ERMrest: A web service for collaborative data management

Karl Czajkowski
USC Information Sciences Institute
karlcz@isi.edu

Carl Kesselman
USC Information Sciences Institute
carl@isi.edu

Robert E. Schuler
USC Information Sciences Institute
schuler@isi.edu

Hongsuda Tangmunarunkit
USC Information Sciences Institute
hongsuda@isi.edu

ABSTRACT

The foundation of data oriented scientific collaboration is the ability for participants to find, access and reuse data created during the course of an investigation, what has been referred to as the FAIR principles. In this paper, we describe ERMrest, a collaborative data management service that promotes data oriented collaboration by enabling FAIR data management throughout the data life cycle. ERMrest is a RESTful web service that promotes discovery and reuse by organizing diverse data assets into a dynamic entity relationship model. We present details on the design and implementation of ERMrest, data on its performance and its use by a range of collaborations to accelerate and enhance their scientific output.

CCS CONCEPTS

• **Information systems → Entity relationship models**; **Database web servers**; **RESTful web services**; *Digital libraries and archives*;

KEYWORDS

Metadata, data management, asset management

ACM Reference Format:
Karl Czajkowski, Carl Kesselman, Robert E. Schuler, and Hongsuda Tangmunarunkit. 2018. ERMrest: A web service for collaborative data management. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM'18).* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Increasingly, scientific collaboration is driven by an iterative process of collecting, analyzing, and disseminating large and complex data sets. Data may originate from diverse instruments, such as DNA sequencers or microscopes, external databases or as the result of computational simulations or analyses. Data are assembled into collections, subject to analyses or refinement, the results recorded, and shared with collaborators within a laboratory, consortium, or a global-scale research community. The process repeats itself, evolving and adapting as new avenues are pursued, new types of data are created, and new measurement or analytic results obtained.

Data intensive scientific discovery can be accelerated and collaboration enhanced when the data associated with the discovery are **F**indable, **A**ccessible, **I**nteroperable, and **R**eusable [14]—the so called FAIR principals. FAIR data is **F1**) findable when it is identified by a unique identifier and characterized by rich metadata that describe the details of the data and its relation to other data, **F2**) accessible via standard protocol with access control and its metadata can be accessed even if the data cannot, **F3**) interoperable by using standardized terms to describe it, and **F4**) reusable by providing accurate and relevant attributes.

FAIR principals are often thought of in the context of sharing final published data. However, there is no reason to assume that the benefits of FAIR data apply only to sharing and reuse of published data. Creating FAIR data as part of the daily process of a scientific investigation has the potential to enhance a data driven collaboration by streamlining the exchange and reuse of data across a research team, reducing the potential for error by maintaining accurate data context, enhancing provenance for reproducibility and improving the quality of data that are ultimately published.

In this paper, we propose an approach to integrate FAIR principals to the entire data life cycle — from when data are being generated and analyzed and as they are shared across scales from a research group, a consortium or global community. Supporting FAIRness as part of the data infrastructure used on a daily basis by a research team has the potential to greatly improve adoption and application of FAIR guidelines. With this goal in mind, we have developed ERMrest, a RESTful web service for collaborative data management that promotes continuous application of FAIR guidelines into daily collaboration.

ERMrest promotes FAIR data production by: providing rich metadata using an Entity-Relationship model to express relationships between diverse data elements (**F1**); offering rich access control and access to metadata via standard HTTP web service interfaces (**F2**); integrating with standardized terms defined by collaborators, consortium or communities (**F3**); and supporting dynamic model evolution so that the data presented accurately represents the current structure and state of knowledge within an investigation (**F4**). ERMrest is part of the Deriva scientific asset management system and has been applied to diverse application domains [10].

The main contributions of this paper are to:

- identify the requirements for a collaborative data management service that supports continuous FAIRness;
- present the design and implementation of the ERMrest service that meets these requirements; and
- provide performance results that show ERMrest is practical for daily use in a wide range of application domains.

In Section 2 we define the requirements for a FAIR data service. In Section 3 we describe our approach to data modeling and describe

SSDBM'18, July 2018, Italy
2018. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

ERMrest as a web service in Section 4. Application of policy is covered in Section 5 and our approach to managing update history is discussed in Section 6 and creation of identifiers in Section 7. In Section 8 we present the ERMrest architecture and implementation. Performance results are provided in Section 9. We conclude with related work and conclusions.

2 REQUIREMENTS

We consider the characteristics of FAIR data within the context of an ongoing data-driven collaboration in order to identify the features a collaborative data management service should have. Our goal is to define a data management service such that all of the data produced by a collaboration are FAIR, supporting the full life cycle of scientific data including early experiment design; early and production data acquisition; ad hoc and repeated analyses; through publication. We specifically wish to support the "long tail" of e-Science [3], where many small collaborations may each involve merely dozens of data-producing clients, dozens to hundreds of data-consuming clients during the active phases of research, and an unknown number of casual or single-use data-consuming clients in later, passive phases of scientific libraries and archives.

Data ecosystem and separation of concerns. FAIR principals specify that metadata has a lifetime that is independent from the underlying data, in that metadata be accessible even if the data are not (**F2**). In a scientific asset-management paradigm, this means that services holding metadata should be capable of tracking bulk files when they are used in research and continue tracking them even if the files are eventually discarded. However, when rich metadata catalogs are used to drive scientific work, their metadata content should also be subject to this same principle. If the metadata content is used to drive research decisions, that use should be recorded and tracked so that these decisions can be understood retrospectively, even if the metadata catalog content has in the meantime undergone changes due to ongoing collaboration and curation. Just as object stores can hold multiple versions of a file and unambiguously determine that a file or version is no longer available, rich metadata catalogs should be able to hold multiple versions of metadata and unambiguously determine when metadata is no longer available.

Pervasive naming. FAIR dictates that all data be identified by a unique and stable identifier (**F1**). Web services inherently generate URLs as identifiers, but uniqueness and stability depend on the design of each service. Most web services for storing objects provide unique URLs for each immutable version of a stored file. However, object stores may not track already-deleted files and entire services may come and go in a long-lived project setting. In contemporary distributed systems, an additional layer of resolver services are often introduced to provide *persistent identifiers* which can be used to track and resolve the current URL for known resources or unambiguously return a "tombstone" record for previously-known resources which are no longer accessible. These same requirements and goals can be applied to dynamic query results, as described in [9], and we argue that this also applies to rich metadata catalog queries. Thus, we need mechanisms to version and timestamp all data including query results, provide a unique identifier for these data, and to generate corresponding citations. Most of these requirements can be met by ensuring that stable identifiers are issued as part of each record, tracking versions of records, assigning a version-based naming scheme to record retrieval and other query results, and supporting a form of historical access such that those earlier query results can be retrieved again by name, even if the latest metadata catalog content has been changed.

Model-driven organization and discovery of assets. FAIR principles require detailed descriptions of data, leveraging standardized vocabulary terms (**F1**, **F3**). Information captured in metadata provides significant scientific context to underlying assets, e.g. recording information about events, protocols, and materials. Detailed metadata in FAIR data requires that the metadata models directly represent the concepts and relationships in the problem domain being worked in. Data and metadata may be sourced from existing data collections, outputs of computations, specialized instruments and lab information management systems, or sources like spreadsheets, text files, and manual data entry. Data assets can share generic metadata concepts such as file name, URL, size, checksums, or file type. Ultimately, basic file metadata is far from sufficient to find and ultimately use or reuse data from scientific experiments. Depending on the level of formality in projects, different kinds of provenance and quality-control metadata may also be useful.

