be able to support the discovery of the metadata attributes of the objects stored in the catalog as well as the discovery of attributes of a particular object and the discovery of objects with particular attributes.

It is also useful to organize the metadata in some logical fashion, For example all the data relating a particular region of the sky may be placed within one group (collection). Attributes may also be associated with such groupings for ease of discovery. The aggregation is useful because it allows for the grouping of objects that are related to each other in some fashion. This may facilitate data discovery, because the discovery can be first done at the collection level: find all collections that have the particular characteristics and then refine the search within these collections. If the search was performed on a flat object space, then it is possible for the queries to return objects that have no particular relation to each other. Performance of searches may also be improved, because in general, there will be fewer collections than individual data items, thus making queries for collections with particular attributes more efficient than queries across all data items. The grouping may be hierarchical, representing the aggregation of collections. In the case of data publication by the collaboration the collections are usually organized by the publishing body. They may also be the work of scientists using published and consortium data who are collecting together all of the objects relevant to an interest, e.g. galaxies with a recorded type 1A supernova, quarks with observable lensing, binaries where one star is a pulsar, suspected satellite tracks, etc. These collections are typically dynamic, intersecting and associated with particular scientific interpretations.

As part of this organization, the collaboration may impose access control on entire collections, on collection groups, or on individual items. A Metadata Service must thus implement authentication and apply policies to metadata access and publication. Authentication and authorization allow control over who is allowed to add, modify, query and delete mappings in the Metadata Service. Auditing information that may be maintained by the Metadata Service includes information about creators and creation times of metadata mappings as well as a log of all the accesses to a particular metadata mapping, including the identity of the user and the action that was performed. Different types of users need to have different access to the metadata.

Although collections allow the publishers to organize the metadata, they do not allow customization by individuals who are part of the collaboration. In general, scientists within a collaboration may have different research goals and may want

to organize the data in a way that is most appropriate for them. This individual-based view of the metadata should not affect the structure or authorization policies imposed by the publisher. Rather, they should be layered on top of the existing collections.

Data sets and their metadata are quite large today and ever growing in size. Often, metadata is stored independently of the data itself. Metadata services need to provide data handles that can be resolved by other services that perform the data access. Metadata services must provide the ability to store information about millions of data objects and provide good performance. They should provide short latencies on query and update operations and relatively high query and update rates. To support reliable access to metadata, the services may need to be replicated.

III. ROLE OF GRID SERVICES

Grid services are used in many application domains today to deliver computing power and data management capabilities needed by large-scale science. Grid services extend standard web services by providing support for associating state with services, managing the lifetime of service instances, and standard mechanisms for subscription and notification of state changes. Grid service interfaces are being standardized as part of an overall Open Grid Services Architecture (OGSA)[6] through the Global Grid Forum. Large scale testbeds such as the Teragrid [7] and iVDGL [8] are deploying Grid services that allow authentication [9], remote job scheduling [10], data access [11], data replication [12] and others. These are basic services available as part of the Globus Toolkit 3 (GT3) [13], the de-facto standard for Grid services.

One particularly relevant Grid service is the OGSA Database Access and Integration (DAI) Service being developed by the Edinburgh Parallel Computing Center (EPCC). Service interfaces are being standardized through the DAIS Working Group of the Global Grid Forum [14]. The OGSA-DAI service provides a common Grid service access interface to a variety of data resources ranging from relational databases to XML databases and eventually to structured files. The DAI service is intended to provide a basis for higher-level services to be constructed, for example, to provide federation across heterogeneous databases. The OGSA-DAI service has three main components: a service registry for discovery of service instances, a data factory service for representing a data resource and a data service for accessing a data resource, such as a relational database. The OGSA-DAI service uses an extensible activity framework. These activities can be extended by other developers to provide additional functionality.

There are several reasons for providing grid service-based metadata services:

1. Integration with other Grid services: Resource access in Grids is authenticated using the Grid Security Infrastructure (GSI) [9] that is based on PKI. To integrate databases with other services on the Grid, GSI authentication needs to be performed by the database service. OGSA-DAI supports this type of authentication and maps authenticated users to database roles. Integration with other services is also enabled by OGSA-DAI's XML-based communications.

2. Service discovery: In a distributed environment, there may be several metadata services. Discovering the appropriate service in necessary. Grid services and OGSA-DAI support the use of service registries and the discovery of services based on attributes published by services (service elements).

3. Federation of multiple databases: As mentioned above, there may several relevant metadata services. It is important to be able to query across them in search of desired data. Grid services and OGSA-DAI provide support for service data publication and notification of changes in the values of the service data, thus providing the basic mechanism for federating multiple OGSA-DAI-based services.

In this work we developed a Metadata Catalog Service (MCS) that builds on the existing Grid services. Fig. 1 shows the software layering of the MCS. Below, we discuss the data management model supported by MCS.

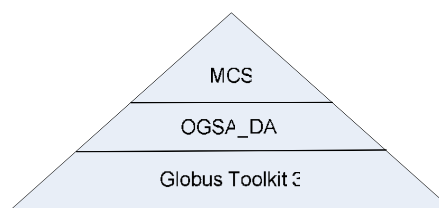

Fig. 1: Layering of Services to support Metadata Services.

