Emergent Semantics: Towards Self-Organizing Scientific Metadata

Bill Howe, Kuldeep Tanna, Paul Turner, and David Maier

OGI School of Science & Engineering at
Oregon Health & Science University
Beaverton, Oregon
(bill,kuldeep,maier)@cse.ogi.edu, pturner@ccalmr.ogi.edu

Abstract. Tasked with designing a metadata management system for a large scientific data repository, we find that the customary database application development procedure exhibits several disadvantages in this environment. Data cannot be accessed until the system is fully designed and implemented, specialized data modeling skills are required to design an appropriate schema, and once designed, such schemas are intolerant of change. We minimize setup and maintenance costs by automating the database design, data load, and data transformation tasks. Data creators are responsible only for extracting data from heterogeneous sources according to a simple RDF-based data model. The system then loads the data into a generic RDBMS schema. Additional grouping structures to support query formulation and processing are discovered by the system or defined by the users via a web interface. Discovered and imposed structures constitute emergent semantics for otherwise disorganized information.

1 Introduction

When a group of environmental scientists requested our help designing a metadata management solution for their scientific data repository, we speculated that the database community's flagship technology, relational database management systems, would solve the problem neatly. We changed our assessment after further investigation, judging that the high cost of database deployment and maintenance made the likelihood of adoption rather low. As an alternative, we present a metadata collection, organization, and query architecture that lowers the cost of entry by reordering steps in the database design methodology. Metadata is gathered and loaded immediately using a simple RDF-based data model. After the data is loaded, they are partitioned according to their *signature*, which is inferred by the system. Additional grouping structures (views) can be defined by users through a web interface. Organizing data via signatures results in a form of schema that facilitates query expression and allows efficient query evaluation. By performing schema design semi-automatically only after the data is loaded, we sidestep the primary obstacles to database adoption in a scientific environment. Additionally, the schema structures we discover can be adapted to changes in the metadata stream, avoiding schema evolution problem.

The context for our work is the CORIE Environmental Observation and Forecasting System, designed to support study of the physical processes of the Columbia River estuary. The CORIE system both measures and simulates the physical properties of the estuary, generating 5GB of data and over 20,000 data products and subproducts daily, including visualizations, aggregated results and derived datasets. The data products are consumed for many purposes, including salmon habitability studies and environmental impact assessments. The number of files in the repository is around 6 million and growing by tens of thousands a day. The existing repository is organized as a collection of directory structures on a Linux filesystem. Descriptions for these files are found encoded in the file name, in the content of the file itself, or in accompanying files.

For example, the data products generated from the simulation outputs have descriptive information encoded within their filenames. Figure 1 shows an example of descriptions found in and around the target file.

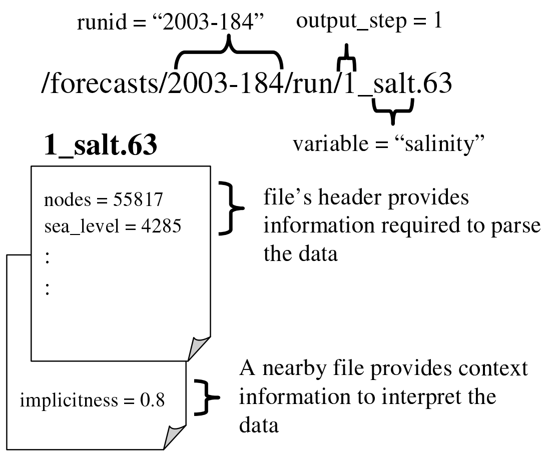

runid = "2003-184"
output_step = 1
/forecasts/2003-184/run/1_salt.63
variable = "salinity"
1_salt.63
nodes = 55817
sea_level = 4285
:
:
file's header provides information required to parse the data
implicitness = 0.8
A nearby file provides context information to interpret the data

Fig. 1. Metadata encoded in file names, headers, paths, "nearby" files.

The data repository is exhibiting growth in both the number and types of files stored. We have encountered at least the following kinds of files:

- Observation data downloaded from field sensors.
- Simulation inputs downloaded from external servers, in varying stages of pre-processing.
- Simulation parameter files.
- Simulation control information such as logs, status flags, and saved checkpoints.
- Raw simulation outputs in multiple formats.
- Derived data products including reduced datasets, images, visualizations, reports, and aggregations of multiple datasets.

Additionally, there is a growing need for access to these data by people other than the data creators; the heterogeneity of the sources of metadata across these file types is an impediment.

To support general data access for non-expert users through these *scattered* metadata, we describe two alternatives. Our initial plan was to build a straightforward database application. However, there was a possibility that such a system would be difficult to deploy and maintain without considerable ongoing help from us. After further thought, we aimed higher: a system for automatically and adaptively organizing metadata.

1.1 The "Obvious" Solution

Our first thought was to define a schema to house the metadata in an RDBMS. Next, data creators populate the schema by inserting a tuple for every file in the repository. Then, a database application is constructed and tested for searching the database.

There are several disadvantages to this approach. The data creators are not necessarily data modelers; schema design requires somewhat specialized skills. Even proficient data modelers would have trouble: The schema should be tuned to declared use cases, but since new users are coming online continuously, not all use cases are known at design time. We could undertake the modeling ourselves, but the effort would still require a lot of interaction with the scientists, who do not have much time to spare.

Data loading is also problematic; we cannot ask the scientists to manually enter 6 million tuples. Portions of the data-loading task can be automated since the data creators are often proficient programmers, but their knowledge does not necessarily include RDBMS languages or APIs. Data loading constitutes a significant up-front cost to this approach which must be paid before any benefit is realized, so it is difficult to persuade the scientists to diligently record metadata during the interim. Accurate test data with which to proof the system is also needed but has an even lower priority.

Once the data is loaded, the schema may be difficult to validate since the database is free to diverge from what is encoded in the filenames, file content, etc. Database APIs and a specialized schema constitute a "wide" interface between data creators and metadata managers: Small requirements changes can result in significant interface changes. A question of responsibility also arises: if a new metadata field is required, who must update the database and possibly modify or reload data?

