ObsDB: A System for Uniformly Storing and Querying Heterogeneous Observational Data

Shawn Bowers[1], Jay Kudo[1], Huiping Cao[2], Mark P. Schildhauer[2]

[1]*Deptartment of Computer Science, Gonzaga University*
[2]*National Center for Ecological Analysis and Synthesis, UC Santa Barbara*
jkudo@gonzagau.edu, bowers@gonzaga.edu, cao@nceas.ucsb.edu, schild@nceas.ucsb.edu

***Abstract*—Earth and environmental scientists collect and use a wide range of observational data. This data often exhibits high structural and semantic heterogeneity due to the variety of data collected and the ways in which observational datasets are structured in practice. However, to address questions at broad temporal, geographic, and biological scales, researchers often need to access and combine data from many observational datasets. This paper presents a system called *obsdb* that helps to address these challenges by providing an integrated environment for storing, querying, and analyzing heterogeneous data based on a semantic observational model. The model allows for ontology-based descriptions of observational datasets and provides a common representation for storing observational data. The obsdb system is built on top of standard relational database technology and provides a declarative query language for accessing observations. Integrated support is also provided for exploratory data analysis, allowing users to call analytical scripts created using the R system over stored observational data.**

I. Introduction

Many analyses within the earth and environmental sciences require access to a wide range of data. These studies often reuse existing data collected by different researchers and research groups to study phenomena at broad geospatial, temporal, and biological scales.[1] However, while there are a large number of repositories for storing earth and environmental data (e.g., [1] contains over 25,000 datasets alone), researchers still struggle to find relevant datasets and combine them into an integrated dataset for analysis.

Discovery of relevant datasets is often a multi-step process that starts by looking for data that matches the user's desired criteria at a coarse-grain level, e.g., ensuring that each dataset contains the general kinds of observations needed. Once candidate datasets are found, users then move to a more involved *exploratory analysis* phase where the details of each dataset are examined. Exploratory analysis often involves, e.g., verifying the contents of the dataset, determining the number of observations of a particular type, running statistical summarizations, and visually comparing different dataset variables.

A key challenge in enabling data discovery and integration involves dealing with the high-degree of structural heterogeneity found in observational data. This includes differences at the format level (e.g., where data can be stored as rasters, tables, etc.), as well as at the schema level (e.g., where datasets containing the same types of observations can be stored using different naming conventions as well as fundamentally different attribute structures). Further, because of the variety of phenomena observed and the types of experiments performed, data semantics play a critical role in discovery and integration (e.g., to specify as unambiguously as possible the types of observations of interest).

Increasingly, high-level observational data models are being developed to help address structural and semantic differences found in observational data (e.g., [2]–[8]). These models aim at providing general approaches for describing and representing observations and measurements found in underlying datasets by defining common "core" concepts (e.g., denoting the entities or features being observed, measurement units and protocols, and context relationships between observations [2], [3]). A key aim of these models is to enable interoperability and uniform access to heterogeneous observational data by abstracting away the underlying representation details found across observational datasets.

Our goal in this work is to extend these approaches by providing useful data-management services for researchers based on a high-level observational data model. Specifically, we are interested in developing tools and techniques to store, query, and access observational data using an observational data model to improve (exploratory) data discovery and integration for earth and environmental scientists.

Contributions. In this paper, we describe a system called ObsDB that (i) extends a generic observational model [3] with query and analysis capabilities, (ii) allows observation and measurement types to be described and searched using semantic-web standards; and (iii) seamlessly combines relational database technology, semantic-web tools, and a popular analytical application (the R system) within a single observational database infrastructure. A key contribution of this work is a formal, declarative query language (ObsQL) for accessing observational data in a simple and declarative manner. A benefit of the language is that query results can easily be converted into tabular data suitable for processing within a system such as R. We also describe an initial feasibility study of the current implementation of ObsDB, and discuss opportunities for future optimization.

Organization. This paper is organized as follows. The observational model used in ObsDB is described in Sec. II.

[1]As simple examples, a researcher may be interested in studying how atmospheric and climate conditions influence tree allometry (i.e., growth rate), or how nitrogen fertilization influences productivity across grasslands.

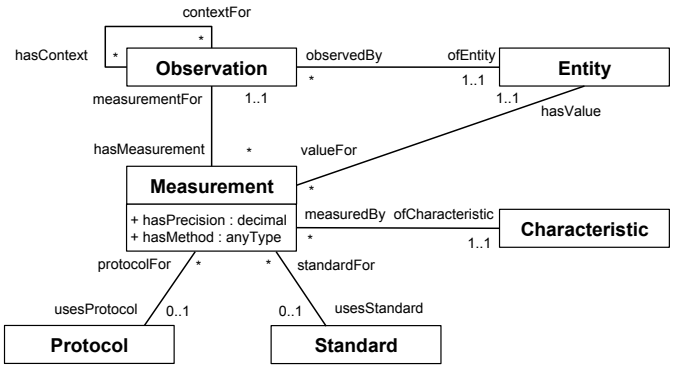

Fig. 1. The observational model used in ObsDB: observations are of made of entities; observations can consist of a set of measurements and participate in a set of context relationships with other observations; and each measurement consists of a characteristic, standard, protocol, precision, and method.

