

# Information Extraction and Integration from Heterogeneous, Distributed, Autonomous Information Sources – A Federated Ontology-Driven Query-Centric Approach

Jaime A Reinoso Castillo, Adrian Silvescu, Doina Caragea, Jyotishman Pathak, Vasant G Honavar  
 Artificial Intelligence Research Laboratory, Department of Computer Science  
 Iowa State University, Ames, Iowa, 50011

**Abstract-** This paper motivates and describes the data integration component of INDUS (Intelligent Data Understanding System) environment for data-driven information extraction and integration from heterogeneous, distributed, autonomous information sources. The design of INDUS is motivated by the requirements of applications such as scientific discovery, in which it is desirable for users to be able to access, flexibly interpret, and analyze data from diverse sources from different perspectives in different contexts. INDUS implements a federated, query-centric approach to data integration using user-specified ontologies.

**Keywords-** Data Integration, Federated Database, Ontology

## I. INTRODUCTION

Development of high throughput data acquisition in a number of domains (e.g. biological sciences, space sciences, commerce) along with advances in digital storage, computing, and communication technologies have resulted in unprecedented opportunities in data-driven knowledge acquisition and decision making. The effective use of increasing amounts of data from disparate information sources presents several challenges in practice [1]:

- a) Data repositories are large in size, dynamic, and physically distributed. Consequently, it is neither desirable nor feasible to gather all of the data in a centralized location for analysis. Hence, there is a need for algorithms that can efficiently extract the relevant information from disparate sources on demand.
- b) Data sources of interest are autonomously owned and operated. Consequently, the range of operations that can be performed on the data source (e.g., the types of queries allowed), and the precise mode of allowed interactions can be quite diverse. Hence, strategies for obtaining the necessary information (e.g., statistics needed by data mining algorithms) within the operational constraints imposed by the data source are needed.
- c) Data sources are heterogeneous in structure (e.g., relational databases, flat files) and content. Each data source implicitly or explicitly uses its own ontology (concepts, attributes and relations among attributes) [22] to represent data. For example, domain-specific ontologies are being developed in many areas (e.g., the gene ontology project, <http://www.geneontology.org>, is aimed at development of ontologies and their XML encodings for use in Bioinformatics). Thus, effective integration of information from different sources

bridging the syntactic and semantic mismatches among the data sources is needed.

- d) In many applications (e.g., scientific discovery), because users often need to examine data in *different contexts from different perspectives*, there is no single universal ontology [22] that can serve all users, or for that matter, even a single user, in every context. Hence, methods for context-dependent dynamic information extraction and integration from distributed data based on user-specified ontologies are needed to support knowledge acquisition and decision making from heterogeneous distributed data.

This paper describes the data integration component of INDUS (Intelligent Data Understanding System) – a modular, extensible, platform independent environment for information integration and data-driven knowledge acquisition from heterogeneous, distributed, autonomous information sources. INDUS when equipped with machine learning algorithms for ontology-guided knowledge acquisition can accelerate the pace of discovery in emerging data-rich domains (e.g., biological sciences, atmospheric sciences, economics, defense, social sciences) by enabling scientists and decision makers to rapidly and flexibly explore and analyze vast amounts of data from disparate sources.

The rest of the paper is organized as follows: Section II briefly introduces the data integration problem and describes the considerations that had an impact on the choice of the overall approach to data integration in INDUS. Section III provides more precise definitions of the relevant terminology and operations on data. Section IV describes the implementation of the data integration component of INDUS. We conclude in Section V with a summary and related work.

## II. DATA INTEGRATION SYSTEMS

Data Integration systems [2,5,10] attempt to provide users with seamless and flexible access to information from multiple autonomous, distributed and heterogeneous data sources through a unified query interface. Ideally, a data integration system should allow users to specify *what* information is needed without having to provide detailed instructions on *how* or from where to obtain the information. Thus, in general, a data integration system must provide mechanisms for the following:

- a) Communications and interaction with each data source as needed.
- b) Specification of a query, expressed in terms of a user-specified vocabulary (ontology), across multiple heterogeneous and autonomous data sources.
- c) Specification of mappings between user ontology and the data-source specific ontologies.
- d) Transformation of a query into a plan for extracting the needed information by interacting with the relevant data sources.
- e) Integration and presentation of the results in terms of a vocabulary known to the user.