1.2 An Alternative

Since the metadata sources may change frequently, those responsible for the changes must also be responsible for their extraction. However, aside from extraction, all other tasks should be automated and adaptive to change.

In our approach, data creators are only responsible for "descriptor extraction" using a very simple data model based on RDF. Descriptors are extracted using *collection scripts* written by the data creators in their choice of programming language. These descriptors are loaded as-is into a generic RDBMS schema. Generic schemas are difficult to query and browse, and exhibit poor performance. We therefore partition the data into groups according to patterns found in the metadata. These groups act as a schema to expose structure, simplify query expression, enable browsing, and improve response time. Additional groupings of data can be defined by users through a web interface providing a personalized view of the repository.

This approach offers several advantages. A narrower interface between metadata managers and data creators is required: To accommodate changing metadata requirements, simply update the collection scripts and the system reacts appropriately. Metadata can be harvested prior to finalizing a schema making benefits more immediate. A single, generic metadata delivery interface, controlled by collection scripts can both load new data and update existing data. Since the schema is built dynamically, new views can be easily defined to support new users or new tasks. We use the term *emergent semantics* to describe the paradigm of harvesting data first and organizing it second. By organizing and publishing the gathered metadata back to users, they retain responsibility and control over validation and adjustments. The collection scripts used to gather metadata can be modified and re-executed to update the database.

Several challenges must be overcome to realize these benefits. Queries over a generic RDF schema are expensive and difficult to express. Techniques for identifying patterns in the metadata stream must be devised. Once patterns in the metadata are found, we must convert them into structures (signatures and signature extetnts) to facilitate query expression and processing. Users may also wish to impose their own structure on the data. As new users and new use cases are introduced, requirements may change. If the metadata stream changes, can the previously found structures adapt? In some cases, users may wish to promote previously found structures into hard constraints, rejecting non-conforming metadata. This paper presents an architecture that takes initial steps towards resolving these issues.

2 Harvesting Metadata

To extract metadata from scattered sources, we rely on collection scripts to assign $\langle property, value \rangle$ pairs to each file. The output of collection scripts are then interpreted as $\langle subject, property, object \rangle$ triples. In our application, collection scripts are written primarily, but not exclusively, by the environmental scientists. They are proficient programmers and are most familiar with the file description encoding schemes at the sources.

Collection scripts may be written in any language. To fire the collection scripts and harvest the metadata we provide a program *harvest*. The harvest program accepts a set of *applicability rules* and a target directory as input, and produces $\langle file, property, value \rangle$ triples for each file in the directory. An applicability rule consists of a regular expression, the path to a collection script, and a path to an interpreter for the collection script, if necessary. The harvest program recursively walks the target directory, testing the rules for each file. A rule is activated if its regular expression accepts the string formed by the path and file name being tested. For each activated rule, the program executes the appropriate collection script. Multiple rules may fire for the same file, and the same collection script might be triggered by multiple rules.

For the file in Figure 1, a simple rule matching the ".63" extension triggers a collection script that emits triples from the file name (e.g., `<1_salt.63, variable, salinity>`), from the content (e.g., `<1_salt.63, nodes, 55817>`), and from nearby files (`<1_salt.63, implicitness, 0.8>`). Alternatively, these three sources could have been accessed by three separate collection scripts (requiring three rules), all triggered by the same regular expression. Rules give collection script writers significant flexibility with regard to software design. For example, these three collection scripts may be written by different authors with different metadata requirements.

Each script is expected to accept one command line argument, a file path, and call a function `Assert(property, value, type)` for each metadata item it wishes to record for the file. The `Assert` function is provided by the system for a few languages (Perl, Python, C). If scripts are written in a language for which the assert function has not been defined, the script may simply emit the property, value, and type arguments to standard out in a comma-delimited format.

Initiating a separate process for each script execution is prohibitively expensive, as there are around 20,000 files per run and a large repository of existing runs. To improve performance, the harvest program uses embedded interpreters for the two most popular languages, Perl and Python. For simplicity, the users still write their code as if it is to be run from the command line; the harvest program wraps each script in a virtual environment and executes it in the same address space.

The `Assert` function, called from within a collection script, emits a triple through temporary files or stdout. The stream of triples produced by a series of calls made by amultiple collection scripts is then delivered to a relational database for analysis, as described in Section 4.

3 Modeling Metadata

To interpret extracted metadata, we must choose a data model to capture the descriptions. Rich data models that can capture complex relationships, constraints, and operations allow the same real world concept to be modeled in many different ways. To choose an appropriate translation into the data model's structures, modeling experts meet with domain experts, clarify the details, and construct a prototype. This offline coordination is expensive in terms of time required, and usually involves several iterations. Other confounding issues impede the data modeling efforts: dirty data, redundancy, ambiguity. Finally, if changes are expected to be frequent, this design process might be repeated several times.

We sidestep these issues at this stage by adopting a very simple data model. Very little transformation need be performed and very few initial modeling decisions need be made to get the metadata recorded and queryable. Once the descriptions are in a uniform, machine-readable format, richer modeling features can be applied. Figure 2 contrasts the manual effort required by the two approaches.

To accommodate the heterogeneity of the data sources and simplify the scientists' collection scripts, we adopt a data model based on the Resource Description Framework (RDF) [7] consisting of $\langle subject, property, object\rangle$ triples. We sacrifice expressive power for simplicity and uniformity: Simple facts can be recorded without regard to overall structure.

Of course, the additional expressive power of richer data models is important for query formulation and processing. To recover these benefits, we must derive relationships encoded in the $\langle subject, property, object\rangle$ triples. For example, if several files all have the same set of properties, we can group the files together to simplify queries and improve their performance.