IV. DATA MANAGEMENT MODEL

In the Requirements Section, we outlined a scenario for the use of metadata services for data publication and access. It is clear that metadata attributes could be represented in some database technology, such as relational or XML, and that the discovery of data objects be mapped into queries. We believe, however, that in many instances, it is desirable to have a more specialized metadata management service. We argue that in Grid environments, we need to have dedicated Metadata Services because of several concerns, such as usability and ease of schema discovery. Databases provide a very general and flexible infrastructure for data management. However, one has to be familiar with query languages such as SQL or XQuery to be able to efficiently interact with a DBMS. Many domain scientists in physics, astronomy, biology, etc. simply are not comfortable with query languages. Users and applications also need to know the internal database structure to be able to pose appropriate queries. Our schema and API (described below) provide an easy way to discover attributes and interact with the system.

Given the benefits of providing an extensible metadata model, we present a model that supports a variety of interactions, allowing users to describe attributes of data items and organize them in ways that are needed by a collaboration and by individual users. We also describe the API that supports the data model and is used to manage objects within the Metadata Service.

The model also supports a flexible set of attributes. We define the basic object within a metadata service as a data item. This *data item* may for example represent an individual image. Fig. 2 shows a UML diagram of the objects supported within MCS. The following sections give more details about the role of each of the entities.

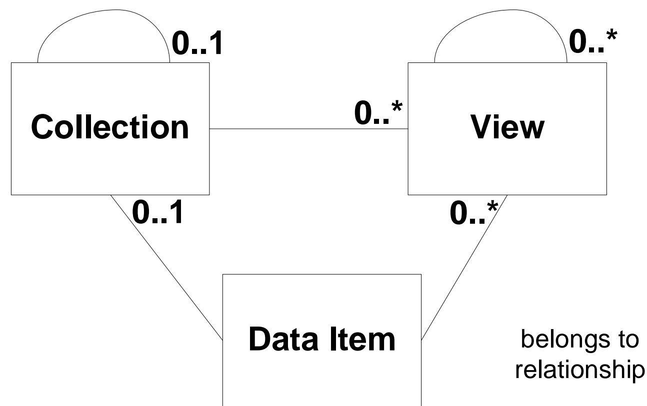

Fig. 2: A Data Model for Metadata Services. The depicted relationship refer to the "belongs to" relationship.

A. Users

In general we distinguish between three types of users: the collaboration, the members of the collaboration and the general public. The collaboration is in charge of publishing the data sets. Individuals or groups within a collaboration may provide additional attributes and annotation and may structure the metadata in a personalized way. Finally, members of the general public (or community at large) may query the metadata services.

B. Structuring Data Items and Imposing Authorization Policies

It is often convenient to be able to refer to a set of data items with a single name. This can play an important role in improving the scalability. For example, authorization can be attached to that single, without having to impose it on every individual item. Name sets can also be useful in the data publication process. They allow the data publisher (the collaboration) to impose a logical structure on the data items being published. For example, a collaboration may want to assign a single name to all the data collected during a particular run of an instrument. For these reasons, our metadata model includes a concept of *collections.* Collections allow a name and attributes to be associated with an arbitrary set of data objects.

Collections can aggregate a set of data items or other collections. The collaboration may also impose authorization on data items and data collections. In our model, the authorization on the data items and collections may be described through the authorization on the collection the data belongs to. This allows for defining authorization in a scalable way. To assure consistent authorization, a particular data item or data collection may belong to only one parent collection. If a data object could belong to multiple collections, then determining the authorization for the object would involve examining a set of possibly contradictory policies.

In our model we view collections as being defined by the collaboration. However, individual members of the collaboration may want to organize and name the published metadata in a customized way. Individuals may also want to associated additional attributes with the named entities. The organization designed by individuals should not affect the organization and authorization imposed by the collaboration as a whole. To support this functionality, we introduce the notion of a *view*. A view allows data items and collections to be organized by members of the collaboration into groups that are relevant to them. Views do not have any effect on the way that the metadata is published to the community or how access to the data is authorized. Views may be described by attributes, be annotated and made part of other views. Because there is no authorization imposed by a view, data items and collections may belong to several views. In general, members of the general public would only be able to query metadata services and would not be able to create views. Both collections and views are acyclic.

C. Flexible Schema

There is no common set of attributes that can describe data in a variety of domains. Usually, a collaboration agrees on a set of terms that describe their particular data set. However, metadata services need to span domains and collaborations and thus need to support a dynamic attribute set. Even within a collaboration, some flexibility may need to be supported. For example, members of the collaboration may come up with additional ways of describing the data, providing annotations and other attributes that are necessary for data interpretation.

Flexibility in the attribute set is needed for all data objects managed by the metadata catalog: data items, collections and views.

Metadata attributes can be divided among a set of core attributes, such as those described in Dublin core [5] and additional domain-specific attributes that can vary depending on the underlying application domain: astronomy, physics, etc.

