Quarrying Dataspaces: Schemaless Profiling of Unfamiliar Information Sources

Bill Howe[#1] **David Maier**[*2] **Nicolas Rayner**[*3] **James Rucker**[*4]

[#]Oregon Health & Science University, Center for Coastal Margin Observation and Prediction
20000 NW Walker Road, Beaverton, Oregon
[1]howeb@stccmop.org

[*]Portland State University, Department of Computer Science
1900 SW 4th Avenue, Portland, Oregon
[2]maier@cs.pdx.edu
[3]raynern@cs.pdx.edu
[4]jgrucker@cs.pdx.edu

***Abstract*— Traditional data integration and analysis approaches tend to assume intimate familiarity with the structure, semantics, and capabilities of the available information sources before applicable tools can be used effectively. This assumption often does not hold in practice. We introduce dataspace profiling as the cardinal activity when beginning a project in an unfamiliar dataspace. Dataspace profiling is an analysis of the structures and properties exposed by an information source, allowing 1) assessment of the utility and importance of the information source as a whole, 2) assessment of compatibility with the services of a dataspace support platform, and 3) determination and externalization of structure in preparation for specific data applications.**

In this paper, we define dataspace profiling and articulate requirements for dataspace profilers. We then describe the Quarry system, which offers a generic browse-and-query interface to support dataspace profiling activities, including path profiling, over a variety of data sources with minimal setup costs and minimal a priori assumptions. We show that the mechanisms used in Quarry deliver strong performance in large-scale applications. Specifically, we use Quarry to efficiently profile 1) a detailed standard for medication nomenclature supplied under a generic schema and 2) the metadata for an environmental observation and forecasting system, and conclude that in these contexts Quarry offers advantages over existing tools.

I. Introduction

We are constantly adapting existing information sources to new or extended uses. When such adaptation includes multiple sources, a classical approach is "engineered" data integration [1] over selected sources, carefully combined and validated, with uniform capabilities over the integrated data. An alternative is the *dataspace* approach [2], [3], which opts for comprehensive inclusion of all information sources from an enterprise or endeavor, but with fewer guaranteed capabilities, at least initially – keyword indexing [4], or possibly just cataloging. However, over time there can be an incremental increase in capabilities (fielded search, structure-based queries, instance identification, constraint checking) or efficiency (index creation, materialization) for incremental investment – a "pay-as-you-go" approach [5], [6], [7].

In any dataspace of significant size, it is unlikely that anyone has intimate knowledge of all its sources. (In fact, there may be some sources for which no one has a good understanding any longer.) Available scheme, documentation, UIs, examples, existing applications and domain knowledge must be brought to bear to understand a source sufficiently to adapt it to new needs. Our interest is *dataspace profiling*, analogous to *database profiling*, which Marshall defines as "the process of analyzing a database to determine its structure and internal relationships" [8]. However, most database profiling tools assume a starting point of relational data with a domain-specific schema. In dataspace settings, however, the starting point may have no a priori schema, or only a generic one. An example of the former is collections of entity-property-value triples (as in RDF) resulting from shredding a semi-structured source or running an information extraction tool over a document base. An example of the latter is the Unified Medical Language System (UMLS) Metathesaurus [9], where the main content is captured in three files: one for concepts and atoms (MRCONSO), one for relationships (MRREL), and one for attributes of concepts and atoms (MRSAT). Concept and relationship types are represented as ordinary data in records in these files, rather than as domain-specific column names.

As part of dataspace profiling in such settings, we are interested in discovering or verifying latent structural characteristics and regularities. For example, we might discover that every entity with property P1 also has properties P2 and P3, or that every entity participating in a relationship of type R has property P4 equal to value V1 or V2. As a more concrete example, the UMLS Metathesaurus uniquely identifies each abstract *concept* with a *concept unique identifier* (CUI). It also identifies each *atom*, which represents an occurrence of an entry representing a concept in a source vocabulary, with an *atom unique identifier* (AUI). The MRREL file can store relationship instances connecting either concepts (via CUIs) or atoms (via AUIs). A dataspace profiling task here might be to determine if a particular segment of UMLS, say relating to medication nomenclature or drug classes, relates entities via CUIs or AUIs (or possibly both).

There are multiple uses for the knowledge gained by structural profiling of a dataspace. One is simply improved understanding of a data source, or the connection among sources, to help assess suitability or utility of particular information for a given task or application. A second is verifying conditions

needed as prerequisites to adding a certain capability over one or more data sources. For example, a common index on author over a combination of bibliographic sources would require that they all support an author property or an equivalent. A third, of high interest to us, is dataspace customization. Data sources often need to be "massaged" in various ways to be suitable for a particular use. (The UMLS Metathesaurus fact sheet notes that "the Metathesaurus *must* be customized for effective use in most specific applications" [9].) By *customization*, we intend modifications, augmentations and other enhancements of information sources to make them better suited for particular tasks and applications. Customizations include formatting, subsetting, partitioning, pivoting data to schema (and vice versa), annotation, model translation, joining and schema or constraint enforcement.