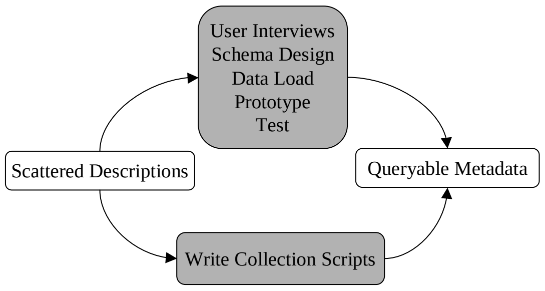


Fig. 2. Two methods of organizing data with different amounts of offline activity.

3.1 RDF: Shortest Route to Machine-Processable Metadata

We have been using the RDF terminology of "subject," "property," and "object." The word "object" connotes a fundamental feature of the RDF data model: Properties can reference arbitrary resources, making the overall data model a graph. This feature distinguishes RDF from tree-based data models, e.g., LDAP [15].

RDF was designed to enable interoperability and automation by adorning web resources with simple machine-processable metadata. Our domain is similar: We adorn files, the substrate of observation and simulation systems, with metadata to enable access.

Formally, an RDF graph is a set of triples of the form (s, p, o). The property (or *predicate*) p is drawn from a set of Uniform Resource Identifiers R. The subject s is drawn from the union of R and a set of *blank nodes* B. The object

o is drawn from the union of R, B, and a set of literals L. Blank nodes act as existentially quantified variables over the domain of a particular graph. RDF graphs can be drawn as a graph using the union of R, B, and L as nodes and drawing an edge for every triple. Edges are labeled with the property URI, and non-blank nodes are labeled with either the URI or the string literal in quotes. Blank nodes are, as might be expected, unlabeled.

We forego the use of more advanced features of RDF, such as RDF Schema for defining classes and class membership, and reification for asserting RDF statements about other RDF statements. Use of these features would again require the up-front modeling effort we hope to avoid. We do allow literals to carry a type as supported in RDF. Note that our interest is in the RDF model rather than the XML-based syntax for RDF [14].

Even without the advanced features, the RDF data model can be used to capture the information expressed in much richer data models, though redundancy is often introduced, and query facilities are sacrificed.

3.2 Interpreting RDF in our Application

To model our metadata in RDF, we must first specify the sets R, B, and L. Each file for which metadata is collected is given a URI derived directly from the file's path. Additionally, each unique property emitted is assigned a URI of the form `property:<`*propertyname*`>`. Our "objects" may be string literals or file URIs. Figure 3 shows an example of an RDF graph used to model our domain.

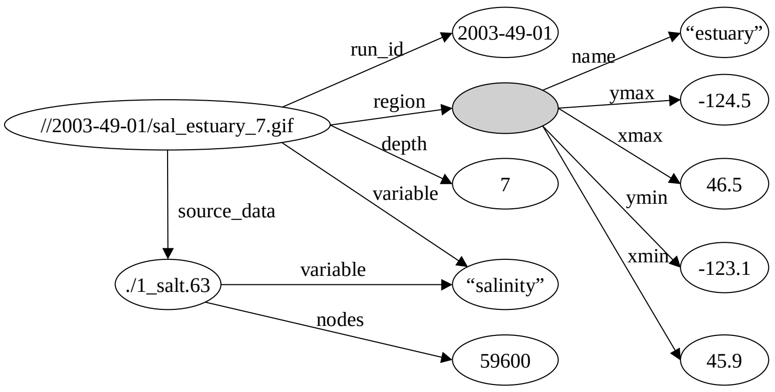


Fig. 3. An example of RDF used to model files and properties in our domain.

The RDF data model allows a URI to appear as both a property and an object in two triples x and y. In our application, however, property URIs may not appear as objects, though file URIs may. Files that reference other files are especially useful for tracking *provenance*; data products can reference the datasets they are derived from, and datasets can reference the executable of the simulation code used to generate it as well as the inputs.

The graph structure of RDF is one prominent feature RDF; the other is the concept of blank nodes. Blank nodes are used to model anonymous structured types. In Figure 3, we use a blank node (the gray oval) to capture the idea that the region named "estuary" corresponds to the bounding coordinates xmin = 45.9, ymin = -123.1, xmax = 46.5, ymax = -124.5.

Collection scripts are not capable of emitting blank nodes directly. To retain simplicity, elements of structured property values are attached to the files directly. The metadata stream can be seen as "denormalized" in the sense of relational database design. We sacrifice such advanced modeling features at this stage, favoring instead simple collection scripts. In Section 4.4, we describe how these structured types can be correctly abstracted.

4 Analyzing RDF using RDBMS

We have modeled our domain of files and metadata using RDF. Our use of RDBMS technology to store and manipulate RDF data is the subject of this section. As reported in the literature, RDBMS are an appropriate choice for managing RDF data. However, since we have no schema information at this stage, we adopt a generic RDF schema that captures the RDF triples directly. This approach has been called the *vertical representation* [4], *generic schema*, or *edge schema* [11] in the literature for various data models and applications. We discuss querying the data in terms of SQL: Query languages for RDF have been proposed [23, 16], but either tend to rely on schema information encoded with RDF Schema [8], or have not demonstrated scalability.

Two variants of the generic RDF schema appear in Figure 4a. A `Triples` table stores RDF triples in terms of string representations of the URIs. To improve performance, a `Resources` table might abstract the string representations of uris and leave integer keys in the `Triples` table for faster processing of set operations on resources (Figure 4b).