In practice, the details in metadata will vary greatly from collaboration to collaboration. The types of data tracked, contextual details, and relationships between data will all evolve as a collaboration proceeds. Early-phase, exploratory projects need quick setup and simple models while researchers establish experiment protocols, collaboration methods, shared terminology, and data collection standards. Over time, the understanding or conceptualization of science tasks may mature and along with it the representation of activities and results must also evolve. Hence, if a data management service is to be used for the life time of a data-driven investigation, the metadata model that it captures must be capable of evolving over time while providing suitable access methods (**F4**).

Rights management and access control. FAIRness does not imply public access to all data, particularly as we extend these principles to earlier phases of scientific research. Membership in research collaborations will change over time, while the roles and associated rights of the members will also vary. For example, a post-doc in a group may be allowed to create new data elements to share with her lab, but only the PI is allowed to approve the release of the data to other members of a research consortium. In addition, scientific data sharing may involve data use agreements, access to proprietary data, time driven data embargoes, and different user roles within and across collaborations. Support for access control and associated policy may be required at levels of granularity that go from an entire data set, to a single data element to provide access (**F2**). These different policies must be applicable in a fine-grained way, so that multiple classes of assets and metadata can be managed in one project and given different access policies.

3 ENTITY RELATIONSHIP MODEL

The organization of metadata benefits from well-defined concepts and structure to represent (or model) real-world information about assets. For example, a researcher may wish to find all pairs of sequence reads for each replicate in RNA experiments where the

biosamples had a specific genotype and mutation. In addition, they need enough detail to process data in a pipeline such as the read direction, strandedness, and the species of biosamples. This requires structured information about the files (direction, read number), the experiments (molecule type, strandedness, etc.), replicates (biological or technical replicate number), and biosamples (species).

While it is possible one could "flatten" these details into key-value pairs or other types of standardized structure, such a limited meta-model offers little expressive power in which to define and query metadata; and may lead to many data quality issues due to lack of database "normalization" (delete/update anomaly, redundancy, inconsistency, etc.). Instead, ERMrest adopts the Entity Relationship Model (ERM) as its core meta-model for allowing researchers to structure their metadata catalogs.

ERM as Tabular Meta-Model. Scientists need conceptually simple data structures balanced with the ability to evolve them to represent rich domains. Most researchers are familiar with tabular data in spreadsheets and research papers. In our experience, they readily adjust to tabular data-entry forms and table-oriented query results, where the tabular characteristics are enforced more rigidly than in spreadsheets or typographic tables.

The introduction of a tabular data store governed by an ERM can help structure interactions between collaborators, improving reusability of data and also providing a tangible nomenclature for discussion between collaborators about data collection techniques and data quality requirements. Furthermore, it enables model-driven software tools which consume an ERM description and automatically present a customized interface. These tools consume an ERM and adapt reusable table presentation techniques rather than having any ERM-specific structure designed into them. Such adaptive software enables a fluid collaboration style where data models can evolve throughout the life cycle of a research project.

Flexible Data Modeling. Our approach allows each collaboration to customize and evolve their own ERM enabling them to create detailed and meaningful metadata that is relevant to the needs of the collaboration, promoting reusability (**F4**). ERMs are collected into a collaboration-specific *catalog* which includes the current ERM schema, all current metadata values along with update history (discussed in Section 6 and policy (discussed in Sections 5).

To help with the construction and operation of model-driven software, we augment our ERM with *annotations* via convenient key-value stores associated with each node in an ERM. These allow software designers and data modelers to collaborate and experiment on more advanced presentation features by annotating parts of the ERM with machine-readable hints about semantics or desired presentation style. This enables us to "tweak" presentation and improve user experiences while preserving more rational ERM choices, e.g. keeping normalized entity and relationship data in ERMrest presenting denormalized content on screen.

ERM and Vocabulary. To promote findability (**F1**) and interoperability (**F3**) an ERM can easily be crafted to facilitate annotation of entities and data elements using standardized vocabularies (i.e., ontologies). The ERM naturally models vocabulary terms (as entities) and their relationships (e.g., is-a, part-of, etc.) either via direct references or through "relationship sets," which may be modeled as tables to define relationships with additional properties or more complex cardinalities.

ERMrest System Columns. To simplify certain aspects of ERMrest software development and to introduce scientific data-management practices which are not always obvious to users creating new ERMs, we have defined a small set of ERMrest *system columns* which we require in every table definition in our systems:

- `RID` or "row ID" is an opaque and stable identifier unique to each entity;
- `RCT` or "row creation time" records the birth-moment of each entity;
- `RMT` or "row modification time" records the moment when each entity was most recently revised;
- `RCB` or "row created by" records the authenticated identity of the client who added an entity;
- `RMB` or "row modified by" records the authenticated identity of the client who most recently revised each entity.

Our service automatically manages the content of these special system columns so that they represent system-managed provenance and do not vary depending on the choices of other clients.

4 ERMREST AS A WEB SERVICE

To promote accessibility (**F2**), ERMrest is a RESTful web service using secure HTTP as the protocol layer. We represent content and configuration as a set of web resources, each named by a URL, and define all client access in terms of simple hypertext transfer protocol (HTTP) requests performed against these resources.

We organize the web resources hierarchically into *catalogs* and catalog *snapshots*, each with its own ERM, security policies, and stored content. Requests from web browsers and other clients can manipulate these resources to learn the catalog configuration, reconfigure it, add content, remove content, or query content.

Catalog resources. Catalogs are the mutable data stores of the service. New catalogs can be created, a brief catalog description retrieved, or an entire catalog deleted. The catalog URL is a prefix for the URL of each of the other catalog-specific resources.

Snapshot resources. Snapshots are siblings to mutable catalogs. Rather than representing a data store, each snapshot represents content as it was known at a moment in the past. Each snapshot resource hierarchy mirrors its parent catalog resource hierarchy at the time the snapshot is created. Because snapshot resources represent content rather than storage, they only support retrieval operations.

History resources. Each mutable catalog has history management resources, allowing an administrative client to amend the historical record and alter the set of available snapshots en masse.

ERM resources. The ERM resources of a catalog represent the model which governs its storage. Model descriptions can be retrieved, and the model revised at will by creating or deleting individually named nodes in the ERM hierarchy: schemas, tables, columns, and key or foreign key constraints.

Policy resources. Access-control policies are managed relative to nodes of the ERM, allowing the administrator to express general policies affecting an entire catalog as well as more specific policies affecting only individual schema, tables, or columns. The policy system is described in more detail in Section 5.

Rights resources. Rights summaries are tri-state[1] predictions of future access-control decisions. These summaries augment individual ERM nodes and allow clients to anticipate likely outcomes for requests involving each ERM concept, without exposing clients to policy definitions nor a difficult interpretive task. A model-aware client such as our companion graphical user interface can use these predictions to tailor the options presented to each user.

Data resources. Tabular data representations are exchanged to retrieve or update relational data content of the catalog. ERMrest splits the data resource space into several interpretations of the relational storage: entity (whole row) access; projected (partial row) access; computed aggregates, e.g. counts or min/max; and grouped projections, i.e. access to combinations of group keys and their associated values or aggregates.

Each of these data APIs exposes a rich hierarchy of sub-resources denoting a range of query idioms including: simple access to one table; relational joins of multiple tables; filter-based row selection; column-based projection; configurable sort order; and pagination. A small subset of these resources support insertion, update, or deletion of data in one table at a time.