There are two broad classes of approaches to data integration: *Data Warehousing* and *Database Federation* [4].

In the data warehousing approach, data from heterogeneous distributed information sources is gathered, mapped to a common structure and stored in a central location. In order to ensure that the information in the warehouse reflects the current contents of the individual sources, it is necessary to periodically update the warehouse. In the case of large information repositories, this is not feasible unless the individual information sources support mechanisms for detecting and retrieving changes in their contents. This is often an unreasonable expectation in the case of autonomous information sources. The warehousing approach to data integration has another important drawback in the case of applications such as scientific discovery in which users often need to analyze the same data from multiple points of view: The data warehouse relies on a single common ontology for all users of the system. This ontology is typically specified as part of the design of the data warehouse. Each user queries the warehouse using a common vocabulary and a common query interface.

In the case of Database Federation, information needed to answer a query is gathered directly from the data sources in response to the posted query. Hence, the results are up-to-date with respect to the contents of the data sources at the time the query is posted. More importantly, the database federation approach lends itself to be more readily adapted to applications that require users to be able to impose their own ontologies on data from distributed autonomous information sources. Because our focus is on data integration for scientific application, which requires users to be able to flexibly integrate data from multiple autonomous sources, we adopt the *Database Federation* approach to information integration.

Typically, a query posted by the user must be decomposed into a set of operations corresponding to the information that needs to be gathered from each data source and the form in which this information must be returned to the

system. To accomplish this, data integration systems must support two basic set of operations:

- *get()* to query the information sources; and
- *transform()* for mapping the results in the desired form.

These operations should be capable of dealing with syntactic and semantic mismatches between the vocabulary (names of entities and relationships) of the user (user ontology or global ontology) to query for information and the vocabulary understood by each information source (source-specific ontologies or local ontologies).

There are two basic approaches for dealing with semantic mismatches between global ontology and local ontologies: *Source-Centric Approach* and the *Query-Centric Approach* [5]. In the case of the source-centric approach, each individual data source determines how the concepts in a local (source-specific) ontology are mapped to concepts in the global ontology. Thus, the user has little control on the true *meaning* of concepts in the global ontology (and hence the results of a query). In other words, the semantics are *source-centric*. This frees the user or the administrator of the integration system from the task of specifying the transformations between global concepts and local concepts – these transformations are specified by the local source(s). In contrast, in the query-centric approach to information integration, concepts in the global ontology are defined in terms of concepts in local ontology (source-specific ontologies). Thus, the query-centric approach is better suited for data integration in applications in which the users need the ability to impose the ontologies (and semantics) of their choice to flexibly interpret and analyze information from autonomous sources. But this requires the user or administrator of the integration system to specify precisely how global concepts can be composed from local concepts.

Consider the case of a scientist who has defined a set of queries  $Q$  to obtain information about a set of proteins. Assume that a new data source  $A$  becomes available. In this case, the scientist may want to decide whether (and how) to utilize the new data source in answering queries in  $Q$ . This decision may be based on the way proteins have been classified by the data source and what such classification means to the scientist relative to concepts in the global (user) ontology. The source-centric approach puts the information sources in control of the semantics. In contrast, the query-centric approach puts the user in control of semantics. Hence, we adopt the Query-Centric approach to data integration in INDUS.

Thus, INDUS offers a federated, query-centric approach to information integration from heterogeneous, distributed, autonomous information sources. The overall architecture of INDUS is shown in Figure 1.

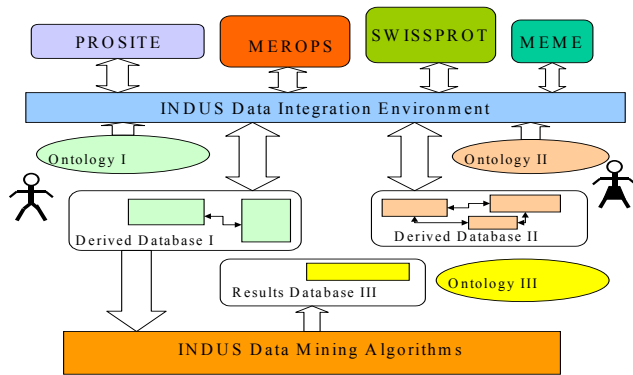


Fig. 1 INDUS (Intelligent Data Understanding System) for Information Extraction, Integration, and Knowledge Acquisition from Heterogeneous, Distributed, Autonomous Information Sources. PROSITE, MEROPS, SWISSPROT, and MEME are examples of data sources used by Computational Biologists.