4.1 Performance

The use of a single table for storing the RDF graph faithfully supports the RDF specification, but makes queries over RDF difficult to write. For example, to retrieve all the files (subjects) which exhibit the properties `variable`, `region`, and `plottype`, 2 self-joins (in standard SQL) are required. Figure 5 shows the SQL for such a query over the schema of Figure 4a. To ask for files that exhibit an n-attribute signature, an $n-1$-way join is required. More generally, each property to be viewed (projection) or used to filter results (selection) contributes a join on the `Triples` table. The SQL is awkward to generate or write, and performance is terrible with respect to the number of conditions. Our current database captures 30 million RDF triples describing 6 million files, so performance is important.

a) **Triples**

subject	property	object
file://forecasts/2003-184/images/anim-sal_estuary_7.gif	property:region	estuary
file://forecasts/2003-184/images/anim-sal_estuary_7.gif	property:variable	salt
file://forecasts/2003-184/images/anim-sal_estuary_7.gif	property:plottype	animation
file://forecasts/2003-184/images/anim-sal_estuary_7.gif	property:source	file://forecasts/2003-184/run/1_salt.63

b) **Resources**

id	uri
1	file://forecasts/2003-184/images/anim-sal_estuary_7.gif
2	file://forecasts/2003-184/run/1_salt.63
3	property:region
4	property:variable
5	property:plottype
6	property:source
7	estuary
8	salt
9	animation

Triples

subject	property	object
1	3	7
1	4	8
1	5	9
1	6	2

Fig. 4. Two possible schemas for modeling RDF in RDBMS.

```
SELECT r.subject as file, r.object as region,
       p.object as plottype, v.object as variable
FROM   statements r, statements p, statements v
WHERE  r.subject = p.subject
  AND  p.subject = v.subject
  AND  r.property = ‘property:region’
  AND  p.property = ‘property:plottype’
  AND  v.property = ‘property:variable’
```

Fig. 5. SQL to find resources exhibiting properties `variable`, `region`, and `plot type`.

Using a single `Triples` table, we can express arbitrary RDF graphs. In practice however, we observe that the files have one of several *signatures*. For example, most data products derived from simulation output are associated with a variable (e.g., salinity, velocity, temperature), a region (e.g., estuary, far, plume), and plot type (isolines, transect, timeseries). Figure 6 shows a transect plot (a) and an isolines plot (b) for the salinity variable in the estuary region.

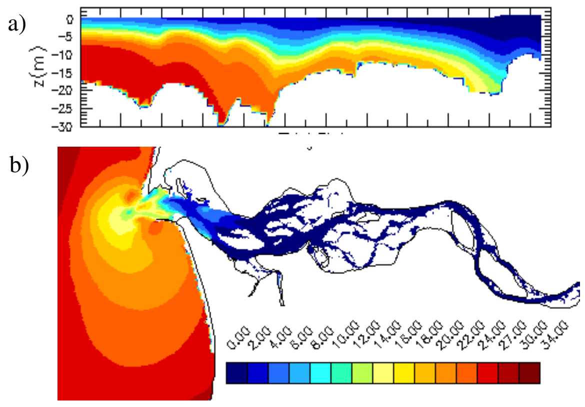


Fig. 6. Two data products from the CORIE simulation system.

Each of these plots exhibit the three metadata attributes involved in the query of Figure 4. If this signature is common, we can improve performance by materializing the signature extent proactively, reducing the query to "`SELECT * FROM <signature name>`"

4.2 Signatures

A *signature* is the set of properties used to describe a particular file. A *signature extent* is a set of files that all exhibit the same signature. Computing the extents involves finding the unique signatures exhibited in the current database instance of the `Triples` table. We materialize these extents as tables in the relational schema. This approach is only wise if the number of signatures is small relative to the number of files. Considering that there are intuitively related sets of files such as data products, simulation outputs, etc., this assumption appears valid. Note, however, that the number of signatures is not simply equal to the number of scripts. Multiple scripts, possibly written by different people, can fire for the same file. Further, nothing constrains a script to output the same properties for every file. For example, we use collection scripts that output different parameters depending on which version of the simulation is being considered.

Computing signatures is similar to extracting association rules in a data mining application. Instead of identifying which item groups tend to be found together in a single customer order, we find properties that tend to be found associated with a single file. However, we need not invoke the full power of data mining, since we are only interested in association rules of 100 percent confidence; most data mining algorithms use sophisticated techniques to estimate the confidence level and prune the search space.

Computing the signatures might seem to require application code, but we can exploit set-valued attributes to express the computation using a `GROUP BY` in SQL. We simulate set-valued attributes in Postgres [26] using sorted arrays. The SQL for this computation is given in Figure 7. The `GROUP BY` operator uses an aggregation function (`array_accum`) that constructs an array dynamically from the elements of the group. If the data are sorted prior to evaluating the `GROUP BY`, the arrays may be compared for equality, and therefore duplicate arrays may be removed.

```
SELECT DISTINCT array_accum(property) as signature
FROM (
  SELECT subject, property
  FROM   Triples
  ORDER BY subject, property
)
```

Fig. 7. SQL to compute unique signatures.

For each signature, a signature extent can be computed with attributes for each property in the signature. The materialized view can be computed using SQL of the form in Figure 5, but a much faster method again uses set-valued attributes and a *crosstab* operation. The crosstab operation is known by various names in the literature. The designers of the SchemaSQL language [18] propose a *fold* operator, and Online Analytical Processing (OLAP) systems include a CUBE operator [12].

The SQL in Figure 7 can be adjusted slightly to generate the profile for every subject in the triples table (Figure 8a) rather than the set of unique signatures. We can materialize this adjusted query, calling the materialized table FileSignatures. Now, for each unique signature s, we want to 1) find the set of file URIs F_s whose signature is s, 2) find the set of triples T_s whose subject is in F, 3) sort the results by subject and property[1], and 4) perform the crosstab operation to convert the properties into attribute names in the query results. Steps 1-3 are captured by the SQL in Figure 8b. The query result can be materialized as a signature extent and is available for further querying. This approach is far more efficient than the SQL in Figure 5, since that approach requires $O(n)$ joins, where n is the number of attributes in the signature. The crosstab algorithm is linear with respect to the number of files and the number of triples.

performance results comparing join-based signature materialization vs. crosstab signature materialization

[1] The single-pass crosstab algorithm requires the data be sorted.

a)

```
SELECT subject,
       array_accum(property)
       as signature
FROM (
  SELECT subject, property
  FROM   Triples
  ORDER BY subject, property
)
GROUP BY subject
```

b)

```
SELECT t.subject,
       t.property,
       t.object
FROM   FileSignatures fs,
       Triples t
WHERE  fs.signature = <s>
  AND  fs.subject = t.subject
ORDER BY t.subject, t.property
```

Fig. 8. SQL to compute the signature of each subject.

