# Database Support for Enabling Data-Discovery Queries over Semantically-Annotated Observational Data*

Huiping Cao[1], Shawn Bowers[2], and Mark P. Schildhauer[3]

[1] Dept. of Computer Science, New Mexico State University
hcao@cs.nmsu.edu
[2] Dept. of Computer Science, Gonzaga University
bowers@gonzaga.edu
[3] NCEAS, University of California Santa Barbara
schild@nceas.ucsb.edu

**Abstract.** Observational data plays a critical role in many scientific disciplines, and scientists are increasingly interested in performing broad-scale analyses by using observational data collected as part of many smaller scientific studies. However, while these data sets often contain similar types of information, they are typically represented using very different structures and with little semantic information about the data itself, which creates significant challenges for researchers who wish to discover existing data sets based on data semantics (observation and measurement types) and data content (the values of measurements within a data set). We present a formal framework to address these challenges that consists of a semantic observational model (to uniformly represent observation and measurement types), a high-level semantic annotation language (to map tabular resources into the model), and a declarative query language that allows researchers to express data-discovery queries over heterogeneous (annotated) data sets. To demonstrate the feasibility of our framework, we also present implementation approaches for efficiently answering discovery queries over semantically annotated data sets. In particular, we propose two storage schemes (in-place databases *rdb* and materialized databases *mdb*) to store the source data sets and their annotations. We also present two query schemes (*ExeD* and *ExeH*) to evaluate discovery queries and the results of extensive experiments comparing their effectiveness.

## 1 Introduction

Accessing and reusing observational data is essential for performing scientific analyses at broad geographic, temporal, and biological scales. Classic examples in earth and environmental science include examining the effects of nitrogen treatments across North American grasslands [20], and studying how changing environmental conditions affect bird migratory patterns [22]. These types of studies often require access to hundreds of data sets collected by independent research groups over many years. Tools that aim to help researchers discover and reuse these data sets must overcome a number of

* This work was supported in part through NSF grants DBI-0743429 and DBI-0753144, and NMSU Interdisciplinary Research Grant #111721.

A. Hameurlain et al. (Eds.): TLDKS VI , LNCS 7600, pp. 198–228, 2012.

| *site* | *plt* | *size* | *ph* | *spp* | *len* | *dbh* |
|---|---|---|---|---|---|---|
| GCE6 | A | 7 | 4.5 | piru | 21.6 | 36.0 |
| GCE6 | B | 8 | 4.8 | piru | 27.0 | 45 |
| ... | ... | ... | ... | ... | ... | ... |
| GCE7 | A | 7 | 3.7 | piru | 23.4 | 39.1 |
| GCE7 | B | 8 | 3.9 | piru | 25.2 | 42.7 |
| ... | ... | ... | ... | ... | ... | ... |

| *yr* | *field* | *area* | *acidity* | *piru* | *abba* | ... |
|---|---|---|---|---|---|---|
| 2005 | f1 | 5 | 5.1 | 20.8 | 14.1 | ... |
| 2006 | f1 | 5 | 5.2 | 21.1 | 15.2 | ... |
| ... | ... | ... | ... | ... | ... | ... |
| 2010 | f1 | 5 | 5.8 | 22.0 | 18.9 | ... |
| 2005 | f2 | 7 | 4.9 | 18.9 | 15.3 | ... |
| ... | ... | ... | ... | ... | ... | ... |

**Fig. 1.** Typical examples of similar (but not identical) observational data sets consisting of study locations (plot, field), soil acidity measurements, and height and diameter measurements of trees

significant challenges: (1) observational data sets exhibit a high level of structural heterogeneity (e.g., see Fig. 1), which includes the use of various terms and conventions for naming columns containing similar or compatible information (e.g., "dw", "wt", "m", "biomass" may each be used to denote a "mass" measurement); and (2) semantic information about data sets, which is crucial for properly interpreting data, is typically either missing or only provided through natural-language descriptions.

Despite these challenges, a number of efforts are being developed with the goal of creating and deploying specialized software infrastructures (e.g., [5,1]) to allow researchers to store and access observational data contributed from various disciplines. While a large number of data sets are stored using these repositories, these sites provide primarily simple keyword-based search interfaces, which for many queries are largely ineffective for discovering relevant data sets (in terms of precision and recall) [8].

In this paper, we present a formal semantic annotation and data discovery framework that can be used to uniformly represent and query heterogeneous observational data. We adopt an approach that is based on a number of emerging observation models (e.g., [3,14,10,18]), which provides canonical representations of observation and measurement structures that researchers can use to help describe, query, and access otherwise heterogeneous data sets. Here we consider the use of description-logic (i.e., OWL-DL) based ontologies for domain-specific terms to specify observation and measurement types. These types can be used to both annotate data sets and to specify data-discovery queries.

Semantic annotations in our framework define concrete mappings from relational data sets to a uniform observational model specialized by domain-specific terms. The annotation language was designed to support annotations created either manually or automatically (e.g., by employing attribute similarity measures or data-mining techniques). The annotation language is currently being used to store annotations created (via a graphical user interface) within a widely used metadata editing tool [2] for earth and environmental science data sets. A key contribution of our annotation approach is that it provides a declarative, high-level language that follows the "natural" way in which users describe their observational data sets semantically, i.e., by focusing on attribute-level metadata, and then by automatically inferring remaining structural relationships. We also support data-discovery queries posed over both the types of observations and measurements used to annotate data sets as well as over (possibly summarized) data-set values. For instance, using our framework, it is possible to express

queries that range from simple "schema-level" filters such as "*Find all data sets that contain height measurements of trees within experimental locations*" to queries that access, summarize, and select results based on the values within data sets such as "*Find all data sets that have trees with a maximum height measurement larger than 20 $m$ within experimental locations having an area smaller than 10 $m^2$*".

Finally, we describe different storage and query evaluation approaches that have been implemented to support the framework. We consider two main approaches. The first is a "data warehouse" approach that uses a single "materialized" database to store underlying observational data sets, where query evaluation involves rewriting a discovery query into a query over the warehouse. In the second approach, semantic annotations are treated as logical views over the underlying data set schemas, where query evaluation involves rewriting the original query using the annotation into corresponding queries over the underlying data sets. Based on our initial experimental results, we demonstrate the feasibility of querying a large corpus using these approaches, and that querying data in place can lead to better performance compared with more traditional warehousing approaches. This paper is an extended version of [11] that describes in detail query evaluation strategies for complex data-discovery queries and additional corresponding experimental results.

The rest of this paper is organized as follows. In Sect. 2 we present the observational model, semantic annotation language, and data-discovery language used within our framework. In Sect. 3 we describe implementation approaches. In Sect. 4 we present our experimental results. In Sect. 5 we discuss related work, and in Sect. 6 we summarize our contributions.

## 2 Semantic Annotation and Discovery Framework

Fig. 2 shows the modeling constructs we use to describe and (depending on the implementation) store observational data. An *observation* is made of an *entity* (e.g., biological organisms, geographic locations, or environmental features, among others) and primarily serves to group a set of measurements together to form a single "*observation event*". A *measurement* assigns a value to a *characteristic* of the observed entity (e.g., the height of a tree), where a value is denoted through another entity (which includes primitive values such as integers and strings, similar to pure object-oriented models). Measurements also include *standards* (e.g., units) for relating values across measurements, and can also specify additional information including collection protocols, methods, precision, and accuracy (not all of which are shown in Fig. 2). An observation (event) can occur within the *context* of zero or more other observations. Context can be viewed as a form of dependency, e.g., an observation of a tree specimen may have been made within a specific geographic location, and the geographic location provides important information for interpreting and comparing tree measurements. In this case, by establishing a context relationship between the tree and location observations, the measured values of the location are assumed to be constant with respect to the measurements of the tree (i.e., the tree measurements are dependent on the location measurements). Context forms a *transitive* relationship among observations. Although not considered here, we also employ a number of additional structures in the model for representing complex

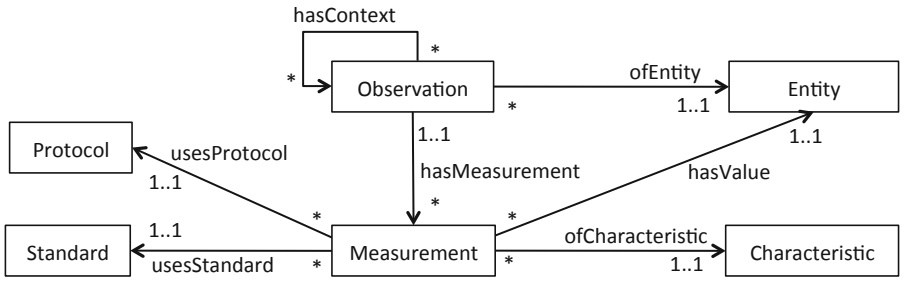



**Fig. 2.** Main observational modeling constructs used in semantic annotation and data discovery

units, characteristics, and named relationships between observations and entities [10]. When describing data sets using the model of Fig. 2, domain-specific entity, characteristic, and standard classes are typically used. That is, our framework allows subclasses of the classes in Fig. 2 to be defined and related, and these terms can then be used when defining semantic annotations.

A key feature of the model is its ability for users to assert properties of entities (as measurement characteristics or contextual relationships) without requiring these properties to be interpreted as *inherently* (i.e., *always*) true of the entity. Depending on the context an entity was observed (or how measurements were performed), its properties may take on different values. For instance, the diameter of a tree changes over time, and the diameter value often depends on the protocol used to obtain the measurement. The observation and measurement structure of Fig. 2 allows RDF-style assertions about entities while allowing for properties to be contextualized (i.e., the same entity can have different values for a characteristic under different contexts), which is a crucial feature for modeling scientific data [10]. Although shown using UML in Fig. 2, the model has been implemented (together with a number of domain extensions) using OWL-DL.[1]

### 2.1 Semantic Annotation

Semantic annotations are represented using a high-level annotation language in which each annotation consists of two separate parts: (1) a *semantic template* that defines specific observation and measurement types (and their various relationships) for the data set; and (2) a *mapping* from individual attributes of a data set to measurement types defined within the semantic template. A semantic template consists of one or more observation types $O$ specified using statements of the form

$$O ::= \texttt{Observation}\ [\ \{\texttt{distinct}\}\ ]\ \texttt{id}_o : \texttt{EntType}\ [\ , \texttt{ContextType}\ ]^*\ [\ \texttt{MeasType}\ ]^*$$

where square brackets denote optional elements and $*$ denotes repetition. In particular, an observation type consists of an optional *distinct* constraint, a name (denoted $\texttt{id}_0$), an entity type, zero or more context types, and zero or more measurement types. Entity, context, and measurement types are specified using the following syntax

[1] e.g., see `http://ecoinformatics.org/oboe/oboe.1.0/oboe-core.owl`

$$\texttt{EntType} ::= \texttt{Entity} = e$$
$$\texttt{ContextType} ::= \texttt{Context}\ [\ \{\texttt{identifying}\}\ ] = \texttt{id}_\texttt{o}$$
$$\texttt{MeasType} ::= \texttt{Measurement}\ [\ \{\texttt{key}\}\ ]\ \texttt{id}_\texttt{m} : \texttt{Characteristic} = c\ [\ , \texttt{Standard} = s\ ]$$

where $e$, $c$, and $s$ are respectively entity, characteristic, and standard types (e.g., drawn from an OWL ontology), $\texttt{id}_\texttt{o}$ is an observation type name (defined within the same annotation), and $\texttt{id}_\texttt{m}$ is a measurement type name. Both the optional *identifying* and *key* keywords represent constraints, which together with the *distinct* constraint, are defined further below. A mapping $M$, which links data-set attributes to measurement types, takes the form

$$M ::= \texttt{Map}\ a\ \texttt{to}\ \texttt{id}_\texttt{m}\ [\ , a\ \texttt{to}\ \texttt{id}_\texttt{m}\ ]^*$$

where $a$ is an attribute name within the data set and $\texttt{id}_\texttt{m}$ is a measurement type name (defined in the template).

