Published in final edited form as:

Proc IEEE Int Conf Escience. 2017 October ; 2017: 510–517. doi:10.1109/eScience.2017.83.

ERMrest: A Collaborative Data Catalog with Fine Grain Access Control

Karl Czajkowski, Carl Kesselman, and Robert Schuler

Information Sciences Institute, Viterbi School of Engineering, University of Southern California, Marina del Rey, CA 90292

Abstract

Creating and maintaining an accurate description of data assets and the relationships between assets is a critical aspect of making data findable, accessible, interoperable, and reusable (FAIR). Typically, such metadata are created and maintained in a data catalog by a curator as part of data publication. However, allowing metadata to be created and maintained by data producers as the data is generated rather than waiting for publication can have significant advantages in terms of productivity and repeatability. The responsibilities for metadata management need not fall on any one individual, but rather may be delegated to appropriate members of a collaboration, enabling participants to edit or maintain specific attributes, to describe relationships between data elements, or to correct errors. To support such collaborative data editing, we have created ERMrest, a relational data service for the Web that enables the creation, evolution and navigation of complex models used to describe and structure diverse file or relational data objects. A key capability of ERMrest is its ability to control operations down to the level of individual data elements, i.e. fine-grained access control, so that many different modes of data-oriented collaboration can be supported. In this paper we introduce ERMrest and describe its fine-grained access control capabilities that support collaborative editing. ERMrest is in daily use in many data driven collaborations and we describe a sample policy that is based on a common biocuration pattern.

I. Introduction

Increasing attention is being paid to the need to create data that is Findable, Accessible, Interoperable, and Reusable (FAIR) [1]. A significant aspect of FAIR is that “to be findable (F) or discoverable, data and metadata should be richly described to enable attribute-based search.” This leads one to ask how these descriptions are obtained and maintained. While some descriptive elements will be properties of the data, potentially carried as metadata elements in rich file formats such as HDF, or in header elements as in TIFF, truly rich descriptions will be dependent on the details of the data, the context it was collected in, and its relationship to other data.

In traditional data publication environments, there is often a curator whose role is to define, populate and maintain important attributes in a data catalog (e.g. [2]–[4]). However, as data volume, complexity and rates of production continue to grow, expecting a dedicated curator for every important data element is not viable, both in terms of cost of generation and the time and overhead associated with curation. In practice, it will be up to the teams that generate and reuse data to maintain the descriptions of the data for purposes of discovery

and reuse within a collaboration, as well as to streamline the process of publication into widely shared repositories. If we accept this premise, we must then consider the question as to how scientific teams can collaboratively create and maintain potentially complex data organizations, including source data, descriptive metadata and additional information for structuring data into complex, multidimensional data sets.

In previous work [5], we have shown how a catalog service built on the entity-relationship model that dynamically adapts to diverse and evolving data models can streamline the process of collaborative data management and promote the creation and reuse of FAIR data. In this paper, we focus on the role of access control in the collaborative creation of data and metadata. Our hypothesis is that collaborative data curation of complex relational data by domain scientists can be achieved securely through policy specifications defined over the resources of the relational model generically without requiring application-specific policy definitions.

This paper makes the following contributions. We:

- briefly describe a collaborative data catalog for richly structured relational data and describe the design goals and potential pitfalls with respect to complex policy management in this context;
- summarize the features for policy management in the collaborative data catalog service and introduce the key concepts for specifying policy;
- present an exemplar of a policy for collaborative data curation in a multi-user data management use case.

The rest of this paper is organized as follows. In Section II, we introduce the ERMrest collaborative data catalog, and in Section III we outline its policy management capabilities. In Section IV, we present an overview of the system implementation. In Section V, we describe an exemplar policy for “self-serve” data curation in a collaborative scenario, which we have found to be a common use case. We review related work in Section VI and offer conclusions in Section VII.