Signatures provide an overview of the data collected with the harvest mechanism. Users can check their assumptions about the metadata that is collected by reviewing the set set of signatures found and which signature a particular file exhibits. For example, in our domain, users expected that all files containing parameters to the simulation would have the same signature. All the parameter files were thought to exhibit the same properties, though with different values. However, two similar but distinct signatures were found by the system, exposing the fact that that parameter files created for different versions of the code had a slightly different format. A new collection script to parse the older parameter files was created.

4.3 Update Semantics

So far, we have discussed how to efficiently compute signature extents from a large set of RDF triples, but how are these data to be maintained as the RDF stream changes? New simulations result in new files being delivered with potentially new signatures. As requirements change, existing collection scripts will evolve and new collection scripts will be written. New applicability rules may also be defined, causing existing collection scripts to operate on new files. In each of these situations, there are three classes of changes that must b accommodated: new files with potentially new signatures, new properties for existing files resulting in a change of signature, or new values for existing properties of existing files. How the system reacts to these possibilities is the subject of this section.

New Files When new files are presented to the system, they may or may not exhibit a previously encountered signature. If they do, then a new tuple may be inserted into the appropriate signature extent, and no schema changes are required. If the signature has not been previously encountered, then a new signature extent can be created and a the new tuple inserted into it.

New Properties, Existing Files If a collection script is modified and then re-executed over existing data, two possibilities must be addressed. First, the newly modified collection script may emit different values for an existing property of an existing file. We describe this case in the sequel. Second, a collection script may emit an entirely new property for an existing file, changing the file's signature. Since we have previously computed signature extents and materialized them, we must massage the database to accommodate this new signature.

One choice is to simply drop and reload the database whenever the collection scripts are modified or another change is made, but an incremental strategy is desirable. If the file's new signature has been previously encountered, then an extent exists for it, and the tuple can be deleted from the old signature extent and inserted into the new one. If the signature has not been previously encountered, then a new signature extent must be created first.

New Values, Existing Properties, Existing Files To support update of property values, we must modify our interpretation of RDF. The RDF data model allows a single property to be asserted more than once for a particular subject. Instead, we enforce that no two triples in a given database instance may have the same subject and property. To accommodate updates to a property's value, we interpret the assertion of a property as a command to overwrite the previous value, if one exists. This interpretation implies that there is a total order on the RDF triples, corresponding to their arrival in the system.

To define the total order, we must consider executions of the harvest program, executions of collection scripts, and the order of assertions within a collection script. Executions of the harvest program can be serialized naturally by the server. However, within a single execution of the harvest program, multiple collection scripts may attempt to assert the same property for the same file. To resolve this ambiguity, we defer to the order of the applicability rules. Recall that these rules specify which collection scripts to fire for which files. The latest rule that causes a duplicate assertion to be made overrides any previous rules. The applicability rules are managed by the data creators just as the collection scripts are, so changes can be made unilaterally. There is also the possibility that a single collection script may assert the same property for the same file twice in a single execution. We rely on the script author to understand that only the later value will be retained.

To minimize redundant work, we keep track of which collection scripts were executed with which execution of the harvest program. If the harvest program is re-executed over existing files, we only fire those collection scripts that are new or have been modified since the last execution. This policy reduces the amount of time required to harvest metadata and reduces the conflicts that must be resolved during loading.

Batches The semantics we have just described can result in a great deal of data reorganization work. Very small changes to a collection script can result in many insertions, updates, and deletions, and potentially require new database tables to be created. There is a tradeoff between processing new metadata incrementally and buffering batches of metadata to process all at once.

If we enact the policies above after every triple arrives, we will obtain the correct semantics, but we will fire multiple SQL statements for every triple. The volume of data is large enough that his approach is unacceptable. Fortunately, triples generally arrive in batches, usually corresponding to an execution of the harvest program. We interpret the triples in a batch as arriving simultaneously, and only compute the new schemas no more than once per batch. As described, there may be duplicate properties for one file asserted in a single batch. We remove these duplicates inside the harvest program, according to the semantics above, prior to database load. Therefore, even though we consider each batch as a simultaneous set of updates, no ambiguity can arise.

In our application there is a natural batching strategy. For each run of the CORIE simulation code, around 20k files and around 100k triples are loaded into the system. Considering these a batch we can balance the tradeoff between stale data and computing signatures that are replaced without ever being accessed.

Deletes Deletes are more problematic. If we do not support deletes, there is no way to remove properties associated with files to support changing requirements, which prevents the correct signatures from being exhibited. But in order to support deletes, we must allow authors of collection scripts to specify that a file should no longer have a given property. (Note that we do not wish to support deletes of files themselves. The fact that a file once existed is important lineage information.)

We entertained two possible semantics for deletes: 1) Whenever a batch of metadata arrives for a file, assume the old metadata is incorrect and drop it all. These semantics agree with the update semantics already specified; new triples will still "replace" old triples since the old triples are deleted beforehand. The problem with these semantics is that collection scripts are prevented from extending the metadata associated with an existing file. They must produce all metadata for a file in every batch. Another choice is to 2) allocate a special URI "value:delete" that is interpreted as a deletion when it appears as the object of a triple. The problem (other than the inelegance of requiring a specially interpreted value) is that the collection scripts must produce "delete commands" as well as standard metadata, complicating their design and implementation. Although the collection scripts are expected to undergo refinement and change, using them as an interface to perform "one time only" database mutations is awkward. Currently, our implementation uses the second semantics.

4.4 Blank Nodes Revisited

Recall that blank nodes represent placeholders for unnamed subjects and objects in an RDF graph. Semantically, blank nodes are existentially quantified variables. In our design, collection scripts can not emit make use of blank nodes. The subject of every RDF triple represents a physical file. Collection scripts have no way of expressing triples whose subject is a blank node.