Example: Returning to UMLS, suppose we determine that for an identifiable subset of the Metathesaurus, relationship instances of type `has_dose_form` always connect a concept of type *clinical drug* with a concept of type *dose form*. Further, assume that we have verified that the `dose_form_of` relationship is the inverse of the `has_dose_form` relationship. Then we might customize this source by factoring out all of the `has_dose_form` and `dose_form_of` relationship instances into a domain-specific table `DrugInForm(clinicalDrug, doseForm)`. Such a table might be useful in checking electronic prescriptions for errors.

In this paper, we are investigating techniques for property and path analysis in a dataspace, from a starting point with limited or no explicit schema information. Paths arise naturally in property analysis when a property can be entity-valued (thus essentially acting as an instance of a binary relationship). Thus chains of such properties can be navigated. For the current discussion, we view a path type as simply a sequence of property names, for example `has_ingredient.consists_of.tradename_of`. Path characteristics of interest in profiling include:

Path presence Do instances of a particular path type T exist for every entity in a collection C?

Path typing Do instances of path type T leading from a member of entity collection C_1 always lead to an entity or value in collection C_2?

Path multiplicity How many instances of path type T originate from collection C?

Knowing these characteristics can enable a variety of customizations, such as materializing a path with direct links, use of a path value as a key for a collection, and designation of preferred paths between entity collections. We give here a detailed example of dataspace customization based on path profiling, which is supported by the InfoSonde dataspace workbench. In InfoSonde, profiling and customization options are coupled in *modules* structured as (probe, switch, check) triples. The *probe* component determines or verifies a particular characteristic of one or more data sources; the *switch* component offers valid customizations of the source based on its characteristics, and the *check* verifies that a chosen customization is still valid after an update.

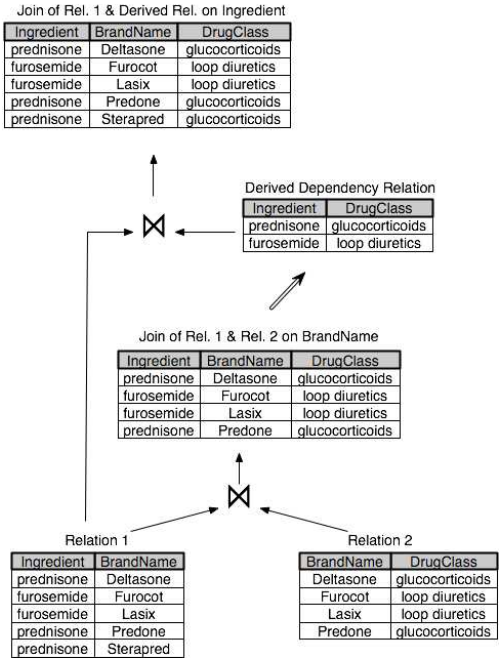

Fig. 1. Illustration of the link augmentation procedure, an example of a customization of data sources for a new purpose.

In a relational setting, path following generally manifests as a join between relations. Generally the goal of such a task is to create the maximal relation instance with the relevant attributes, linking as many values as possible of one attribute set to the corresponding values in another attribute set. One factor that can limit such linking is a relative sparseness of values in one source relation for a shared attribute. For example, in Figure 1, we are not able to connect all tuples in Relation 1 with a DrugClass in Relation 2 due to missing BrandName values in the latter.

In some cases, however, underlying regularity in a domain may permit a natural augmentation of a join limited for this reason. Such regularity can emerge as a functional dependency in the joined relation. The dependency $X \rightarrow Y$ should be between attributes X from source relation R and attributes Y in the other source relation S, where S is the relation missing join values and Y contains the attributes we are attempting to link to attributes in R. A dependency of this kind allows an alternate join path to the Y-values of interest for more values of the join attribute.

The validity of this augmentation relies on the judgement that the functional dependency observed in the joined relation instance holds because it is a true property of the domain. A single update to either source could disprove this assumption, but in doing so it would have to cause a violation of the functional dependency. Thus a module supporting this customization is well-suited to the architecture of InfoSonde, since the necessary semantic judgement would be made by a human analyst (in the probe and switch stages), with the system providing for the ongoing verification of the functional

dependency used (as the check).

Figure 1 shows how this augmentation procedure might benefit the task of connecting drug brand names with drug classes. Relation 1 also provides each drug's active ingredient. Despite the limited coverage of BrandName values in Relation 2, we observe that Ingredient → DrugClass in the join result. A relation instance derived from this dependency can then join with Relation 1 on Ingredient to give the expanded linkage shown in Figure 1.

We have applied this customization to expand the linkage between drug names given a drug nomenclature standard, RxNorm [10] with drug classes in a drug classification source, NDF-RT [11]. The resulting linkage is used to help cluster prescriptions by class in a tool, RxSafe, that provides a consolidated view of an individual's medications.