The overall architecture of the ObsDB system is described in Sec. III, and the current implementation is described in Sec. IV. Work related to ObsDB is presented in Sec. V, and we summarize our contributions in Sec. VI.

II. Observational Data Modeling

The heterogeneity of observational data is due to a number of factors [3]: (1) observational data are largely collected by individuals, institutions, or scientific communities through independent (i.e., uncoordinated) research projects; (2) the structure of observational data is often chosen based on collection methods (e.g., to make data easier to record "in the field") or the format requirements of analysis tools, as opposed to standard schemas (*structural heterogeneity*); and (3) the terms and concepts used to label data are not standardized, both within and across scientific disciplines and research groups (*semantic heterogeneity*). A key aim of observational data models (e.g., [2], [3], [9], [10]) is to help alleviate structural heterogeneity by providing a single, uniform structure for representing otherwise heterogeneous datasets. In a similar way, the goal of ontology-based approaches for observational data (e.g., [4], [8], [11]) is to help with semantic heterogeneity by providing a common set of terms and relationships among terms. The ObsDB model combines these two approaches by providing basic constructs for representing observational data (see Fig. 1) that can be further extended with domain-specific ontology terms.

Fig. 1 shows the main constructs and relationships of the observation model used within ObsDB. Note that while Fig. 1 uses UML notation, the model is implemented as an OWL-DL[2] ontology in which classes (e.g., `Observation`) in Fig. 1 denote OWL classes and relationship roles (e.g., `ofEntity`) denote OWL object properties.

An *observation* consists of an *entity* (denoting the object observed), a set of *measurements*, and *context* relationships to other observations. Specifically, an observation represents an assertion that a particular entity was observed and that the corresponding set of measurements were recorded (as part of the observation). Context relationships state that an observation was made within the scope of other "contextualizing" observations. The measurements associated with a context observation are assumed to be constant for the contextualized observation. For instance, if a tree was measured within the context of a plot whose area was recorded, then this is the area assumed for the plot with respect to the tree observation (where it is possible for the plot to be resized for different studies). Context represents a transitive relationship among observations. Namely, all observations serving as context for an observation x are also considered context for an observation with x as its context. Additional constraints with respect to observations, measurements, and contexts are defined in [3].

A measurement consists of exactly one *characteristic* (i.e., an attribute or property of the observed entity) and one value. Taken together, the measurement asserts that the observed entity had the given value for the characteristic. A measurement can optionally contain a measurement *standard* (e.g., a unit), a *protocol* giving the standard procedure used to obtain the measurement, a *precision* denoting the accuracy of the measurement, and a plain-text *method* description denoting the actual procedure used while carrying out the measurement. As shown in Fig. 1, a value is given by an entity such that a designated subclass of entities represent primitive values (e.g., strings and integers), which is similar to how primitive values are treated within pure object-oriented models, and is an often used convention within OWL ontologies [12]. Using entities as values also allows for characteristics to be defined between entities (e.g., to state that one entity was "adjacent to" another), however, we only consider the use of simple values here.

Example 1 (Observations and Measurements). Consider a simple dataset that consists of height and diameter measurements taken of trees within (experimental) plots spread across different geographic sites. Let `Tree`, `Plot`, and `Site` be types of entities (i.e., `Entity` subclasses); `Height`, `DBH` (a type of `Diameter`), `Area`, and `Name` be types of characteristics; and `Meter`, `Centimeter`, `MeterSquared`, and `SiteCode` be types of measurement standards (where `SiteCode` is a simple example of a "nominal" measurement scale [13] in this case denoting a catalog of site names). The following expression gives the measurement and observation types for the dataset, where "→" denotes a context relationship among observations.

```
Tree[Height Meter, DBH Meter] →
    Plot[Area MeterSquared, Name] →
        Site[Name SiteCode]
```

This expression (which follows the notation used in the query language described in Sec. III) is shorthand for an equivalent set of OWL-DL class definitions. For instance, the expression "`Site[Name SiteCode]`" corresponds to a class $\texttt{Site}_{obs}$ with the following DL definition [14].

[2]http://www.w3.org/TR/owl-ref/

$$\texttt{Site}_{obs} \sqsubseteq \texttt{Observation} \sqcap \forall \texttt{ofEntity.Site} \sqcap \exists \texttt{hasMeasurement.}(\forall \texttt{ofCharacteristic.Name} \sqcap \exists \texttt{usesStandard.SiteCode})$$

Based on these types, we can define the following measurements of trees within the same plot and site (again using a shorthand notation based on the query language).

$$(\texttt{Tree } e_1[\texttt{Height} = 19.3\ \texttt{Meter}, \texttt{DBH} = 11.2\ \texttt{Centimeter}], \texttt{Tree } e_2[\texttt{Height} = 20.8\ \texttt{Meter}, \texttt{DBH} = 14.1\ \texttt{Centimeter}]) \rightarrow \texttt{Plot } e_3[\texttt{Area} = 10\ \texttt{MeterSquared}, \texttt{Name} = \text{'A'}] \rightarrow \texttt{Site } e_4[\texttt{Name} = \text{'GCE6'}\ \texttt{SiteCode}]$$

Here each e_i denotes a resource identifier of type `Entity` (e.g., e_1 is of type `Tree`). Further, the expression "`Tree` e_1[`Height` = 19.3 `Meter`, ...]" denotes an instance of an observation of entity e_1 and a measurement of characteristic `Height` and unit `Meter` with the value 19.3. For convenience, we omit the corresponding observation, measurement, characteristic, and unit resource identifiers above.