II. ERMrest

ERMrest (Entity Relationship Model via Representational State Transfer) is a relational data service for the Web, and allows general entity-relationship modeling and manipulation of data resources by RESTful access methods. ERMrest serves as the metadata catalog of Deriva [5] and enables the evolving and dynamic data models needed for describing, contextualizing, and linking scientific assets. We designed ERMrest to serve as the hub of collaborative data science endeavors analogous to how a wiki serves as a hub for collaborative documentation projects. Using ERMrest, researchers can share data and data models and build data collections iteratively as their research matures and as they refine their conceptualization of their particular investigation.

ERMrest enables non-expert users to create and evolve data models that represent the semantic concepts in their domain. For example, a set of assets (i.e., data files) might be

related in that they were generated from a specific biological sample, and in turn a set of samples may be aggregated to form an experiment. Many use cases can map into simple models with only a handful of entities and relationships and non-experts can easily think about their domain in terms of the main concepts that they want to manipulate. By providing methods for incrementally creating these data models, and by allowing users to express domain concepts directly in the catalog, ERMrest offers a platform where the data models can be created and maintained by the user community.

ERMrest is designed as a RESTful Web service. ERMrest maps *Entity*, *Attribute*, *Schema*, *Table*, *Column*, and other relational concepts to Web resources, which are referenced by Web resource identifiers (i.e., URIs). It supports the following REST interfaces:

- *catalog*: reference, retrieve, create, alter, delete “catalog” resources, each of which is an independent relational data store;
- *schema*: reference, introspect, and alter entity-relationship models (i.e., schema name spaces, table and column definitions, key and foreign key constraints, etc.);
- *entity*: reference, query, and manipulate entity records (i.e., rows of a table);
- *attribute*: reference, query, and manipulate projected attribute records (i.e., subsets of attributes of relations);
- *attributegroup*: reference and query projected attribute group records (i.e., attributes grouped by a subset of attributes of relations);
- *aggregate*: reference and query projected aggregates with supported *aggregate functions* such as count, min, and max.

A. Design Goals

Within ERMrest, we provide fine-grained control of operations on any resource, differentiating rights of one user from another in a shared system. The design goals of the ERMrest policy facilities may be categorized as follows:

1. **Visibility.** Restrict the visibility of catalog contents, e.g., hide an entire catalog, a table, a column of a table, or specific rows of a table.
2. **Modification.** Prevent modification of some content, e.g., add/remove columns of a table, add/remove table constraints, insert/update/delete rows, or change some but not all columns of a row.
3. **Delegation.** Delegate some rights within a community, e.g., authorize additional owners for sub-resources, delegate creation of new schema, or delegate creation of new tables.
4. **Manageability.** Make sure simple policies are still simple to manage, e.g., make an entire catalog visible to a group, or make an entire table mutable by one group.

B. Potential Pitfalls

Defining and implementing complex policies in a richly structured relational data service such as ERMrest and implementing facilities to enforce complex policies imposes some significant complexities that must be resolved in order to provide a consistent yet usable policy model. For example, consider data visibility.

- Most forms of data access depend on access to other resources within a catalog. For example, before you can compose a data access operation, you have to interrogate the schema to understand what data is available, you may need to access related data in order to figure out how to make sense of reference constraints, or you must see the current policy to make sense of what access mechanisms are available.
- References between entities can potentially expose “hidden” data. For example, a policy may restrict access to a row in a table, but if that row is referenced via a foreign key from another visible table, the referenced values in the hidden row are exposed.
- Integrity constraints can potentially expose “hidden” columns. For example, consider a table on which access to a specific attribute (column) by a user is not allowed by a user who is allowed to insert rows into the table. Inserting a new row by the user that is not permitted to see such a column may result in a uniqueness constraint violation error that could indirectly leak information (i.e., the existence of the column).

At present, policy administrators must be aware of these potential pitfalls and reconcile most of these problems by thoughtfully correlating policy specification with data model definitions. In the future, we expect to improve usability by automatically detecting conflicts and flagging potential issues and/or reconciling them if possible without manual intervention.

III. Policy Specification

Policy in ERMrest is defined by a set of *access control lists* (ACLs). ACLs may be specified to govern any type of resource in the data catalog, i.e., whole catalog, a schema, table, column, reference, or individual rows. At a minimum, the catalog ownership ACL must be specified.