II. Quarry

Quarry is a data-property explorer we have developed, initially as a means to browse and refine harvested metadata in scientific domains at the granularity of individual data sets or derived products [5], [12]. Quarry makes no a priori assumptions about domain-specific schema. Rather, it accepts resource-property-value triples (initially provided through simple metadata-harvesting scripts created by the domain scientists). The Quarry storage engine supports efficient access to this information by automatically recognizing common property patterns for resources (called signatures) and materializing a multi-column table for each signature. Searching for resources that satisfy particular property-value conditions is generally fast, as it can usually be performed without joins, and often using a small subset of the materialized tables.

Quarry was designed to promote hands-off operation, supporting iterative improvement in metadata quality with low overhead and requiring minimal database expertise. Revisions of the triple base are easily accommodated by automatic reanalysis, reformulation and update of Quarry-storage-engine structures. Quarry provides a "table-less" API for query that only relies on properties and values, but no knowledge of how properties are grouped in internal tables. Further, the API supports inspecting the set of properties available on a collection of resources, and also the possible values of a given property over such a collection. We will describe the API in detail in Section II-C, but we will introduce two functions here to facilitate discussion. The Properties function returns the set of unique properties used to describe a restricted set of resources, and the Values function returns the set of unique values for a given property over a restricted set of resources. In both cases, the set of resources can be described by a *path expression* where each node in the path is a conjunctive query of property-value pairs.

We have used this API to build a browser that supports filtering down to relevant resources, through alternating interactions to select a property of interest, and then choose a value for that property. (There is a also the ability to show all the properties for a selected resource.)

Example: The CORIE Environmental Observation and Forecasting System integrates continuous ocean circulation models with data from an extended array of fixed and mobile observation platforms. The procedures to collect these raw data and transform them into useful *data products* create and consume a variety of intermediate datasets, images, log files, configuration files and so on. Navigating this data landscape proves difficult for developers, students, visitors, and scientists. We solicited simple harvest scripts from each developer, independently, that would derive a set of descriptive triples from the resources they are familiar with. We harvested and processed these triples using Quarry, then inspected the results through the browser. Reviewing the complete list of properties, we see `variable` (and a potential synonym, `quantity`). We select `variable` and see the values `SAL`, `sal`, and `Salinity`, among others. Each of these seems to refer to the measurement of salinity in sea water. Selecting `SAL`, we are returned to a list of properties – the union of those properties shared by all resources with `variable=SAL`. The list includes properties `station`, `plottype`, `instrumentcomparison`, `year`, and `day`, among others. The property `plottype` suggests that the resources described are all visualizations of data. The property `instrumentcomparison` suggests that these visualizations compare model results with observed sensor data. Selecting the universal distinguished property `userkey`, we are provided with a URL for each of these data products. The initial list of URLs is inconveniently long, however, so we return to the previous list (via the back button) and select additional criteria: `year=2007` and `station=`dsdmna (which we recognize as referring to the station located at Desdemona Sands). This example demonstrates the "vanilla" operation of Quarry: navigation of a large triple base to arrive at individual resources.

While Quarry was originally conceived to provide browse and query services over a set of triples, we realized early on that it had potential for dataspace profiling. Since it makes no a priori schema assumptions, it is not biased toward any particular domain. We have used it to explore sources in our medication dataspace. For example, we loaded the RxNorm [10] subset of UMLS. While there is a browser for RxNorm, called RxNav [13], it has limited support for inspecting attributes of RxNorm concepts and atoms. Quarry helped us discover useful patterns in these attributes and their values.

Example: One of the available attributes for RxNorm was `DDF`. From its name and our domain knowledge, we assumed it represented dose forms for drugs. (The `DDF` attribute is distinct from the `has_dose_form` and `dose_form_of` relationships mentioned earlier.) Selecting `DDF` and browsing the possible values confirmed this supposition. We saw values such as `Capsule`, `Tablet` and `Infusion`. However, we also noticed that the same dose form appeared in different lexical variants, such as `AEROSOL` and `aerosol`. By selecting these variants in turn, we saw that each seemed correlated with the `SAB` property, which indicates the contributing vocabu-