There are also attributes that are specific to the Grid environment, where data may be replicated. As we already mentioned metadata discovery is the process of mapping a set of attributes to one or more identifiers that locate the data objects that posses the specified attributes. This location specification, which we call a logical name, can then be further resolved, using mechanism such as the Replica Location Service [12] to specific data object instances.

D. Metadata Interfaces

Fig. 3 illustrates and categorizes the range of schema models that can support metadata publication and discovery. One the left side of the graph, we show a rigid, fixed schema and the associated API. In the fixed schema all the metadata are known and encoded ahead of time. This schema does not provide a very flexible data model. If the collaboration wants to modify the attribute set, the entire schema needs to be re-worked and a new set of access and discovery mechanisms may need to be provided. This restricted model may, however, facilitate data

discovery, because the data model can be easily exposed Metadata Interfaces

At the other end of the spectrum, we have a fully dynamic schema, where users may create data objects and attribute structures on the fly. This is a more general and flexible model that requires a very general interface. Such an interface would also have to expose the internal structure of the underlying database. This may make the process of discovery complex, especially for users that fall into the general public category.

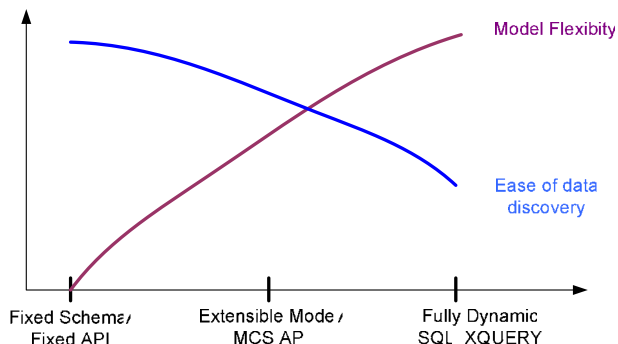

Fig. 3: Classification of Metadata Model Flexibility and Ease of Data Discovery.

In our work we aimed to design an interface that hides the implementation details from the end user. This also provides us with the flexibility in how we structure the database tables, in order to optimize performance, for example. Section VI.D explores alternative table layout we designed to support a flexible attribute set. We believe that with our approach we are combining the benefits of ease of publication and discovery of the fixed schema model with some of the model flexibility of the fully dynamic schema.

We propose an API that can allow for metadata publication, discovery and management of authorization. Publication includes creating and deleting logical objects: data items, collections and views. Attributes can be defined, undefined and set on all the logical objects. As part of publication, the content of a collection or view can be modified.

The API also supports a variety of metadata discovery methods. The API allows clients to discover the set of attributes defined within the Metadata Service and to search for logical objects based on attributes. The attributes of a particular object can be retrieved. The API also supports the discovery of the content of a collection or a view. Parent collections and views of a particular data object can be found as well.

The API also provides the granting and revocation of authorization on data objects as well as the service itself. For example, the API supports authorizing users and user groups to define new data objects.

The data model and the API we have described here are not a general solution. Rather, they support a set of functionality that can satisfy a class of applications that conform to the metadata publication, annotation and discovery requirements and scenarios we described in Section II. This model is also limited in the way it handles attributes. Our attribute namespace has a flat structure and does not support more complex attribute structuring schemes.

V. Related Work

The Storage Resource Broker (SRB) from the San Diego Supercomputing Center [15] and its associated MCAT Metadata Catalog [5] provide metadata and data management services. SRB supports a logical name space that is independent of physical name space. The logical objects, logical files in the case of SRB, can also be aggregated into collections. SRB provides various authentication mechanisms to access metadata and data within SRB.

However, our Metadata Catalog Service model differs from MCAT in significant ways. Perhaps most significantly, the architectural models of the two systems are fundamentally different. MCAT is implemented in tight integration with other components of SRB and is used to control data access and consistency as well as to store and query metadata. MCAT cannot be used as a stand-alone component. In addition, MCAT stores both logical metadata and physical metadata that characterizes file properties as well as attributes that describe resources, users and methods. By contrast, we have designed our Metadata Services to be one component in a layered, composable Grid architecture. We have factored this Grid architecture so that the Metadata Services contain only logical metadata attributes and appropriate handles that can be resolved by a data location or data access service.

The RepMec (Replica Metadata) catalog developed by the European DataGrid's Reptor project [16] is similar in its design and function to our metadata service. The RepMec catalog is built upon the Spitfire database service. The RepMec catalog stores logical and physical metadata. Among other functions, this catalog is used within the EDG project to map from user-provided logical names for data items to unique identifiers called GUIDs. RepMec is used in the Reptor system in cooperation with a replica location service.

VI. Implementation Issues

In our previous work [17] we presented an initial design of the Metadata Catalog Service (MCS) and reported performance numbers related to an implementation of the service based on the Apache web service[18]. The web service implementation lacked some of our desired functionality, including the GSI authentication, service element publication and notification mentioned in Section III.

In our previous study we showed that layering a web service interface on top of a DBMS (in our case MySQL) results in an order of magnitude in performance degradation over accessing the database directly via ODBC. We measured the service performance in terms of add and query rates performed by multiple clients. In our current study, we continue to use relational technologies and evaluate the use of Grid services to support metadata management services on the Grid.