The left side of Fig. 3 gives an example annotation for the first table of Fig. 1. Here we define four observation types denoting measurements of sites, plots, soils, and trees, respectively. A site observation contains a simple (so-called "nominal") measurement that gives the name of the site. Similarly, a plot observation records the name of the plot (where a plot is used as an experimental replicate) as well as the plot area. Here, plots are observed within the context of a corresponding site. A soil observation consists of an acidity measurement and is made within the context of a plot observation (although not shown, we would typically label the context relation in this case to denote that the soil is part of the plot). A tree observation consists of the taxonomic name of the tree along with height and diameter measurements in meters and centimeters, respectively. Finally, each attribute in the data set of Fig. 1 is mapped (via the `Map` statement) to its corresponding measurement type.

The right side of Fig. 3 gives a visual representation of the relationship between (a portion of) the semantic template (top) and the attribute mapping (dashed-lines, middle)

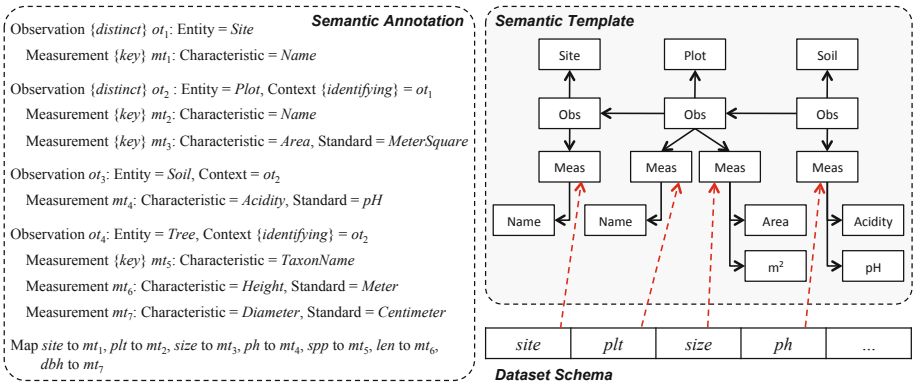



**Fig. 3.** Semantic annotation of the first data set of Fig. 1 showing the high-level annotation syntax (left) and a graphical representation of the corresponding "semantic template" and column-mapping (right)

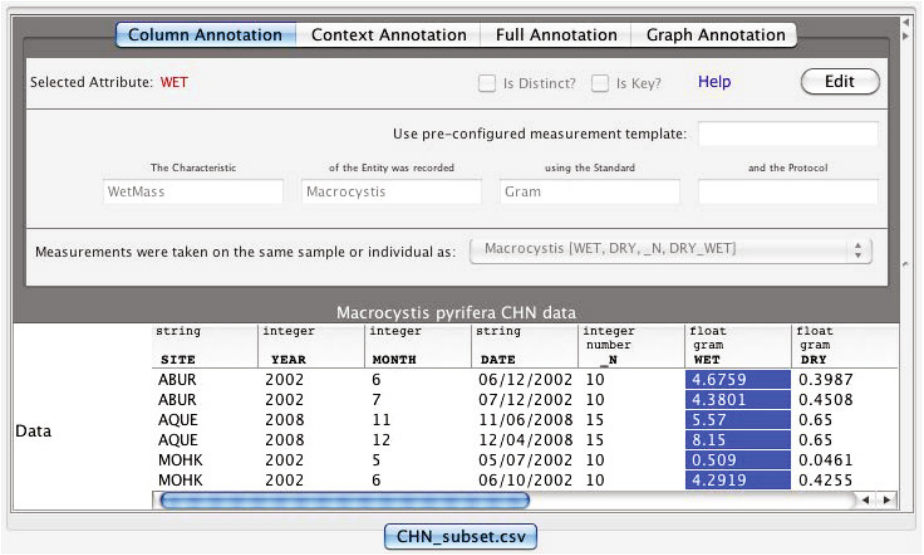



**Fig. 4.** The semantic annotation user interface developed within the Morpho metadata editor

from the underlying data set schema (bottom) to the template. The visual representation of the template informally matches the observational model shown in Fig. 2, where arrows denote relationships between classes. As shown, each attribute is assigned to a single measurement type in the template. This approach follows the typical view of attributes in data sets as specifying measurements, where the corresponding entities, observation events, and context relationships are implied by the template. To help users specify semantic annotations, we have also developed a graphical user-interface within [2] that allows users to specify attribute-level mappings to measurement types and the corresponding measurement and observation types of the data set. An example of the interface is shown in Fig. 4 in which the measurement associated with the column labeled "`WET`" is being specified. The annotation language is used to store (via an XML serialization) the mappings and semantic templates generated by the interface.

The meaning of a semantic annotation can be viewed as the result of processing a data set row-by-row such that each row creates a valid instance of the semantic template. We refer to such an instance as a *materialization* of the data set with respect to the annotation. For example, in the first row of the data set as in Fig. 1, the site value "GCE6" implies: (1) an instance $m_1$ of the measurement type $mt_1$ whose value is "GCE6"; (2) an instance $c_1$ of the Name characteristic for $m_1$; (3) an instance $o_1$ of an observation (corresponding to type $ot_1$) having measurement $m_1$; and (4) an instance $e_1$ of the Site entity such that $e_1$ is the entity of $o_1$. Similarly, assuming the plot attribute value "A" of the first row corresponds to an observation instance $o_2$ (of observation type $ot_2$), the context definition for $ot_2$ results in $o_2$ having $o_1$ as context.

The key, identifying, and distinct constraints are used to further specify the structure of semantic-template instances. These constraints are similar to key and weak-entity

constraints used within ER models. If a measurement type is defined as a *key* (e.g., $mt_1$ in Fig. 3), then the values of instances for these measurement types identify the corresponding observation entity (similar to key attributes in an ER model). For example, consider the first table in Fig. 1. Both the first and the second row have the same site value of "GCE6". Let $e_1$ be the entity instance of the first-row's corresponding observation. Let $o_1$ and $o_3$ be the observation instances for the site attribute in the first row and the second row respectively. The key constraint of $mt_1$ requires that $e_1$ be an entity of both $o_1$ and $o_3$. Similarly, an *identifying* constraint requires the identity of one observation's entity to depend (through context) on the identity of another observation's entity (similar to identifying relationships in an ER model). In our example, plot names are unique only within a corresponding site. Thus, a plot with name "A" in one site is not the same plot as a plot with name "A" in a different site. Identifying constraints define that the identity of an observation's entity is determined by both its own key measurements and its identifying observations' key measurements.

**Definition 1 (Key measurement types).** *The set of key measurement types* $\texttt{Keys}(O)$ *of an observation type* $O$ *are the measurement types whose values can distinguish one entity instance (of which the observation is made) from another. Given an observation type* $O$*, let* $\texttt{M}_{\texttt{keys}}(O)$ *be the set of measurement types of* $O$ *that are specified with a key constraint. Similarly, let* $\texttt{C}_{\texttt{id}}(O)$ *be the set of context observation types of* $O$ *that are specified with an identifying constraint. The set of key measurement types of* $O$ *are thus*

$$\texttt{Keys}(O) = \texttt{M}_{\texttt{keys}}(O) \cup \{M \mid O' \in \texttt{C}_{\texttt{id}}(O) \wedge M \in \texttt{Keys}(O')\}.$$

*Example 1 (Key measurement type example).* Given the semantic annotation in Fig. 3, we can derive the key measurement types for the observation types. First, $ot_1$'s key measurement type is $\{mt_1\}$ because (a) $ot_1$ does not have any identifying constraint, and (b) $ot_1$'s direct key measurement type is $mt_1$. Second, $ot_2$'s key measurement types are $\{mt_3, mt_2, mt_1\}$ because (a) $ot_1$ is the identifying constraint of $ot_2$, (b) $ot_1$'s key measurement type is $mt_1$, and (c) $ot_2$'s direct measurement key types are $mt_2$ and $mt_3$. Similarly, we can derive that $ot_4$'s key measurement types are $\{mt_5, mt_3, mt_2, mt_1\}$.

The *distinct* constraint on observations is similar to a key constraint on measurements, except that it is used to uniquely identify observations (as opposed to observation entities). Distinct constraints can only be used if each measurement of the observation is constrained to be a key. Specifically, for a given set of key measurement values of an observation type specified to be distinct, the set of measurement values uniquely identifies an observation instance. Thus, for any particular set of measurement values, there will be only one corresponding observation instance. In Fig. 3, each row with the same value for the site attribute maps not only to the same observed entity (via the key constraint) but also to the same observation instance (via the distinct constraint).

**Materialization Relations.** Given a data set and an annotation, we *materialize* the data set with respect to the annotation into a set of relation instances. We represent annotations using the following schema.

- *Annot*($a$, $d$) states that $a$ is an annotation of data set with id $d$.
- *ObsType*($a$, $ot$, $et$, *isDistinct*) states that $ot$ is an observation type in annotation $a$, has entity type $et$, and whether it is declared as having the distinct constraint.

– *MeasType*($a$, $mt$, $ot$, $ct$, $st$, ..., *isKey*) states that $mt$ is a measurement type in $a$, is for observation type $ot$, and has characteristic type $ct$, standard type $st$, etc., and whether $mt$ is defined as a key.
– *ContextType*($a$, $ot$, $ot'$, *isId*) states that observation type $ot'$ is a context type of observation type $ot$ in $a$, and whether the context relationship is identifying.
– *Map*($a$, $attr$, $mt$, $\varphi$, $v$) states that data set attribute $attr$ is mapped to measurement type $mt$ in $a$, where $\varphi$ is an optional condition specifying whether the mapping applies (based on the values of attributes within the data set) and $v$ is an optional value to use for the measurement (instead of the data set value).

*Example 2 (Example of materialized annotations).* The annotations in Fig. 3 can be captured in the following core relations.

ObsType

| $a$ | $ot$ | $et$ | *isDistinct* |
|---|---|---|---|
| $a_1$ | $ot_1$ | `Site` | `true` |
| $a_1$ | $ot_2$ | `Plot` | `true` |
| $a_1$ | $ot_3$ | `Soil` | `false` |
| $a_1$ | $ot_4$ | `Tree` | `false` |

MeasType

| $a$ | $mt$ | $ot$ | $ct$ | $st$ | $\cdots$ | *isKey* |
|---|---|---|---|---|---|---|
| $a_1$ | $mt_1$ | $ot_1$ | `Name` | *null* | $\cdots$ | `true` |
| $a_1$ | $mt_2$ | $ot_2$ | `Name` | *null* | $\cdots$ | `true` |
| $a_1$ | $mt_3$ | $ot_2$ | `Area` | `MeterSquare` | $\cdots$ | `true` |
| $a_1$ | $mt_4$ | $ot_3$ | `Acidity` | `pH` | $\cdots$ | `true` |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |

ContexType

| $a$ | $ot$ | $ot'$ | *isId* |
|---|---|---|---|
| $a_1$ | $ot_2$ | $ot_1$ | `true` |
| $a_1$ | $ot_3$ | $ot_2$ | `false` |
| $a_1$ | $ot_4$ | $ot_2$ | `true` |

Map

| $a$ | $attr$ | $mt$ | $\varphi$ | $v$ |
|---|---|---|---|---|
| $a_1$ | `site` | $mt_1$ | *null* | *null* |
| $a_1$ | `plt` | $mt_2$ | *null* | *null* |
| $a_1$ | `size` | $mt_3$ | *null* | *null* |
| $a_1$ | `ph` | $mt_4$ | *null* | *null* |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |

We use the following relations to represent instances of semantic templates.

– *Entity*($d$, $e$, $et$) states that entity $e$ in data set $d$ is an instance of entity type $et$.
– *Obs*($d$, $o$, $ot$, $e$) states that observation $o$ in data set $d$ is of type $ot$ and is an observation of entity $e$.
– *Meas*($d$, $m$, $mt$, $v$, $o$) states that measurement $m$ in $d$ is of measurement type $mt$, has the value $v$, and is a measurement for observation $o$.
– *Context*($d$, $o$, $o'$) states that observation $o$ is within the context of $o'$ in $d$.