INDUS enables a scientist to view a collection of physically distributed, autonomous, heterogeneous data sources (regardless of their location, internal structure, and query interfaces) as *though* they were relational databases, (i.e. a collection of inter-related tables), structured according to an ontology supplied by the scientist. The current prototype of INDUS makes explicit, the implicit ontologies associated with each of the data sources. Each data source is viewed as a repository of instances of concepts associated with the data source. Each concept is simply a collection of instances or records in a relational database i.e. a set of tables, and a set of associations between pair of tables. The input from a typical user (scientist) includes: an ontology that links the various data sources from the user’s point of view, executable code that performs specific computations (if they are not supported directly from data sources), and a query expressed in terms of user-specified ontology. This allows the scientist to extract and combine data from multiple data sources and store the results in a relational database which is structured according to his or her own ontology and can be manipulated using application programs or relational database operations.

### III. DATA INTEGRATION IN INDUS

As noted above, INDUS implements a federated, query-centric approach to information extraction and integration from heterogeneous, distributed and autonomous data sources. The system uses a three-layer architecture consisting of the physical layer, the ontological layer, and the user-interface layer.

The physical layer allows the system to communicate with the information sources. It is based on *federated database architecture*. The ontological layer contains global ontology (or ontologies) specified by users and their mappings to local ontologies associated with the information sources. It automatically transforms queries expressed in terms of concepts in a global ontology into execution plans. The plans describe what information to

extract from each data source and how to combine the results. Finally, the user interface layer enables users to interact with the system, define ontologies, post queries and receive answers. All the complexity associated with the process of gathering the information is hidden from the final user. In what follows, we formally introduce the notions of a concept, ontology and query as we use them in INDUS.

#### A. Concepts

A “concept” in INDUS is equivalent to the mathematical entity for a relation underlying the relational model. Thus, a concept is a subset of the Cartesian product of a list of domains, i.e., if  $D_1, \dots, D_n$  is a list of domains, then  $X \subseteq D_1 \times \dots \times D_n$  is a concept. Here a domain is a set of values. Each domain is assumed to be finite, but is typically unknown a priori.

If  $X$  is a concept, the *structure of a concept*,  $X_{\text{atts}}$ , is described by the list of domains (called also attributes). The elements of this list are drawn from  $\Theta$ , the set of all domains. The  $i$ -th element of  $X_{\text{atts}}$  is represented by  $X_{\text{atts}[i]}$ ; its name by  $X_{\text{atts}[i].\text{name}}$  and the associated domain by  $X_{\text{atts}[i].\text{domain}}$ . For example,  $X_{\text{atts}} = ((\text{“name”}, \mathcal{S}), (\text{“age”}, \mathcal{N}))$ , where  $\mathcal{S}$  represents the set of strings and  $\mathcal{N}$  represents the set of natural numbers, is the structure of a concept with two attributes.

The *extensional definition* of a concept  $X$ , denoted by  $X_{\text{insts}}$ , is the enumeration of all instances of concept  $X$ . Each instance is represented by a list of attribute values. For example, a concept  $X$ , based on  $\mathcal{D}_1 \times \mathcal{D}_2 \times \mathcal{D}_3 = \{(a,b,d), (a,b,e), (b,b,d), (b,b,e)\}$ , may be extensionally defined as  $X_{\text{insts}} = \{(a,b,d), (a,b,e)\}$ .

The *intentional definition* of a concept  $X$ ,  $X_{\text{I}}$ , consists of a description of instances that belongs to that concept. Thus, an intentional definition for the concept  $X$  used before may be:  $X_{\text{I}} = \{i \in \mathcal{D}_1 \times \mathcal{D}_2 \times \mathcal{D}_3 \mid \text{the first element of the } i \text{ tuple is an ‘a’}\}$ . In general, intentional definitions offer a shorter representation than extensional definitions.