The disadvantage of this design is that collection scripts must redundantly emit the same information for many files. For example, all region information (xmin, xmax, ymin, ymax, region name) is linked directly to the file rather than linked to a blank node as in Figure 3. The concept of region is a separate entity and should be abstracted using a blank node in the RDF graph. In a relational setting, we note that region name functionally determines xmin, ymin, xmax, and ymax. Normalization procedures prescribe that we decompose a signature extent involving these properties by extracting a “region” table.

Since the collection scripts cannot express this feature of the data, we rely on tools to discover it after the data has been loaded. Several algorithms exist for discovering these functional dependencies [21]. The results of these algorithms can be used to further refine the schema. The tradeoff is twofold: First, a more complex schema may make data exploration more difficult for users unfamiliar with relational principles. Second, the processing of new RDF tuples becomes more expensive as the schema becomes more complex. The logic to transform a batch of RDF triples into a set of tuples becomes complex when considering updates, deletes, and now decomposed tables. Since some changes to collection scripts may require a significant amount of data to be corrected or reloaded, minimizing the complexity of the load operation is important.

For these reasons, we compute functional dependencies and perform prescribed decomposition only offline after a batch of data has been loaded. Further, we publish the data in terms of signatures, even though the underlying schema may be more complex. Modern RDBMS support this form of logical data independence quite well.

Performance currently is adequate considering the database represents over 6 million files and over 30 million RDF triples. After applying the techniques discussed in this paper for speeding up execution of collection scripts, identification of signatures, and population of signatures, we can re-load the entire repository, including re-harvesting the RDF triples, in a few hours. This level of performance is necessary, since there is a possibility that the majority of the database must be effectively reloaded if significant changes to collection scripts are made at one time.

5 Web Interface

Having imposed some structure on the metadata stream, we publish the data to the web for validation, exploration, and query. Generally changes to a database schema require changes to an application since applications reference table names and column names explicitly. However, an application can be designed generically by retrieving table names and column names from the catalog rather than hard-coding them. The HTML to display a table’s information is generated dynamically from the catalog. We took this approach, extending source code for a free generic database interface [24]. The entry point is a list of tables in the current database.

5.1 Validation Using Signatures

In Postgres, materialized views are not directly supported, and are therefore implemented using tables. The initial view in the web interface shows the signatures inferred by the system. In addition to a system generated name, the properties that make up the signature are displayed, giving an overview of how the files are organized in the database.

The list of signatures computed by the system provides immediate information about what sort of data is being captured. For example, users can see that some signatures are very similar, differing in only one attribute (e.g., `<plottype`, `region`, `variable`, `animation>` versus `<plottype`, `region`, `variable>`). This configuration suggests that the presence of the `animation` property is acting as a boolean. If the `animation` property exists, then the file type is an animated gif; if not, it is a single frame. The value of the `animation` property may be irrelevant. Perhaps a better configuration is for files in both signature extents to assert a `GIFtype` property, with either the value `animation` or `single frame`. If this configuration improves understanding, appropriate adjustments to the collection scripts can be made, and the database schema will adapt.

5.2 Query

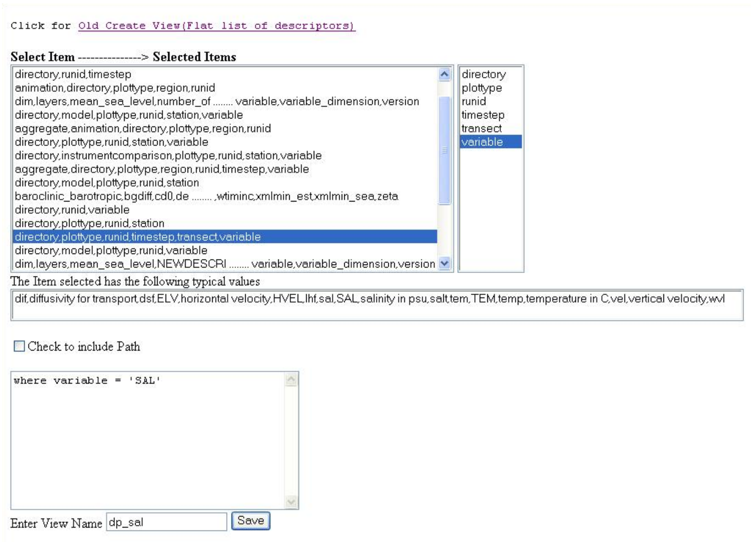
The interface also allows users to construct simple queries via the signatures. Properties encountered in the metadata stream are shown, along with a range of values. This kind of summary information is important for validation as well as query formulation. For example, one of the parameters to the simulation, implicitness, indicates a coefficient controlling the weights of implicit and explicit methods for solving for each timestep. Those unfamiliar with the application can see the range of values found in the database before issuing a query. In fact, those unfamiliar with the term have used these value ranges as clues to its meaning.

Figure 9 illustrates the query interface. To construct a query, users select signatures from the left, which populates a list of properties on the right. Undesired properties from the signature can be removed by double clicking on them. If a single extent is selected, then these properties simply form the select clause of a SQL statement. Selection conditions can be added at the bottom of the screen. The query can also be saved as a materialized view, which is refreshed at system-configurable intervals.

After retrieving the results of queries, the user can view the file (useful for images) or make annotate the file. Figure 10 shows the results of a query, and demonstrates how the user can click a link access the file itself.

If multiple signatures are selected, the natural join between the two is computed. Allowing only the natural join limits the expressiveness of this query interface, but has suited our purposes for the application thus far. Defining the precise query features necessary in this domain is an open problem.