We can then evaluate the mapping defined by a semantic annotation $a$ over a data set $d$ using the algorithm in Fig. 5, which results in populating the above relations for template instances.

**MateralizeDB Algorithm.** As shown in Fig. 5, while processing each row we create measurement instances for each mapped attribute (cell) in the row (Step 2a), link them to their related observation instances (Step 2b to Step 2(d)ii), and then create proper context links between observation instances (Step 2e). The *EntityIndex* is used to ensure only unique entities are created within the data set (based on the values for measurements having a key constraint). Thus, before an entity instance is created (Step 2(d)i),

**Algorithm MaterializeDB**($a$, $d$)

1. $EntityIndex = \emptyset$; /* an index of the form $\{\langle ot, keyvals\rangle \rightarrow e\}$ */
2. **for each** $row = \langle attr_1, attr_2, \cdots, attr_n\rangle \in d$
   (a) $MeasSet$ = CreateMeasurements($a$, $row$);
   (b) $MeasIndex$ = PartitionMeasurements($a$, $MeasSet$);
   /* partition measurements based on observation types,
   returns index $\{ot \rightarrow \{m\}\}$*/
   (c) $ObsIndex = \emptyset$; /* an index of the form $\{ot \rightarrow o\}$ */
   (d) **for each** $ot \rightarrow \{m\} \in MeasIndex$
      i. $e$ = CreateEntity($a$, $ot$, $\{m\}$, $EntityIndex$); /* updates $EntityIndex$ */
      ii. CreateObservation($a$, $ot$, $e$, $ObsIndex$); /* updates $ObsIndex$ */
   (e) ConnectContext($a$, $ObsIndex$);

**Fig. 5.** Materialize a data set $d$ given its semantic annotation $a$

the *CreateEntity* function first checks using the index if the entity has already been created from a previous row. The *CreateMeasurements*, *CreateObservation*, and *ConnectContext* functions are straightforward and each uses the annotation's semantic template to create and connect the corresponding instances.

*Example 3 (Example of MaterializeDB algorithm).* Using the first data set in Fig. 1 and the annotation in Fig. 3 (whose tabular representation is in Example 2) as input, the following illustrates the execution of the algorithm *MaterializeDB*.

1. Process the first row (GCE6, A, 7, 4.5, piru, 21.6, 36.0).
   (a) *CreateMeasurements* creates seven measurement instances $m_1, \cdots, m_7$ for the seven cells in this row respectively. The tabular representation of these measurement instances is shown as follows.

Meas

| $d$ | $m$ | $mt$ | $v$ | $o$ |
|---|---|---|---|---|
| $d_1$ | $m_1$ | $mt_1$ | GCE6 | *null* |
| $d_1$ | $m_2$ | $mt_2$ | A | *null* |
| $d_1$ | $m_3$ | $mt_3$ | 7 | *null* |
| $d_1$ | $m_4$ | $mt_4$ | 4.5 | *null* |
| $d_1$ | $m_5$ | $mt_5$ | piru | *null* |
| $d_1$ | $m_6$ | $mt_6$ | 21.6 | *null* |
| $d_1$ | $m_7$ | $mt_7$ | 36.0 | *null* |

   Note that the observation instances ($o$) for each measurement instance ($m$) is set to *null* because they are unknown at this step. The $o$ values are assigned after Step 2d.
   (b) *PartitionMeasurements* then partitions these seven instances according to their observation types by using the *MeasType* relation in Example 2. In the *MeasType* relation, $mt_1$ is the only measurement type for $ot_1$. So $ot_1$ only corresponds to the instance $m_1$ of type $mt_1$ (i.e., $ot_1 \rightarrow \{m_1\}$). Next, $ot_2$ contains two measurement types $mt_2$ and $mt_3$. Thus, $ot_2$ corresponds to the instances $m_2$ and $m_3$, which are of types $mt_2$ and $mt_3$ (i.e., $ot_2 \rightarrow \{m_2, m_3\}$). Similarly, we can obtain $ot_3 \rightarrow \{m_4\}$, $ot_4 \rightarrow \{m_5, m_6, m_7\}$.

Entity

| $d$ | $e$ | $et$ |
|---|---|---|
| $d_1$ | $e_1$ | Site |
| $d_1$ | $e_2$ | Plot |
| $d_1$ | $e_3$ | Soil |
| $d_1$ | $e_4$ | Tree |

Obs

| $d$ | $o$ | $ot$ | $e$ |
|---|---|---|---|
| $d_1$ | $o_1$ | $ot_1$ | $e_1$ |
| $d_1$ | $o_2$ | $ot_2$ | $e_2$ |
| $d_1$ | $o_3$ | $ot_3$ | $e_3$ |
| $d_1$ | $o_4$ | $ot_4$ | $e_4$ |

Meas

| $d$ | $m$ | $mt$ | $v$ | $o$ |
|---|---|---|---|---|
| $d_1$ | $m_1$ | $mt_1$ | GCE6 | $o_1$ |
| $d_1$ | $m_2$ | $mt_2$ | A | $o_2$ |
| $d_1$ | $m_3$ | $mt_3$ | 7 | $o_2$ |
| $d_1$ | $m_4$ | $mt_4$ | 4.5 | $o_3$ |
| $d_1$ | $m_5$ | $mt_5$ | piru | $o_4$ |
| $d_1$ | $m_6$ | $mt_6$ | 21.6 | $o_4$ |
| $d_1$ | $m_7$ | $mt_7$ | 36.0 | $o_4$ |

Context

| $d$ | $o$ | $o'$ |
|---|---|---|
| $d_1$ | $o_2$ | $o_1$ |
| $d_1$ | $o_3$ | $o_2$ |
| $d_1$ | $o_4$ | $o_2$ |

**Fig. 6.** Instance relations after processing the first row of the first data set in Fig. 1 using algorithm *MaterializeDB*

(c) The for loop in Step 2d generates entity and observation instances. First, for $ot_1 \rightarrow \{m_1\}$, it generates an entity instance $e_1$, whose corresponding tabular information in *Entity* is $(d_1, e_1, Site)$, and an observation instance $o_1$ (with tabular information $(d_1, o_1, ot_1, e_1)$ in *Obs*). In addition, when executing *CreateObservation*, the measurement instance $m_1$'s corresponding observation instance is changed to $o_1$ (originally it is *null*). After applying similar steps to the other $ot \rightarrow \{m\}$ mappings, we get the instances in *Entity*, *Obs*, and *Meas* relations in Fig. 6.

(d) The last step in processing this row is to link the context relationship. Since $ot_1$ is the context of $ot_2$ and $ot_2$ is the context of $ot_3$ and $ot_4$ (See the *ContextType* relation in Example 2). We get the context instances as shown in the *Context* relation in Fig. 6.

2. The process of the second row (GCE6, B, 8, 4.8, piru, 27.0, 36.0) is similar to that of the first row although they differ in dealing with the measurement instances for values "GCE6" and "piru". First, *CreateMeasurements* creates seven measurement instances $m_8, \cdots, m_{14}$ (see Fig. 7) and *PartitionMeasurements* generates $ot_1 \rightarrow \{m_8\}$, $ot_2 \rightarrow \{m_9, m_{10}\}$, $ot_3 \rightarrow \{m_{11}\}$, and $ot_4 \rightarrow \{m_{12}, m_{13}, m_{14}\}$. Then, the for loop creates entity instances and observation instances.

(a) For $ot_1 \rightarrow \{m_8\}$, we do not need to create a new entity instance since $m_8$ shares the same value ("GCE6") with $m_1$, and $\{mt_1\}$ is the key measurement type of $ot_1$. In addition, since $ot_1$ has a distinct constraint, which uniquely identifies observations, we do not need to create a new observation instance for it. Because of this, $m_8$ is not needed and can be discarded.

(b) For $ot_2 \rightarrow \{m_9, m_{10}\}$ and $ot_3 \rightarrow \{m_{11}\}$, we create entity instances ($e_5$ and $e_6$) and observation instances ($o_5$ and $o_6$).

(c) When coming to $ot_4 \rightarrow \{m_{12}, m_{13}, m_{14}\}$, we observe that $m_{12}$'s value ("piru") is the same to that of $m_5$ (for observation instance $o_4$). Nevertheless, $ot_4$'s key measurement types are $\{mt_5, mt_3, mt_2, mt_1\}$ (see Example 1). The key values for $ot_4$ are different for the two rows (the key values for the first row are "$piru, 7, A, GCE6$" and the key values for the second row are "$piru, 8, B, GCE6$". Thus, we need to create a new entity and observation instance for $ot_4$.

Entity

| $d$ | $e$ | $et$ |
|---|---|---|
| $d_1$ | $e_1$ | Site |
| $d_1$ | $e_2$ | Plot |
| $d_1$ | $e_3$ | Soil |
| $d_1$ | $e_4$ | Tree |
| $d_1$ | $e_5$ | Plot |
| $d_1$ | $e_6$ | Soil |
| $d_1$ | $e_7$ | Tree |

Obs

| $d$ | $o$ | $ot$ | $e$ |
|---|---|---|---|
| $d_1$ | $o_1$ | $ot_1$ | $e_1$ |
| $d_1$ | $o_2$ | $ot_2$ | $e_2$ |
| $d_1$ | $o_3$ | $ot_3$ | $e_3$ |
| $d_1$ | $o_4$ | $ot_4$ | $e_4$ |
| $d_1$ | $o_5$ | $ot_2$ | $e_5$ |
| $d_1$ | $o_6$ | $ot_3$ | $e_6$ |
| $d_1$ | $o_7$ | $ot_4$ | $e_7$ |

Meas

| $d$ | $m$ | $mt$ | $v$ | $o$ |
|---|---|---|---|---|
| $d_1$ | $m_1$ | $mt_1$ | GCE6 | $o_1$ |
| $d_1$ | $m_2$ | $mt_2$ | A | $o_2$ |
| $d_1$ | $m_3$ | $mt_3$ | 7 | $o_2$ |
| $d_1$ | $m_4$ | $mt_4$ | 4.5 | $o_3$ |
| $d_1$ | $m_5$ | $mt_5$ | piru | $o_4$ |
| $d_1$ | $m_6$ | $mt_6$ | 21.6 | $o_4$ |
| $d_1$ | $m_7$ | $mt_7$ | 36.0 | $o_4$ |
| ~~$d_1$~~ | ~~$m_8$~~ | ~~$mt_1$~~ | ~~GCE6~~ | ~~*null*~~ |
| $d_1$ | $m_9$ | $mt_2$ | B | $o_5$ |
| $d_1$ | $m_{10}$ | $mt_3$ | 8 | $o_5$ |
| $d_1$ | $m_{11}$ | $mt_4$ | 4.8 | $o_6$ |
| $d_1$ | $m_{12}$ | $mt_5$ | piru | $o_7$ |
| $d_1$ | $m_{13}$ | $mt_6$ | 27.0 | $o_7$ |
| $d_1$ | $m_{14}$ | $mt_7$ | 45 | $o_7$ |

Context

| $d$ | $o$ | $o'$ |
|---|---|---|
| $d_1$ | $o_2$ | $o_1$ |
| $d_1$ | $o_3$ | $o_2$ |
| $d_1$ | $o_4$ | $o_2$ |
| $d_1$ | $o_5$ | $o_1$ |
| $d_1$ | $o_6$ | $o_5$ |
| $d_1$ | $o_7$ | $o_5$ |

**Fig. 7.** Instance relations after processing the first two rows of the first data set in Fig. 1 using algorithm *MaterializeDB*

After processing the second row, the updated relations are shown in Fig. 7.

3. The algorithm continues until it finishes processing all the rows in the data set.

**Analysis of MaterializeDB.** The MaterializeDB algorithm of Fig. 5 runs in $O(n \log m)$ time where $n$ is the number of rows in a data set and $m$ ($\ll n$) is the number of distinct keys within the data set. The algorithm uses $O(nc)$ space where $c$ is the number of columns in the data set (thus, $nc$ is the total number of cells).