An *operational definition* of a concept  $X$ ,  $X_{\text{D}}$ , is a procedure that specifies how to obtain the set of instances of  $X$ .

We use relational databases to store instances of a concept. Therefore, the set of instances that belong to a concept (relation) are rows of the corresponding table. Two types of concepts are defined in INDUS: *Ground Concepts* and *Compound Concepts*.

The *ground concepts* are those whose instances can be retrieved from one or more data sources using a set of pre-defined operations. The operational definition of a ground concept describes a procedure for retrieving  $X_{\text{inst}}$  from a set of relevant data sources. In order to accomplish this task,

INDUS provides an extensible set of components called *instantiators* that are able to interact with data sources and retrieve a set of instances for a particular concept. They encapsulate the interaction of the system with the data sources through a common uniform interface. This common interface allows the system to invoke *instantiators* and also to receive instances from the data sources.

The core of an instantiator is an *iterator*. An iterator is completely specified in terms of the name of the program to be executed and the parameters that control the behavior of the iterator. Let  $\mathbf{T}$  be the set of iterators provided by INDUS. If  $\tau \in \mathbf{T}$ , then  $\tau.name$  corresponds to the name of the program to be executed and  $\tau.param$  specifies the list of parameters associated with  $\tau$ . The  $i$ -th parameter is represented by  $\tau.param[i]$ .

In order to fully specify an *instantiator*, the following information must be provided:

- An iterator,
- An assignment of values for the iterators parameters,
- A mapping indicating how to create an instance of a particular concept  $X$  based on the information returned by the iterator, and
- The query capabilities offered by the data sources when it is accessed through this instantiator.

Formally, if  $X$  is a ground concept, its *operational definition* ( $X.D$ ) corresponds to the set of *instantiators* that can be used to retrieve instances of  $X$ . If  $\iota \in X.D$ ,

- The concept associated with the instantiator must be  $X$  ( $\iota.concept \equiv X$ ).
- $\iota.iterator \in \mathbf{T}$ .
- $\iota.values$  is the list of values assigned to the parameters of  $\iota.iterator$ . Note that  $|\iota.iterator.param|$  must be equal to  $|\iota.values|$ , and the value assigned to the  $i^{th}$ -parameter,  $\iota.iterator.param[i]$ , corresponds to  $\iota.values[i]$ .
- $\iota.mapping$  specifies how to build an instance of  $X$  based on the values returned by the instantiator. Therefore,  $\iota.mapping$  is a list where each element specifies which attribute returned by the instantiator must be assigned to which attribute of  $X$ . Thus, any information returned by the instantiator which is marked as  $\iota.mapping[i]$  must be assigned to the corresponding  $X.att[s[i]]$ .
- $\iota.queryCapabilities$  is the list of conditions associated with the instantiator  $\iota$  where the  $i^{th}$ -element of the list is  $\iota.queryCapabilities[i]$ . If  $b \in \iota.queryCapabilities$ , then  $b.attribute \in X.att$  and  $b.operator \in \mathbf{O}$ , where  $\mathbf{O}$  is the set of operators supplied by INDUS.

The definition of a *compound concept*  $X$  specifies the set of operations that must be applied over a set of instances of other previously defined concepts in order to determine the set of instances of  $X$ . INDUS uses four operations for operationally defining new compound concepts based on the existing concepts: selection, projection, vertical integration and horizontal integration. We say that two concepts  $X$  and  $Y$  have *equivalent structure* if:

- $|X.atts| = |Y.atts|$ , and
- $\forall i \leq i \leq |X.atts|$  the  $i^{th}$ -element of both lists has the same domain or there is a natural transformation between the corresponding domains of  $X$  and  $Y$ .

*Selection:* Given two concepts  $X$  and  $Y$  that have equivalent structure, the operational definition of  $Y$  in terms of some selection operation on  $X$  can be described as  $Y.D := \sigma_s (X)$ , where  $s$  is a conjunction of built-in predicates. A built-in predicate is of the form (*argument operator argument*), where *argument* follows the format  $function_1(attribute_1, attribute_2, \dots)$ . Thus, the previous definition implies that the set of instances of  $Y$  is the set of instances of  $X$  that satisfy the condition expressed by  $s$ .