The query interface allows us to make efficient use of the scientists' time. We can meet with them for an hour, browse through signature extents, and construct views on the fly. In several of these meetings, we have discovered anomalies in the



Click for Old Create View(Flat list of descriptors)

Select Item --------------> Selected Items

```
directory,runid,timestep                                                    directory
animation,directory,plottype,region,runid                                   plottype
dim,layers,mean_sea_level,number_of ....... variable,variable_dimension,version   runid
directory,model,plottype,runid,station,variable                             timestep
aggregate,animation,directory,plottype,region,runid                         transect
directory,plottype,runid,station,variable                                   variable
directory,instrumentcomparison,plottype,runid,station,variable
aggregate,directory,plottype,region,runid,timestep,variable
directory,model,plottype,runid,station
baroclinic_barotropic,bgdiff,cd0,de ....... ,wtiminc,xmlmin_est,xmlmin_sea,zeta
directory,runid,variable
directory,plottype,runid,station
directory,plottype,runid,timestep,transect,variable
directory,model,plottype,runid,variable
dim,layers,mean_sea_level,NEWDESCRI ....... variable,variable_dimension,version
```

The Item selected has the following typical values

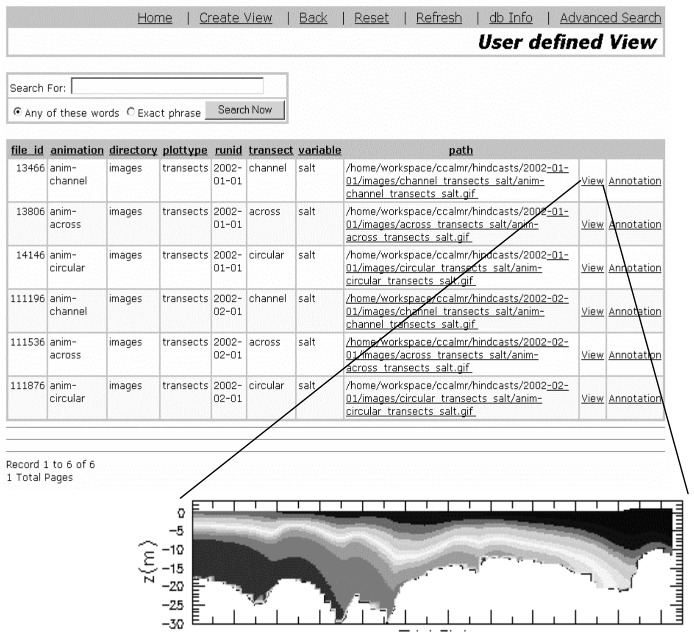
dif,diffusivity for transport,dsf,ELV,horizontal velocity,HVEL,lhf,sal,SAL,salinity in psu,salt,tem,TEM,temp,temperature in C,vel,vertical velocity,wvl

☐ Check to include Path

```
where variable = 'SAL'
```

Enter View Name dp_sal Save

Fig. 9. To express queries and saved views, users select signatures on the left and add selection conditions at the bottom.



Home | Create View | Back | Reset | Refresh | db Info | Advanced Search

User defined View

Search For:

(● Any of these words) ○ Exact phrase Search Now

file_id	animation	directory	plottype	runid	transect	variable	path		
13466	anim-channel	images	transects	2002-01-01	channel	salt	/home/workspace/ccalmr/hindcasts/2002-01-01/images/channel_transects_salt/anim-channel_transects_salt.gif	View	Annotation
13806	anim-across	images	transects	2002-01-01	across	salt	/home/workspace/ccalmr/hindcasts/2002-01-01/images/across_transects_salt/anim-across_transects_salt.gif	View	Annotation
14146	anim-circular	images	transects	2002-01-01	circular	salt	/home/workspace/ccalmr/hindcasts/2002-01-01/images/circular_transects_salt/anim-circular_transects_salt.gif	View	Annotation
111196	anim-channel	images	transects	2002-02-01	channel	salt	/home/workspace/ccalmr/hindcasts/2002-02-01/images/channel_transects_salt/anim-channel_transects_salt.gif	View	Annotation
111536	anim-across	images	transects	2002-02-01	across	salt	/home/workspace/ccalmr/hindcasts/2002-02-01/images/across_transects_salt/anim-across_transects_salt.gif	View	Annotation
111876	anim-circular	images	transects	2002-02-01	circular	salt	/home/workspace/ccalmr/hindcasts/2002-02-01/images/circular_transects_salt/anim-circular_transects_salt.gif	View	Annotation

Record 1 to 6 of 6
1 Total Pages

Fig. 10. Query results allow access to the file itself.

metadata that reflected real anomalies in the source data. We receive comments such as the following:

– "Why are there only 3 salinity data products for this simulation run? I was expecting 4 thousand!"
– "Why are there two copies of the parameter file for every simulation run?"
– "The date should be divided into month and year; we should change that in the collection scripts."

6 Related Work

Our base representation of RDF triples is directly related to the work of Agrawal et al. in managing E-commerce data [4]. Their system also copes with frequently changing attributes by adopting a vertical representation of data that is isomorphic to our base representation of RDF triples. The authors observe that such a representation makes queries difficult to express, as we have also argued. A horizontal view is constructed over the vertical representation of the data to ease this difficulty. The authors' notion of horizontal representation is very different from our signature extents, however. The horizontal representation is a single table that exhibits all attributes exhibited by any item in the database, acting as an implementation of the Universal Relation [20]. Null values are used to fill the table where an attribute has not been defined for an item. The resulting table has thousands of attributes, and many of the tuples are very sparse. Our extents actually partition the data among many "horizontal" tables, each having only a handful of columns and significantly fewer tuples. Using extent tables, we can achieve interactive query performance with at least an order of magnitude more data than Agrawal et al.

RDF databases are gaining popularity in both research and commercial contexts [9, 17, 16]. These systems organize the data according to RDF Schema information provided along with the RDF data. Contrast this situation with our domain, where the schema information is not recorded explicitly and is frequently changing. Systems that allow querying of unstructured RDF [23] operate via graph matching and have not yet demonstrated scalability. Further, the users are required to "fish" for query results, since no schema structures are present to guide query formulation. Guha has done some early work on a scalable native RDF database that does not rely on RDF Schema, but thus far results are not provided [13].

More recent work by the same group advocates views over RDF databases constructed as virtual RDF classes [19]. However, users are charged with building the views using a specialized language. We are taking steps towards deriving appropriate views given only the base RDF data.

Relational representations of XML data are also of interest. Some depend on Schema information, which we do not have. Earlier work compares generic representations of XML. The *edge schema* captures parent-child relationships between nodes in the XML tree, and is very similar to the common representation of RDF triples we have adopted. Binary representations involve separate tables for each element label, with results constructed via joins. Several groups have shown these representations to perform poorly in all but highly specialized domains.