5 FINE-GRAINED POLICY

We allow fine-grained policies to adapt each catalog to the needs of its community. Each policy grants a privilege, allowing a specific actor or role to use an access mode in order to operate on one or more resources. By "fine-grained," we mean that granted privileges can be scoped broadly (e.g. entity set) or narrowly (e.g. attribute) according to the goals of the administrator and collaboration.

These access decisions cover a range of use cases including:

- Hide certain columns from "less privileged" users;
- Row-ownership, e.g. client matches `RCB`;
- Curation status, e.g. row is marked with coded values indicating that the row has been approved for release;
- User-managed ACLs, where rows extend access to explicitly listed users, and users are given the right to further adjust the embedded ACL to re-share.

We use access control lists (ACLs) to associate policies with specific user or group identifiers. Commonly, groups are used to define a role subject to a set of related policies. Then, users can be added and removed from these groups as their role in a project changes, avoiding frequent changes to policy definitions.

Policies are attached to ERM nodes in order to define their scope, e.g. a policy can affect a whole catalog, a specific schemata, a specific table, or one column of a table. Policies are inherited as default by descendants in the ERM, but each node can override with a custom policy which changes the default for its descendants.

ERMrest supports two types of policies: *static* and *dynamic*. Static policies grant rights to clients matching an ACL associated with the ERM. Static policies grant access to ERM nodes (for introspection and ERM management) and also grant access to associated storage (for data access). Dynamic policies address a common use case in which policy based on the state of the metadata. For example a policy that says that a dataset can be released only after it has been reviewed by a project PI, as indicated by the value of an `approved` metadata attribute on a data set. Dynamic policies include data pattern tests which must be met during request processing before any data-access right is granted.

[1]Binary access control decisions of allow/deny are represented as boolean values `true`/`false`. A third access control prediction of "unknown" is represented as the `null` value, meaning that a specific request must be attempted in order to determine data-dependent decisions.

5.1 Access Modes

We use access modes to define the kinds of access granted by a specific policy. The modes for static policies are summarized in Table 1, and dynamic access modes are summarized in Table 2. Dynamic mode names overlap with static modes and slightly adjust their meaning. The static `enumerate` mode controls the visibility of an ERM node for introspection and any other model-driven actions. The static `owner` mode includes administrative privileges needed to manage a catalog or ERM. A dynamic policy can only grant rights to data storage resources, never affecting ERM access.

Many modes are specific to certain kinds of resource. However, access modes can also be configured on parent resources to employ inheritance, e.g. `select` on a catalog or schemata sets a default policy for tables contained therein, and may have no effect at all if every table has a customized policy. Furthermore, some access methods imply other lesser privileges, e.g. `select` and `create` imply `enumerate`; `update` and `delete` imply `select`; and `owner` implies all privileges.

Table 1: Static access modes and resource scopes.

Mode	Resource	Granted access permission
`enumerate`	catalog	Observe catalog.
	schemata	Observe schemata.
	table	Observe table.
	column	Observe column.
`select`	table	Query and retrieve rows.
	column	Query and retrieve cells.
`insert`	table	Create rows.
	column	Initialize in new rows.
`update`	table	Mutate existing rows.
	column	Mutate in existing rows.
`delete`	table	Remove existing rows.
`create`	catalog	Add schemas to catalog.
	schemata	Add tables to schema.
`owner`	catalog	Admin. access.
	schemata	Admin. access to sub-tree.
	table	Admin. access to sub-tree.

5.2 Dynamic Policy Details

Each dynamic policy has a scoping ACL to determine relevance in a given request, but also has additional content describing how a data-dependent check should be performed. For each stored row that is being processed, a dynamic policy describes a method to project some content out of the database and finally grant access only if a special access test is satisfied by this contextual content.

A policy architect can create a modular policy which defers certain details to data-dependent tests. Then, less privileged users can be given limited access to manage the data values which control

Table 2: Dynamic access modes and resource scopes.

Mode	Resource	Granted access permission
`select`	table	Observe row.
	column	Observe value.
`insert`	for. key	Reference during insertion.
`update`	table	Mutate row.
	column	Mutate field.
	for. key	Reference during update.
`delete`	table	Remove row.
	column	Reset field to default.
`owner`	table	`select`, `update`, `delete`.
	column	`select`, `update`.

these tests, reusing existing data access interfaces for user-initiated changes to the policy enforcement system. Of course, these policy-influencing data changes are also subject to additional policy checks to prevent abuse. This lets the users have limited control over how resources are shared, without giving them wholesale access to scorrupt or bypass important access rules. This is similar in spirit to defining role-based policies in terms of a group, and delegating group administration to users at large. Those deputized users can add or remove group members but they cannot arbitrarily change other policies affecting themselves or others.

Projected context. Every dynamic policy describes how to project contextual content from the database as part of policy enforcement. Ultimately, the source of projected context is always a named column in the database. The simplest context is a column within the same row for which access is being determined. Chained projections introduce joined tables based on foreign key relationships to the table being controlled. In this case, a multi-table pattern of linked row instances is located, and the final projected context can be drawn from any one of these linked rows. Filtered projections introduce additional column-operator-value predicates, masking the context value if these value tests do not hold. The filters are useful for making a row-level decision conditional on a coded value, e.g. only consider this context when the row is "curated."

Decision types. ERMrest supports two decision types when processing contextual content. An `acl` decision interprets the projected content as another row-specific ACL which should be matched against the requesting client, just as we match static ACLs found in policy definitions. These dynamic `acl` decisions are useful when restricting access based on row-level provenance or by user-controlled permission lists. For contextual content which does not represent client or role identifiers, a `non-null` decision type is also available and simply tests that a context value has been found.

The simplistic `non-null` decision type is usually used in combination with filtered projections. When the actual access decision can be expressed in the form of a static ERMrest query, that query is written into the projection rule so that it returns context only when access is desired and returns `NULL` otherwise. However, filtered projections can also be combined with `acl` decisions when you want to conditionally enable or disable consideration of ACLs embedded in row content as a basis for workflow state restrictions.

Foreign key policies. Policies on foreign keys, as indicated in Table 2, enable scenarios where coded values are enshrined in a community and used in data-dependent policies. Rather than giving users all-or-nothing access to modify a field containing such coded values, the policy architect can control which users are able to express which coded values. These policies are scoped to a foreign key rather than the vocabulary table, allowing the policy architect to restrict the use of the same terms distinctly at each point of use in the ERM.

Dynamic selection. When dynamic policies grant the `select` privilege on all relevant rows and columns, request processing proceeds with the same behavior as if static policies granted access (except for performance penalties from enforcing the dynamic policy). However, when dynamic policies fail to grant `select` privilege on a row, that row is excluded from query processing and the request behaves as if the row does not exist. The client effectively observes the subset of table rows which they are permitted to see. When a dynamic policy fails to grant `select` privilege on a column, the SQL `NULL` value is substituted for whatever value is actually present. The client effectively observes a masked version of the enclosing row. In contrast, dynamic authorization failures on data mutation requests will cause the entire request to fail.

6 HISTORY

To support FAIRness of metadata, we track mutation results in each catalog and express them as catalog snapshots. Our objective is to automatically and coherently capture catalog snapshots and support stable reference to query results.