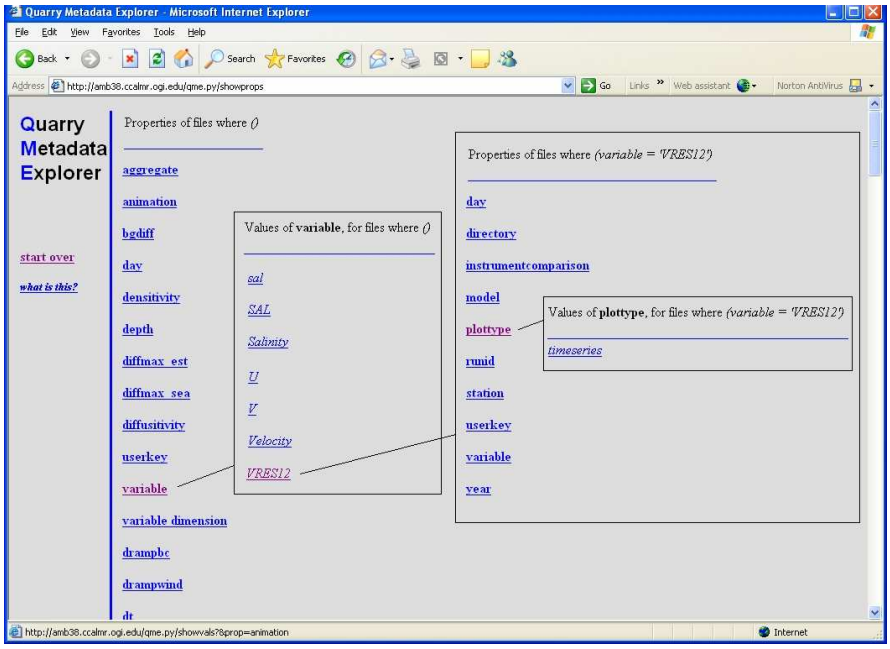

Fig. 2. The Quarry Metadata explorer interface, with insets illustrating the result of multiple user interactions. QME alternately displays the unique properties and the unique values for a selected property corresponding to a set of resources. QME naturally supports profiling activities through set-oriented browsing: Each click allows inspection of an aspect of a related set of resources.

lary to the Metathesaurus. For instance, all resources with `DDF=AEROSOL` also had `SAB=MMSL`. In contrast, for resources with `DDF=aerosol`, we always have `SAB=MTHFDA` or `SAB=VANDF`. We suspected a pattern here. Starting over from the top, and exploring the `SAB` axis first, we did indeed see pattern, which can be summarized as:

- For `SAB=MMSL`, all `DDF` values are lower case (`capsule`).
- For `SAB=MMX`, all `DDF` values have initial capitals (`Capsule`).
- For `SAB=MTHFDA`, `DDF` values are in upper case, frequently with multiple commas (`CAPSULE, DELAYED RELEASE`).
- The case for `SAB=VANDF` is similar to `SAB=MTHFDA`, but with some abbreviations and no space after commas (`CAP,ORAL`).
- Other cases, such as `SAB=SNOWMEDCT`, led to resources without the `DDF` property.

The Quarry API includes a Traverse function to navigate resource-valued properties. Given a collection of resources (specified by property-value pairs), we can select any property for traversal, which results in the collection of resources that are the value for that property for any resource in the original collection. (Traversing a non-resource-valued property just leads to the empty collection. Quarry offers this option because early in the profiling process we might not even know whether a property is resource-valued or not.) In conjunction, we have also added cardinality information to call results to support profiling. Currently, we can see the number of instances when traversing a property.

In the remainder of this section, we will describe the design and implementation of Quarry, beginning from the initial ingest of triples.

A. Signatures and Signature Extents

Once a set of (resource, property, value) triples have been presented to the Quarry backend, they are loaded into a PostgreSQL table with three string attributes: resource, property, and value. The *Signature Manager* is then invoked to identify the signature $S(r)$ of each resource r; the signature of a resource is simply the set of properties used to describe it. We then create a materialized view for each distinct signature and populate it with one tuple for each resource having that signature. For example, a particular model output file from a CORIE ocean circulation simulation is associated with seven (property, value) pairs: {`year=2003`, `day=184`, `output_step=1`, `variable=salinity`, `nodes=55817`, `sea_level=4285`, `implicitness=0.8`}. These values are deposited in a materialized view with seven attributes. This materialized view will have at least one tuple in it: (`2003, 184, 1, salinity, 55817, 4285, 0.8`).

The performance of our query processing strategy depends on the condition

$$|Signatures| << |Resources| \quad (1)$$

Although data owners are unrestricted in their choice of properties and values, we hypothesize that there is a natural convergence on fewer, more meaningful signatures; that is, that Relation (1) holds in general. Indeed, we find that our hypothesis does hold for the UMLS and CORIE datasets. For the UMLS dataset, we find 817 signatures over 74k resources

described by 10M properties. For the CORIE dataset, we find 62 signatures over 4M resources described by 28M properties.

We perform most processing in SQL to ensure that large results need not be materialized in memory. To efficiently compute signatures, we model the signature as a set-valued attribute implemented using the array features of PostgreSQL.

We considered (but rejected) an even less restrictive metadata model: keyword tags. Tagging schemes provide a "bare minimum" approach to metadata: users provide arbitrary terms (or phrases) and associate them with resources [14], [15], [16]. Such schemes were designed to encourage metadata attachment by minimizing overhead: A user need not struggle with rigid metadata standards or even consider the semantics of his or her tags. He or she simply describes resources using terms from his or her own mental model of the data. This approach works well with very large numbers of users (and hence very large numbers of mental models). In the aggregate, patterns emerge in the tags that enable querying. However, there is evidence that tags provide too much freedom. Tagging communities are beginning to adopt conventions to emulate a richer data model: the tag "geo:lat=39.234" represents a namespace ("geo"), a property ("lat"), and a value ("39.234"). Our source data tends to have more structure than, say, a vacationer's photos, so we adopt a data model of (resource, property, value) triples rather than (resource, keyword) pairs. We do not use namespaces, however, since shared namespaces are a kind of global schema that we cannot assume in the general case.

B. Quarry Metadata Explorer

Figure 2 illustrates the Quarry Metadata Explorer (QME), a canonical application for efficiently browsing a Quarry instance.