The observation model is divided into a number of different ontology modules (each defined in a separate OWL file, with a separate namespace, etc.). The core ontology defines the basic constructs shown in Fig. 1; a separate module defines a basic subclass structure for characteristics and entities as well as for defining units and unit conversions; and another module defines standard units and corresponding characteristics and conversions (including SI units and base dimensions). As ongoing work we are defining ontologies to represent commonly used measurements in plant and marine ecology, among others domains. We note that the observation model is compatible with a number of other efforts, including, e.g., the Phenotypic Attribute Trait Ontology (PATO), which is an OBO Foundry [15] ontology containing a large number of commonly used measurement types [10].

In prior work, we have developed approaches to map tabular datasets (via "semantic annotations") into instances of the observational model [3], [16]. These mappings can be used to convert tabular data into, e.g., an RDF representation denoting an instance of the model for the observations contained within the dataset. The O&M [2] observational model (which has similar constructs as those of Fig. 1) provides a number of representation schemes (including XML and OWL [11]) for directly representing observational data originating from sensor networks as well as other data sources. Similarly, approaches such as [6] allow users to input data directly into an observational model similar to Fig. 1. ObsDB compliments these approaches by providing additional management services over data that has been stored (e.g., using one of the above approaches) into a common model.

III. Managing Scientific Observations

Given the model presented in Sec. II, we now define the basic architecture and query language of the ObsDB system. In the following section we provide additional details on the current implementation of ObsDB.

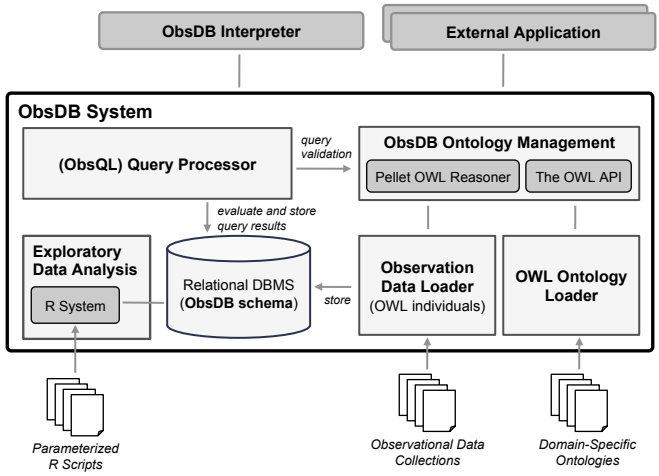

Fig. 2. High-level overview of the obsdb architecture.

A. System Architecture

As shown in Fig. 2, ObsDB is comprised of the six main components described below.

Loading ontologies. Ontologies can be registered with and loaded into the system using the *OWL Ontology Loader*. Once an ontology is loaded, users can access, store, and query observations using terms from the loaded ontology.

Loading observational data. Similarly, users can add observational data to ObsDB using the *Observation Data Loader*. In general, data can be loaded into the system using a number of different formats, including those described in the previous section. However, the current implementation assumes observational data is represented as OWL individuals (i.e., RDF triples denoting observation instances). Ontology definitions and observational data can also be loaded in one step, e.g., when both the type information and the data are stored within a single OWL file. Observational datasets are logically organized into *data collections* within ObsDB, where each collection contains a set of observation instances.

Storing observational data. Loading observational data into the system using the data loader results in the creation of a collection (if the collection has not already been created). The data collection together with the corresponding observations are stored using the ObsDB schema within a relational DBMS. The underlying database of ObsDB manages observations and collections as well as query results.

Processing observation queries. A separate query processor component is used to translate high-level observational queries (expressed in the ObsQL language; see below) into SQL statements that are evaluated over the underlying database. The query language allows users to select observations stored within ObsDB based on a number of conditions. The result of a query is a set of observations, and query results can be saved as (virtual) collections within ObsDB.

Executing exploratory analyses. The *Exploratory Data Analysis* component allows users to run analytical scripts (in the

current implementation, as R scripts) over observational data stored within ObsDB. To execute an analysis, users select a subset of observations (which may span multiple collections) via a query, and then apply the analysis to the query result. ObsDB automatically creates the appropriate data from the query result that is then used to call the analysis. We describe how this is currently implemented in Sec. IV.

Managing ontologies. The *Ontology Manager* is used by the ontology loader, data loader, and the query processor. Specifically, the manager provides a number of basic services (via Pellet[3] and the OWL API[4]) for reading and writing OWL/RDF files, checking consistency, and for generating class hierarchies. The ontology and data loader both use the manager for parsing OWL/RDF files and determining whether a given dataset and ontology is consistent. The query processor uses the ontology manager to help validate and evaluate query expressions. For example, the class hierarchies produced by the manager are used by the query processor to "expand" queries, i.e., to find observations and measurements having subtypes of the types given in a query expression. The ontology manager is also used to export data collections as OWL/RDF files (not shown in Fig. 2).