However, our experience tells us that data stewards and system administrators will need pragmatic options to control costs, mitigate risks, and prevent abuse. We introduce history amendment features to allow for these eventualities: to revise mistakes in policy configuration, to expunge old history when enforcing data-retention policies, and to redact specific values which were mistakenly stored in a catalog. Without these features, we know that administrators will face all-or-nothing decisions to close down or destroy valuable catalogs where they cannot afford the cost or risk of keeping snapshots.

We also find it worth discussing one significant non-goal for ERMrest history interfaces. We are not building a generalized temporal database nor focusing on temporal knowledge models. Snapshots are meant to represent drafts or revisions of the evolving catalog content, much as in a version-control system for source code or documents. If a project needs to track time-series data or model the time at which facts are known, those concepts should be reflected in the asset or ERM structures they use.[2]

6.1 Catalogs and Catalog Snapshots

We distinguish a mutable catalog and each of its many catalog snapshots. In effect, every modification to the mutable store creates a new catalog snapshot. At any given moment in time, the mutable catalog contains the same content as the most recently created snapshot.

In essence, each catalog snapshot represents the set of row versions existing at the snapshot time. We can consider the entire

[2] Put succinctly: The revision-history of a book is quite distinct from the history related by the book.

history of a catalog as a linear sequence of snapshots, each corresponding to one distinct moment in time when row versions were created and/or rows deleted. If we introduce an identifier for each of these possible snapshot moments, we can include that within query URLs in order to name different queries expressed against the same snapshot or conversely the same logical query expressed against different snapshots.

Because we support evolution of models, we account for differences in the ERM between each snapshot and the live catalog. Furthermore, because policy is scoped to ERM nodes, we cannot assume that the latest catalog policy can be applied to an older snapshot which does not even possess the same ERM concepts. Instead, snapshots will also hold a snapshot of the catalog policy in place when they are created. Administrators will need to use history management mechanisms to intervene if they have a need to retroactively change the policy on previously held snapshots.

6.2 History Management

To address the need for system operators to control their history storage, we define a set of interface functions that enable examination and manipulation of a history timeline by system administrators. These include:

Range Discovery. Administrators need to be able to probe a catalog to determine the range of snapshots currently stored.

Truncation. To reclaim space or observe limited data retention policies, administrators can instruct ERMrest to delete older snapshots based on a data-retention horizon.[3]

Policy Amendment. To address retroactive changes to data-access policy, administrators can address an ERM node within a historical interval to modify their respective policies en masse to a new, common configuration. This interface allows editing of static or dynamic policy elements.

Data Redaction. To address redaction of stored values in historical snapshots, administrators can address a table column within a historical interval to delete its content. This revises stored row version snapshots to replace the stored column value with a `NULL` value. The administrator can apply this operation to all entities within the historical interval, or optionally can include a simple filter condition to selectively redact only row versions that hold some other value. Two common idioms would be: redaction of all row versions for a specific entity with a fixed `RID` value; or redaction of all row versions containing a specific inappropriate value in the column to be redacted, regardless of which entity it belongs to.

7 PERSISTENT IDENTIFIERS

Every entity item within an ERMrest catalog is automatically assigned a globally unique persistent identifier when the `RID` is assigned by the server. By combining the `RID` with the snapshot ID for the catalog, we can uniquely identify an exact instance of an entity. Given that the snapshot captures the state of the entire catalog, we have identified not only a single entity, but all of the other entities that might be connected to it via a relationship in the ER model.

[3]Of course, further administrative tasks may be needed to complete this objective after ERMrest has deleted data under its control. System operators may need to also purge database backups, audit logs, or copy-on-write storage systems used in their deployment.

Snapshot IDs combined with `RID`s and the web service interface allows us to construct data retrieval URLs for entities and more complex queries at a particular point in time. These URLs act much like version-qualified object references to an object store, making stable reference to a previously held value. However, the history management mechanisms described in Section 6 and the other web service mechanisms described below introduce additional complexity. While conventional object stores will either serve older versions or refuse as a binary decision, ERMrest is capable of serving an amended version of referenced historical content.

8 INTERFACE AND IMPLEMENTATION

Like any web service, ERMrest supports well-defined communications regarding its resources, using HTTP request-response messages. To design the service interface, we must define the resource types, their identifiers, their representations, and the behavior of the HTTP methods used to manipulate them. Aside from the methods, e.g. `GET` or `PUT`, there are many standard HTTP headers which can modify the meaning of a request or response.

We take a pragmatic stance in binding relational catalog concepts to web concepts. At present, the ERMrest service interface's target audience is programmatic clients who understand ERMrest rather than traditional web browsers which expect server-rendered content suitable for human presentation. In order to support users with web browsers, we have developed Chaise [10], a companion suite of single-page web applications which present a graphical user interface and translate user-level goals into combinations of ERMrest requests and client-side presentation.[4]

Every web service and client must know that HTTP itself is not reliable, since any request or response message may be lost or incompletely transmitted. We support a number of standard HTTP features which allow reliability to be synthesized over a stream of requests. These features include well-defined status codes to signal error classes, so-called `ETag` headers to name individual resource states, conditional processing instructions to detect concurrent change, and standard HTTP methods which define rules for safe error-handling procedures.

HTTP methods. In general, we use the `GET` method to retrieve representations of data, ERM, and policy resources in a side-effect free manner; `PUT` for in-place mutation of data or policy resources; `DELETE` for destruction of data, ERM, and policy resources; and `POST` for creation of new sub-resources under a named parent resource.[5]

URLs name queries. We define a rich set of query resources where the URL structurally encodes query terms. The URL identifies the access API and table (or joined tables) being queried; imposes data filters as attribute-operator-value filter predicates applied to each row; establishes a sorting order for results; identifies a page size or limit for the query response; and identifies a pagination boundary if addressing a position other than the beginning of the sorted result stream. For example, the following query identifies an `entity` set in catalog `1` at snapshot `Y-1Z4B` from table `bar` joined with `foo` with filter predicate on attribute `x` greater than `7`.

[4]A typical Chaise "page load" makes a sequence of ERM and data requests to build up an intuitive, model-driven rendering of catalog content.

[5]Our use of `PUT` and `POST` for tabular data is a pragmatic compromise to simplify the interface for clients. Ideally, these should use the `PATCH` method to be more compliant with REST principles. We may revisit this design decision in the future.

`ermrest/catalog/1@Y-1Z4B/entity/foo/bar/x::gt::7`

This query-based URL syntax supports access idioms for mutable storage including: stable identifiers, e.g. to locate a row by its immutable `RID`; unstable attribute-based naming, e.g. locating a row by an application-specific, mutable key; or pattern-based naming, e.g. a set of rows sharing certain direct or linked attributes. All of these URLs support retrieval via `GET`, most support deletion of content via `DELETE`, and only a very reduced subset of URLs support creation or mutation operations.

URLs name data. Given mutation, it is best to think of data URLs as representing the result of a query process. The result has a well-defined semantics but the precise content depends on the current state of storage. When the storage state is fixed, one can conflate a query process with the resultant data. In these cases, one may think of a query URL as a name for data.

The catalog snapshot mechanism provides explicit URLs to fix the state of storage and reference such named data over long periods of time. However, due to content-negotiation, access-control, and history management, even snapshot data URLs may name different data depending on when and by whom a request is made. Only in combination with end-to-end validation (such as externally-imposed checksums) can an exact data value be shared by reference.

Differential content retrieval. The same URL can retrieve different representations depending on the outcome of content negotiation, e.g. JSON or CSV tabular data. This standard HTTP mechanism, influenced by client choices, is called content negotiation. In addition, operations on the same URL can retrieve different data or cause different status codes depending on the client's identity and configured access-control policies.