QME may be viewed as a virtual hierarchy alternating between properties and values. Each node in the hierarchy corresponds to a set of resources that match the current criteria, plus a set of properties describing those resources, or a set of values for a property used to describe the resources. The top level is an exhaustive list of all unique properties used to describe any resource from any data source. Selecting a property p returns the list of unique values used for that property for any resource from any source. Subsequently selecting a value v returns another list of unique properties, this time restricted to those properties asserted for resources satisfying the condition $p = v$. This alternation between properties and values provides an intuitive way to navigate through the triple-space, narrowing the results as conditions are appended.

Returning to the CORIE example at the beginning of Section II, consider the values of the `variable` property selected from the top-level list. The unique values of `variable` include the lexical variants of salinity, `SAL`, `sal`, and `Salinity`. The values `U` and `V` are also returned; these might represent components of a velocity vector. The value `VRES12` may not hold significance for the user (it did not for us). To investigate, we select `VRES12` and then `plottype`, finding that only one value is returned, `timeseries`. This result implies that every resource with `variable=VRES12` also has `plottype=timeseries`, implying a functional dependency. Selecting `VRES12` then `userkey` displays a partial list of file paths for resources satisfying the criterion `variable=VRES12`, along with an estimate of the total number of matching resources. The `userkey` property is distinguished: It represents a globally unique identifier such as a URI. We attempt to resolve the URI by clicking on it, finding that an image is displayed. From the axis labels and our domain knowledge, we are able to deduce that `VRES12` must indicate 12-hour residual velocities (i.e., tidal-averages).

C. Query Model

We adopt a simple API rather than a complete query language. The API consists of four functions: Describe, Properties, Values, and Traverse. Describe returns the (property, value) pairs associated with a resource. Properties returns the union of properties for all resources matching given conditions. Values returns the unique values of a given property for any resources matching given conditions. Traverse is used to find resources referenced by a property; Traverse is used to dereference relationships such as `has_dose_form`. The Values and Properties functions are evaluated by dispatching a SQL query to each signature extent whose signature subsumes the set of properties involved in the query. For example, the query

$$\mathsf{Values}(\{region = estuary, year = 2004\}, plottype)$$

returns the values of the property $plottype$ associated with resources in any table with attributes $region$ and $plottype$. The Describe function looks up the signature of the resource and then looks up the resource itself by scanning the appropriate signature extent. The Traverse function performs a join between signature extents (e.g., "find all values of `has_dose_form` that refer to resources." We will be more precise about how these functions are evaluated in Section II-D, after we present their semantics.

The semantics of the API is simplest to express in terms of a single table of triples. In the definitions below, we adopt a Haskell-like notation for expressing types, and for pattern matching on list arguments: $x : xs$ appearing as an argument means that x is bound to the head of the list and xs is bound to the tail of the list. Also, $\{\}$ and [] denote the empty set and the empty list, respectively.

To denote types, we use the convention that $\{X\}$ ($[X]$) indicates a set (list) of elements of type X, and $A \to B \to C$ indicates a curried function accepting an argument of type A, an argument of type B, and returning a value of type C. The relevant type aliases are

$$\begin{aligned} Resource, Property, Value &= String, String, String \\ Condition &= (Property, Value) \\ Node &= ([Condition], Property) \\ Path &= [Node] \end{aligned}$$

The API, plus the helper function filter, have the following types.

$\mathsf{Filter} :: \{Condition\} \rightarrow \{Resource\} \rightarrow \{Resource\}$

$\mathsf{Properties} :: Path \rightarrow \{Condition\} \rightarrow \{Property\}$

$\mathsf{Values} :: Path \rightarrow \{Condition\} \rightarrow Property \rightarrow \{Value\}$

$\mathsf{Traverse} :: Path \rightarrow \{Condition\} \rightarrow \{Resource\}$

Let Σ be a set of (resource, property, value) triples. We define a helper function $\mathsf{Filter}(C, R)$ that returns a subset of resources R that satisfy the conditions C, where C is a set of (property, value) pairs representing a conjunctive query. Then the API is defined as

$$\mathsf{Filter}(C, R) = \{r \mid (p, v) \in C,\ (r, p, v) \in R\}$$
$$\mathsf{Traverse}([], C) = Filter(C, \Sigma)$$
$$\mathsf{Traverse}((D, p) : ns, C) =$$
$$\{r \mid (s, p, r), (r, \_, \_) \in \Sigma,\ s \in Filter(C, X)\}$$
$$\text{where } X = Traverse(ns, D)$$
$$\mathsf{Properties}(P, C) =$$
$$\{p \mid (r, p, \_) \in \Sigma,\ r \in Traverse(P, C)\}$$
$$\mathsf{Values}(P, C, p) =$$
$$\{v \mid (r, p, v) \in \Sigma,\ r \in Traverse(P, C)\}$$

In Quarry, the set Σ is not materialized; rather, the resources are partitioned into signature extents and the queries are actually evaluated over those. In addition to each signature extent table, Quarry uses two auxiliary tables during query processing, $Signature(tablename, signature)$ and $Resource(resource, signature)$, where $tablename$ is a string identifier for the signature used as the name of the table holding the signature extent.