A. ACL Policy Statement

The ACL is the basic statement of a policy specification in a data catalog. An ACL takes the form:

```
ACL = (resource, permission, roles)
```

where:

- `resource` is the model-level resource to which the ACL is attached in order to identify the scope of the access governed (i.e., a schema, table, column, or reference);
- `permission` indicates the access permission (e.g., insert, update, etc.) granted by the ACL; and
- `roles` indicates a set of user or group identities or the wild card ‘ * ’ which matches any role. If the roles are empty, then the specified permission for that resource is denied to everyone.

As an example, `(foo.bar, write, curator)` would “grant ‘write’ permission on table ‘foo.bar’ to the ‘curator’ group.”

B. Static vs. Dynamic ACLs

Two forms of ACLs are supported by the policy specification: *static* ACLs and *dynamic* ACLs.

A *static ACL* is a *name-based access control* statement, meaning that it expresses an access control policy directive without regard to the contents (i.e., data) of the catalog but instead identifies the resources based on their names (i.e., table name, column name, etc.). For example, the policy administrator may wish to limit access to a particular `table` resource in the catalog to a certain user role defined in the policy. The statement instructs the service to restrict access independent of the contents (i.e., rows and attribute values) of the table.

A *dynamic ACL* is a *content-based access control* statement, meaning that it expresses an access control policy directive based on the contents of the catalog. For example, the policy administrator may wish to limit write-access to certain *rows* of a particular table resource based on the value of each row’s `creator` column. The policy could state that the current user role must match the value of the `creator` column of a row in order to update the values of those matching rows.

In a dynamic ACL, the `roles` property in the `(resource, permission, roles)` statement is specified as a *query over the catalog that returns a list of roles* rather than a statically defined list of roles in the policy specification itself. In this way, the effective role of the ACL may be computed dynamically for every row, column, or constraint based on the results of a query.

C. Resources and Inheritance

Each ACL is associated with a specific resource in the model hierarchy: catalog, schema, table, column, and constraint. In the absence of any additional ACLs, the access rules for a resource are inherited from the rules that are in force for the *parent* resource in the hierarchy. For example, unless further restricted, policy access to a column in a table will be the same as the policy for access to the table itself. Inheritance of ACLs simplifies the overall policy specification and eliminates redundancy and manual effort.

Policy administrators can *restrict* an inherited ACL by specifying an ACL of the same permission but with an empty `roles` list. For example, if most tables in a schema allow *write* permission, it is simpler to grant write on the schema, e.g., `(foo, write, curator)` and then *restrict* it at the table level, e.g., `(foo.baz, write,)`.

In general, inherited policy can be restricted or extended by adding an ACL to a sub-resource. For example:

- A resource policy can *restrict* access compared to the parent resource; e.g., one table is read-only in an otherwise read-write schema, or one column is hidden in an otherwise read-write table;
- A local policy can *broaden* access compared to the parent resource; e.g., one table is writable in an otherwise read-only schema, or one column is mutable in an otherwise read-only table;
- A more specific access mode can be *extended* to more clients; e.g., one table is writable by a curator group and another group can only insert new rows, or one column is writable by a curator group, and another group can only set values during row insertion.

D. Permissions

Operations on a catalog resource are governed by specific permissions:

- `owner` is a special permission that subsumes all other permissions defined below. In addition, only users with the owner permission for a resource are allowed to create, edit, or delete the ACLs for that resource;
- `create` a new child model element in a catalog or schema, e.g., add a new table definition;
- `enumerate` existing model elements in a catalog or schema, e.g., view the definition of a table;
- `select` or read existing data; and
- `write` data, e.g., change rows or attribute values. Write operations can be further refined into narrower permissions: `insert` new data; `update` existing data; and `delete` existing data.

The precise meaning of some permissions varies by resource type (see Table I for additional details). Reference (i.e., constraint) resources, which are used to describe a relationship between two entities, differ from row and column-level access controls. Dynamic ACLs on rows and columns can be expressed in terms of current values, while dynamic ACLs on references are applied to the proposed new value; e.g., to restrict what entities can be related to one another.