We decided to use the OGSA-DAI service as the basis for the MCS implementation. By leveraging off the significant development efforts of the EPCC group as well as the OGSA developers of the Globus team, we are able to build upon a general purpose database access service with well-defined and extensible interfaces. We avoid having to implement our own Grid service for a metadata catalog service with GSI authentication, lifetime management, etc.

The success of our MCS implementation is based on the extensible activity scheme of the OGSA-DAI implementation [19]. Activities provide a way to add functionality that is not provided by OGSA-DAI by allowing customized functions to be called in response to application-specific queries passed through the DAI interface. In order to layer MCS on top of OGSA-DAI we defined an MCS-specific activity that contains functions corresponding to the MCS API. The activity includes functions that can ,add, delete and query logical items, collections or views and their associated attributes.

This section describes implementation issues related to layering MCS on top of the OGSA-DAI.

A. *Extending OGSA-DAI*

In order to add a new activity, an XML schema for the activity has to be specified and the implementation of the activity has to be provided. Currently the activities that can be performed over a data resource need to be specified in a configuration file prior to service initialization. The OGSA-DAI Grid Service instances can execute the activities. We implemented a new activity called *mcsActivity* that includes the definition and implementation of the MCS API. The results returned by the activity are also in XML format.

OGSA-DAI supports both synchronous and asynchronous activities. *mcsActivity* is implemented as a synchronous activity. The results are returned in the response sent to the client. The client blocks until the response is received.

B. *Support for Authorization*

OGSA-DAI supports authorization by mapping users to database roles. This authorization model is useful in general but does not provide the granularity of control required by MCS. Thus we have implemented an authorization model in MCS which provides finer grained control, at the level of logical objects. The following are various permissions that a user can have in MCS

- MCS create permission: this permission is required in order to create a logical item, collection or view. A user having this privilege can grant other users similar privilege. Initially this permission has to be granted out of band to one user, who can then grant permissions to others.
- Write permission: A user can have write permission on a particular logical item, collection or view. Write permission on any object (logical item, collection, or view) allows the user to modify attributes of that object, to grant (or revoke) read or write permission on that object to (or from) other users. In addition, write permission on a collection or view allows the user to add objects to and delete objects from that collection or view, and write permission on a collection also conveys write permission on all logical objects in the collection. The creator of a logical object is granted write permissions over it.
- Read permission: A user can have read permissions on a particular logical item, collection or view. Read permissions allow the user to query the attributes of the object.

One of the features of the authorization model is that permissions on a logical collection are also valid on the objects in the collection. Thus a user having write permissions over a collection automatically gets write permissions over the objects under the collection.

The Distinguished Name (DN) from the certificate that the user presents for authentication identifies a user in MCS. If the user does not use any authentication in accessing the Grid Data Service, then the user is mapped to an anonymous DN. The union of the permissions granted to the anonymous DN and the user's DN is considered while evaluating the authorization for the user.

C. *MCS Client-side Tools*

The OGSA-DAI client side tools can be used to access the MCS Grid Data Service. The OGSA-DAI client side tools take the perform document as an argument. New operations can be added under the MCS Grid Data Service. Only the schema for the new operations needs to be published. The user needs to compose the required perform document based on the new schema.

We developed a wrapper around the OGSA-DAI client code to expose a simple API interface. Each operation that can be invoked using *mcsActivity* is exposed as a method call in the MCS-API. We have also developed command line tools. When using the MCS-API the user need not be aware of the syntax of the XML perform documents exchanged between the user and the MCS Grid Data Service.

Using the OGSA-DAI framework provides a very convenient mechanism for extending the MCS schema and executing arbitrary SQL operations over the extended schema. The *relationalResourceManager* activity allows the users to add new tables in the MCS schema. The *sqlQueryStatement* and *sqlUpdateStatement* activites can be used to execute SQL statements on the newly defined tables. We have used that functionality to explore various alternative implementations of the extensible schema.

D. *Support for a Extensible Attribute Set*

The key to representing the metadata is to capture the common attributes while making it possible to add additional attributes that represent domain-specific metadata. One way to achieve this within our relational technology-based implementation, is to create a basic table that contains the common attributes as table columns and then create additional tables for a fixed set

of predefined attribute types, such as integer, string, float, etc. In our implementation we support 6 different attribute types: String, Integer, Float, Date, Time, and DateTime. Each attribute-type table (static attribute table) contains three columns: object id, attribute name and attribute value. When attributes are added to an object, the name and value of the attribute are placed into the appropriate table along with an object id that identifies the row in the common attribute table this new attribute belongs to. All of the attributes for an entry can then be found by searching all tables for entries with matching object identifiers.

This table set up is simple and allows for an easy addition of new attributes by adding rows to the table. However, as the size of the database grows into millions, the attribute tables can grow large. For example, for a 5 million database size, if each logical item has 10 attributes, and 5 of them are of type string, then the string table will grow to 25 million rows. Obviously searching such a large table can become inefficient.