D. Query Processing

Queries are evaluated by constructing SQL statements over the signature tables. Only signature tables whose attributes (properties) are involved in the query must be accessed during query evaluation; this restriction is the key to good performance on datasets involving tens of millions of triples. In addition to a set of property attributes, each signature table contains a distinguished *userkey* attribute representing the resource itself. This attribute is the primary key for each signature table. SQL statements for each API call are constructed as follows. When the SQL is complex or platform-specific, we will write the expression as a set comprehension. For example, the UnNest function below involves processing set-valued attributes. In PostgreSQL, these attributes are implemented using array types and delimited strings.

$\mathbf{Properties}(P, C)$ = $PropQuery(Traverse(P, C))$

$\mathbf{Values}(P, C)$ = $TravQuery((C, p) : P)$

$\mathbf{Traverse}([], C)$ = $ValuQuery(\{$`'userkey'`$\}, C)$

$\mathbf{Traverse}(P, C)$ = $TravQuery((C,$ `'userkey'` $) : P)$

$\mathbf{Describe}(r)$ =

```
SELECT * FROM s WHERE userkey = r
```
where $(r, S) \in$ `Resource`, $(s, S) \in$ `Signature`

$PropQuery(R)$ = $UnNest($
```
SELECT signature FROM Resources
WHERE userkey IN R)
```
$ValuQuery(A, C)$ =
```
SELECT A
FROM SigUnion(A ∪ {p | p, v ∈ C}) WHERE C
```
$TravQuery([])$ =
```
SELECT r.userkey FROM Resources r
```
$TravQuery((C, p) : ps)$ =
```
SELECT k.p
FROM Resources r, UnNest(X)as k
WHERE r.userkey = k.p
```
where $X =$
```
SELECT p FROM ValuQuery([p], C)
WHERE userkey IN TravQuery(ps)
```
$SigUnion(A)$ =
```
SELECT A FROM s0
UNION
SELECT A FROM s1
...
SELECT A FROM sn
```
where $s_i \in \{id \mid (id, S) \in$ `Signatures`, $A \subseteq S\}$

$UnNest(X) = \{x \mid x \in Y,\ Y \in X\}$

The Properties and Values functions can both be thought of as operating on a *working set* of resources. The conditions and path arguments specify a set of resources R. Properties returns the union of all properties used to describe R, and Values returns the values of a property p used to describe some resource in R. In Section II-B, we describe the QME interface as navigating a virtual hierarchy formed from alternating calls to Properties and Values. Each node in this virtual hierarchy is therefore associated with the working set R of the API call that led there. Profiling activities involve drawing conclusions about the relationship between working sets. For example, comparing the cardinality of the working set after a traversal of `has_dose_form` with the cardinality of the working set after a subsequent traversal of `dose_form_of` could demonstrate an inverse relationship.

E. Experimental Results and Performance Enhancements

Initial processing of triples can be expensive, but is entirely automated. Once indexed, queries turn out to be quite efficient relative to existing generic triple-store methods. The YARS system for storing arbitrary RDF graphs has been shown to outperform other popular triple stores [17]. YARS uses a collection of B-Trees implemented in BerkeleyDB to efficiently answer complex RDF queries with joins. However, as with relational systems, the performance of YARS is sensitive to the choice of join order. In Figure 3, we compare the performance of Quarry and YARS for a set of randomly generated conjunctive queries over the RxNorm dataset. Note that the y-axis is logarithmic. Those queries for which response times are under the dotted line (one second) do not indicate a significant difference between Quarry and YARS; interactivity is maintained in both cases. However, when YARS selects the

wrong join order, the response time is no longer interactive and therefore unsuitable for our purposes.

There are several techniques we are considering implementing with the intended effect of improving the performance of Quarry. The first addresses the situation where there are many, similar, low-cardinality signatures. Such signatures can be merged together to reduce query response time. For example, if S_1 and S_2 are signatures with properties $ABCD$ and $ABCDE$, respectively, it may be advantageous to merge them into a single signature $ABCDE$, where the property E is padded with NULL for those resources originating from signature S_1.

A second technique we are considering addresses the issue of "promiscuous" properties, meaning properties possessed by most or all of the resources in the collection, requiring the $SigUnion$ function above to involve a potentially large number of tables. To compensate for this weakness, we can sequester each promiscuous property in its own *property table* (similar to those used in, for example, Jena [18]) that maps values to userkeys. Queries involving these promiscuous properties could use the corresponding property table as an index to prune the list of signature extents that must be searched.

Currently Quarry uses only very simple caching. When a request takes longer than a certain threshold in processing time, its result is cached so that subsequent *identical* requests will be fulfilled from the cache. Since only identical matches result in cache hits, a query that shares a path prefix with a cached query will result in a cache miss. The intermediate results of the $TravQuery$ calls could instead be cached to increase the hit rate. For example, consider a request R involving two traversals, the first from `tty=DF` traversing over `dose_form_of`, and the second with the condition `tty=SCD` over `has_dose_form`. If a previous request involved a traversal `tty=DF` over `dose_form_of` and the result was cached, that result can be used to evaluate R. To implement this improved cache, the recursive function $TravQuery$ will store intermediate results rather than construct a single complete SQL statement.