State tracking. An `ETag` names the instantaneous version of a resource being accessed. In practice, we derive this from the most recent change to any ERM, policy, or data resource involved in the request. The `ETag` header is returned in retrieval responses to identify the state being served and in mutation responses to identify the new state created as a result of the operation.

A known `ETag` can be sent by the client with conditional processing request headers to ask the server to process the request only if the current resource state holds a certain relationship to the version known by the client. The `IfMatch` header is used in mutation requests to skip processing and signal an error if there has been any concurrent change to the resource that the client is not aware of. Conversely, the `IfNoneMatch` header is used in retrieval requests to skip processing if the resource has not changed, instructing the client to continue using cached content.

Atomic requests. Each request provides atomic, consistent, isolated, and durable updates or inspection of catalog state. Side-effects of successful requests are visible to all subsequent requests by all (authorized) clients. Concurrently submitted requests will be serviced with an apparent serialized order.

Requests which produce a non-success status code leave the resource unchanged. However, a client may need to perform queries of resource state or employ conditional processing headers to reestablish a coherent understanding of catalog state if response messages are lost. Most mutation operations are not idempotent, and blind repetition following a lost success response may cause errors or other undesired results.

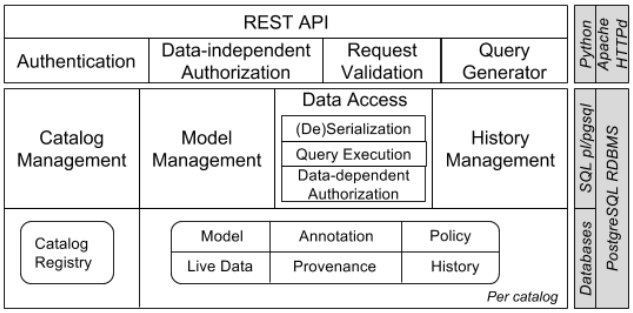

Figure 1: ERMrest system architecture.

Web service implementation. The current ERMrest software implements the preceding web resources and HTTP binding principles as a Python-based web service which is tightly coupled to the PostgreSQL database management system. We operate the service as a `web.py` framework application running in `mod_wsgi` daemon mode behind the Apache HTTPD web server, with HTTP/2 protocol enabled for efficient handling of many small requests from web browsers.

Figure 1 depicts the overall ERMrest architecture. For performance reasons, we delegate tabular data processing to the database. The Python web service performs meta-model processing to validate and plan requests, but any tabular data input or output is simply relayed as byte streams.

Through a mixture of dynamically generated SQL, stored procedures, and triggers, the database engine performs the significant work of processing data. It deserializes inputs, enforces data-dependent access policies, produces side-effects denoted by data mutation requests, enforces integrity constraints, tracks data changes and provenance, computes data results, and serializes outputs.

Model-driven processing. ERMrest generates SQL statements through a process that resembles a syntax-driven compiler, using the ERM and policies to validate the request prior to generating and executing commands on the database. Each request URL references a number of ERM concepts, and the resulting syntax tree defines the "shape" of the request processing that will proceed if access is granted. This compiler-like meta-processing is what allows ERMrest to adapt incrementally to ERM changes made by clients during an active collaboration.

Model storage and evolution. The ERM is *reified*[6] so that ERM nodes are each issued their own stable `RID`. Each table in the active catalog ERM maps directly to a table in PostgreSQL with a `RID` primary key, with a corresponding history table containing historical copies of rows, each keyed by `RID` and a visibility interval delineating their birth and death moments.

We reify the ERM and stored it in tables so that we can also apply history tracking to the ERM. Unlike normal SQL "information schema" which reify only the current model of a database, our ERM storage tables are backed by history tables and allow us to determine the previous state of the ERM at any point in the recorded history. A history of policy definitions accompanies the ERM history.

[6] Reification here simply means that the ERM concepts are themselves modeled as data within the same system.

Policy storage. We treat policies for a catalog as part of the ERM and store them alongside other meta-model content in the database. Policy management is treated much like model management, updating the authoritative representation in the database on each policy change. All service request processing is designed to provide a transactionally-consistent view of ERM, policy, and data.

Authentication. For each request, we establish a primary client identity and any secondary attributes associated with the client, according to the configured authentication system.

Most of our deployments use an OpenID Connect flow through Globus, allowing users to sign-on via their home institution. In this configuration, secondary attributes reflect the client's group memberships, determined by consulting the Globus Groups service. The group mechanism allows a user community to self-manage named sets of clients, and use these group names as roles in role-based policies.

Authorization. To allow access, we must find a policy granting access to the client. For data-independent policies, such tests are performed by the Python service logic. For data-dependent policies, tests occur first in Python to determine relevance of each policy, and relevant tests are then compiled into the SQL executed in the database to determine final access.

Logically speaking, ERMrest policies are a disjunctive set of rules granting access privileges. If the service can find a data-independent decision to grant access, it can perform a faster SQL operation lacking any data-dependent enforcement. If there are no relevant data-dependent policies, the service can skip SQL processing entirely and reject access. Finally, if a static decision cannot be determined, the service produces SQL with the disjunctive policy checks encoded in such a way that the PostgreSQL query planner can optimize its plan based on which data-dependent checks are cheaper to execute.

History tracking. Historical row versions are automatically stored using SQL triggers which intercept each row touched by `INSERT`, `UPDATE`, and `DELETE` statements and update row version data in a history table shadowing each live storage table. When a table is deleted from the active catalog ERM, its shadow history table remains. Catalog snapshots are logical resources formed by querying these history tables selectively to only consider row versions visible at a certain point in time.

To handle evolution of table definitions, we resort to a "blob" storage method for historical row versions. Aside from the row version key, all other content goes into a single `rowdata` column stored using PostgreSQL's `jsonb` column type. The fields in this data blob are named by the `RID` of each column so they are unambiguous even if column names are reused over time.

We also use SQL triggers to record requests which make any change to mutable table content, ERM definitions, or policy. This simplified audit log allows ERMrest to always know the entire set of valid snapshot identifiers. This record of requests may also support future enhancements to browse the history of changes or retrieve details of each change for auditing or formulation of "undo/redo" requests.

Longitudinal amendment. The longitudinal history amendment interfaces directly manipulate history storage tables. History truncation deletes every row version with a death moment earlier than the truncation boundary. Amendment and redaction update row versions in place to change policy or attribute content, respectively.

History change tracking. We record history truncation, amendment, and redaction requests in parallel with snapshot tracking. We use this amended record to determine the latest revision time for any given snapshot, so we can produce a new `ETag` and invalidate any cached representation of a snapshot after an administrator revises history. For history without any revisions, the revision time for a snapshot is the same as the snapshot time.

Fast cache hits. We rely on state-tracking for cache invalidation. In our experience, clients are chatty and repeatedly poll resources of interest much more frequently than resources change. Therefore, we want a low-latency but correct method to determine the current `ETag` and handle condition processing headers. We address this concern with several related techniques:

- ERM caching in the Python service.
- Indexed query to determine current version.
- Connection pooling for SQL plan caching.
- Composite `ETag` combining version, client identity, and content-negotiation values to avoid incorrect cache hits.

These features allow cache hits to be handled on a fast path through the service logic which has almost no wasted work when a mismatch is detected and the request processing expands with generalized (and slower) logic to complete handling of the request.