**Semantic Annotations as Schema Mappings.** The semantic annotation language can easily be expressed using standard schema mapping approaches [19], i.e., annotations have a straightforward reduction to source-to-target tuple-generating dependencies and target equality-generating dependencies. A source-to-target tuple-generating dependency (*st-tdg*) is a first-order formula of the form $\forall \bar{x}(\varphi(\bar{x}) \rightarrow \exists \bar{y}\, \psi(\bar{x}, \bar{y}))$ where $\varphi(\bar{x})$ and $\psi(\bar{x}, \bar{y})$ are conjunctions of relational atoms over source and target schemas, respectively, and $\bar{x}$ and $\bar{y}$ are tuples of variables. We can use st-tdgs to define instances of semantic templates, e.g., the following rule maps the first attribute in the data set of Fig. 3 to measurement type $mt_1$, where $R$ is used as the name of the data set relation:

$$\forall \bar{x}(R(\bar{x}) \rightarrow \exists \bar{y}\, \mathrm{Meas}(\mathtt{d}, y_1, \mathtt{mt}_1, x_1, y_2) \wedge \mathrm{Obs}(\mathtt{d}, y_2, \mathtt{ot}_1, y_3) \wedge \mathrm{Entity}(\mathtt{d}, y_3, \mathtt{Site}))$$

Here we assume $x_1$ is the first variable in $\bar{x}$, each $y_i$ in the rule is a variable of $\bar{y}$, and $\mathtt{d}$, $\mathtt{mt}_1$, $\mathtt{ot}_1$, and $\mathtt{Site}$ are constants. A target equality-generating dependency (*t-egd*) takes the form $\forall \bar{y}(\phi(\bar{y}) \rightarrow u = v)$ where $\phi(\bar{y})$ is a conjunction of relational atoms over the target schema and $u, v$ are variables in $\bar{y}$. We can use t-egds to represent key, identifying, and distinct constraints, e.g., the following rule can be used to express the key constraint on measurement type $mt_1$:

$$\forall \bar{y}(\mathrm{Meas}(\mathtt{d}, m_1, \mathtt{mt}_1, v, o_1) \wedge \mathrm{Obs}(\mathtt{d}, o_1, \mathtt{ot}_1, e_1) \wedge \mathrm{Meas}(\mathtt{d}, m_2, \mathtt{mt}_1, v, o_2) \\ \wedge \mathrm{Obs}(\mathtt{d}, o_2, \mathtt{ot}_1, e_2) \rightarrow e_1 = e_2)$$

The annotation language we employ is also similar to a number of other high-level mapping languages used for data exchange (e.g., [13,6]), but supports simple type associations to attributes (e.g., as shown by the red arrows on the right of Fig. 3) while providing well-defined and unambiguous mappings from data sets to the observation and measurement schema.

### 2.2 Data Discovery Queries

Data discovery queries can be used to select relevant data sets based on their observation and measurement types and values. A *basic discovery query* $Q$ takes the form

$$Q ::= \texttt{EntityType}(\texttt{Condition})$$

where `EntityType` is a specific entity OWL class and `Condition` is a conjunction or disjunction of zero or more conditions. A condition takes the form

$$\begin{array}{rcl} \texttt{Condition} & ::= & \texttt{CharType} \; [\; \texttt{op value} \; [\; \texttt{StandardType} \; ] \; ] \\ & | & f(\texttt{CharType}) \; [\; \texttt{op value} \; [\; \texttt{StandardType} \; ] \; ] \\ & | & \texttt{count}(\; [\; \texttt{distinct} \; ] \; *) \; \texttt{op value} \end{array}$$

where `CharType` and `StandardType` are specific characteristic and (measurement) standard OWL classes, respectively, and $f$ denotes an aggregation function (sum, avg, min, or max). A data set is returned by a basic discovery query if it contains observations of the given entity type that satisfy the corresponding conditions. We consider three basic types of conditions (for the three syntax rules above): (1) the observation must contain at least one measurement of a given characteristic type (`CharType`) with a measured value satisfying a relational (i.e., $=, \neq, >, <, \geq, \leq$) or string comparison (e.g., `contains`); (2) the aggregate function applied to measurements of the characteristic type (`CharType`) for all observations of the entity type must satisfy the relational comparison; and (3) the number (`count`) of all observations of the entity type must satisfy the relational comparison (where `distinct` restricts the set of observations to those within unique entities). For instance, in the following basic discovery queries

Tree(TaxonName = 'piru')
Tree(TaxonName = 'piru' $\wedge$ count(distinct $*$) $\geq 5$)

the first query selects data sets with at least one Tree observation labeled as having the (abbreviated) taxon name "piru", and the second query restricts the returned data sets of the first query to contain at least five such observations.

A *contextualized discovery query* generalizes basic discovery queries to allow selections on context. A contextualized query $Q_C$ for $n \geq 1$ has the form

$$Q_C \; ::= \; Q_1 \rightarrow Q_2 \; \cdots \rightarrow Q_n$$

where each $Q_i$ is a basic discovery query and $\rightarrow$ denotes a context relationship. A data set satisfies a contextualized query if it satisfies each basic query $Q_i$ and each matching

$Q_i$ is related by the given context constraint. In a contextualized query, there exists at most one aggregation function in a basic query. In addition, since each $Q_i$ has $Q_{i+1}$ as its context, the entity of $Q_1$ is *logically* contained in its context queries' entities. So, the aggregation condition applies to $Q_1$ (the finest level of entity) if $Q_C$ has an aggregation condition. To illustrate, the following examples can be used to express the two queries of Sect. 1:

Tree(Height) $\rightarrow$ Plot()
Tree(max(height) $\geq$ 20 Meter) $\rightarrow$ Plot(area $<$ 10 MeterSquare).

That is, the first query returns data sets that contain height measurements of trees within plots (experimental locations), and the second returns data sets that have trees of a maximum height larger than 20 $m$ within plots having an area smaller than 10 $m^2$. For a collection of data sets $D$ and a contextualized discovery query $Q$, in the normal way, we write $Q(D)$ to denote the subset of data sets in $D$ that satisfy $Q$. Note that $Q(D)$ can be computed on a per data set basis, i.e., by checking each data set in $D$ individually to see whether it satisfies $Q$.

## 3 Approaches for Evaluating Data Discovery Queries

In this section we describe two different strategies for evaluating data discovery queries over annotated observational data sets. As shown in Fig. 8, both strategies utilize semantic annotations (and corresponding ontologies) to answer discovery queries. We assume annotations are stored using the relations described in Sect. 2, namely the *Annot*, *ObsType*, *MeasType*, *ContextType*, and *Map* tables. The two approaches differ in how they utilize different representations of the underlying data sets. The first query strategy (*rdb*) stores each data set in its "raw" form (i.e., "in place") according to its defined schema. Thus, each data set is stored as a distinct relation in the database. For instance, both data sets of Fig. 1 are stored as tables without any changes to their schemas. The second query strategy (*mdb*) materializes each data set using the *MaterializeDB* algorithm. In this approach, the observations and measurements stored in each data set are represented using the *Entity*, *Obs*, *Meas*, and *Context* tables described in Sect. 2.1.

In what follows, we first present our approaches for evaluating basic discovery queries using these strategies (Subsections 3.1 and 3.2, respectively). Then, in Subsection 3.3 we extend these strategies for evaluating complex discovery queries.

### 3.1 Query Evaluation over In-place Database

Evaluating a basic discovery query $Q$ in the *rdb* approach consists of three steps. Here we assume that each data set (denoted by id $d$) is stored in a relation $R_d$. The first step prunes the search space of candidate data sets to select only those data sets with the required entity, characteristic, and standard types specified within $Q$. The candidate data sets (i.e., those that potentially match $Q$) are selected by accessing the semantic-annotation relations $ObsType$, $MeasType$, and $Annot$. This pruning step can help decrease the cost of evaluating discovery queries by reducing the number of data sets

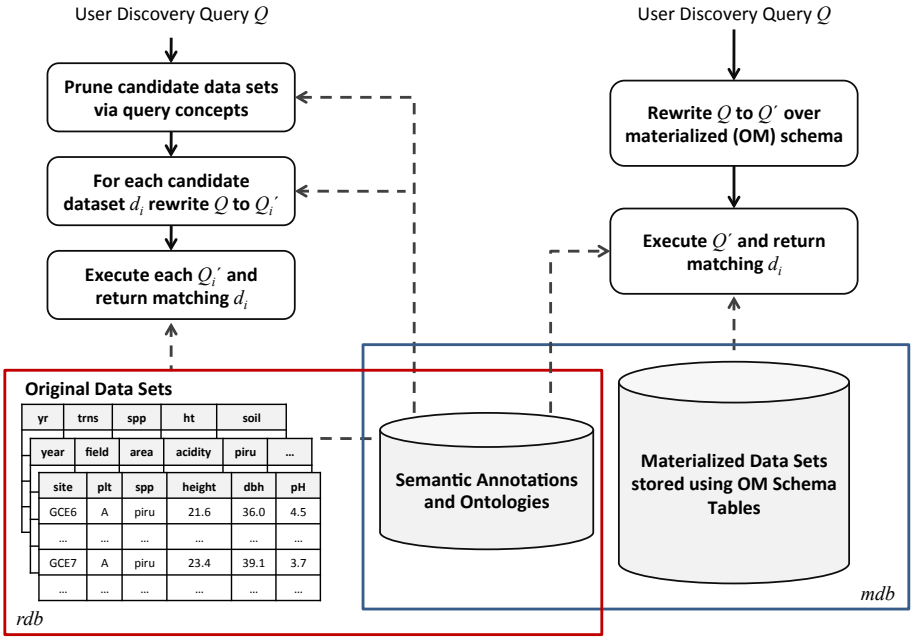



**Fig. 8.** Different strategies for answering a data discovery query $Q$: the first strategy stores each data set in its "raw" form (left) and the second strategy materializes each data set into a common schema (right)

whose values must be accessed (for those queries that select data sets based on data values). The effects of this step are illustrated by the experimental results in Fig. 17.

The second step translates $Q$ into an SQL query $Q'$ over each candidate data set relation $R_d$. After the first step, we obtain the measurement types (*mt*) related to the discovery query $Q$ (via the *MeasType* relation). We then find the attributes $attr_q$ in each candidate relation $R_d$ via the $Map$ relation, which contains correspondences between attributes $attr$ and measurement types $mt$. If a basic query does not have any aggregation operators, these attributes are sufficient to form the resulting SQL over $R_d$. Otherwise, we must obtain the key measurement types $attr_{key}$ for the observation types of $Q$. Recall that (Definition 1) the key measurement types of an observation type consist of its own key measurement types and those of its identifying observation types, which must be retrieved by traversing the identifying context chain (i.e., by searching the $ObsType$ and $ContextType$ relations). Once the needed attributes for each candidate data set $d$ are obtained, we apply three transformation rules to translate $Q$. First, the `SELECT` clause retrieves only the id of the data set when there is no aggregation function in $Q$. Otherwise, `SELECT` clause may retrieve the $attr_{key}$ values. Second, the `WHERE` clause contains all the conditions for the `EntityType`, `CharType`, and `StandardType`. Third, if $Q$ has an aggregation function, we need to include a `GROUP BY` and a `HAVING` clauses to perform the aggregation. With these rules, $Q$ is translated into the following SQL query (where square brackets denote optional clauses).

```
SELECT DISTINCT d[, attr_key] FROM R_d
WHERE non-aggregation conditions
[GROUP BY attr_key]
[HAVING aggregation condition];
```

The third step executes the SQL query over the database after transforming $Q$ to SQL query $Q'$.