B. A Query Language for Observational Data

A major feature of ObsDB is its ability to allow users to express and evaluate high-level queries for accessing observations stored within and across collections of observations (i.e., datasets). Queries also form an integral part of exploratory data analysis, e.g., by allowing users to discover and then select relevant portions of data collections for further analysis. Below we first present examples demonstrating the syntax of the *observation query language* (ObsQL) supported within ObsDB, followed by a more formal definition of the language.

Query examples. Within ObsDB users can associate an OWL ontology namespace with a prefix name. As in XML, the prefix (followed by a colon) is appended to the class name to uniquely identify the class. For example, the following ObsQL query selects all observations whose entities are of type `Tree` (where `Tree` is assumed to be defined within the ontology denoted by the prefix name `o1`).

```
o1:Tree
```

It is also possible within ObsDB to omit the prefix name, in which case the system matches all classes of the given name across the loaded ontologies. To simplify the examples below, we omit the prefix labels.

The following query consists of an entity type (`Tree`) and a simple measurement type (denoted by square brackets) consisting of a characteristic type (`Height`).

```
Tree[Height]
```

This query returns all tree observations (i.e., of `Tree` entities) having at least one height measurement (i.e., with a `Height` characteristic). The query above can also include a measurement standard, e.g.,

```
Tree[Height Meter]
```

returns the set of observations of trees with height measured in meters. It is possible to use multiple measurement constraints within a single observation query. For example, the query

```
Tree[Height Meter, Diameter Centimeter]
```

returns the set of observations of trees that contain both a height measurement in meters and a diameter measurement in centimeters. In this case, since `DBH` is a subclass of `Diameter`, observations containing measurements of `DBH` would also be returned.

In addition to queries over entity and measurement types, ObsQL also supports value-based queries. For example,

```
Tree[Height > 20 Meter]
```

selects tree observations with height measurements greater than 20 meters. Other valid comparison operators include $<$, $\leq$, $>$, $\geq$, `LIKE`, and `BETWEEN` (i.e., the standard set of SQL value comparison operators). Multiple value comparisons can be used, as shown in the following query.

```
Tree[Height > 20 Meter, DBH between 12 and 25]
```

Each of the above queries involve simple observation selection conditions. More complex queries can be constructed by combining observation selection conditions through union and context constraints. For instance, the query

```
(Tree[Height Meter], Soil[Acidity pH])
```

returns a set of observations such that each observation is either (1) of a tree entity with a height measurement in meters, or (2) of a soil entity with acidity measured using the pH scale. Similarly, a context query such as

```
Tree[Height Meter] → Soil[Acidity pH]
```

returns tree and soil observations, but where each tree observation has a corresponding soil observation as context. Note that the tree and soil observations may be directly or indirectly connected through context (i.e., tree observations may be directly connected to an observation that is connected to the soil observation), since context relationships are transitive. Using ObsQL, context constraints can also be chained together. For instance, the query

```
Tree → Plot → Site
```

returns tree, plot, and site observations in which tree observations have a corresponding plot observation as context, and plot observations have a corresponding site observation as context (again, where these observations may be directly or indirectly connected via context relationships). It is also possible to combine both types of queries, e.g., the query

```
(Tree, Soil) → Plot → Site
```

[3]http://clarkparsia.com/pellet/
[4]http://owlapi.sourceforge.net/

returns tree, soil, plot, and site observations such that the tree and soil observations are related through context to the same plot observation, and each plot observation is related through context to a corresponding site observation. Similarly, the following query

```
(Tree, Soil) → (Plot, Zone)
```

returns tree, soil, plot, and (ecological) zone observations such that pairs of tree and soil observations have the same plot and zone observation context.

In the above examples, each query returns a "flat" set of observations. In Sec. IV we show how results can additionally be formatted based on the structure of the query to enable further computation (i.e., for exploring query results through simple aggregation or more complex R scripts). ObsQL also allows "placeholder" variables in query expressions to access observations and measurements within a query result. Placeholder variables do not influence query results. As an example, the following query uses the placeholder variables `$t` and `$h` (where `t`, `h` can be arbitrary labels and '`$`' denotes a variable).

```
Tree $t[Height $h > 20 Meter]
```

Here, the variables can be used in additional expressions within ObsDB to compute aggregates (or in general, run R scripts) over query results. For example, from this query we can issue the ObsDB command "`count $t`" which gives the number of observations returned by the query. In this example, the variable `$t` denotes a set of observations, whereas the variable `$h` denotes a set of measurements.

Syntax and semantics. More formally, the abstract syntax of ObsQL is defined by the following grammar rules where e denotes an entity type, c a characteristic type, and s a measurement-standard type; where types denote sets of conforming instances such that if r is of type t, then $r \in t$.

$$\begin{aligned} \mathtt{q} &::= \mathtt{t} \mid \mathtt{t} \rightarrow \mathtt{q} \\ \mathtt{t} &::= e \mid e\mathtt{[m]} \mid \mathtt{t,\ t} \\ \mathtt{m} &::= c \mid c\ s \mid c\ op_v \mid c\ op_v\ s \mid \mathtt{m,\ m} \end{aligned}$$