We do not currently implement any response caching on the server side. In the presence of fine-grained, data-dependent policies, the responses may be unique for each client and therefore would limit the benefit and increase the cost of server-side caching. However, client-side caching works very well to reduce processing costs on our servers, as described in Section 9.

9 EVALUATION AND DISCUSSION

We have developed and operated ERMrest and companion client software in several ongoing bioinformatics research projects. The service saw its first pilot application in late 2015 and has remained in continuous use. Here, we report qualitative observations from five deployments for the period 2018-01-13 to 2018-02-12 and also quantitative results for a synthetic benchmark effort.

9.1 Deployments

ERMrest has been used in scientific collaborations spanning from small teams engaged in data collection and analysis for basic science, to large, multi-national consortium producing data repositories. These include:

- The microscopy core for the Center for Regenerative Medicine and Stem Cell Research (CIRM), offering microscope slide-scanning as a service;
- The NIDCR FaceBase project, organizing a central repository for data generated by a number of spoke sites;
- The GPCR Consortium, an international collaboration to discover and analyze G-Protein Coupled Receptor molecular structures;
- The NIDDK GUDMAP/RBK projects, curating microscope imagery and sequencing data with assessments and annotations by domain-experts;

- Mapping the Dynamic Synaptome, an NIH-funded multidisciplinary effort to develop methods for in vivo measurement of the synaptome.

We report here on activity logs collected from our production catalogs over a representative one month period.

Table 3: Catalog ERM and data characteristics for five projects.

Metrics	CIRM	FB	GPCR	GUDMAP	Synapse
#tables	29	107	85	20	29
#cols tot.	246	1.0k	845	2.6k	265
widest table	50	37	83	48	33
95^{th} table	16	20	31	31	33
50^{th} table	9	7	6	8	10
in avg. row	13	16	7	20	25
#rows tot.	39k	23k	7.5M	12M	3.8k
largest table	22k	4.1k	5.8M	11M	2k
95^{th} table	6.9k	788	88k	37k	825
50^{th} table	11	28	357	7	13

Catalog characteristics. Our long-tail science catalogs are quite small in systems-engineering terms, and the complexity of the ERM can vary greatly depending on the community and their problem domain, and particularly when "legacy" metadata has been imported from long-running projects.

Table 3 summarizes the basic shape of our five production catalogs which have each accumulated 1-3 years of ERMrest-based collaboration. Shown are the total number of tables, column count statistics (across the whole ERM; for the maximum, 95^{th}, and 50^{th} percentile table widths; and for an average row across all tables), and row count statistics (across the whole catalog, and for the maximum, 95^{th}, and 50^{th} percentile table lengths). In GUDMAP and GPCR, the two catalogs with millions of rows rather than mere thousands, the sets of user-managed entities are qualitatively similar to those in our smaller project catalogs. The millions of extra rows come from bulk import of external data: large term lists from external ontologies and detailed manifests of large digital assets such as gene expression micro-array assays and protein sequence data.

Apart from GPCR, all catalogs have extended their ERMs with the five extra system columns described in Section 3. In general, the non-vocabulary tables grow over time at the somewhat steady rate that experiments are performed by human participants. Vocabulary and other bulk-imported tables can grow in fits and spurts as collaborators revisit nomenclature debates and evolve their catalog's ERM.

Observed workload. Human-driven science catalogs are also quite idle by contemporary web service standards. Table 4 breaks down the month of logged requests by category. During the reporting month, our production ERMrest services experienced a peak request load of 627 requests in one minute and 175 requests in one second. More significantly, these servers averaged less than 1.5 requests per minute, meaning that there are long periods of inactivity punctuated by periods of modest load. These catalogs saw a combined data read:write request ratio ratio of over 22:1, and a combined cache hit rate of 39.9% and 78.1% on data and ERM reads, respectively.

Table 4: Request workload 2018-01-13 to 2018-02-12.

Metrics	CIRM	FB	GPCR	GUDMAP	Synapse
#reqs	13.6k	62.6k	14.6k	51.1k	56.8k
peak minute	0.14k	0.57k	0.21k	0.58k	0.63k
peak second	45	175	72	80	74
#data writes	0.93k	1.1k	1.9k	0.94k	2.9k
#data reads	11k	55k	12k	44k	52k
cached	34%	28%	46%	32%	59%
#ERM writes	0	33	0	0.16k	0.13k
#ERM reads	1.5k	2.9k	0.78k	5.5k	1.6k
cached	75%	73%	73%	81%	82%

Table 5: Request latency logged on server 2018-01-13 to 2018-02-12 for data writes, cached data reads, and uncached data reads.

Metrics (ms)	CIRM	FB	GPCR	GUDMAP	Synapse
avg. write	187	27	14,485	370	22
avg. cached	14	14	11	28	14
avg. uncached	116	34	121	587	36
50^{th} uncached	37	18	31	70	15
90^{th} uncached	183	76	235	682	59
99^{th} uncached	1,314	196	2,226	4,900	555

Observed performance. Our project servers are not running identical ERMrest releases, but still have qualitatively similar performance. Median data read latency is well under 100 milliseconds and 90% of reads under 1 second, suitable for interactive use in web-based user interfaces. Table 5 breaks down the logged data requests by category and latency. Most projects also show very reasonable data write latency; GPCR is an outlier with slow writes due to an unusual legacy workload involving frequent bulk insert-or-update requests to synchronize with a large external table source. All projects show a long-tail distribution of data read latencies with the majority being quite fast and a small percentage increasing in cost due to much more complex structural queries. All catalogs show consistently low latency for cache hits.

All five deployments run on one physical host, each within a separate virtual machine (VM) running Fedora 26 and PostgreSQL 9.6 or newer. Each VM is allocated 8-16 GB of RAM which is shared between ERMrest, PostgreSQL, Apache HTTPD, and other project-specific software. The backing storage for all the PostgreSQL instances in the VMs is capacity-optimized rather than speed-optimized with a hardware RAID 6 configuration on bulk nearline storage.

9.2 Benchmarks

Our production systems vary in ERM complexity, table sizes, workload, software versions, and policy configuration. It is difficult to draw quantitative lessons from their empirical performance, other than the general observation that they are "fast enough" and support their ongoing collaborations. The general performance characteristics of a database system like PostgreSQL are a complex topic

in their own right, well beyond the scope of this article. We developed a simple, synthetic benchmark to examine the practical cost of several ERMrest features under a simplified model of typical metadata catalog operations.

Compared features. We permute the following ERM features to form a family of benchmarking scenarios:

- Baseline: assume k columns of user-driven content;
- System columns: add 5 standard columns;
- Row-ownership: a data-dependent access rule;
- B-tree: `btree` indexing on each column;
- Tri-gram: `trgm` indexing on each column.

We calculate incremental costs as differences between the median round-trip times measured for scenarios differing in only one feature. The added indexes do not benefit the simple test requests in any way, but have substantial maintenance costs we wish to measure. We know that having such indexes present can help our most complex user-driven queries, and we want to understand the performance penalty incurred by aggressively generating column indexes rather than expecting administrators to carefully design a minimal indexing scheme for each catalog.

Test environment. Our test machine for synthetic measurement has a single 6-core Intel Xeon E5-1650v3 processor and 64 GB of error-correcting RAM, approximating an affordable mid-range server. To better represent a modern but modest storage system, we also chose a mid-range, SATA-based Intel DC S3500-series SSD. We tested our ERMrest development branch on Fedora 27 using PostgreSQL 10.2.