*Projection:* Let  $X$  and  $Y$  be two concepts and let  $p$  be a list with equivalent structure with  $Y$ . The operational definition of  $Y$  in terms of a projection over the concept  $X$  can be expressed as:  $Y.D := \pi_p (X)$ , where  $p$  is a list of functions applied over  $X$  attributes. Thus, when  $Y$  is instantiated, the set of instances of  $Y$  is the set of instances of  $X$  after applying the functions specified in  $p$  to the attributes in  $X$ .

*Vertical Integration:* Vertical fragmentation occurs when the instances of a concept are fragmented across two or more data sources. Thus, each data source stores values of a subset of attributes of the concept. We assume the existence of a special attribute (corresponding to a unique key or index) that is stored at each source so that the corresponding fragments of each instance can be combined. Vertical integration of two concepts  $A$  and  $B$  into a new concept  $AB$  involves combining each instance of  $A$  with the corresponding instance of  $B$  followed by selection and projection operations (as needed). Let  $X$ ,  $Y$  and  $Z$  be three concepts such that  $|X.atts| = |Y.atts| \cdot |Z.atts|$ , where “ $\cdot$ ” represents the list concatenation operation. The operational definition of  $X$  in terms of the Cartesian product of  $Y$  and  $Z$  can be written as  $X.D := Y \times Z$ . Thus, the operational definition of  $X$  is described in terms of the vertical integration of  $Y$  and  $Z$  as follows:  $X.D := \pi_p (\sigma_s (Y \times Z))$ . In general, a concept  $X$  may include  $n$  concepts  $Y_1, Y_2, \dots, Y_n$  in its operational definition using a vertical integration operation, as follows:  $X.D := \pi_p (\sigma_s (Y_1 \times Y_2 \times \dots \times Y_n))$ .

*Horizontal Integration:* In the case of horizontal fragmentation, instances of a concept  $X$  are distributed across several information sources. If  $X$ ,  $Y$ , and  $Z$  are three concepts with equivalent structure  $X.D$ , the operational definition of  $X$ , is defined in terms of the concepts  $Y$  and  $Z$  using a horizontal integration operation, as follows:  $X.D := Y \cup Z$ . Thus, the set of instances of  $X$  is obtained by taking the union of the instances of  $Y$  and instances of  $Z$ . A more general definition of a horizontal integration operation may include a selection, a projection and the union of more than two concepts. Thus, if  $X$  is a concept, its operational definition can be based on a horizontal integration operation as follows:  $X.D := \pi_p (\sigma_s (Y_1 \cup Y_2 \cup \dots \cup Y_n))$ . We assume the union operation is applied over bags (not over

sets) in our current prototype. This means that duplicated instances are not eliminated.

Note that the selection and projection operations are special cases of the vertical integration or the horizontal integration. Therefore, the general form of the operational definition of a *compound concept* is of the form:  $X_{.D} := \pi_p(\sigma_s(Y_1 \bullet Y_2 \bullet \dots \bullet Y_n))$ , where  $X$  stands for a compound concept,  $s$  is the selection criteria,  $p$  is the projection criteria, and each  $Y_i$  makes reference to a predefined concept and  $\bullet$  is one of the compositional operations defined in INDUS (e.g.,  $\cup$  or  $\times$  in our current implementation).

## B. Global Ontology

In general, an *Ontology* specifies terms and relationships among terms [22]. In INDUS, a global ontology consists of the set of concepts that are used to describe entities and relationships in the domain of discourse (e.g., molecular biology). In principle, the global ontology can be tailored to suit the needs of each user or each group of users that share a common vocabulary. *Queries* are expressed in terms of concepts in the global ontology. The global ontology can be extended by defining new concepts in terms of existing concepts using a well-defined set of compositional operations. From a user's perspective, the global ontology used in INDUS hides the complexity of accessing and retrieving the information from the data sources. Semantics of user-defined concepts in the global ontology are mapped to the semantics associated with the ground concepts associated with the individual information sources (i.e., concepts in the respective local ontologies) using compositional operations and/or predefined or user-supplied functions.