As shown above, we often use parenthesis to group expressions. A query $\mathtt{q} : R \rightarrow R$ over a set R of observation resource identifiers returns the subset of identifiers $[\![q(R)]\!] \subseteq R$ that satisfy q. For convenience, in the following we omit R. We first define the semantics of simple entity and characteristic expressions:

$$\begin{aligned} [\![e]\!] &\equiv \{r_o \mid \exists r_e : \mathtt{ofEntity}(r_o, r_e), r_e \in e\} \\ [\![c]\!] &\equiv \{r_o \mid \exists r_m, r_c : \mathtt{hasMeasurement}(r_o, r_m), \\ &\qquad \mathtt{ofCharacteristic}(r_m, r_c), r_c \in c\} \\ [\![c\ s]\!] &\equiv \{r_o \mid \exists r_m, r_c, r_s : \mathtt{hasMeasurement}(r_o, r_m), \\ &\qquad \mathtt{ofCharacteristic}(r_m, r_c), r_c \in c, \\ &\qquad \mathtt{usesStandard}(r_m, r_s), r_s \in s\} \\ [\![c\ op_v]\!] &\equiv \{r_o \mid \exists r_m, r_c, v' : \mathtt{hasMeasurement}(r_o, r_m), \\ &\qquad \mathtt{ofCharacteristic}(r_m, r_c), r_c \in c, \\ &\qquad \mathtt{hasValue}(r_o, v'), (v'\ op_v) \equiv \mathtt{true}\} \\ [\![c\ op_v\ s]\!] &\equiv \{r_o \mid \exists r_m, r_c, v', r_s : \mathtt{hasMeasurement}(r_o, r_m), \\ &\qquad \mathtt{ofCharacteristic}(r_m, r_c), r_c \in c, \\ &\qquad \mathtt{hasValue}(r_o, v'), (v'\ op_v) \equiv \mathtt{true}, \\ &\qquad \mathtt{usesStandard}(r_o, r_s), r_s \in s\} \end{aligned}$$

In particular, an entity class expression returns observation identifiers consisting of entities of the type e. We note that $r \in e'$ and $e' \sqsubseteq e$ implies $r \in e$ (according to the standard definition of "*is-a*"). Similarly, characteristic class expressions return observation identifiers with at least one measurement containing the given characteristic type c. A value expression op_v consists of a comparison operator (see above) and a value v (or a range of values in the case of `BETWEEN`). A measurement standard further filters a measurement requiring that the given standard be used.

For an expression $\mathtt{m}_1, \mathtt{m}_2$ we define $[\![\mathtt{m}_1, \mathtt{m}_2]\!] \equiv [\![\mathtt{m}_1]\!] \cup [\![\mathtt{m}_2]\!]$, and similarly for $\mathtt{t}_1, \mathtt{t}_2$ we define $[\![\mathtt{t}_1, \mathtt{t}_2]\!] \equiv [\![\mathtt{t}_1]\!] \cup [\![\mathtt{t}_2]\!]$. Composite expressions are defined as follows.

$$\begin{aligned} [\![e\mathtt{[m]}]\!] &\equiv \{r_o \mid r_o \in [\![e]\!], r_o [\![m]\!]\} \\ [\![\mathtt{t} \rightarrow \mathtt{q}]\!] &\equiv \{r_{o1}, r_{o2} \mid r_{o1} \in [\![\mathtt{t}]\!], r_{o2} \in [\![\mathtt{q}]\!], \\ &\qquad \mathtt{hasContext}(r_{o1}, r_{o2})\}. \end{aligned}$$

Thus, an entity class expression combined with one or more measurement type expressions selects observations that match both the entity class and the measurement types (which is equivalent to the intersection of $[\![e]\!]$ and $[\![\mathtt{m}]\!]$). Similarly, two observations matching a type $\mathtt{t}_1$ and a type $\mathtt{t}_2$ satisfy a context query $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ if they participate in a context relationship. In this case, we return the set containing both the observations of $\mathtt{t}_1$ and $\mathtt{t}_2$. This can easily be extended to queries of the form $\mathtt{t} \rightarrow \mathtt{q}$ as shown above.

IV. System Implementation

The current implementation of the ObsDB system represents a functional prototype of the architecture and ideas presented in the previous section. The current system is written as an open-source Java application that is built over the Apache Derby embedded relational database system. The ObsDB framework is primarily designed to be used within an external application, e.g., where ObsDB is used as an internal library or wrapped as an external (e.g., web) service. In addition, we have developed a command-line interpreter that can be used to directly run the ObsDB system.

Here we describe the details of the implementation, focusing on the physical schema of the underlying database, the steps involved in answering ObsQL queries, the integration of the R system for providing exploratory analysis support, and a preliminary evaluation of the system (with respect to query and analysis support).

A. Physical Schemas

We first describe the physical schemas used to store ontologies and observational data.

Ontologies. Registered ontologies are stored using the relations:

- `ontology`$(\underline{u}, p_s, p_l)$ where u is the ontology URI, p_s is the original location of the ontology (given as a file path or URL), and p_l is the local (internal) path to the ontology.
- `prefix`$(\underline{n}, u)$ denotes a mapping between an a prefix name n and a namespace URI u.

Note that when an ontology is loaded, a local copy is created and the system verifies that all imports (dependencies) of the ontology are registered within ObsDB.

Observations. When data is loaded into ObsDB the corresponding OWL/RDF file is parsed and shredded into the following relations.