Operating regime. Observing our real-world catalogs, we know that most user-managed tables have fewer than 100k rows, and most requests are simple data reads easily supported by primary key indexing on tables. On those catalogs small percentage of read requests are complex ERM-specific queries which are infeasible to reproduce in a simple, synthetic test procedure.[7]

Other test variables. We simulate a catalog at different phases in its lifecycle of steady growth by repeating measurements against tables prepared with different numbers of pre-existing rows. The ERMrest interface is tuned for paginated access to data. Our production systems typically experience request sizes of 1 or 5-20 rows for interactive browsing of single entities or small search result sets, respectively. We simulated a range of pagination scenarios by testing with different request sizes, but present most results below for a fixed scenario with 10 rows per request.

Test procedure. Our benchmark sweeps through a set of table-definition scenarios, table-size, and request-size parameters to gather ten samples at each point in the parameter space. These tests focus on simple, single-table access scenarios which are somewhat symmetric between insert, update, read, and delete operations. We force a synchronous `VACUUM` command to simulate rest periods which occur on production systems. The high duty-cycle of writes in our benchmark would otherwise overwhelm the background table maintenance in PostgreSQL and distort our results. Each benchmark run takes approximately 10 hours to complete.

[7] We expect to evaluate these more challenging queries in future empirical studies of real-world systems. Such studies will benefit from longer-running ERMrest catalogs collecting historical data and query logs, so that we may reproduce different storage configurations and replay different test workloads in a controlled laboratory setting.

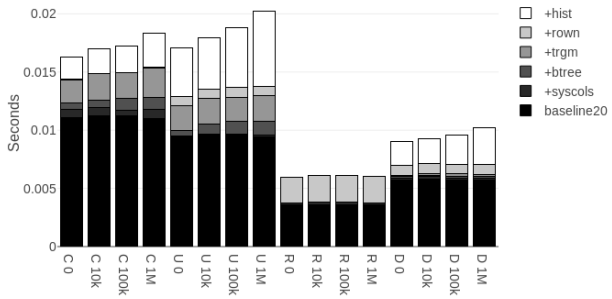

Figure 2: Round-trip HTTPS times to (C) create; (U) update; (R) read; and (D) delete 10 rows per request for varying table sizes. Each stacked bar represents differential costs of ERMrest features.

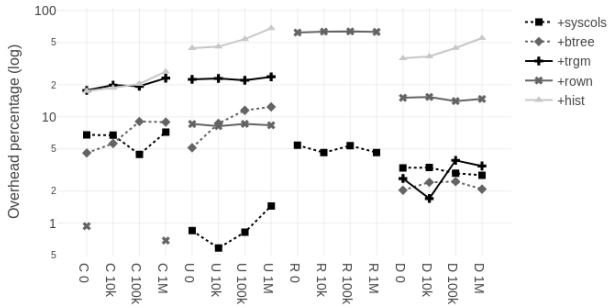

Figure 3: Overhead percentages for (C) create; (U) update; (R) read; and (D) delete while varying table size for a fixed request size of 10 rows.

Our service implementation does not provide any real-time latency guarantees, and there are costs sporadically incurred on requests due to various levels of resource reclamation and re-initialization within service components. Our benchmark introduces randomized, unmeasured "filler" requests to ensure that we are measuring a typical steady-state service condition and that any of these periodic and aperiodic latency penalties are distributed over all measurements, rather than accidentally biasing a particular step in the benchmark parameter sweep.

In practice, our benchmark resets the entire ERMrest service state to a known ground state when destroying the catalog in each iteration of the outer test loop. The inner loops proceed with a single sequential, connection-pooling Python client using HTTPS 1.1 protocol.

Measured results. Figure 2 shows median feature costs with request size fixed at 10 rows per request and a representative table definition including 20 columns of non-system metadata. For the

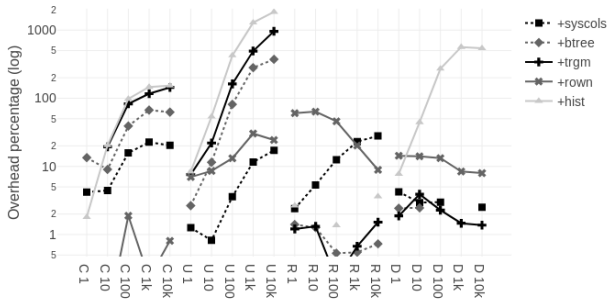

Figure 4: Overhead percentages for (C) create; (U) update; (R) read; and (D) delete while varying request size for a fixed table size of 100k pre-existing rows.

four data access methods, round-trip times are tested at 5 different table sizes (0, 10k, 100k, and 1M pre-existing rows). This test scenario coincides with typical paginated access patterns we might see in our production catalogs.

Across the tested range of table sizes, data retrieval with static access control is consistently beneath 4ms and remains under 7ms with a simple row-ownership dynamic access control check. Baseline data mutation requests are all slower by a factor of 2-3x, with deletion being cheapest and insertion the most expensive. The addition of system columns adds approximately 0.2ms to each read, update, and deletion request due to the increase in row size from 20 to 25 columns, while adding 0.4-0.8ms to insertion requests. There is no discernible relationship between table size and baseline request costs in the tested range.

The indexing and history-collection features have essentially no impact on read requests.[8] These features have substantial cost for data mutation requests, because they introduce additional database "write amplifying" maintenance work that PostgreSQL performs as part of each change to catalog storage. The `btree` indexing is cheapest, followed by `trgm` indexing, and then by history collection. These maintenance costs also show a clear trend where costs increase with the size of the pre-existing table, as maintenance work includes adjusting tree-like structures in the database rather than purely localized changes to the affected rows. Update requests become the most expensive access method on large tables, because they involve twice the history maintenance cost as old row snapshots and new row snapshots must both be written to record each change.

In spite of the higher mutation costs, all access methods perform well with less than 25ms of round-trip latency for this regime of small paginated requests. Figure 3 depicts the same measurements as in Figure 2 using a vertical scale representing feature overhead as a percentage of the baseline request round-trip time. An overhead of 0% would have no cost, while an overhead of 100% would indicate a doubling of round-trip time. We believe that overhead values at or around 1% are insignificant, as these differences in median time are well within the typical request-to-request variance.

Finally, Figure 4 depicts a complementary set of feature overhead measurements, where we vary the request size (1, 10, 100, 1k, and 10k rows per request) while fixing the pre-existing table size at 100k rows. Here, the per-request cost of the features dominates requests with 1-10 rows. However, as request sizes increase from 100-1k, the per-row costs begin to dominate and most overhead curves begin to roll off towards a fixed cost percentage. We believe that this linear behavior continues for several orders of magnitude, based on isolated experiments with the database; however, such large requests have too high of a round-trip latency to be useful in a synchronous web service, where messages may be lost and require re-transmission, and most HTTP middleware will report timeout errors and abandon requests due to lack of timely response.

Concluding observations. Our synthetic tests confirm our expectation that history-collection and aggressive indexing features have manageable overhead in our target operating regime with many small requests in a read-heavy duty-cycle. These features only incur a penalty for write access and we argue that this penalty is worthwhile due to the potential enrichment it can bring to subsequent data consumption. Evaluation and measurement of these enrichments is left for future work.

Only at large request sizes do the feature costs reported here begin to interfere with service utility. Baseline data mutation requests that may have been acceptable (even if slow) may become 20-50x slower with all optional features enabled, causing requests to become too slow for practical use in a synchronous HTTP setting. However, such large requests are not the product of interactive human use and so should not be considered a road-block. Batch ingestion of large metadata can be implemented by unattended processes which can generate streams of smaller requests to work well with the current ERMrest APIs, or asynchronous web interfaces could be defined for cases where large batch requests must be performed under atomic access guarantees.