This mapping process allows the user to resolve semantic mismatches among the different sources. Examples of semantic mismatches include: the use of the same term to describe two semantically different concepts, or when two different concepts are denoted by the same term. INDUS also supports transformations between values of attributes that are used to define concepts. Such transformations can be used to map values of attributes associated with instances retrieved from different sources so that they are expressed in terms of a common unit (e.g., to transform temperature values from degrees Fahrenheit to degrees Celsius) or to compute values of an attribute associated with instances of a compound concept in terms of values of one or more attributes in the corresponding instances of its component concepts.

The ontological layer includes the global ontology, the data-source specific ontologies (i.e., the corresponding set of ground concepts) and the information needed for obtaining instances of a ground concept from a corresponding data source. Figure 2 shows the definition of a simplified ontology for the PROSITE database, composed of two basic concepts: Family and Motif. Each concept is

described in terms of the corresponding attributes (which define the structure of the concept).

## C. Queries

In INDUS, a query over a concept  $X$  allows users to obtain instances of  $X$ . Formally, if  $Q$  is a query over a concept  $X$ , it is specified by a projection and selection operation over instances of  $X$  as follows:  $Q := \pi_p(\sigma_s(X))$ .

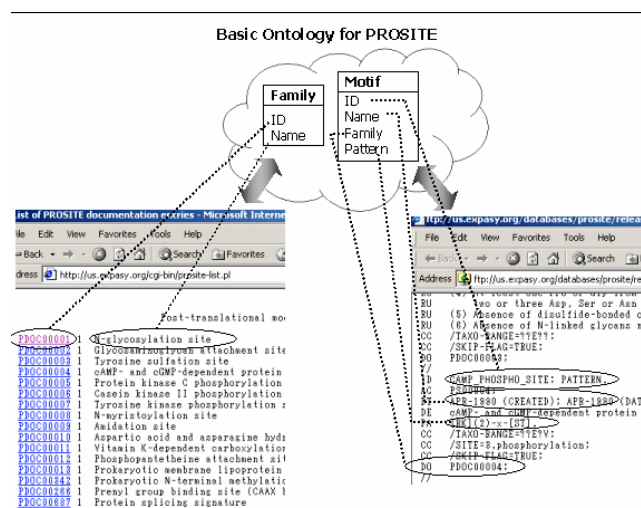


Fig. 2 Basic Ontology for PROSITE

INDUS provides a query-centric algorithm that takes as input a query  $Q$  and returns the set of instances that satisfy  $Q$ . We illustrate the query-centric approach through an example. Assume a global concept corresponding to a table called PROTEIN with columns ID, NAME, TYPE. Assume that there are two data sources A and B that contain information about proteins. In a query-centric approach, the PROTEIN concept is described in terms of ground concepts associated with the sources A and B as follows:

```

Create Or Replace View PROTEIN (Id, Name, Type) As
Select Id, Name, Type From A.Protein
Union
Select Id, Name, Type From B.Protein;

```

If a new data source C contains only Enzyme proteins, a subset of the general concept, it can be added to the definition as follows:

```

Create Or Replace View PROTEIN (Id, Name, Type) As
Select Id, Name, Type From A.Protein
Union
Select Id, Name, Type From B.Protein
Union
Select Id, Name, 'Enzyme' From C.Protein;

```

Finally, if a new data source D stores different chemical compounds (including proteins) in a table, it can be added to the definition as follows:

```

Create Or Replace View PROTEIN (Id, Name, Type) As
Select Id, Name, Type From A.Protein
Union
Select Id, Name, Type From B.Protein
Union
Select Id, Name, 'Enzyme' From C.Protein

```

```

Union
Select Id, Name, Type From D.chemical_compound
Where D.kind = 'Protein';

```

The procedure for answering a query  $Q$  finds an equivalent rewriting  $Q'$  of  $Q$  expressed in terms of ground concepts and compositional operations (vertical integration and horizontal integration in our current prototype).  $Q'$  is represented by an *expression tree*, where each internal node corresponds to a well-defined operation (union, projection etc.) and each leaf node corresponds to a ground concept.