- `observation`$(\underline{u, o_{id}}, e_{id})$ denotes the observation instances where (u, o_{id}) is the full resource identifier for the observation such that u is the namespace URI and o_{id} is the fragment id. The entity instance is given by the fragment id e_{id}, which we assume has the same namespace URI as the observation.
- `entity`$(\underline{u, e_{id}}, et_u, et_{id})$ denotes entity instances with resource id (u, e_{id}), and (et_u, et_{id}) denote the resource id of the named OWL class declared as the entity type.
- `context`$(\underline{u, o_{id1}, o_{id2}})$ denotes context relationships between observations, where both are assumed to be declared within the same namespace URI u.
- `measurement`$(\underline{u, m_{id}}, o_{id}, ct_u, ct_{id}, \ldots, s_v, n_v)$ denotes measurement instances where (u, m_{id}) is the resource id of the measurement, o_{id} is the id (with respect to u) of the observation the measurement is for, $(ct_u, ct_{id}$ is the resource id of the characteristic type, and s_v and n_v represent non-numeric and numeric measured values, respectively (not shown are standard types, protocol types, method strings, and precision values). For a measurement, the measured value is stored in either of s_v or n_v depending on the value type.

In addition to the above tables, we also store the full (transitive) classification hierarchy in a separate table for each type (e.g., characteristic, entity, standard, etc.). These tables are updated as new ontologies are loaded into the system, and are used in query evaluation.

B. Query Evaluation

Each query q is rewritten into a corresponding SQL expression over the physical schema. Given a query, the ObsDB query processor performs the following steps.

1. Check validity: This step performs query parsing and ensures that the query is well-formed. During this step, it is also possible to perform simple type checking, e.g., to ensure that the query uses appropriate types for characteristics, entities, and so on.

2. Remove variables: When running queries, users can optionally store the query result within the database. In this case, a separate table is used to also store the query expression corresponding to the result table. The query expression (containing variables) is used later for additional operators supported by ObsQL and when performing external R functions. Thus, once the query is parsed, we remove the variables, which are not needed for query evaluation.

3. Rewriting: Once the query is parsed and the variables are removed, we rewrite the query expression into an equivalent SQL expression over the physical schema. The rewriting step largely follows the semantics of ObsQL given in the previous section with two exceptions. First, we incorporate as part of rewriting the classification tables described above to match observations that use subclasses of those used in the query. Second, instead of returning a single set of observations, we return a table that contains two columns per observation type given in the original query (where the first column gives the namespace URI and the second the fragment id of the observation instance). For example, the query

$$\texttt{(Tree, Soil)} \rightarrow \texttt{Plot}$$

is rewritten into an SQL query that returns three sets of columns corresponding to tree observations, soil observations, and plot observations, respectively. Note that, similar to the semantics given in the previous section, only observations are returned by a rewritten query.

4. Execute SQL query: The generated SQL query is executed over the database and the result of the query is stored as a separate, temporary table. We then run an additional SQL query over this table to remove potential duplicate matches, i.e., for cases in which the same set of observations match multiple observation types within a query (the original query eliminates cases where the same observation matches multiple types). As a simple example, the query

$$\texttt{(Tree, Plant)} \rightarrow \texttt{Soil}$$

could result in duplicate matches since trees are considered types of plants. If o_1 and o_2 are both tree observations contextualized by the soil observation o_3, the result of the initial SQL query would contain (omitting namespace URIs):

$$\{(o_1, o_2, o_3), (o_2, o_1, o_3)\}$$

assuming these were the only observations in the database. The result of removing duplicate matches is stored in another temporary table.

5. Convert query result to R data frame: Using the result of the previous step, we then create an R data frame. The data frame is constructed by rejoining the query result table with the measurements specified in the original query to obtain the measured values for the observations. (Alternatively, we could also select the measurements as part of the query result and use this to directly generate the corresponding data frame, which may provide a slight speed-up in overall query response time.) The resulting data frame is stored in a local file and indexed

within the database (for the case when the user wishes to store the query result). Finally, the data frame is displayed to the user within the ObsDB interpreter.

C. R System Integration

The ObsDB system provides "built-in" commands for summarizing query results. The system also supports analysis through the use of ad-hoc R scripts, which can be dynamically loaded into ObsDB. We support the standard aggregate operators of SQL (i.e., count, sum, avg, min, max) as well as additional operators for computing value ranges, standard deviations, medians and modes, and for basic graphing (plots, barcharts, plot matrices, etc.). An aggregation expression takes the form

$$op\ \$m\ [\ by\ \$o\ ]\ in\ q$$

where *op* is an aggregate operator (excluding count), $\$m$ is a measurement variable, $\$o$ is an observation variable, and q is the name of the stored query result. The count operation differs in that $\$m$ is required to be an observation variable, i.e., using count we total the number of observations as opposed to the number of measured values. The *by* clause is optional and restricts the scope of the aggregate operator to observations that have a context relationship with $\$o$. Similar to the `GROUP BY` clause in SQL, the result is the aggregate value for each corresponding context observation. Aggregate operations are implemented in ObsDB using SQL (for the standard SQL operators) and through R (for the remaining operators; via the data frame support described above). For example, consider the following ObsDB query expression.

```
Tree $t[Height $h] → Plot $p → Site $s as q1
```

Here, the query result is stored within a table named `q1`. We can now perform, e.g., the following summarizations over `q1`

```
count $t by $s in q1
avg $h by $p in q1
stddev $h by $s in q1
```

which return the number of tree observations in each site, the average height of trees within each plot, and the standard deviation of tree heights by site, respectively, for observations returned by the query `q1`.