An alternative implementation we considered is to represent the domain-specific attributes in individual tables (*dynamic attribute tables*). When a new attribute, (identified by a name and a type) is created, then a new table is created. Subsequently, if the attribute is used to describe data objects, the value of the attribute and the corresponding data object are entered in the table. This approach considerably reduces the size of the tables. However, it increases the number of tables that need to be searched to one per attribute (name, type) pair rather than one per attribute type. However, this organization eliminates the need to do a join across multiple entries within a table as only one entry per object id will be found in each table. The drawback of this approach is that more tables will have to be stored in the database. Additionally there is no efficient way to find all the attributes of a particular object, because all the dynamic attribute tables may need to be searched. As a result we created an additional table that contains records consisting of the object id, attribute name and attribute type.

VII. Performance Study

With this study we aimed to address two issues:

1. How to most efficiently support a flexible schema with a variable number of attributes? Which approach: the static or dynamic attribute tables, provides efficient addition and deletion of attributes of a particular object as well as discovery of objects based on their attributes?
2. What overheads are being imposed by grid services vs. web services? Obviously, adding features such as authentication must require additional server-side processing.

To evaluate the alternative table designs, we performed a series of comparisons measuring the performance of various MCS APIs with the two schemas.

To address the second issue of web service versus Grid service overheads, we compare the two MCS implementations. We measured the performance of representative MCS APIs with both versions of MCS. In this case we used the fixed attribute table schema.

A. *Experimental Setup*

We experimented with databases of three sizes. For each size, we created logical collections with 1000 data items per collection. With each item, we associated 10 user-defined attributes of different types (string, float, integer, date and datetime), and in some cases we evaluated the performance of the system as a function of the number of attributes. Likewise, we associated 10 attributes with each collection. We loaded databases with a total of 100,000, 1,000,000, and 5,000,000 data items and their associated collections and attributes. Since we maintained a constant 1000 items per collection, there were 100 collections in 100,000 entry database, 1000 collections in the 1 million entry database, and 5000 collections in the 5 million entry database.

In all the results below we evaluate the performance of 4 critical metadata operations: add, simple query, complex query, and get user attributes. The add operations add a logical item with ten associated user-defined (dynamic) attributes of various types. To maintain the size of the database, we follow each add operation with a delete operation. The simple query operation does a value match for a single static attribute associated with a data item. The complex query operation does value matches for all ten user-defined attributes associated with a data item. Get user attributes returns the user-defined attributes of a particular data item. The rates of operations per second are measured at the client-side.

The MCS was installed on a dual-processor 2.2 GHz Intel Xeon workstation running RedHat Linux 8.0. The web-services-based MCS is built upon the Apache Jakarta Tomcat 4.1.24 server and the OGSA-DAI based implementation uses OGSA-Dai v. 3.0.2. Each implementation uses MySQL 3.23.49 relational database. We built indexes on item names, collection names and views (not used for these performance tests). We also built indexes on the database-assigned identifiers for these items and on (name,ID) pairs.

B. *Optimizing and Supporting a Dynamic Attribute Set*

Before we present the performance results, we need to touch upon the issue of the number of different attribute names. In our tests, we synthetically populated the attributes for the data objects drawing names based on a uniform distribution over the set of names. The values were partially related to the names in case of string attributes and we used the current date and time for the date time attributes. In the case of the static table schema (a table for each attribute type and object type), the number of different attribute names does not affect the size or number of the tables. However, in the dynamic attribute tables implementation, the number of tables (one for each attribute name and type) can vary based on the number of different attribute names. Obviously, the size of the tables is affected then as well. Based on our experiences with climate modeling applications (ESG, [20]) we chose for our study a set of a 1,000 and 5,000 different attribute names. Other applications such as

gravitational wave physics [21] usually have fewer distinct attribute names.

In our previous work, the 5 Million database size showed the greatest sensitivity to complex queries. Thus for the study of different schema implementations we focus on that database size.

Fig. 4 shows the performance of the add operation for the static and dynamic schema implementation with the 1,000 and 5,000 attribute namespaces. We varied the number of clients performing the adds from 1 to 24. We measured the number of operations per second that could be sustained on the client-side. We notice that the static schema is on the average 1.7 times faster than the dynamic schema. In the latter case, each time a new attribute is added (up to 1,000 or 5,000 depending on the size of the attribute namespace) a new table is created. This is pure overhead on top of the operation of adding a row into a table.

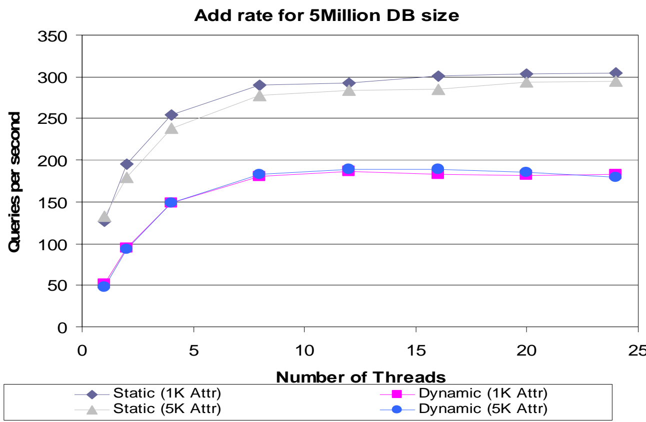

Fig. 4 Performance of Add Operation. The Database Size is 5 Million Items. The number of client threads is varied.