Any compound concept that appears in the query is recursively replaced by its definition to arrive at a plan (an expression tree) in which only the ground concepts appear as leaves. Therefore, each non leaf represents an operational definition described by  $\pi_p(\sigma_s(\text{node}_1 \bullet \text{node}_2 \bullet \dots \bullet \text{node}_n))$ , where  $\text{node}_i$  represents a descendant node of  $\text{node}_1$ , for  $i > 1$ . Here,  $\bullet$  corresponds to a cross product operations ( $\times$ ) if vertical integration operation is required. Similarly, a union operation ( $\cup$ ) is used if a horizontal integration is needed. In the case of a leaf node  $\text{node}_i$ , the associated operational definition of  $\text{node}_i$  corresponds to the formula  $\pi_p(\sigma_s(X))$  where  $X$  is a ground concept. The algorithm for finding the expression tree of a query  $Q$  is presented Figure 3.

```

ConstructTree(Q){
  Create the tree  $T$ ;

  Create a node1 as a root of  $T$  based on the operational definition of  $Q$ .

  Create a node2 as a descendant of node1 based on the operational definition of the concept in  $Q$ .

  Transform the selection and projection lists associated with node2 such that only conditions involving the identity function are maintained in the selection list. Similarly, only those attributes required in node1 are maintained in the projection list.

  ExpandTree(node2);
}

```

Fig 3. The *ConstructTree* algorithm

```

executeTree(node){

  for each child of node{
    executeTree(child);
  }
  if node is a ground concept{

    node.createTable();
    node.chooseIterator();
    node.populateTable();
  }
  else{
    node.createView();
  }
}

```

Fig. 4 Pseudo-algorithm for Executing the Expression Tree.

After the plan is created, the next step is execution. The instances that correspond to the extensional definition of each ground concept associated with an information source are extracted from the respective source. The instances corresponding to each internal node of the plan (execution tree) are constructed by appropriately combining the set of instances returned by its descendents. The process terminates when the set of instances for the root of the tree is obtained. The algorithm for executing an expression tree is shown in Figure 4. For example, consider a query posted over the PROTEIN concept as follows (Figure 5):

```

Select Id, Name From PROTEIN Where type= 'Enzyme'

```

The first step is to create an expression tree for the query based on the definition of the PROTEIN concept. The definition of the PROTEIN concept is recursively rewritten until it is expressed in terms of ground concepts associated with the available data sources. The resulting plan (execution tree) is traversed in post-order fashion. The leaf nodes corresponding to ground concepts are 'executed' by invoking the corresponding instantiators. One of the common optimization approaches we adopt is to push down the selection and projection operations as close to the leaves of the tree as possible.

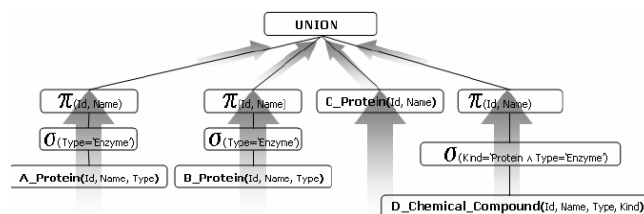


Fig. 5 Execution order of the expression tree

#### IV. IMPLEMENTATION OF INDUS

This section describes the implementation of the data integration component of INDUS. This component of INDUS consists of five principal modules as shown in Figure 6: graphical user interface, common global ontology area, instantiator library, query resolution module, and private user workspace.

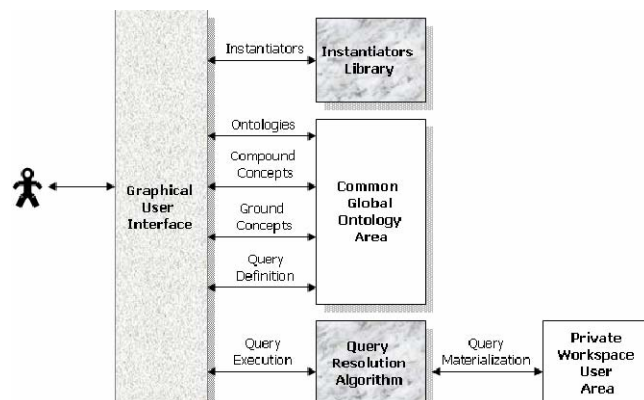


Fig. 6 INDUS modules

The *graphical user interface* allows the users to interact with the INDUS. It enables users to describe ontologies, define operational definitions of ground concepts; compound concepts and queries, register the iterators, and execute queries.