ObsDB also allows ad hoc R scripts to be registered with the system and then run over query results. To register an R script, special comments must be added to the header of the script to denote variable names (and optional descriptions), which are replaced by appropriate values when the script is run within ObsDB. After a script is registered with ObsDB, the header of the script is parsed and indexed. Users can then list registered scripts (which displays the name of the script and the associated variables and comments). The script can then be called from within ObsDB by issuing the command

$$\texttt{run}\ s\ a_1\ a_2\ \ldots\ \texttt{in}\ q$$

where s is the name of the script, each a_i is an argument, and q is the name of the query the script is run over. Arguments are passed via placeholder variables from the query. ObsDB executes the script by replacing script variables with the corresponding columns of the data frame associated to the query variables.

D. System Evaluation

To test the feasibility of our query rewriting implementation of ObsDB, we performed an initial experimental evaluation over a corpus of test datasets (see Fig. 3). The datasets ranged from approximately 40 to 1100 observations with twice the number of context relationships for each dataset (thus, we considered a relatively large number of context relations). Each test dataset had context relationship "chains" of length 3. In addition, we considered three types of queries: (1) simple queries of the form "`Tree[Height > 20]`" (SQ); (2) simple context queries of the form "`Tree[Height > 20] → Plot[Name LIKE 'Plot%']`" (SCQ); and (3) context queries similar to SCQ but with two context relations (MCQ). Each query had approximately the same selectivity ranging from 13% to 20%. All tests were performed on a 2.13 GHz Linux machine with 2 GB of memory.

The results of executing these queries over the test data are shown in Fig. 4, which shows total execution time including the initial rewritten query, temporary table creation, and R data frame creation. As shown, the overall query execution times increase significantly as the number of context relationships within the query increase. In addition, the simple queries (SQ and SCQ) take considerably less time than the queries with multiple contexts (MCQ) for larger numbers of observations. For each query, generating the temporary tables and the R data frame contributed only a small (and largely linear) portion of the overall time. The primary bottleneck in answering SCQ and MCQ queries for larger datasets is due to the number of joins introduced in the initial SQL rewriting step (where MCQ queries involving two context relations contain 17-way joins). While promising, the results for multiple context queries highlight the need for additional optimizations; without optimization, these queries are only feasible for smaller sized datasets. An obvious approach is through denormalization (e.g., combining the measurement and observation tables of the physical schema). As ongoing work, we have developed optimization techniques with demonstrated scalability and query responce time improvements for MCQ queries.[5] We also intend to explore approaches based on techniques used in column and RDF triple stores [17].

V. Related Work

The need for more uniform mechanisms to describe observational data has led to a number of proposals for observational data models (e.g., [2], [6], [7], [9]), and corresponding ontologies (e.g., [4], [8], [10], [11]). The work presented here is complementary to these efforts. Specifically, ObsDB provides a framework for managing observational data according to a

[5]Our preliminary results (not shown here due to lack of space) reduce MCQ query time to less than 1 second for databases of 500,000 observations (i.e., 100 datasets each with 5,000 rows containing 10 observations per row). This work uses PostgreSQL instead of Derby).

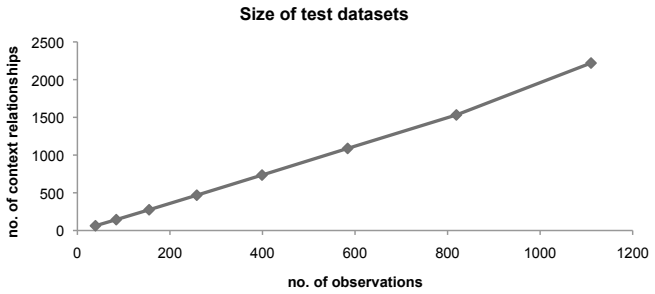

Fig. 3. Number of observations and context relations for each test dataset.

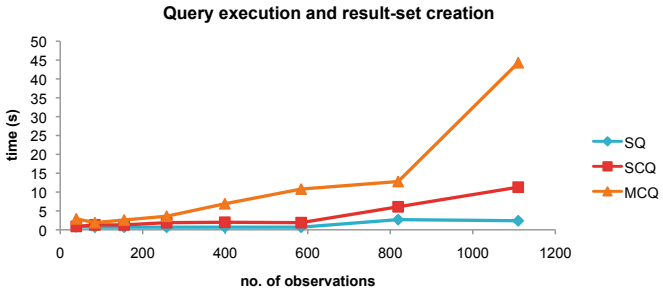

Fig. 4. Query execution times (including result-set creation and formatting) for simple observation (SQ), single context (SCQ), and multiple context (MCQ) queries over the test datasets of Fig. 3

generic observational data model, supports the use of domain-specific ontologies for describing this data, and supports a novel high-level query language for discovering and accessing observations (within and across datasets). In addition, ObsDB supports exploratory data discovery tasks (through its R support) that are crucial for determining the relevance of data within broad-scale scientific studies.