E. Policy Scoping and Resource Dependencies

An ACL specified on a resource (e.g., table, column, etc.) governs access to that resource.

Because many catalog operations depend on access to multiple resources, a complete request

will be granted or denied based on the conjunction of ACLs governing each resource involved in the operation.

For example:

- Joined relations can only be viewed if all involved relations (i.e., rows) can be viewed.
- Rows can only be viewed if their table is visible.
- A table can only be accessed if its schema is visible.
- A schema can only be accessed if its catalog is visible.
- A column can only be modified if its (reference) constraints allow the new value.

F. Model-level Policies

Here we describe some of the details and nuances for defining policies on different resources in the hierarchical catalog model.

1) Catalog ACLs—A catalog-level ACL describes the access permitted on the whole catalog. Catalogs do not inherit ACLs from elsewhere, so an unspecified catalog policy is not allowable. During catalog creation the *owner* defaults to the requesting client – no other ACLs are granted if not explicitly specified.

2) Schema ACLs—A schema-level ACL describes the access permitted on the whole schema. When not specified, the effective schema-level ACL is inherited from the catalog. During schema creation, the *owner* defaults to the requesting client. The owner will not be explicitly specified if the client is also an owner of the catalog at the time of creation. All other schema ACLs are unspecified by default, leaving it to inherit the policy from the catalog resource.

3) Table ACLs—A table-level ACL describes the access permitted on the whole table. When not specified, the effective table-level ACL is inherited from the schema. During table creation, the *owner* defaults to the requesting client. Again, if the client is also an owner of the schema at the time of table creation then the owner is only implied rather than explicitly specified. All other table ACLs are unspecified by default, thus inheriting from its schema resource. Dynamic ACLs (see Section III-B) can augment static ACLs at the table-level to enable access to only a subset of its rows. All static and dynamic table ACLs are disjunctively considered when deciding row access.

4) Column ACLs—A column-level ACL describes the access permitted for a given column of a table. When not specified, the effective column-level ACL is inherited from the table. Columns are considered part of their table resource and do not have separable ownership. Dynamic ACLs (see Section III-B) can augment static ACLs at the column-level to enable access to only a subset of column values throughout a table.

5) Reference ACLs—A reference-level ACL specifies whether to permit foreign key references to be expressed (i.e., make references to or be referenced by other model

resources). By default, reference ACLs are granted to all (i.e., ‘ * ’) if not otherwise specified. In practice, reference ACLs are less often restricted and so this default simplifies common policy specifications. A model that needs to restrict expression of foreign keys should explicitly override the default ACLs. To completely disable expression of foreign keys, the reference ACLs for the *insert* and *update* permissions should have empty role lists. Reference constraints are considered part of the enclosing table resource and do not have separable ownership. Dynamic ACLs (see Section III-B) can augment static ACLs at the reference-level to restrict expression of only a subset of data in a column governed by that reference constraint. These ACLs are actually managed on foreign key reference constraints, but their effect is to limit what new values can be expressed in the foreign key’s constituent columns.

IV. System Implementation

The service is written in the Python language using the web.py (www.webpy.org) web framework, and is hosted in the Apache web server via `mod_wsgi` daemon processes. It uses PostgreSQL databases to back each catalog. A simple registry tracks all provisioned catalogs and their backing database connection parameters. We also add a hidden database schema, `_ermrest` in each catalog database to store internal catalog management state, including the catalog policy.

A. Authentication and Group Management

We integrate with web-based identity and attribute providers through the standard Open Authentication (OAuth) protocol in order to get web client identity and group membership. The service logic includes server-side plug-ins for different authentication and group identity providers. In most of our active deployments, we use Globus (www.globus.org) for its federated identity provider (IdP) and group management service [6]. Globus federates many campus and research IdPs, which allows users to authenticate to ERMrest using their campus accounts for example. Globus Groups complements the IdPs with user-managed and delegated group management to allow users to create and manage membership for projects. The *roles* used in the policy specification may reference the user or group identity reported by the IdP.