*Example 4.* We illustrate the rewriting process with the following example. Consider the semantic annotations in Fig. 3 and the basic discovery query

Tree(TaxonName = 'piru' $\wedge$ count(distinct $*$) $\geq 5$)

from Subsection 2.2. The first step finds the attributes involved in the condition (i.e., TaxonName = 'piru'). From the entity type "Tree" and the characteristic "TaxonName", we can find the corresponding observation type "$ot_4$", measurement type "$mt_5$", and the attribute "spp". We then find the key attributes to perform the aggregation. The key measurement types for entity type "Tree", whose observation type is $ot_4$, come from $ot_4$ and $ot_4$'s identifying observation types $ot_2$ and $ot_1$. Since these observation types' key measurement types are $mt_5$, $mt_3$, $mt_2$ and $mt_1$, the key attributes $attr_{key}$ are $spp$, $size$, $plt$, and $site$ (from the $Map$ relation). The resulting SQL query is expressed as follows where the data set id is $d_1$ and the corresponding relation is denoted as $R_{d1}$.

```
SELECT DISTINCT d_1 FROM R_d1
WHERE spp = 'piru'
GROUP BY d_1, spp, size, plt, site
HAVING count(*) ≥ 5;
```

Finally, in the third step of the *rdb* approach, we execute the SQL query for each of the candidate data sets such that the answer for $Q$ is the union of these results.

**Cost Analysis.** The major computation cost using the *rdb* approach is to send multiple SQL queries to the database server to search the needed information for all the different candidate data tables. Given a basic data discovery query $Q$, it can be translated to $|D|$ $Q'$ queries that must be evaluated over the different data sets in $D$. Thus, the complexity is $O(|D|)$ where $|D|$ is the total number of data sets in the database. Thus, the cost increases with larger numbers of candidate data sets which we can see from Fig. 21. Note that $Q'$ does not require any join operations, which makes the evaluation of each $Q'$ efficient.

### 3.2 Query Evaluation over Materialized Database

The second query strategy (*mdb*) evaluates a given discovery query $Q$ over the materialized database by directly rewriting $Q$ into an SQL query expressed over the annotation and instance relations of Subsection 2.1. This approach differs from *rdb*, which requires obtaining the underlying attributes for *each* individual candidate table and where one SQL query is constructed per candidate table. Instead, using *mdb*, a single SQL query is created to answer the entire basic data discovery query.

The way in which a basic query is rewritten depends on whether it has an aggregation condition. For a basic query without an aggregation condition, we access the $Entity$ and $Obs$ relations to check the entity conditions, and search the *MeasType* and *Meas* tables for characteristic and standard conditions. For a basic discovery query with an aggregation condition, we perform the aggregation by grouping $Entity.e$ or $Obs.o$ depending on whether the aggregation is on a distinct entity instance or observation instance. Thus, a discovery query $Q$ can be re-written to an SQL query $Q'$ using the *mdb* approach as follows.

```
SELECT DISTINCT Annot.d [, Entity.e][, Obs.o]
FROM Annot, Entity, Obs, MeasType, Meas
[WHERE table join condition [AND selection condition]]
[GROUP BY Annot.d[, Entity.e][, Obs.o] ]
[HAVING aggregation condition ];
```

where:

```
table join condition = (Annot.a = MeasType.a)
   AND (Annot.d = Meas.d) AND (MeasType.mt = Meas.mt)
   AND (Meas.o = Obs.o) AND (Obs.e = Entity.e)
```

In this table join condition, there is no need to include the join conditions $Meas.d = Obs.d$ and $Obs.d = Entity.d$ because the observation id $Obs.o$ and the entity id $Entity.e$ are globally unique identifiers. That is, no observation or entity instances share the same identifier even though they are from different data sets. If a basic query contains multiple measurement conditions (of the same entity type), the corresponding SQL query must be combined using "INTERSECTION" or "UNION" operations to answer the basic query of one entity type.

*Example 5.* Consider the following query again

Tree(TaxonName = 'piru' $\wedge$ count(distinct $*$) $\geq 5$)

It is rewritten using the *mdb* approach as:

```
SELECT DISTINCT Annot.d
FROM Annot, Entity, Obs, MeasType, Meas
WHERE table join condition
    AND MeasType.ct='TaxonName' AND Meas.v = 'piru'
GROUP BY Annot.d, Obs.o
HAVING COUNT(*)>5;
```

**Cost Analysis.** The major computation cost in *mdb* involves the cost of joining over the type and instance relations, and the selection cost over the measurement values. In *mdb*, a basic data discovery query $Q$ is translated to *only one* SQL query $Q'$. This is more efficient compared with *rdb* strategy which translates $Q$ to $|D|$ $Q'$ queries. However, executing $Q'$ in *mdb* involves joining several large instance and type relations. These join operations are generally expensive. We do not include the complexity for performing the join operation because the join strategy is decided by the database system.

**Horizontally Partitioned Database.** Let the above described $mdb$ approach be denoted as $mdb2$ (as opposed to $rdb1$). Note that $mdb2$ is a storage scheme consisting of all the type and instance relations that are described in Sect. 2.1. In the $mdb2$ storage scheme, the measurement relation $Meas$ contains all the data values in one column, thus these data values share the same data type no matter whether their original data types are the same or not. This design of using uniform data type for different data values requires utilizing type casting functions provided by the database system to perform type conversion for evaluating queries with algebraic or aggregation operators. Such type conversion incurs a full scan of the $Meas$ table regardless of whether there is an index on the value columns or not. Consequently, the query execution over the simple storage scheme of $mdb2$ is very costly. To alleviate this issue, we propose to partition the measurement instance table $Meas$ to several horizontally partitioned instance tables according to the different data types (e.g., numeric, char, etc.) of the data values in their original data sets. The relations coming after partitioning $Meas$ table are assigned new relation names which start with ***Meas*** and have data types as a suffix, e.g., $MeasNumeric$, $MeasChar$, etc. The new relations are almost the same to that of $Meas(d, m, mt, v, o)$ with the same attributes except that the data types for the data value attribute $v$ are different. The partition is done when loading data into the databases (i.e., when materializing the data sets). We use $mdb3$ to denote the storage scheme with the partitioned $Meas$ relations and all the other unpartitioned type and instance relations. The $mdb3$ scheme does not incur additional space overhead compared with $mdb2$ because it stores the same information as $Meas$, but using several $Meas$-data-type relations. It can support the same data discovery queries that $mdb2$ supports. When $mdb3$ is used to evaluate a data discovery query e$Q$, the algorithm translates $Q$ to an equivalent SQL expression by replacing all occurrences of $Meas$ to the corresponding $Meas$-data-type relations.

**De-Normalized Materialized Database.** The join condition shows that a large portion of the query evaluation cost comes from the join operation over the measurement instance, observation instance, and entity instance relations. To reduce the join cost, we propose to de-normalize multiple relations into a single relation. Considering the most frequently used joins on a basic discovery query, we examine two de-normalization strategies. The first ($mdb4$) is to de-normalize the $Entity$, $Obs$, and $Meas$ relations into a single table to avoid the cost of joining them. The second ($mdb5$) de-normalization strategy leverages the characteristic type information by also including (de-normalizing) the $MeasType$ table with the instance tables. The de-normalized relation contains the same number of rows as that in $Meas$ although it contains more columns than $Meas$. In the $mdb4$ scheme, the de-normalized relation contains all the attributes from the $Meas$ relation and three more columns $(ot, e, et)$ from the $Obs$ and $Entity$ relations. That is, it has eight columns (three columns more than that of the original $Meas$). The new columns in the de-normalized relation may contain duplicate values of $ot$, $e$, or $et$. However, the amount of value duplication is generally moderate. Thus, $mdb4$ may still use less space than $mdb2$ compared with the three instance relations which contain twelve columns in total. In $mdb5$, the de-normalized relation contains all the columns of the de-normalized relation in $mdb4$, and includes more columns ($ct$, $st$, $\cdots$) from the $MeasType$. These additional columns again contain

**Table 1.** Different storage schemes for materialized database

| Abbreviation | Storage schemes |
|---|---|
| mdb2 | $mdb$ with measurement values in uniform type |
| mdb3 | $mdb$ with horizontal partition of measurement instance tables with data types |
| mdb4 | De-normalize the three instance tables ($Entity$, $Obs$, and $Meas$) |
| mdb5 | De-normalize the three instance tables ($Entity$, $Obs$, and $Meas$) and measurement types ($MeasType$) |
| mdb3i | *mdb3* with index on search keys |
| mdb4i | *mdb4* with index on search keys |

duplicate values for the same measurement type. The de-normalization is done only once after materializing the databases. Table 1 summarizes the different storage schemes for the materialized database approaches.

The de-normalization strategies are a trade-off between storage space and the query evaluation efficiency. Using the de-normalized scheme, the cost of performing joins can be reduced since the join operation over several relations is already done during the de-normalization process. Alternatively, de-normalized relations consist of columns with duplicated values. For example, the characteristic type $ct$ and the standard type $st$ for the same measurement type $mt$ are duplicated wherever the same measurement type is used. Such duplication may greatly increase the number of data cells in the de-normalized relation and increase the evaluation cost of selection operation over measurement types. In our experiments in Sect. 4, we compare the space used by different materialization strategies (Fig. 13) and examine the effect of each storage scheme with different query settings (Tests 1–3 for *queries over the materialized database* in Sect. 4.2).

### 3.3 Evaluating Complex Discovery Queries

The above approaches can be used to rewrite and evaluate a *basic discovery query* with one entity type and multiple (including one) measurement type conditions (as shown in Examples 4 and 5). However, we have found that complex data-discovery queries (e.g., involving context relationships [8]) are important for helping to improve the relevance of query results (query precision). These queries consist of complex operations such as context relationships, or conjunctions and disjunctions of basic/contextualized queries. For instance, $Q_3 \wedge Q_1 \rightarrow Q_2 \vee Q_4$ is a complex discovery query, where $Q_1, \cdots, Q_4$ are basic data discovery queries. According to the query semantics, the operations within this query listed by ascending precedence order are context, conjunction, and disjunction. In this example, applying the operations in this order results in the equivalent query $(Q_3 \wedge (Q_1 \rightarrow Q_2)) \vee Q_4$. The complicated operations (or components) in a complex discovery query make the direct employment of the basic query evaluation strategies inapplicable. In this subsection, we present approaches for executing complex discovery queries by utilizing the basic query evaluation procedures as building blocks.

A complex query can be viewed as a logical formula by considering every basic and contextualized query as a literal (or a term) in this logical formula. Since every logical formula can be formulated in disjunctive normal form (DNF) over its terms, a complex

query can also be converted to a DNF expression. Making use of this property, we can evaluate a complex data-discovery query by converting it into DNF, evaluating each DNF clause, and merging (through union) the results of all the DNF clauses. Theoretically the conversion of a logical formula to DNF can lead to an exponential explosion of the formula. However, the approach of parsing a query into DNF is still reasonable because in practice complex queries generally do not include large numbers of logical terms.

Each clause in the DNF of a complex query $Q$ is a query block. It is a conjunctive normal form (CNF) of basic or contextualized discovery queries. In a special situation, a CNF may contain only one basic data discovery query or one contextualized query. For instance, $(Q_3 \wedge (Q_1 \rightarrow Q_2)) \vee Q_4$ has two clauses in its DNF: $DNF_1 = Q_4$, which is a basic query, and the other is $DNF_2 = Q_3 \wedge (Q_1 \rightarrow Q_2)$, which is the conjunction of a basic query $Q_3$ and a contextualized query $Q_1 \rightarrow Q_2$.

To evaluate a contextualized query $Q_C = Q_1 \rightarrow Q_2 \cdots \rightarrow Q_n$, we need to consider two situations. In the first situation, there is no aggregation condition in any of the basic query components. In this case, the execution of basic query component $Q_i$ is not affected by any of its context (as described in Subsections 3.1 and 3.2). So, the correct results of $Q_C$ can be calculated by evaluating each $Q_i$ independently and intersecting their results. However, when there is an aggregation condition, to correctly execute the basic query with the aggregation condition, we need to first apply the conditions in the context queries which do not contain any aggregation operator. For instance, given the query

$$\text{Tree(max(height)} \geq \text{20 Meter)} \rightarrow \text{Plot(area} < \text{10 MeterSquared)},$$

the maximum aggregation only applies to trees in a plot with size less than $10m^2$ as denoted in Subsection 2.2.