The ObsQL query language employed by ObsDB is similar to a number of emerging graph-based query languages (e.g., [18]), and with its support for summarization, ObsDB is similar in spirit to OLAP approaches and approaches for graph summarization (e.g., [19]). An important difference is ObsDB's focus on observational data (e.g., where context has a specific semantics compared to arbitrary graph arcs) and its use of ontologies. Further, unlike in traditional OLAP systems, ObsDB contributes a novel approach for combining querying and aggregation over graph-based data, which is a relatively unexplored area in database systems research (e.g., see [19]). Finally, ObsDB shares similar goals to scientific workflow systems (e.g., [20]–[22]), which support the definition and execution of complex data analyses. Many of these systems provide R support as well as the ability to use ontologies to help discover and chain together analytical components. These systems, however, do not provide explicit support for modeling and querying observational data (as presented here). Additionally, the goal of ObsDB differs from workflow systems in that instead of focusing on generic data-analysis support, ObsDB aims to provide data discovery and integration capabilities in which researchers use analytical functions to explore whether a given dataset is relevant for their work. After this step, a researcher may then proceed by executing a larger analysis over the data using a workflow system. ObsDB and workflow systems are thus complementary, and one could easily imagine embedding ObsDB within existing workflow systems.

VI. Summary

We extended our prior work [3] by presenting an architecture and implementation for managing observational data (ObsDB). The system provides novel approaches for uniformly storing and querying heterogeneous observational data. We also introduced a new query language (ObsQL) that provides users with a formal, high-level, declarative approach for discovering and accessing observational data. In addition, ObsDB seamlessly integrates with the R system for enabling exploratory data analysis tasks over query results. ObsDB provides an important step towards leveraging the emerging development of ontologies in the earth and environmental sciences for data discovery and integration. As future work, we are exploring optimization approaches as well as adding support in ObsDB for importing data in different underlying formats (e.g., via semantic annotation).

References

[1] "The Knowledge Network for Biocomplexity (KNB)," http://knb.ecoinformatics.org/index.jsp.
[2] OGC, "The OpenGIS Observations and Measurements Encoding Standard (O&M)," http://www.opengeospatial.org/standards/om.
[3] S. Bowers, J. Madin, and M. Schildhauer, "A conceptual modeling framework for expressing observational data semantics," in *ER*, 2008.
[4] P. Fox, *et al.*, "Ontology-supported scientific data frameworks: The virtual solar-terrestrial observatory experience," *Computers & Geosciences*, vol. 35, no. 4, pp. 724–738, 2009.
[5] J. Balhoff, *et al.*, "Phenex: Ontological annotation of phenotypic diversity," *PLoS ONE*, vol. 5, 2010.
[6] J. Cushing, *et al.*, "Component-based end-user database design for ecologists," *J. Intell. Inf. Syst.*, vol. 29, no. 1, pp. 7–24, 2007.
[7] D. Tarboton, J. Horsburgh, and D. Maidment, "CUAHSI community observations data model (ODM), version 1.0 design specifications," 2007, http://water.usu.edu/cuahsi/odm/.
[8] "Semantic Web for Earth and Environmental Terminology (SWEET)," http://sweet.jpl.nasa.gov/sweet/.
[9] Unidata, "network Common Data Form (netCDF)," http://www.unidata.ucar.edu/software/netcdf/.
[10] C. Mungall, *et al.*, "Integrating phenotype ontologies across multiple species," *Genome Biology*, vol. 11, no. R2, 2010.
[11] A. P. Sheth, C. A. Henson, and S. S. Sahoo, "Semantic sensor web," *IEEE Internet Computing*, vol. 12, no. 4, pp. 78–83, 2008.
[12] A. Rector, *et al.*, "OWL Pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns," in *EKAW*, 2004, pp. 63–81.
[13] S. Stevens, "On the theory of scales of measurement," *Science*, vol. 103, pp. 677–680, 1946.
[14] F. Baader, *et al.*, Ed., *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
[15] "The obo foundry," http://www.obofoundry.org/.
[16] C. Berkley, S. Bowers, M. B. Jones, J. S. Madin, and M. Schildhauer, "Improving data discovery for metadata repositories through semantic search," in *CISIS*, 2009, pp. 1152–1159.
[17] L. Sidirourgos, *et al.*, "Column-store support for RDF data management: not all swans are white," *PVLDB*, vol. 1, no. 2, pp. 1553–1563, 2008.
[18] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD*, 2008, pp. 405–418.
[19] C. Chen, *et al.*, "Graph OLAP: a multi-dimensional framework for graph data analysis," *Knowl. Inf. Syst.*, vol. 21, no. 1, 2009.
[20] B. Ludäscher, *et al.*, "Scientific workflow management and the kepler system," *Conc. and Comp.: Pract. & Exp.*, vol. 18, no. 10, 2006.
[21] T. Oinn, *et al.*, "Taverna: A tool for the composition and enactment of bioinformatics workflows." *Bioinformatics*, vol. 20, no. 17, 2004.
[22] B. Bavoil, *et al.*, "VisTrails:Enabling interactive multiple-view visualizations," in *IEEE Visualization*, 2005.