B. Policy Enforcement

We implement access control in the service logic, using policy stored in the management schema within the catalog so that policy, model, and data are consistently under transaction control. Databases like PostgreSQL do allow fine-grained access to be granted, but we have found that the performance on the typical types of policy do not perform well due to the nature of the query and the known limitations with PostgreSQL’s implementation of “row-level security,” which interferes with query optimization. Our approach is not to rely on a general row-level policy engine, but rather to rewrite the REST queries on-the-fly before passing them off to the query optimizer. This approach has advantages over the native row-level security approach:

1) Fast-track policy decisions—Many policy decisions can be made by the service-logic even before rewriting the REST query into a SQL statement. For example, static ACLs are fully enforceable by inspecting the model elements used in the REST query and checking against the static policy statements. If the static policy grants the necessary access mode for the resource, then the query may be executed without any further modification for policy enforcement. If no access mode is granted by the static policy, and there are no dynamic policies, then the query can be denied without further evaluation.

2) Query optimization—When dynamic policy statements are needed, the query can be rewritten and optimized to use database indexes for fast query processing. A known limitation in PostgreSQL, at the time of this writing, is that its native “row-level security” (RLS) prevents the usage of query indexes when joining tables during a query; thus the query plan degrades to a table scan so that the row-level policy may be evaluated against each individual tuple in order to avoid potentially leaking restricted data. This limitation can result in order(s) of magnitude query performance degradation.

3) Simplified deployment—SQL Views complicate native row-level security because they can leak information by circumventing RLS enforcement. When Views are created they inherit the permissions of the role that created them. In a typical web service implementation such as ERMrest, a daemon user account is used to connect to the database server. If the service were to create a SQL View on behalf of a client, then the SQL View would essentially inherit the privileges of the daemon user, which generally has an elevated level of permissions. As such, in our experiments with PostgreSQL RLS, we needed to introduce a secondary daemon account with restricted privileges in order not to circumvent the RLS policies. However, by implementing the policy enforcement in the service-logic and query rewriting, the service now can be deployed with a single daemon account which simplifies deployment significantly.

V. Example Policy: Self-Serve Curation

This section describes an example policy for facilitating “self-serve data curation” with minimal intervention required by system operators. We use the term self-serve data curation as *the curation of complex data performed directly by domain scientists*. Traditionally, data curation has been considered the exclusive domain of trained curators and information science personnel. The problem, however, with this prevailing model for data curation is *limited scalability and sustainability*. While a complete discussion of this issue is beyond the scope of this paper, we note it as a particular objective that we believe fine-grained access control over relational data services can address. We use this as a significant motivating use case and present here a policy specification used to facilitate self-serve curation.

A. Roles and Responsibilities

Here, we sketch out a baseline set of roles and responsibilities to be used in the policy for self-serve curation. Below, when we use the term “project” in a role name we mean a particular science application which may be an individual lab or a member of a consortium.

- `admin`: This role is for system administrators or operators. It will serve as the catalog owner. This would normally be the one role that non-project support staff would belong to. Because the `owner` can do anything up to and including deletion of the catalog itself, it is prudent to limit this role to qualified support staff in general, ideally those with some training in database administration.
- `<project>-curator` : This role is for a project's data curator group. The curator is generally concerned with data quality across an entire catalog. The curator may not enter data themselves but rather oversee the data entry and monitor data submissions from research participants (i.e., project PIs and their staff). The curator will however need to be able to write (insert, update, and delete) to all schemas and their tables, with exceptions as dictated by the project requirements, in order to correct mistakes or make other edits to maintain the quality of the data catalog. The curator role can be assigned to project staff, preferably with some expertise or training in either scientific data curation (e.g., a "biocurator") or informatics (e.g., a "bioinformatician"). The curator could also be granted limited schema modification permissions, such as adding a column, adding a table (mostly additive operations rather than potentially destructive ones), as curators may not be experienced database administrators and lack sufficient depth of knowledge to alter the schema of a database correctly and consistently.
- `<project>-writer` : This role is for anyone in a project who does data entry (i.e., data submission) such as investigators (PIs), research assistants, and other personnel. The writer has static insert on all schemas and their tables, again with exceptions as dictated by the project. The writers should be able to enter new data into a catalog but they should only be able to modify their own data.
- `<project>-reader` : This role is for projects that need to limit the read-only users of their catalogs (i.e., where 'anonymous' usage is not allowed).