F. Extensions

We are considering several other extensions to Quarry to support additional types of profiling. We list three here.

1. **Signature sets**: Since we explicitly store the signature for each resource, it would not be difficult to provide the count or list of different signatures for the current resource collection. A very simple example is determining, say, all the signatures that include the `DRT` (drug route) property.

2. **Cardinalities**: Quarry currently reports cardinality when traversing a property. There are other places where counts would be useful, such as the number of occurrences of each property within the working set of a property list in QME, or the number of occurrence of each value in a value list. The former capability would probably be supplied "on demand," rather than automatically, as it implicitly requires exhaustively drilling down one more level on every property.

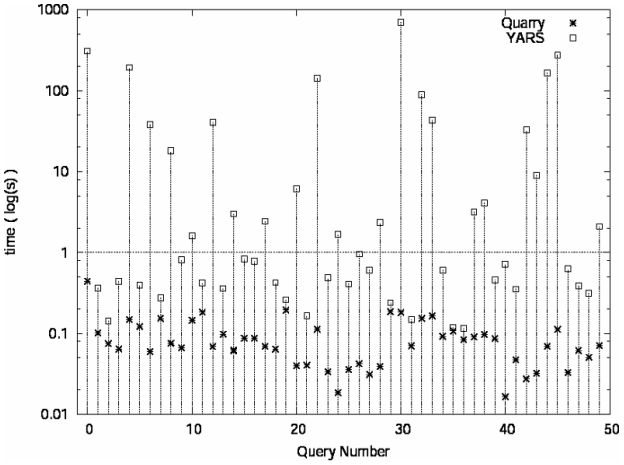

Fig. 3. Conjunctive query performance using Quarry and YARS. YARS uses a generic index scheme based on redundant B-Trees, while Quarry clusters resources by their signature. When YARS chooses the wrong join order, the system cannot respond at interactive speeds.

3. **Collection and partition comparisons**: There are cases where we would like to compare two result collections, for example, as obtained by navigating two different paths, such as `tradename_of` → `consists_of` versus `consists_of` → `tradename_of`. At a finer level, we would like to know if one partition of a resource collection (say defined by a property or path) is a refinement of another partition. For example, we might want to know if the partition induced by tradename (which corresponds to the path `tradename_of` → `str`) is a refinement of the partition induced by the `VA_CLASS_NAME` property.

III. Related Work

Database profiling [19], [8] tools and concepts aim to support analysis of data properties of the tables in a relational database: independent and joint value distributions, null counts, uniqueness tests, inclusion tests. We seek to adapt and refine these ideas for dataspace profiling, where far less structure can be assumed to exist a priori. Further, our use of path-oriented profiling extends the capabilities of table- and column-oriented profiling tools.

Our use of triples as a core data model echoes the RDF model underlying research on the Semantic Web. RDF storage and inference engines [18] adhere to the RDF syntax and semantics, which place additional restrictions on the kind of data suitable for storage. Specifically, the requirement that a reference to a resource must be encoded as a URI. We allow the user to remain agnostic about which values he or she will interpret as resource keys until runtime. That is, any set of literal values can be "tested" for use as references and keys as part of a profiling activity.

The RDF storage systems also do not in general report performance at the scale at which we operate (20-30 million triples). The YARS system [17], for example, reports better query performance than Jena [18] and Sesame [20], but requires significant space and is rarely competitive with the

Quarry system for the profiling queries we test. The SemWeb library [21] reports strong performance results on a similar scale (6.9 million triples) using a MySQL backend, but we have not yet compared performance directly.

The dataspace concept is receiving significant attention from the database community. The PAYGO architecture [22] uses clusters of related schema to fill the gap between structured query (e.g., SQL) and fully unstructured search (e.g., keyword query). The iTrails system [23] also uses a triples model to integrate heterogeneous source data, but the focus is on improving the precision and recall for path queries using user-supplied augmentations of the knowledge base rather than supporting the initial profiling activities that allow the user to write appropriate augmentations.

The theme of incremental structuralization of unstructured data was recently explored by Chu [24]. Chu et al. supply a "workbench" for interactively experimenting with structural alternatives through extraction, integration, and clustering operators. Although we share the pay-as-you-go philosophy, our targeted users are different: we aim to serve users who know nothing about logical or physical database design, and do not care to. Quarry is intended to operate in a "hands-off" manner, providing application developers and web users browse and query services over large-scale datasets without having to think about how the data is stored or managed. Further, we do not require users to write SQL to manipulate the data: We argue that a simpler API can express the majority of profiling tasks while leading directly to a simple, intuitive user interface.