Using this approach, a complex query can be evaluated in two steps.

- Step 1: The first step is to parse a query into its DNF representation. Further, since each DNF clause (a query block) is still complicated when it contains conjunctions and context, this step further decomposes each DNF clause into basic queries.
- Step 2: In this step, we evaluate and integrate the query blocks by utilizing the query capacity of a DBMS. For this step, we present two query schemes, *ExeD* and *ExeH*.

In what follows, we present our method in detail for evaluating a complex query.

Fig. 9 shows the procedure **QueryParse** for parsing a complex query $Q$ into query blocks. This algorithm first converts a query to its DNF representation. Then, it parses each DNF clause to smaller query units. Each query unit is either a basic query or a contextualized query with an aggregation operation. In the contextualized query with aggregation, the left hand side of the context chain "$\rightarrow$" is a basic query with an aggregation operation, and its context is merged together. Step 2(b)iii details the process. Recall (Subsection 2.2) that a contextualized query can have at most one aggregation condition. In addition, when a contextualized query $Q_i$ contains an aggregation condition, its first basic query $Q_{i1}$ contains the aggregation operation. So, all the other basic queries in the context chain can be put together into $Q'$ as a CNF expression. Consider the above query $Q = (Q_3 \wedge (Q_1 \rightarrow Q_2)) \vee Q_4$. The *QueryParse* procedure generates

**Function QueryParse**($Q$)
/* Output: query blocks of $Q$ */

1. Rewrite $Q$ to DNF;
2. **For** (each DNF clause $DNF_i$) /* Decompose each $DNF$ clause*/
   (a) $QB_i = \emptyset$; /* Query block for $DNF_i$*/
   (b) **For** (each clause $Q_i \in DNF_i$)
   /* Each $DNF_i$ is a CNF of basic queries and/or contextualized queries*/
      i. **If** $Q_i$ is a basic query, add $Q_i$ to $QB_i$;
      ii. **If** $Q_i$ is a contextualized query without aggregation
         – **For** each $Q_{ij}$ in $Q_i$, add $Q_{ij}$ to $QB_i$;
      iii. **If** $Q_i$ is a contextualized query *with* aggregation
         A. For each $Q_{ij}$ ($j \geq 2$) in $Q_i$, add $Q_{ij}$ to $Q'$;
         B. Add $Q_{i1} \rightarrow Q'$ to $QB_i$;
   (c) $DNF = DNF \cup QB_i$;
3. **return** $DNF$;

**Fig. 9.** Function to parse a complex data discovery query $Q$ to its query blocks

$DNF = \{QB_1, QB_2\}$ where $QB_2 = \{Q_4\}$. As to $QB_1$, when $Q_1$ does not contain any aggregation operation, $QB_1 = \{Q_1, Q_2, Q_3\}$. Otherwise, $QB_1 = \{Q_1 \rightarrow Q_2, Q_3\}$. The *QueryParse* procedure rewrites a query $Q$ in the form of $QB_1 \vee \cdots \vee QB_{|DNF|}$ where each query block $QB_i$ ($1 \leq i \leq |DNF|$) is in the form of $qu_{i1} \wedge \cdots \wedge qu_{ij} \wedge \cdots$.

**ExeD: Executing a Query Based on Decomposed Query Units.** *ExeD* is the first query scheme to process a complex query. It executes a query by evaluating its most decomposed query units. In *ExeD*, each of the query units of a DNF clause (i.e., query blocks) are rewritten into SQL queries and executed. The result of each such most decomposed query unit is combined (outside of the DBMS) to obtain the result of every DNF clause. The detailed process of the *ExeD* framework is shown in Fig. 10. This framework first utilizes the *QueryParse* function to reformulate the query into a DNF representation such that each DNF clause is a query block. Then, it executes every DNF clause $DNF_i$ by evaluating its query units independently. In particular, when a query unit represents a basic discovery query, the query unit is executed using one of the strategies discussed in Subsections 3.1 and 3.2. When a query unit is a contextualized query with one context in the context chain (Step 3(b)ii), it means that this query has an aggregation operation. *ExeD* first constructs the condition for the "`WHERE`" clause from *all* the constraining queries (i.e., context). Then it adds the constraining "`WHERE`" clause to the SQL expression for the basic query with the aggregation condition, where the construction of this SQL expression (especially the GROUP BY and HAVING clauses) follows the principles in translating a *basic* data discovery query in the previous two sections (Sect. 3.1 for *rdb* and Sect. 3.2 for *mdb*). Next, the SQL query is evaluated using either an in-place database or a materialized database. Finally, the results of the different DNF clauses are unioned together as the final result.

**Algorithm ExeD**($Q$, $db$)

1. $R = \emptyset$;
2. $DNF = \mathbf{QueryParse}(Q)$; /* From the given query, get the DNF. */
3. **For** (each $DNF_i$ in $DNF$) /* Execute every DNF clause and union their results. */
   (a) $R_i = \emptyset$;
   (b) /* Execute every query block in this DNF clause and intersect their results */
   **For** (each query unit $qu$ in the query block of $DNF_i$)
   i. If $qu$ is a basic query, $sql$ = form a basic query from $qu$;
   ii. else, i.e., $qu = q \rightarrow \{cq\}$ is a contextualized query with aggregation,
   – Form a `WHERE` clause from all the conditions in all $cq$s;
   – $sql$ = form a basic query sql from $q$ and add the `WHERE` clause from $cq$;
   iii. $R_{ib}$ = Execute($sql$, $db$);
   iv. $R_i = R_i \cap R_{ib}$;
   (c) $R = R \cup R_i$;
4. Return $R$;

**Fig. 10.** Algorithm to execute a complex data discovery query $Q$ over a database $db$ by executing a query block based on its decomposed query units

The *ExeD* approach may incur unnecessarily repeated scans of the database since it evaluates each query unit using the DBMS and combines the results externally, outside of the system. For instance, consider the query

$$\text{Tree(height} \geq 20 \text{ Meter)} \rightarrow \text{Plot(area} < 10 \text{ MeterSquared).}$$

Using the *ExeD* approach, we must send two basic queries, Tree(height $\geq$ 20 Meter) and Plot(area $<$ 10 MeterSquared), to the same table.

**ExeH: Executing a Query Based on Holistic Sub-Queries.** To overcome the problem of repeated scanning of a table in *ExeD*, we propose a new strategy based on holistic (as opposed to "decomposed") sub-queries. Here, "holistic" means that the query units are combined together (when it is possible) and are evaluated together. Thus, the idea of *ExeH* is to form a "holistic" SQL query for all the possible basic query units and to execute this holistic SQL by taking advantage of the optimization capabilities of the DBMS. When a complex query does not contain any aggregation operation, it can be converted to a holistic SQL by translating conjunction to an "AND" condition or an "INTERSECT" clause, and translating disjunction to an "OR" condition or a "UNION" clause. However, not every complex query can be rewritten to one holistic SQL query. Specifically, queries with aggregations must be performed by grouping key measurements. When we have discovery queries with multiple aggregation operations (e.g., in multiple DNFs), the "GROUP BY" attributes for each aggregation may not be the same. In *ExeH*, whose framework is shown in Fig. 11, we categorize query blocks into those with and without aggregations (Step 3). All the query blocks without aggregation conditions are combined and rewritten to one holistic SQL query (Step 4 to 6), while the query blocks with aggregations are processed individually (Step 7). *ExeH* is very similar to *ExeD* except that it separates the basic queries with aggregations $DNF_{ag}$ and without aggregations $DNF_{nag}$. For queries without aggregations, we can form a

**Algorithm ExeH**($Q$, $db$)

1. $R = \emptyset$;
2. $DNF$ = $\mathbf{QueryParse}(Q)$; /* From the given query, get the DNF. */
3. $DNF_{ag}, DNF_{nag}$ = $\mathbf{PartitionDNF}(DNF)$;

   /* Execute the DNFs without aggregation */
4. $sql$ = FormHolisticSql($DNF_{nag}, db$);
5. $R_{nag}$ = Execute($sql$, $db$);
6. $R = R \cup R_{nag}$

   /* Execute every DNF with aggregation */
7. **For** each $DNF_i$ in $DNF_{ag}$
   (a) $sql$ = construct an SQL statement for $DNF_i$;
   (b) $R_{ag}$ = Execute($sql$, $db$);
   (c) $R = R \cup R_{ag}$;
8. Return $R$;

**Fig. 11.** Algorithm to execute a complex data discovery query $Q$ over a database $db$ by using holistic query units

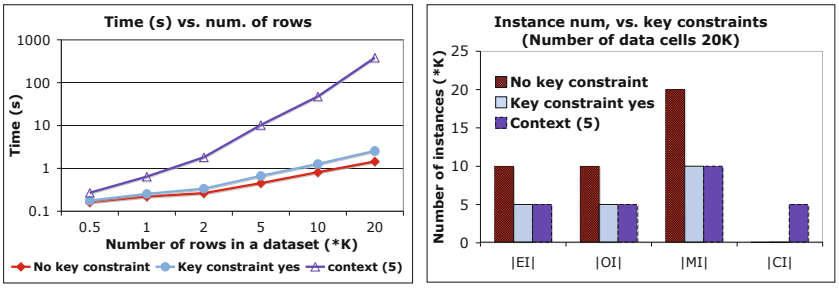
holistic SQL (Step 4). For queries with aggregation, we can form SQL statements as Step 3(b)ii in *ExeD* and execute them.

## 4 Experimental Evaluation

In this section we describe our experimental results of the framework and the algorithms discussed above to evaluate the performance and the scalability (with respect to time and space) of the different query strategies. Our implementation was written in Java, and all experiments were run using an iMac with a 2.66G Intel processor and 4G virtual memory. We used PostgreSQL 8.4 as the back-end database system. To report stable results, all numbers in our figures represent the average result of 10 different runs (materialization tasks or queries) with the same settings for each case.

**Data.** We generated synthetic data to simulate a number of real data sets used within the Santa Barbara Coastal (SBC) Long Term Ecological Research (LTER) project [4]. This repository contains $\sim$130 data sets where each one has 1K to 10K rows and on average 15 to 20 columns. To simulate this repository (to test scalability, etc.), our data generator used the following parameters. The average number of attributes and records in a data set was 20 and 5K, respectively. The average number of characteristics for an entity was two. The *distinctive factor* $f \in (0, 1]$, which represents the ratio of distinct entity/observation instances in a data set, was set to 0.5. We also set the longest length of context chains to be 5 to test the execution of complex queries with context and aggregation. In this way, a data set can have observations with many context relationships as well as observations without any context.

Our synthetic data generator also controls *attribute selectivity* to facilitate the test of query selectivity. In particular, given a selectivity $s \in (0, 1]$ of an attribute $attr$, a




(a) Time (b) Space

**Fig. 12.** Cost of data materialization

repository with $|D|$ data sets will be generated to have $|D| \cdot s$ data sets with attribute $attr$. For example, for a repository with 100 data sets, if an attribute's selectivity is $0.1$, then 10 data sets in this repository have this attribute. In using the *rdb* storage schema, this factor affects the number of SQL queries that need to be sent to the database, thus the number of table scans needed to perform the query. In using the *mdb* storage schema, this factor affects the number of *candidate* instances that are involved in the join and selection condition. To generate a repository with $n$ data sets, we used characteristic selectivity 0.01, 0.05, 0.1, 0.2 and 0.5.

With the above settings, we generated repositories with various numbers of data sets ($|D| = 20$, 50, 100, 1K). Using this synthetic data, we tested the effect of different factors of the data to the query strategies. In our experiments, we first examine the efficiency of the materialization algorithm. Then, we show the effect of the query strategies over data sets with these different settings.