Note that although our policy does include a somewhat traditional *curator* role, the role as defined here serves the purpose of stewardship of the catalog rather than performing of all data entry tasks. We believe that such a model can lead to more scalable and sustainable data curation efforts.

There are of course projects with special requirements that may need to be handled outside the scope of this sketch. For example:

- A larger consortium may have policies for limiting the read access of rows dynamically based on user membership in a sub-project of the consortium; or
- A data sharing repository project may wish to enforce policies for limiting read access for "work in progress" in order to avoid user confusion over seeing data that has been only partly curated. We will illustrate this scenario in the example to follow.

B. Example Relational Schema

For the purpose of this example, we will define a small schema that follows a realistic pattern for biocuration scenarios. This example database includes a schema resource called the “isa” schema which stands for the Investigation-Study-Assay (ISA) pattern common to many biocuration databases. Within the `isa` schema, we will define the following two tables:

- `dataset(id, created_by, status, ...)` : The “dataset” table represents the domain concept pertaining to collections of data in the data catalog, typically associated with a data submission activity. The table’s primary key is defined as the `id` column. The dataset table has an `created_by` column that stores the value of the user role that created the row in the table. Finally, the dataset has a `status` column, which is used to indicate the curation status of the dataset and has a domain restriction to the values: `pending` or `done`. We omit the details of other columns as not pertinent to the example.
- `assay(id, owner, dataset_id, ...)` : The “assay” table represents the domain concept of an assay (e.g., a measure captured during an experiment). It also has a primary key column named `id` and an `owner` column. In addition, it has a `dataset_id` column, which is used in a foreign key reference constraint to refer to the dataset to which this assay belongs.

The complete details of a relational schema for biocuration would, of course, be significantly more complex, but to illustrate the policy specification example these details will suffice.

C. Example Policy Specification

Here we enumerate the static and dynamic ACLs necessary to specify the policies for our “self-serve” data curation example.

- `(catalog, owner, [admin])` : grants ownership of the catalog to the members of the administrator role.
- `(catalog, enumerate, [*])` : grants the `enumerate` access mode to all users including anonymous users that are not logged in to the system (appropriate for public data sharing repositories). The `enumerate` mode permits visibility of the catalog’s data model (i.e., schemas, tables, etc.) but does not permit access to the data itself.
- `(catalog, select, [*])` : grants read-only access to the contents of the catalog to all users including anonymous.
- `(isa, write, [project-curator])` : grants full write access (i.e., insert, update, and delete) to members of the `project-curator` role. This ACL would be repeated for each writable schema by name such as `isa` for example.
- `(isa.dataset, insert, [project-writer])` : grants limited write access mode (i.e., insert only) on the `isa.dataset` table to members of the `project-`

writer role. This policy needs to be repeated for every table that will be writable by the project writers, including the `isa.assay` table.

- `(isa.dataset, write, ["created_by"])`: grants full write access (i.e., update and delete) on individual rows for which the `dataset`'s `created_by` column value (evaluated on a per-row basis) can be found in the current client's set of roles. The "query" here is specified in a short-hand form which tells ERMrest to *project* the `created_by` column of the table resource. This dynamic ACL would be repeated for the `isa.assay` table.
- `(isa.assay, select, [])`: up until this point, the read access has been granted by inheritance from the `catalog` policy, but the specification of this policy with an empty user role effectively blocks the inheritance of the read access and sets up the dynamic ACL specified next. Note that users with even limited write access can read rows implicitly.
- `(isa.assay, select, [{ "outbound": ["isa", "assay_dataset_id_fkey"] }, { "filter": "status", "operand": "done" }, { "read_role" }])`: grants read access on individual rows that satisfy the query which is specified in a structured format:
 - in this example, it specifies that the `assay` table must be joined with the `dataset` table by way of an outbound foreign key constraint named `assay_dataset_id_fkey`;
 - a filter is specified on the referenced `dataset`'s `status` column and its value must equal `"done"`;
 - finally, the `dataset`'s `read_role` column is projected and the currently authenticated user must match one of the roles returned by the query.