IV. Future Work

This paper has presented the concept of dataspace profiling, introduced the Quarry system, and explained how Quarry might serve as a dataspace profiling tool. We see the possibility of giving at least rudimentary dataspace customization capabilities, mainly through the capability to add properties. One such extension we call "path promotion". Here, we want to "promote" the value at the end of a path from a resource to a direct property of the resource. For example, with RxNorm data, a relationship leads from concept unique identifier (CUI) to another CUI. The "name" of that concept is given by the `str` property. Thus, it might be useful to promote the path `dose_form_of` $\rightarrow$ `str` to the direct property `dose_form_name`. Other extensions we are considering include property renaming to support simple integration (e.g., rename the `DDF` property also to `dose_form_name`) and attachment of new properties as specified by a user. This latter capability could be used to name and later retrieve a complex working set. For example, for every resource r that is the `dose_form_of` a resource with the `TTY`=DF relationship, add the triple (r, `kind`, `dose_form`). Afterward, a user could navigate directly to these resources by clicking on the new `kind` property rather than reconstructing their criteria.

V. Acknowledgements

We would like to thank Antonio Baptista and Paul Turner at the NSF Center for Coastal Margin Observation and Prediction. This work supported by NSF ACI 0121475, OCE 0424602, IIS 0534762; a DARPA Information Integration Seedling Project; and a Maseeh Professorship stipend from Portland State University.

References

[1] "Report on the workshop on information integration," Philadelphia, PA, October, 2006. Draft of April 5, 2007. Available at http://db.cis.upenn.edu/iiworkshop/postworkshop/index.htm.

[2] M. J. Franklin, A. Y. Halevy, and D. Maier, "From databases to dataspaces: A new abstraction for information management," *SIGMOD Record*, vol. 34, no. 4, December 2005.

[3] A. Halevy, M. Franklin, and D. Maier, "Principles of dataspace systems," in *PODS*. New York, NY, USA: ACM Press, 2006, pp. 1–9.

[4] X. Dong and A. Halevy, "Indexing dataspaces," in *SIGMOD*. New York, NY, USA: ACM Press, 2007, pp. 43–54. [Online]. Available: http://portal.acm.org/citation.cfm?id=1247480.1247487

[5] B. Howe, D. Maier, and L. Bright, "Smoothing the ROI curve for scientific data management applications," in *CIDR*, 2007.

[6] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu.

[7] D. Maier and N. B. Rayner, "Pay-as-You-Go information integration," Position paper, Workshop on Integration Information, Philadelphia, PA, October 2006.

[8] B. Marshall, "Data quality and data profiling: a glossary of terms," http://www.telusplanet.net/public/bmarshal/dataqual.htm, viewed June 19, 2007.

[9] National Library of Medicine, "UMLS Metathesaurus fact sheet," http://www.nlm.nih.gov/pubs/factsheets/umlsmeta.html, Tech. Rep., March 2006.

[10] S. Liu, W. Ma, R. Moore, V. Ganesan, and S. Nelson, "RxNorm: Prescription for electronic drug information exchange," *IEEE IT Professional*, vol. 7, no. 5, September 2005.

[11] S. H. Brown, P. L. Elkin, S. T. Rosenbloom, C. Husser, B. A. Bauer, M. J. Lincoln, J. Carter, M. Erlbaum, and M. S. Tuttle, "VA national drug file reference terminology: A cross-institutional content coverage study." *Medinfo*, vol. 11, no. Pt. 1.

[12] B. Howe, K. Tanna, P. Turner, and D. Maier, "Emergent semantics: Towards self-organizing scienitifc metadata," in *Proceedings of Semantics for a Networked World: Semantics for Grid Databases*, ser. Lecture Notes in Computer Science, vol. 3226. Springer, June 2004.

[13] K. Zeng, O. Bodenreider, J. Kilbourne, and S. J. Nelson, "Rxnav: A web service for standard drug information," in *Proc. American Medical Informatics Association, [AMIA] Annual Symposium*, Washington DC, November 2006.

[14] Flickr, Inc., http://www.flickr.com/.

[15] Tagcloud, Inc., http://www.tagcloud.com/.

[16] Google, Inc., "GoogleBase," http://base.google.com/.

[17] A. Harth and S. Decker, "Optimized index structures for querying rdf from the web," in *LA-WEB*. Washington, DC, USA: IEEE Computer Society, 2005, p. 71.

[18] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations," in *WWW Alt.* New York, NY, USA: ACM, 2004, pp. 74–83.

[19] Intelligent Search Technology, Ltd., http://www.telusplanet.net/public/bmarshal/dataqual.htm, viewed November 27, 2007.

[20] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: An architecture for storing and querying rdf data and schema information," Semantics for the WWW. MIT Press, 2001. [Online]. Available: citeseer.ist.psu.edu/broekstra01sesame.html

[21] J. Tauberer, http://razor.occams.info/code/semweb/, viewed November 27, 2007.

[22] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu, "Web-scale data integration: You can afford to pay as you go," in *CIDR*, 2007, pp. 342–350.

[23] M. A. V. Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi, "itrails: Pay-as-you-go information integration in dataspaces," in *VLDB*, 2007, pp. 663–674.

[24] E. Chu, A. Baid, T. Chen, A. Doan, and J. F. Naughton, "A relational approach to incrementally extracting and querying structure in unstructured data," in *VLDB*, 2007, pp. 1045–1056.