### 4.1 Materialize Database

We first tested the efficiency (time and space usage) of the materialization method using data sets generated with a distinctive factor $f = 0.5$, number of columns 20, and various number of rows (from 0.5K to 20K). For a given number of rows (e.g., 1K), 10 data sets are generated using 10 different random sequences. We used data sets to collect the average running time and the average number of materialized result instances. The annotations over these data sets are the same on observation and measurement types. They differ in the key constraints. "No key constraint" and "Key constraint yes" refer to cases where either no key or key constraints exist in the semantic annotations of data sets; both of these two cases do not include any context constraints. The "context (5)" represents data sets that are semantically annotated with a context chain of 5 observation types (with implicit key constraints).

Fig. 12(a) shows that the materialization time, where each case (every line) is linear to the number of rows in these data sets. This result is consistent with our analysis in Subsection 2.1. The "no key constraint" uses the least amount of time because it does not need to do any additional computation to enforce the uniqueness of the entity and observation instances. The "context (5)" uses the most amount of time because of the context materialization.

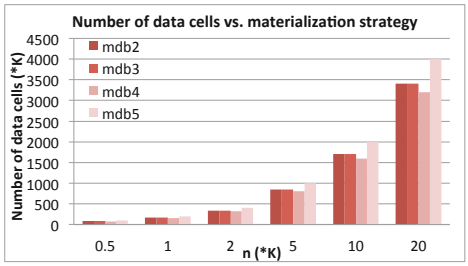



**Fig. 13.** Space complexity of different data materialization strategies

Fig. 12(b) plots the number of instances generated for a data set with 20K data cells. For the case without any key constraints, the number of measurement instances is the same as the number of data cells because it treats every cell as a different instance, even though different cells may carry the same value. However, for the case with key constraints, half the number of instances are used since the distinctive factor $f$ is 0.5. When the context chain is of length 5, the number of context instances is the number of observation instances times the chain length.

Fig. 13 shows the number of data cells used in different materialization strategies for data sets with the number of rows $n$ varying from 500 to 20K. These data cell numbers give an estimation of the space used although the data types of different data columns may be different. The total number of data cells include the cells used for the measurement instance $Meas$, the observation instance $Obs$, the entity instance $Entity$, and the measurement type $MeasType$ relations because these relations are the ones that are changed during partition or de-normalization. For $mdb2$ and $mdb3$, the cell number counts all those from all these relations. Their space usage is the same since $mdb3$ partitions $Meas$ in $mdb2$ to several smaller ones. For $mdb4$, the cells count those in $MeasType$ and in the de-normalized relation for $Meas$, $Obs$, and $Entity$. $mdb4$ uses less space than $mdb2$ because we de-normalize the three relations into one by getting rid of the redundant column information. For $mdb5$, the cells are those from the de-normalized relation for $Meas$, $Obs$, and $Entity$, and $MeasType$. $mdb5$ uses much more space because of the value duplication caused by the measurement type. The experiments in the next Subsection show that the $mdb4$ scheme provides the best query support of those considered.

### 4.2 Querying Databases

Here we test the effectiveness and efficiency of *ExeD* and *ExeH* over the different storage schemas *rdb*, *mdb* and the variations of *mdb* (as shown in Table 1) that were discussed in Subsection 3.2. To test the effect of different factors over the query methods, we generate a batch of synthetic queries. For the purpose of getting stable results, each result value in all our figures is the average result of running 10 different queries with the same setting for each run. We use "QueryMethod (storage schema)" to represent the test of a query method over a given storage schema.

The synthetic query generator generates three different types of queries to test the feasibility of the query strategies. These include queries with different numbers of logic connectors (e.g., AND, OR), with different lengths of context chains, and with aggregation functions.

To test the effect of the different approaches over the storage schema, the query generator controls the query selectivity by using the characteristic selectivity and value selectivity. Characteristic selectivity corresponds to the attribute selectivity in the synthetic data generator. To generate a query with a given characteristic selectivity, we simply retrieve the corresponding attributes that have this selectivity. The value selectivity $s_{val}$ ($\in (0, 1]$) determines the percentage of data rows in a data set that satisfy a given measurement condition. For example, when the selection condition $mval \geq 10.34$ for characteristic "Height" has value selectivity 0.1, then a table with 1K records has 100 rows satisfying this selection condition. Thus, this factor controls the number of *result* instances of a query. To generate a query condition satisfying a value selectivity, we first count the number of distinctive values (or value combinations) for an attribute (or attributes). We then combine them to create conditions with a given selectivity. The synthetic queries are generated with different characteristic selectivities. For data sets with the same characteristic selectivity, we generated queries with value selectivity 0.001, 0.01, 0.1, 0.2 and 0.5.

**Queries over the Materialized Database.** In this series of experiments, we used a data repository with 100 data sets. The queries have fixed characteristic selectivity 0.01. Each query contains two non-aggregation basic data-discovery queries which are connected using one logic connector (in our case "AND").

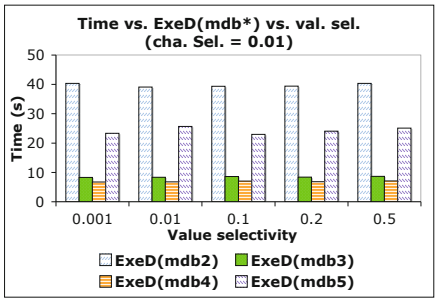



**Fig. 14.** Query materialized DB: time of performing a query over different storage variations.

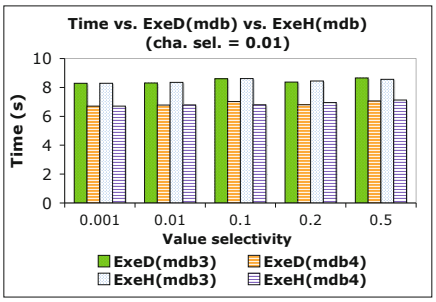



**Fig. 15.** Query materialized DB: time of performing a query using different query strategy: **ExeD** and **ExeH** perform similar

**Test 1:** Our first set of experiments test the effect of the different storage schema of the materialized database using *ExeD*. Fig. 14 shows the results. From this figure, first, we can examine the effectiveness of the instance table partitioning. As shown, $mdb2$ performs the worst because it stores different types of values as string values and any query with a value comparison condition must scan all the values of the table and convert them to their corresponding numeric values. Because of this requirement, a sequential scan over the instance relation is needed for any type of query condition. $mdb3$ outperforms

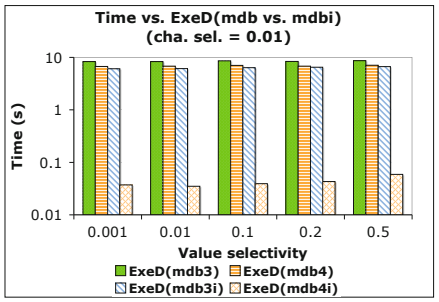



**Fig. 16.** Query materialized DB: time of performing a query using **ExeD** on different *mdb* (with and without index): Indexed mdb supports query in a much more efficient manner

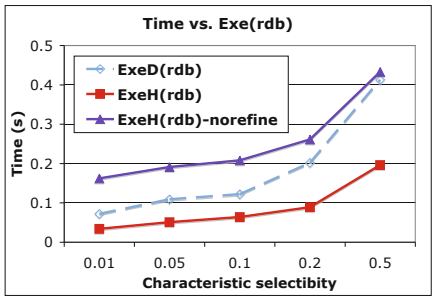



**Fig. 17.** Query raw DB: different strategies: (a) ExeD(rdb) performs worse than ExeH(rdb) (b) ExeH(rdb) perform better than ExeH(rdb)-norefine

$mdb2$ by partitioning the materialized measurement instances according to their value type.

Second, this figure also shows the effect of the de-normalization over the materialized data tables. $mdb4$ de-normalizes the instance tables, thus reducing the join cost over the instance tables compared with $mdb3$. However, this is not the case for the de-normalization strategy of $mdb5$. As shown in Table 1, $mdb5$ materializes one more table ($MeasType$) compared with $mdb4$. However, this de-normalization does not improve the efficiency (and instead, made it worse). This is due to two reasons. First, the $MeasType$ table is much smaller than the instance tables, thus, the de-normalization (i.e., reducing the join operation) does not improve the execution time as compared to joining the instance tables. Second, this de-normalization duplicates the $MeasType$ table with all of the other column information. Thus, the linear scan of the de-normalized table uses much more time.

Because $mdb2$ and $mdb5$ always require more time for query execution than with $mdb3$ and $mdb4$, we do not include them in the tests below.

**Test 2:** The second set of tests examine how the two different query strategies affect the query efficiency over the materialized database with different storage schemas. As mentioned in *Test 1*, $mdb2$ and $mdb5$ always perform worse than *mdb3* and *mdb4*, so in this test we just use the storage schema *mdb3* and *mdb4*. Fig. 15 illustrates that these two query strategies perform similar for both *mdb3* and *mdb4*. This is because the holistic SQL in *ExeH* performs a "UNION" operation on the sub-queries in *ExeD*, and this "UNION" operation is not improved by the database internal optimization techniques.

**Test 3:** The third set of experiments check the effect of using indexing in $mdb3$ and $mdb4$. Fig. 16 shows that the search is much more efficient over a materialized database with indexes. The index helps improve the execution dramatically (comparing $mdb3$ with $mdb3i$, and $mdb4$ with $mdb4i$). However, the index of $mdb3$ does not work as effectively as that in $mdb4$ because the join operation dominates the cost.

**Queries over the Raw Database.** We test the query strategies over the in-pace databases using the same set of queries and database as those in querying materialized databases.

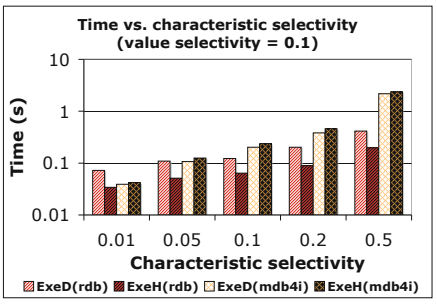



**Fig. 18.** Time of performing a query by varying characteristic selectivity

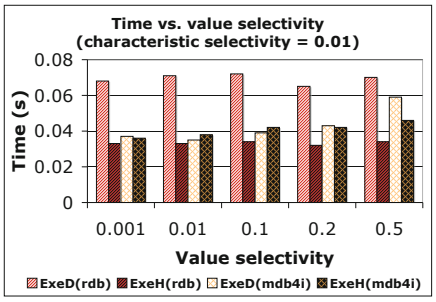



**Fig. 19.** Time of performing a query by varying value selectivity

**Test 4:** In this test, we check how *ExeD* and *ExeH* perform over the raw databases (rdb). Fig. 17 shows the running time that the different strategies used to search raw databases. *ExeH(rdb)-norefine* runs the *ExeH* over the database without the database pruning technique in the first step. Both *ExeH(rdb)* and *ExeD(rdb)* use the database pruning to get the candidate data sets. Comparing *ExeH(rdb)* and *ExeH(rdb)-norefine*, we can see that the database pruning technique improves the efficiency almost by a factor of three. This is because the pruning step reduces the search space by sending the rewriting requests only to the data tables containing the required entity or measurement information. As shown, *ExeH(rdb)* also outperforms *ExeD(rdb)*. This is because for a complex query with $|q_b|$ basic data-discovery queries, *ExeD* needs to send the re-written SQL to the database $q_b$ times and each time the searched data table is scanned once. On the other hand, *ExeH* only needs to send the query to the database once.

**Value Selectivity and Characteristic Selectivity of Different Query Strategies.** The following experiments examine how the different querying approaches are affected by the changes of value selectivity and characteristic selectivity. To perform this test, we used the same database as those for querying materialized databases. The queries also consist of two basic data-discovery queries connected with one logic connector ("AND").