This dynamic ACL effectively makes the rows of the `isa.assay` table visible for users *only* if the assay belongs to a dataset that has been fully curated, as indicated by its `status` attribute.

D. Accessioning for Self-Serve Curation

While the specifics of 'accession numbers' are beyond the scope of this discussion, they are a vital part of most scientific databases. In a self-serve curation scenario, we can support data accessioning through a combination of integrity constraints, default expressions, database trigger functions, and finally policy statements. We specify policy statements to prevent insert, update, or delete of the designated accession number columns in any table that needs them. This allows the system to be the sole generator of accession numbers. During insert of the row, the default expression will ensure that an accession number generator is called in order to define the column value for the inserted row.

VI. Related Work

Research has explored topics of scientific metadata catalogs with key-value models [7]; distributed metadata catalogs with key-value models [8]; and distributed relational database access underpinning metadata catalogs [9]. However, these catalogs support a flat, per-asset description of data, and don't support the structured models that ERMrest does, nor do they provide RESTful interfaces. Research on metadata catalogs has also considered issues of flexible modeling [2], dynamic model generation and integration [3], and incorporating semantic representations [4] into metadata catalogs. While some catalogs [2], [8] provide fine-grained access control, they do so only on *collections* and *entries*.

Leading commercial and open source relational database management systems have fine-grained access control capabilities today. Oracle *Virtual Private Database* (VPD) [10] has features for specifying 'static policy' and 'dynamic policy' which are analogous to the static and dynamic ACLs in ERMrest. VPD also rewrites user queries in order to enact dynamic policies. However, defining policies in VPD requires modification of the database schema, whereas in ERMrest the policy and catalog model are separate but related. This allows policy in ERMrest to be managed and delegated by end-users. In addition, VPD requires users to write database 'functions' to implement dynamic policies, while in ERMrest dynamic ACLs are expressed simply as catalog queries.

Similarly, PostgreSQL supports fine-grained access control through its *Row-Level Security* (RLS) [11] features as discussed earlier in Section IV. Like VPD, RLS requires modification of the database schema in order to set policy, making it inappropriate for delegating management to end-users. As noted earlier, RLS suffers a major performance limitation, at the time of this writing, such that database indexes cannot be used during joins for any relations governed by RLS policies. The approach in ERMrest of rewriting queries avoids this limitation and therefore query performance benefits from the normal query optimization.

Fine-grained access control has been a topic of considerable exploration in database management systems research [12]. In [13], the authors demonstrate the limitations inherent to query modification for answering *authorization-transparent* queries; e.g., aggregate queries may be modified in such a way as to provide misleading query results (e.g., different average or count) if some of the rows are excluded due to fine-grained access control. They present an approach to rewriting queries using *authorization views* and deny queries that would have returned different results based on the policy enforcement rather than returning results that may differ. While ERMrest queries are not *authorization-transparent* by their definition, our use case for aggregate queries is to enable data *discovery*, for example through 'faceted search' interfaces, and in these use cases, it is not necessarily misleading to present partial information; e.g., count of available assays for a given filter set may be less than the total due to policy enforcement, but this is the valid results set for *that given user* based on the permissions. In [14], the authors propose *parameterized views* to encode policy restrictions. This approach is less flexible than our approach, which allows policy to be enforced on any resource, not just limited to tables wrapped by views, and ours allows end-users to administer policy without altering the database schema.

Overall, ERMrest differs from these approaches in the following key aspects: first, the entity-relationship (ER) model provides a significantly more powerful data modeling capability than the simpler key-value models supported by many scientific metadata catalogs; yet, the ER model is more understandable by end users because of its close approximation to spreadsheets, which many scientists use on a daily basis; the ER models are automatically exposed through a RESTful web services architecture capable of expressing many complex queries, unlike traditional relational database services that do not expose a web service protocol; and finally, the fine-grained access control in ERMrest is performed through efficiently rewriting queries, which not only achieves high performance but separates the policies from the database schema thus allowing policy management to be delegated to end-users rather than exclusively limited to database administrators.