Fig. 5 shows the performance of simple queries, as the number of clients is varied from 1 to 12. The performance for both schemas is about the same. The static schema is on the average 9% better for 1,000 attribute namespace, whereas the dynamic schema is 3.5% better on average for 5,000 namespace.

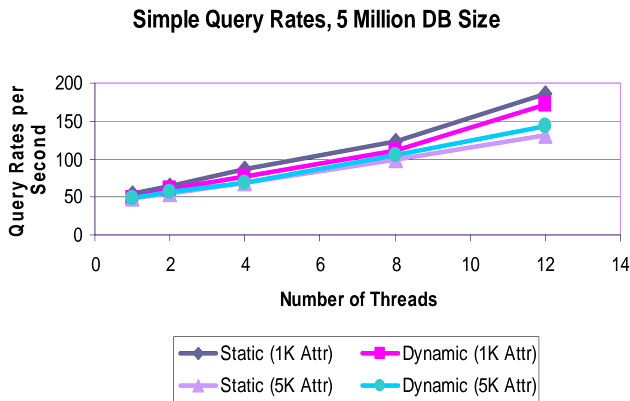

Fig. 5: Simple Query Performance for the Static and Dynamic Schemas.

Fig. 6 shows the complex query performance for the static and dynamic schemas. We notice that the dynamic solution is 1.5 times better on average than the static schema solution. This is because in the static schema, the operation needs to search for the attributes in very large attribute tables. In the dynamic case, there are more tables to be searched, but the tables themselves are smaller.

Fig. 7 shows the performance of the query that returns all the user-defined attributes of a given data item. Because the complex query first needs to search the table that contains records consisting of the object id, attribute name and attribute type and then queries individual tables. As a result the static solution is 1.8 times better on average for the 1,000 attribute namespace and 2.7 times better for the 5,000 namespace.

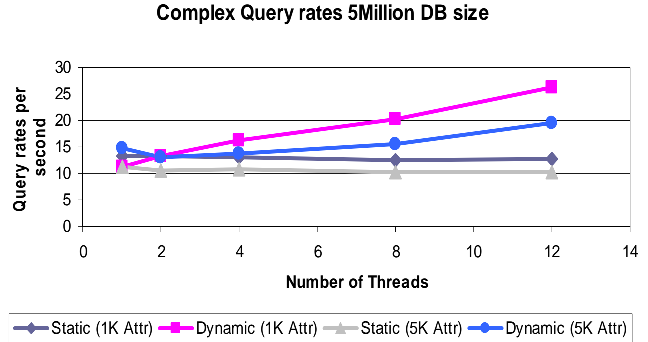

Fig. 6: Complex Query Performance, measured in queries per second.

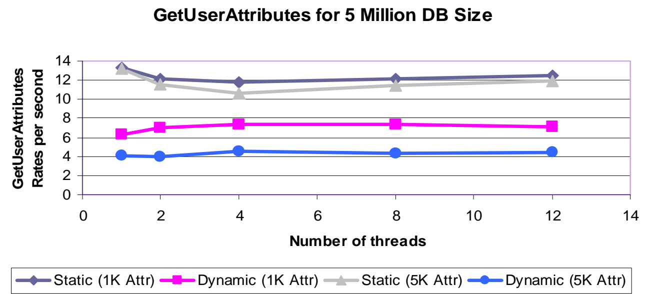

Fig. 7: Performance of the Operation that returns all the user-defined attributes of a given item.

In summary, it is not obvious which schema, dynamic or static, is better. The performance of the dynamic approach depends on the number of unique attribute names. Results in using 1,000 and 5,000 different attribute names, show that the static schema generally performs better for add operations and for querying all user-defined attributes. The dynamic schema performs better for complex queries that match 10 attributes for an increasing number of requesting threads. The two schemes show similar performance for simple queries and for complex queries that match a varying number of attributes. The choice of schema may depend on the expected operation workload for the MCS. We assume that query operations will be the more frequent than add operations. Additionally, we speculate that value matching of desired attributes will occur more frequently than querying the attributes a given data item. Under these assumptions, the dynamic schema solution would be beneficial, especially if there are not too many distinct attribute names.

C. Evaluating Grid Service versus Web Service Performance.

In this section we compare the performance of the MCS implementation layered on top of a standard web service and on top of OGSA-DAI. In all the following experiments we have used the static schema. For this study we varied the size of the database from 100,000 to 1 and 5 Million. In general, because OGSA-DAI performs GSI-authentication and the web service does not, we expect to see better performance from the latter.

Table 9 compares simple query rate performance. Since simple queries are very efficiently handled by the native database (as we have seen in out previous work [17]), this experiment clearly exposes the overhead of the services. We can see that the grid service performs an order of magnitude worse than the web service for all the database sizes.