**Test 5:** We show the results over *rdb* (using the database pruning technique in Subsection 3.1) and *mdb4i*, which provides the best support to query over materialized database. Figure 18 shows that the execution time of the different query methods increase with the increase of the characteristic selectivity. For *rdb*, this is because higher characteristic selectivity means more candidate tables, thus more SQL queries are sent to the tables. For *mdb4i*, it is because the number of candidate instances that are involved in the join conditions increases. However, when we fix the characteristic selectivity and vary the value selectivity, we observe that the execution time is almost constant (Fig. 19) for *rdb* because the number of candidate data tables is the same. For *mdb4i*, the execution time grows slightly with the increase of the the value selectivity also due to the increase in the number of candidate instances.

In addition, these two figures show that the rewriting strategy over the storage schema *rdb* performs better than that over the *mdb4i* schema because the queries over the large

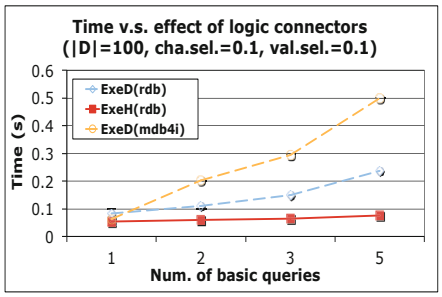



**Fig. 20.** Vary the number of logical connectors in a complex query (Fix $|D|$ =100)

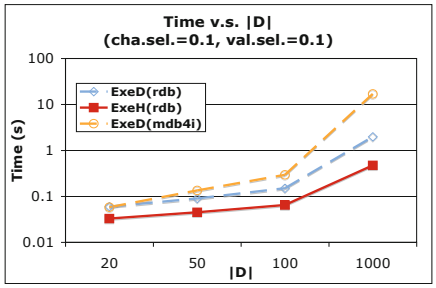



**Fig. 21.** Vary $|D|$ (Number of basic data queries connected by the logic connector is 3)

number of materialized instances use much more time compared with executing SQL queries over the candidate data tables in *rdb*. There are situations that *mdb4i* has a faster query response time than *rdb*, which we discuss in the next test.

**Complex Query Tests.** We also test how the different query strategies perform against the complex queries with logic connectors, context chains, and aggregations.

**Test 6:** Fig. 20 shows the results of performing queries with different numbers of logic connectors. Fig. 20 illustrates that the three methods grow linearly to the number of logic connectors. Initially, queries over a materialized database (with indexes) perform better than over raw databases. This is because the query involves only a simple selection over the materialized database when no logical connector is used. When there are more logical connectors, the materialized database is searched multiple times to get the results. When the number of logic connectors grow, *ExeD(mdb4i)* grows much faster than *ExeD(rdb)*. This is because every basic query in the complex query needs to access the large number of instance tables once for *mdb*. While using *rdb*, when the number of logic connectors is large, *ExeH(rdb)* is still almost constant because the time required to scan the database is almost the same.

Fig. 21 shows how the efficiency of performing a complex query is changed when we change the number of data sets $|D|$ in a repository. The results are consistent with the results in Fig. 20. That is, they grow linearly to $|D|$ once the value selectivity and characteristic selectivity is fixed.

**Test 7:** We compare the execution time over queries with logic predicates and those with context chains (value selectivity=0.1, characteristic selectivity=0.1). Fig. 22 plots the ratio of the execution time of performing a complex query with logic connectors and that of performing a contextualized query with the same number of basic query units. All three methods grow linearly to the number of basic queries. However, the query time over *rdb* grows slower than for *mdb* since the latter must access the context instance table.

As these results show, rewriting queries to the underlying data set schemas outperforms the materialization approach (i.e., the standard warehousing approach) with respect to both the cost of storage (especially due to de-normalization and executing the materialization algorithm) and overall query execution time.

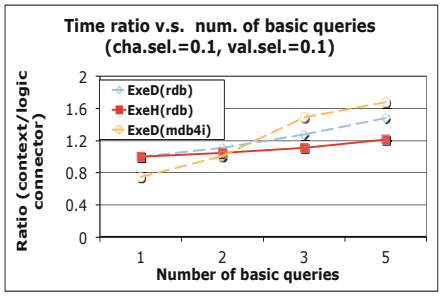



**Fig. 22.** The execution of complex query with logic connectors and context chains

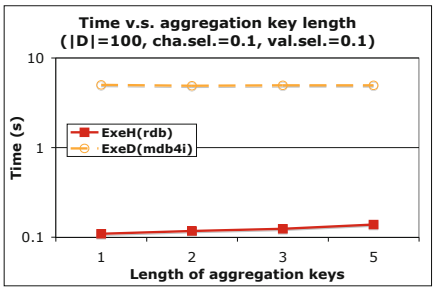



**Fig. 23.** Vary the length of aggregation attributes for aggregation query

**Test 8:** Finally, we test the effect of aggregation with one aggregation function in a complex query, but where the aggregation is over different numbers of key measurements. Fig. 23 shows that the query time is constant when the number of key measurements changes in *mdb4i*. This is due to the fact that the materialization algorithm already leverages key measurements. On the other hand, the query over *rdb* grows slightly linear to the number of key measurements because of the growth of the number of "GROUP BY" attributes.

### 4.3 Summary of Experimental Results

Our experiments test the performance of different storage schemes ($rdb1$, $mdb2$, $mdb3$, $mdb4$, $mdb5$) in supporting query evaluation, compare the efficiency of different query strategies (*ExeD* and *ExeH*), and show the scalability of our approaches. Among the different materialized database schemes ($mdb2$, $mdb3$, $mdb4$, $mdb5$), $mdb2$ has the worst query performance; the scheme $mdb4$, which only de-normalizes instance relations, performs better than any of the other $mdb$ schemes. When comparing the *mdb* and *rdb* storage schemes, the *rdb* storage scheme supports query evaluation more efficiently. When the storage scheme is fixed, the query strategy *ExeH* shows better performance than *ExeD*. To summarize, *ExeH(rdb)* provides the best overall performance, and it also shows *linear* scalability (scale-up) with the number of data sets and the number of basic queries for complex discovery queries.

## 5 Related Work

Data management systems are increasingly employing annotations to help improve search (e.g., [15,21,9]) and for maintaining data provenance (e.g., [9,12]). For example, MONDRIAN [15] employs an annotation model and query operators to manipulate both data and annotations. However, users must be familiar with the underlying data structures (schemas) to take advantage of these operators, which is generally not feasible for observational data in which data sets exhibit a high degree of structural and semantic heterogeneity. A number of systems have been proposed (e.g., [9,12], among

others) for propagating annotations (as simple text attached to data) through queries as well as for providing the ability to query over annotations. Efforts have also been carried out for leveraging annotations, e.g., for the discovery of domain-specific data [16,23]. These approaches are largely based on keyword queries, and do not consider structured searches. Our work differs from these approaches in that we consider a highly structured and generic model for annotations with the aim of providing a uniform approach for issuing structured data-discovery searches.

Our work is closely aligned to traditional data integration approaches (e.g., [17,19]), where a global mediated schema is used to (physically or logically) merge the structures of heterogeneous data sources using mapping constraints among the source and target schemas. As such, the observational model we employ in our framework can be viewed as a (general-purpose) mediation schema for observational data sets. This schema can be augmented with logic rules (as target constraints) and uses the semantic annotations as mapping constraints. However, instead of users specifying logic constraints directly, we provide a high-level annotation language that simplifies the specification of mappings and more naturally aligns with the observation model. While it is possible to express annotations given in the annotation language as first-order dependencies (e.g., [7]) together with object creation, our work focuses on exploring and implementing practical approaches for rewriting and optimizing queries that can include aggregation and summarization operators over our annotation approach.

## 6 Conclusion

We have presented a novel framework for querying observational data based on formal semantic annotations to capture the common observational semantic types and their relationships and a data discovery language that allows structural queries over both schema and data. We also have considered different strategies for efficiently implementing the framework. We examined the effect of different storage schemes. The in-place scheme *rdb* stores data tables for each data set; while the materialized scheme *mdb* materializes the contents in all the data sets into a central warehouse. We also presented two query strategies (*ExeD* and *ExeH*) to search semantically annotated data using the different storage schemes. The query strategy *ExeD* executes a data discovery query by decomposing it into query units and evaluating each query unit independently. *ExeH* on the other hand executes a query by executing its query units in a holistic manner. Our experiments show that in most cases answering queries "in place" (*rdb*) by utilizing *ExeH* outperforms more traditional warehouse-based approaches (using either *ExeH* or *ExeD*) even after applying different indexing schemes to the warehouse data. As future work, we intend to develop other query schemes by utilizing a DBMS's internal statistical information about the data, and to investigate newer parallel-processing approaches (such as MapReduce style distributed processing over key-value data stores) to help improve the efficiency and scalability of queries over semantic annotations.

**Acknowledgements.** We would also like to thank the reviewers for their thoughtful comments on the paper as well as Matt Jones, Margaret O'Brien, and Ben Leinfelder for their contributions to the development and testing of semantic annotation support within Metacat.

## References


1. Knowledge network for biocomplexity (KNB), http://knb.ecoinformatics.org
2. Morpho, M. (ed.), http://knb.ecoinformatics.org
3. OpenGIS: Observations and measurements encoding standard (O&M), http://www.opengeospatial.org/standards/om
4. Santa Barbara Coastal LTER repository, http://sbc.lternet.edu/data
5. The Digital Archaeological Record (tDAR), http://www.tdar.org
6. An, Y., Mylopoulos, J., Borgida, A.: Building semantic mappings from databases to ontologies. In: AAAI (2006)
7. Arenas, M., Fagin, R., Nash, A.: Composition with target constraints. In: ICDT, pp. 129–142 (2010)
8. Berkley, C., et al.: Improving data discovery for metadata repositories through semantic search. In: CISIS, pp. 1152–1159 (2009)
9. Bhagwat, D., Chiticariu, L., Tan, W.C., Vijayvargiya, G.: An annotation management system for relational databases. In: VLDB (2004)
10. Bowers, S., Madin, J.S., Schildhauer, M.P.: A Conceptual Modeling Framework for Expressing Observational Data Semantics. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 41–54. Springer, Heidelberg (2008)
11. Cao, H., Bowers, S., Schildhauer, M.P.: Approaches for Semantically Annotating and Discovering Scientific Observational Data. In: Hameurlain, A., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) DEXA 2011, Part I. LNCS, vol. 6860, pp. 526–541. Springer, Heidelberg (2011)
12. Chiticariu, L., Tan, W.C., Vijayvargiya, G.: DBNotes: a post-it system for relational databases based on provenance. In: SIGMOD, pp. 942–944 (2005)
13. Fagin, R., Haas, L.M., Hernández, M., Miller, R.J., Popa, L., Velegrakis, Y.: Clio: Schema Mapping Creation and Data Exchange. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 198–236. Springer, Heidelberg (2009)
14. Fox, P., et al.: Ontology-supported scientific data frameworks: The virtual solar-terrestrial observatory experience. Computers & Geosciences 35(4), 724–738 (2009)
15. Geerts, F., Kementsietsidis, A., Milano, D.: Mondrian: Annotating and querying databases through colors and blocks. In: ICDE, p. 82 (2006)
16. Güntsc, A., et al.: Effectively searching specimen and observation data with TOQE, the thesaurus optimized query expander. Biodiversity Informatics 6, 53–58 (2009)
17. Halevy, A., Rajaraman, A., Ordille, J.: Data integration: the teenage years. In: VLDB (2006)
18. Balhoff, J., et al.: Phenex: Ontological annotation of phenotypic diversity. PLoS ONE 5 (2010)
19. Kolaitis, P.G.: Schema mappings, data exchange, and metadata management. In: PODS (2005)
20. Pennings, S., et al.: Do individual plant species show predictable responses to nitrogen addition across multiple experiments? Oikos 110(3), 547–555 (2005)
21. Reeve, L., Han, H.: Survey of semantic annotation platforms. In: SAC (2005)
22. Sorokina, D., et al.: Detecting and interpreting variable interactions in observational ornithology data. In: ICDM Workshops, pp. 64–69 (2009)
23. Stoyanovich, J., Mee, W., Ross, K.A.: Semantic ranking and result visualization for life sciences publications. In: ICDE, pp. 860–871 (2010)