VII. Conclusions and Future Work

In this paper, we have described a fine-grained authorization mechanism for a collaborative data catalog and provided an example of how it can be used to author policy for collaborative metadata editing. We are using ERMrest with fine-grained authentication on a daily basis in a range of different use cases ranging from basic scientific investigations, to community developed data repositories. To date, we have found the policy model to be powerful enough to express all of our access control requirements.

Future work will focus primarily on the usability issues associated with policy management. We would like to add new features to ERMrest's API to expose summaries of record-level rights so that clients can anticipate fine-grained privileges without having to blindly attempt resource-modifying requests. Complimenting this, we plan to extend our user interfaces to consume record-level rights summaries so that we can tailor the functions in the interface to only present operations that will be valid with respect to the model, for example, don't show an edit button for a table, if you are not allowed to edit that table. Finally, we need tools for policy authoring and analysis to allow scientists and other responsible staff to effectively customize the authorization system while trying to minimize the risk of inadvertent data release or access violations.

All Deriva tools (deriva.isrd.isi.edu) including ERMrest are open-source and are publicly available on GitHub (<https://github.com/informatics-isi-edu>).

Acknowledgments

The work presented in this paper was funded by the National Institutes of Health under awards 5U54EB020406, 1R01MH107238-01, 5U01DE024449 and 1U01DK107350, by the National Science Foundation under award 1446112, and the GPCR Consortium.

References

1. Wilkinson, MDea. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*. 2016; (3):160018. [PubMed: 26978244]
2. Deelman, E., et al. Grid-based metadata services. *Scientific and Statistical Database Management*, 2004. Proceedings. 16th International Conference on; IEEE; 2004. p. 393-402.
3. Tuchinda, R., et al. AAAI. 2004. Artemis: Integrating scientific data on the grid; p. 892-899.

4. Wang X, et al. Semantic enabled metadata management in PetaShare. *International Journal of Grid and Utility Computing*. 2009; 1(4):275–286.
5. Schuler, R., Kesselman, C., Czajkowski, K. Accelerating data-driven discovery with scientific asset management. *IEEE 12th International Conference on eScience*; IEEE; 2016.
6. Ananthakrishnan, R., et al. Globus nexus: An identity, profile, and group management platform for science gateways and other collaborative science applications. *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*; IEEE; 2013. p. 1-3.
7. Rajasekar A, et al. iRODS primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*. 2010; 2(1):1–143.
8. Koblitz B, Santos N, Pose V. The AMGA metadata service. *Journal of Grid Computing*. 2008; 6(1): 61–76.
9. Antonioletti M, et al. The design and implementation of Grid database services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*. 2005; 17(2–4):357–376.
10. [accessed: 2017-09-08] Using oracle virtual private database to control data access. <http://docs.oracle.com/cd/B2835901/network.111/b28531/vpd.htm#DBSEG007>
11. [accessed: 2017-09-08] PostgreSQL 9.6.5 documentation: Row security policies. <https://www.postgresql.org/docs/9.6/static/ddl-rowsecurity.html>
12. Bertino E, Sandhu R. Database security-concepts, approaches, and challenges. *IEEE Transactions on Dependable and secure computing*. 2005; 2(1):2–19.
13. Rizvi, S., et al. Extending query rewriting techniques for fine-grained access control. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'04; New York, NY, USA: ACM; 2004. p. 551-562.[Online]. Available: <http://doi.acm.org/10.1145/1007568.1007631>
14. Roichman, A., Gudes, E. Fine-grained access control to web databases. *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '07; New York, NY, USA: ACM; 2007. p. 31-40.[Online]. Available: <http://doi.acm.org/10.1145/1266840.1266846>

TABLE I

Permission meaning differs per resource-type.

Perm.	Table	Column	Reference
Insert	Create row	Set value during row insert	Make reference to key (during row insert)
Update	Modify row	Replace value	Make reference to key (during row update)
Delete	Remove row	Reset to default value	N/A
Select	Read row	Read value	N/A