10 RELATED WORK

Data publication systems [4, 11] generally focus on the organization, annotation, and dissemination of scientific data and results. They allow scientists to describe published data objects with simple descriptive metadata in line with Dublin Core, which covers the basic publication metadata. These approaches grew out of the (book) publishing industry and while useful guides for scientific data publication, they only address "findability" in the sense of simple indexing and registries of data. Our approach empowers science users with the ability to generate rich domains of information based on the ERM model in a manner that facilitates early collaboration through publication. While the simple publication schemes only allow for finding data in a registry, our approach allows science users to collaborate throughout active research activities and to publish enough details about their data to enable other users to understand what processes created the data and how to reuse them.

Scientific metadata and metadata catalog systems have been an area of intense research interest. These systems are used as the centerpiece in collaborative data management, often for large e-science

[8] The tested query accesses the live catalog and not a historical snapshot, while using basic row primary keys for selection so the addition or absence of optional column indexes do not affect the PostgreSQL query plan.

applications in physical sciences but increasingly in other domains as well. In many ways, metadata catalog systems share much in common with data publication systems, differing mainly in the intended audience and data life cycle they support. Where data publication focused on public dissemination of finished data products, metadata catalogs have supported data sharing within collaborations during active research. Like publication systems, metadata catalogs have generally offered only simple key-value models for describing data [5, 7, 8], thus they also share their limitations for describing data sufficiently for use and reuse by others outside the collaboration. Some have explored richer semantic models [12, 13], however, we believe the ERM approach is more suitable for scientists who lack in-depth training in semantic web and scientific data curation.

Shared or collaborative database use is another topic that has been explored by others. OGSA-DAI [2] offered science collaborations access to database services for a variety of purposes using Grid protocols. Our approach differs by offering simpler interfaces for data management over standard Web protocols and ours is approachable to users in the "long-tail" who do not necessarily have the training necessary to operate Grid services. SQLShare [6] provides a cloud hosted database service for science users. SQLShare and ERMrest share a focus on enabling users in the "long-tail" with powerful tools for data collaboration. But, SQLShare focuses on data analysis workloads through the SQL language while ERMrest covers workloads involving data creation and offers simpler data interfaces. Finally, others have also attempted to map ERM concepts to Web protocols, such as HTSQL [1]. Unlike ERMrest, HTSQL does not provide support for model introspection, model evolution, content update, history, nor differentiated access control, where individual rows may be visible to some clients but not others.

11 CONCLUSION

We have summarized major challenges faced in many scientific data management and collaboration problems, and we have introduced ERMrest, a web-based metadata catalog addressing these problems. We have presented ERMrest key features, design objectives, architecture, and implementation, and demonstrated how ERMrest can rapidly adapt to many diverse data types and supports continuous FAIRness through mutable storage, fine-grained access control, automated creation of non-mutable global unique IDs for every data element, versioning, and provenance. Qualitative observations from ERMrest deployments and quantitative results from a synthetic benchmark are provided.

We found that ERMrest usually performs fast enough to support interactive use in browser-based web applications in five research consortia. Interactive latency suffers for a small number of complex structural metadata queries we have attempted in our projects, and for those cases we must consider conventional database performance tuning practices such as improving indexes or investing in storage and computing hardware capability. Future work is required in this area, to explore query workloads from real projects and consider whether there are any web service interface changes which might enable the expression of ERMrest searches while generating SQL queries which the database can execute more rapidly.

Not all of our deployments have utilized the recent history feature. We plan to roll this feature out and gain more experiences from our user communities. Possible areas for future work in ERMrest include: refinement of the HTTP data access interface for `PATCH` scenarios; asynchronous interfaces for large batch requests or those involving multiple input or output table representations; performance evaluation and optimization for access to historical snapshot content; new longitudinal history access interfaces to support use cases such as viewing the history of one entity or summarizing the changes made by one request; and lastly, review of model evolution experiences from projects to identify any enhancements to the ERM meta-model which might improve the usability of ERMrest in complex, evolving collaborations.

ERMrest and ERMrest-based tools are open-source and are publicly available on GitHub (github.com/informatics-isi-edu/ermrest).

ACKNOWLEDGMENTS

This work is supported by the National Institutes of Health under awards U54EB020406. 1R01MH107238, 5U01DE024449, and 1U01DK107350

REFERENCES

[1] 2012. HTSQL Web Site. (2012). http://www.htsql.org
[2] Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P Chue Hong, Brian Collins, Neil Hardman, Alastair C Hume, Alan Knox, Mike Jackson, et al. 2005. The design and implementation of Grid database services in OGSA-DAI. *Concurrency and Computation: Practice and Experience* 17, 2-4 (2005), 357–376.
[3] Christine L. Borgman, Milena Golshan, Ashley Sand, Jillian Wallis, Rebekah Cummings, Peter Darch, and Bernadette Randies. 2002. Data Management in the Long Tail: Science, Software, and Service. *International Journal of Digital Curation* 11, 1 (2002), 128–149.
[4] Kyle Chard, Jim Pruyne, Ben Blaiszik, Rachana Ananthakrishnan, Steven Tuecke, and Ian Foster. 2015. Globus data publication as a service: Lowering barriers to reproducible science. In *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE, 401–410.
[5] Ewa Deelman, Gurmeet Singh, Malcolm P Atkinson, Ann Chervenak, NP Chue Hong, Carl Kesselman, Sonal Patil, Laura Pearlman, and Mei-Hui Su. 2004. Grid-based metadata services. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 393–402.
[6] Bill Howe, Garret Cole, Emad Souroush, Paraschos Koutris, Alicia Key, Nodira Khoussainova, and Leilani Battle. 2011. Database-as-a-service for long-tail science. In *International Conference on Scientific and Statistical Database Management*. Springer, 480–489.
[7] Birger Koblitz, Nuno Santos, and V Pose. 2008. The AMGA metadata service. *Journal of Grid Computing* 6, 1 (2008), 61–76.
[8] Arcot Rajasekar, Reagan Moore, Chien-yi Hou, Christopher A Lee, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, et al. 2010. iRODS Primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services* 2, 1 (2010), 1–143.
[9] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Proell. 2016. *Data Citation of Evolving Data.* Recommendations of the Working Group on Data Citation (WGDC). https://doi.org/10.15497/RDA00016
[10] Robert Schuler, Carl Kesselman, and Karl Czjakowski. 2016. Accelerating data-driven discovery with scientific asset management. In *IEEE 12th International Conference on eScience*. IEEE.
[11] MacKenzie Smith, Mary Barton, Mick Bass, Margret Branschofsky, Greg McClellan, Dave Stuve, Robert Tansley, and Julie Harford Walker. 2003. *DSpace: An open source dynamic digital repository.* Technical Report.
[12] Rattapoom Tuchinda, Snehal Thakkar, Yolanda Gil, and Ewa Deelman. 2004. Artemis: Integrating scientific data on the grid. In *AAAI*. 892–899.
[13] Xinqi Wang, Dayong Huang, Ismail Akturk, Mehmet Balman, Gabrielle Allen, and Tevfik Kosar. 2009. Semantic enabled metadata management in PetaShare. *International Journal of Grid and Utility Computing* 1, 4 (2009), 275–286.
[14] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data* 3 (2016), 160018.