Metadata standards such as Dublin Core [2] and those produced by the Federal Geographic Data Committee (FGDC) [3] provide pre-defined schemas for metadata. These standards are usually intended to be human-readable, and are not usually amenable to structured querying. Further, the standards are often intended for large granularity datasets rather than individual files. For example, in our domain, an entire year's worth of simulations may be described by a single FGDC document.

However, the use of metadata standards is inconsistent with specified intents; fields are left blank, misinterpreted, or overloaded with additional meaning. These incompatibilites belie the challenge in defining a general metadata schema for a wide variety of domains. We advocate tools that adapt to changing requirements rather than restrict them.

Research on ontology models and tools has become commonplace in the last few years. The OWL languages for capturing ontological properties [6], based on RDF, have become W3C recommendations. While work on ontologies show promise for capturing complex relationships across domains, adoption has been slow due to practical issues. Usually ontology tools assume that a Ontologist, Knowledge Engineer, or Cybrarian [22] is available to model source metadata in the target system. This task is far from trivial, which is why job titles such as those mentioned have been invented. Our framework streamlines the process of converting scattered source data to machine-processable metadata. Inferring and exploiting the complex relationships supported by ontology models remains future work.

The Grid [10] community has recognized the need for a comprehensive metadata management solution in Grid environments. The MCAT metadata information catalog supports metadata processing in the context of the San Diego Super Computing Center's Storage Request Broker (SRB) [5]. More recently, the Metadata Catalog Service (MCS) [25] has been shown to perform well under heavy query workloads and large database sizes [1]. Our goals differ from these systems, however. We are working toward a system that can adapt to evolving requirements with little or no user intervention. The MCAT system is designed to process metadata queries over using specialized query structures. While the schemas providing structure to the data are not necessarily fixed, users must explicitly update the schema to reflect changing requirements. The ability for MCAT to handle frequent changes of this kind over populated databases is unclear. However, the infrastructure issues such as location transparency and replication addressed by this research are important and complementary to our own work.

7 Future Work

There are several research directions to follow based on this work, primarily in the area of finding and exploiting richer structures in the RDF data. Currently we infer simple groupings from the data patterns, and we discussed using functional dependencies to further refine the schema. However, views defined by users in the web interface provide very important information as to how users will access their data. Beyond simply materializing these views, the schema can be designed to make these user views efficient. Finding automatic techniques to transform the schema based on user-defined views remains open.

Although we acknowledge that inferred functional dependencies can lead to improved schema design, we have not applied the same analysis to other data dependencies from relational theory: join dependencies and inclusion dependencies. In fact, a rigorous interpretation of these dependencies in our domain would be useful.

We exploit patterns in the metadata stream to facilitate query expression and improve performance. However, a schema serves another purpose as well: enforcement of constraints on the data. If patterns found by the system are acknowledged by users, the system could promote the patterns into hard constraints. After this promotion, violations of the constraints in the metadata stream would be rejected outright as errors. Providing a flexible transition between reactive and proactive pattern exploitation presents interesting theoretical and practical challenges.

The query interface provided via the web supports only simple natural joins, making many useful queries impossible to express. We predict that an interface can be designed that supports all relevant queries in our scientific domain without requiring the full power of SQL.

8 Acknowledgements

This work was supported by

References

[1] MCAT - a meta information catalog. http://www.npaci.edu/DICE/SRB/mcat.html.

[2] Dublin core metadata. http://dublincore.org/index.shtml, 1997.

[3] Content standard for digital geospatial metadata. http://www.fgdc.gov/metadata/metadata.html, 2004.

[4] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *The VLDB Journal*, pages 149–158, 2001.

[5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of the Centers for Advanced Studies Conference*, Toronto, Canada, November 1998.

[6] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Web ontology language reference. W3C Recommendation, http://www.w3.org/TR/2003/CR-owl-ref-20030818/, 1999.

[7] D. Beckett and B. McBride. RDF/XML syntax specification. http://www.w3.org/TR/rdf-syntax-grammar/, 2004.

[8] D. Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, 2004.

[9] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. Hendler, editors, *Proceedings of the First Internation Semantic Web Conference*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.

[10] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 1999.

[11] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22:27–34, 1999.

[12] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational operator generalizing group-by, crosstab and sub-totals. In *ICDE*, pages 152–159, 1996.

[13] R. V. Guha. *rdfDB : An RDF Database.* http://www.guha.com/rdfdb/, 2001.

[14] P. Hayes and B. McBride. Rdf semantics. W3C Recommendation, http://www.w3.org/TR/rdf-mt/, 2003.

[15] I. A. Howes, M. C. Smiths, and G. S. Good. Understanding and deploying LDAP directory services. Technical report, 1999.

[16] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *The Eleventh International World Wide Web Conference.*

[17] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying RDF descriptions for community web portals. In *Journees Bases de Donnees Avancees*, pages 133–144, 2001.

[18] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. Schemasql: An extension to SQL for multidatabase interoperability. *Database Systems*, 26(4):476–519, 2001.

[19] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the semantic web through RVL lenses. In *Second International Semantic Web Conference*, pages 20–23, 2003.

[20] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the Universal Relation Model. *ACM Transactions on Database Systems (TODS).*

[21] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12(1):83–99, 1994.

[22] D. L. McGuinness. Conceptual modeling for distributed ontology environments. In *Proceedings of The Eighth International Conference on Conceptual Structures*, 2000.

[23] L. Miller. RDF Squish query language and Java implementation. Public draft, Institute for Learning and Research Technology, 2001. http://ilrt.org/discovery/2001/02/squish/.

[24] E. Robillard. GenericDB. http://www.genericdb.com/.

[25] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Mahohar, S. Pail, and L. Pearlman. A metadata catalog service for data intensive applications. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, November 2003.

[26] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of Postgres. *TKDE*, 2(1):125–142, 1990.