# of Threads	100K items, web service	100K items, OGSA -DAI	1Million items, web service	1Million items, OGSA-DAI	5MiIlion items, web service	5Million items, OGSA-DAI
1	76.72	8.13	70.91	8.13	70.13	7.22
2	90.05	11.81	93.19	11.81	91.36	11.20
4	111.37	14.55	100.30	14.55	110.31	13.30
8	91.40	13.46	98.87	13.46	98.03	14.71
12	93.35	14.63	94.52	14.63	94.92	15.17

Table 1: Performance of Simple Queries for Various Database Sizes. The table shows the number of queries per second.

# of Threads	100K items, web service	100K items, OGSA -DAI	1Million items, web service	1Million items, OGSA-DAI	5MiIlion items, web service	5Million items, OGSA-DAI
1	31.79	5.38	26.97	5.04	11.06	5.55
2	52.24	10.11	35.72	9.20	11.59	11.37
4	62.88	12.47	40.32	11.74	11.42	12.99
8	67.10	14.02	41.45	13.55	10.65	14.28
12	92.82	14.76	36.67	14.59	10.53	14.67

Table 2: Complex Query Rates for Various DB Sizes. The table shows the number of complex queries per second.

The web-service based MCS better than OGSA-DAI-based MCS for all database sizes: 8 times for 100K database size, 13.5 times for 1Million database size, and 13 times for the 5Million database size. We plan to increase the number of threads further in the final version of the paper.

Because complex queries are more costly than simple queries in terms of database performance, the differences between the web service and OGSA-DAI are not as significant as in the simple query case. Also, as the size of the database increases, the average performance difference decreases (Table 2). For the 100K DB size, web-service-based MCS is over 5 times faster on average than the OGSA-DAI solution. Interestingly, for the 5 Million DB, as the number of threads increases, OGSA-DAI outperforms the other solution by a factor of 1.4.

The final set of results compares add performance. Table 3 shows that comparative add rates do not vary much with database size or the number of the client threads. The web-service-based MCS is on the average 2.6 times faster in terms of adds per second than the OGSA-DAI-based MCS.

In summary, we see that OGSA-DAI performs worse than a web-service. However, the implementations are not directly comparable, since OGSA-DAI performs authentication. Also, it is very interesting to see that OGSA-DAI performs well when the number of client threads is high and the database size is large, indicating better scalability than the web-service based version. Additionally, a new version of OGSA_DAI is about to be released. The new version promises performance improvements. We will evaluate the new version for the final version of this paper.

# of Threads	100K items, web service	100K items, OGSA -DAI	1Million items, web service	1Million items, OGSA-DAI	5MiIlion items, web service	5Million items, OGSA-DAI
1	33.96	18.45	32.75	17.00	31.18	18.30
2	48.78	25.66	46.03	23.94	43.61	24.16
4	59.19	26.60	55.93	26.21	54.27	25.24
8	63.65	26.26	62.88	26.03	60.95	25.63
12	66.22	24.19	65.14	24.75	63.39	24.76
16	65.96	23.49	66.07	23.76	60.56	22.83
20	69.54	21.80	68.28	22.23	61.84	21.23
24	71.29	18.26	69.66	14.54	69.45	16.38

Table 3: Add Rate Performance for Various DB Sizes. The table shows the number of adds per second.

VIII. APPLICATION USE CASES

We have used the web service-based MCS in a variety of applications. We are currently in the process of transitioning to the use of the OGSA-DAI implementation. MCS has been used in Earth System Grid (ESG) application [22], in the Pegasus workflow management system [23, 24], and others.

ESG is a climate modeling application. MCS was used as one component in an ESG demonstration that included replica management and storage management services as well as various data storage services. The ESG metadata followed the netCDF convention and was stored metadata in XML format. To store ESG metadata in MCS, we added user-defined attributes to the MCS to correspond to application-specific ESG metadata attributes as well as Dublin Core attributes. Then we parsed or “shredded” the XML metadata files to extract individual attribute values and stored these. The ESG

metadata included 154 user-defined attributes for data items and 27 attributes for collections.

Pegasus [25] is a planning component developed within the GriPhyN project (www.griphyn.org) [21]. Pegasus is used to map complex application workflows onto the available Grid resources. One of the applications that uses Pegasus is LIGO. LIGO (Laser Interferometer Gravitational-Wave Observatory) is a project that seeks to directly detect the gravitational waves predicted by Einstein's theory of relativity. Pegasus uses MCS to discover existing application data products. When the Pegasus planner receives a user request to retrieve data with particular metadata attributes, it queries the MCS to find all logical files with the corresponding properties. For example, a user might request all logical files that corresponding to a particular frequency band, and the MCS will return a list of relevant files to the Pegasus planner. When the workflow generated by the Pegasus planner results in creation of new application data products, Pegasus uses the MCS to record metadata attributes associated with those newly materialized data products. To support LIGO application-specific metadata, we added 23 user-defined attributes to the pre-defined attributes provided by the MCS schema.

MCS is also used by Pegasus to store provenance and performance information about the workflow components that have been executed on the Grid and up-to-date information about a workflow being executed. The information describes the various executables used in the workflow execution, the time it took to execute the workflow components, including the time to perform data movement. MCS can also provide users with various levels of detail regarding a set of workflows or particular workflow instances. As such MCS is used in conjunction with the Pegasus portal by the Montage application [26, 27], an astronomy application that delivers science grade mosaics of the sky on demand.

IX. Conclusions and Future Work

In this paper we described, MCS, a metadata catalog service for metadata management on the Grid. We described the requirements that have driven the design of MCS. We presented the data model, the authorization model and the API used to interact with MCS. We discussed alternative schema designs that can support a dynamic user-defined attribute set. Finally, we evaluated the performance of two alternative schemas and the overhead imposed by a grid service-based implementation in comparison to a web service-based version.

In this work we have focused on a centralized metadata service design. However, in distributed systems, it is often necessary to distribute services to provide reliability and good performance. In our future work we plan to investigate the feasibility of distributing MCS, exploring issues of federation of multiple services in a Grid environment.

Acknowledgements

This research was supported in part by the National Science Foundation under grants ITR-0086044(GriPhyN) and ITR AST0122449 (NVO) and the DOE Cooperative Agreements: DE-FC02-01ER25449 (SciDAC-DATA) and DE-FC02-01ER25453 (SciDAC-ESG.)

References

[1] B. C. Barish and R. Weiss, "LIGO and the Detection of Gravitational Waves," *Physics Today*, vol. 52, pp. 44, 1999.

[2] C.-E. Wulz, "CMS - Concept and Physics Potential," Proceedings of Second Latin American Symposium on High Energy Physics (II-SILAFAE), San Juan, Puerto Rico, 1998.

[3] "NVO," 2004. http://www.us-vo.org/

[4] I. Foster, et al., "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing Applications*, vol. 15, pp. 200-222, 2001.

[5] MCAT, "MCAT - A Meta Information Catalog (Version 1.1)." http://www.npaci.edu/DICE/SRB/mcat.html

[6] I. Foster, et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.," Open Grid Service Infrastructure WG, Global Grid Forum 2002.

[7] "TeraGrid." http://www.teragrid.org/

[8] "International Virtual Data Grid Laboratory (iVDGL)." http://www.ivdgl.org

[9] V. Welch, et al., "Security for Grid Services.," Proceedings of Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), 2003.

[10] K. Czajkowski, et al., "A Resource Management Architecture for Metacomputing Systems," in *4th Workshop on Job Scheduling Strategies for Parallel Processing*: Springer-Verlag, 1998, pp. 62-82.

[11] W. Allcock, et al., "Data Management and Transfer in High-Performance Computational Grid Environments," *Parallel Computing*, 2001.

[12] A. Chervenak, et al., "Giggle: A Framework for Constructing Sclable Replica Location Services.," Proceedings of Proceedings of Supercomputing 2002 (SC2002), 2002.

[13] "Globus Toolkit 3." http://www.globus.org/ogsa/

[14] V. Raman, et al., "Data Access and Management Services on Grid," GGF, DAIS group 2002.

[15] C. Baru, et al., "The SDSC Storage Resource Broker," Proceedings of Proc. CASCON'98 Conference, 1998.

[16] I. Foster, et al., "Giggle: A Framework for Constructing Scalable Replica Location Services.," Proceedings of SC 2002, to appear, 2002.

[17] G. Singh, et al., "A Metadata Catalog Service for Data Intensive Applications," Proceedings of Supercomputing (SC), 2003.

[18] "Apache." http://xml.apache.org/

[19] N. Hardman, et al., "OGSA-DAI: A look under the hood: Part 2: Activities and results," 2004. http://www-106.ibm.com/developerworks/grid/library/gr-ogsadai2/

[20] I. Foster, et al., "The Earth System Grid II: Turning Climate Datasets Into Community Resources," Proceedings of Annual Meeting of the American Meteorological Society, 2002.

[21] E. Deelman, et al., "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," Proceedings of 11th Intl Symposium on High Performance Distributed Computing, 2002.

[22] ESG, "The Earth Systems Grid." http://www.earthsystemsgrid.org

[23] E. Deelman, et al., "Pegasus : Mapping Scientific Workflows onto the Grid," Proceedings of 2nd EUROPEAN ACROSS GRIDS CONFERENCE, Nicosia, Cyprus, 2004.

[24] E. Deelman, et al., "Workflow Management in GriPhyN," in *Grid Resource Management*, J. Nabrzyski, J. Schopf, and J. Weglarz, Eds., 2003.

[25] E. Deelman, et al., "Pegasus: Planning for Execution in Grids," GriPhyN 2002-20, 2002.

[26] R. Williams, et al., "Multi-wavelength image space: another Grid-enabled science," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 539-549, 2003.

[27] B. Berriman, et al., "Montage: A Grid-Enabled Image Mosaic Service for the NVO," Proceedings of Astronomical Data Analysis Software & Systems (ADASS) XIII, 2003.