The *common global ontology area* manages the repository where definitions of ontologies, ground concepts, compound concepts, queries, and iterator signatures are stored.

The *instantiator library* contains the set of functions used to interact with the individual data sources. Each instantiator is based on a particular iterator. The latter interacts directly with a data source. Each iterator is implemented by a Java class. An instantiator supplies the parameters that control the behavior of the corresponding iterator. It also maps the instances returned by the iterator into instances of the corresponding ground concept. Thus, the functionality of an instantiator (together with the corresponding iterator) corresponds roughly to that of a *wrapper*. However, unlike wrapper-based data integration systems that often have ontologies and the required semantic mappings hardwired into the wrappers, INDUS enforces a clear separation of functionality between ontologies defined by the users and the instantiators that call the relevant iterators to retrieve instances associated with ground concepts.

The *query resolution* module accepts a query expressed in terms of concepts in the global ontology as input and returns the answer to the query constructed from the relevant data sources.

Finally, the *user workspace* module allows INDUS to manage the private workspace where users store answers for posted queries. The required partial results are also stored in this area. In particular, the set of instances associated with each ground concept present in the expression tree associated with the query are stored as populated relational tables. Furthermore, each internal node of the expression tree, which represents a set of compositional operations described in the operational definition of a particular compound concept, is materialized as a relational view defined in terms of tables or views previously created and materialized by the query resolution algorithm.

The modular design of INDUS ensures that each module can be updated and alternative implementation easily explored. Modularization enables INDUS to use different network architectures. For example, INDUS may be implemented in a centralized architecture, locating all the modules in a single server, or in an architecture where modules are distributed across several servers. Thus, an *application server* may support the operation of the graphical user interface, the Iterators library and the query resolution modules sharing the same Java virtual machine. A *repository server* may support the operation of the common global ontology area and the user workspace modules sharing the same relational database system. For

the current prototype of INDUS, we used JSP (Java Server Pages) for developing the graphical user interface. It is hosted in an Apache Tomcat 4.0 web server. The common global ontology area was developed under a relational database in order to provide a robust environment to store and manipulate ontologies of large size. Relational database technology also offers us an efficient way to support multiple concurrent users. Standard protocols such as ODBC and JDBC can be used to share the ontology with other applications as needed. The user workspace resides in a relational database. This enables users to manipulate the results of queries using relational database operations. Although the workspaces are private for each user, using mechanisms for granting privileges provided by the relational database management system (RDBMS) enables users to share their results with others if they so desire. The Iterators and the resolution algorithm are implemented in Java. Thus, all essential components of INDUS are platform independent.

At least four different roles may be played by a user when interacting with INDUS. As a *domain scientist*, a user may define ontologies, compound concepts and queries. Also, a user may execute queries and manipulate the retrieved data. In this role, the user is expected to have knowledge of the relevant domain, some familiarity with the data sources and their capabilities, but no deep knowledge of programming. As an *ontology engineer*, a user may expand the iterator library by programming new iterators, define the ground concepts associated with new data sources, or new modes of interaction with existing data sources. As an *administrator*, a user is able to install the INDUS software, including the graphical user interface and the query resolution module, and set up and manage the databases supporting the common global ontology module and the user workspace, which includes adding new users to the system. As a *developer*, a user may add new compositional operations to INDUS, modifying the graphical user interface and the query resolution module. In practice, a given user may play multiple roles.

Setting up the INDUS data integration environment involves installation of INDUS and a relational database system that is to be used by INDUS for the ontologies and user workspace. Incorporation of new data sources into INDUS involves registering the data sources, defining the relevant ground concepts and implementing the necessary instantiators and iterators. Using INDUS to extract and integrate data from multiple sources involves defining the relevant compound concepts and functions and formulating and executing queries expressed in terms of concepts in the global ontology.

## V. SUMMARY AND DISCUSSION

### A. Summary

In this paper, we have described the design and implementation of the data integration component of INDUS (Intelligent Data Understanding System)

environment for flexible information extraction and information integration and knowledge acquisition from heterogeneous, distributed, autonomous information sources. INDUS implements a federated, query-centric approach to data integration. Hence, the information extraction operations to be executed are dynamically determined on the basis of the user-supplied ontology and the query supplied by the user or an application program (e.g. a decision tree learning program which needs counts of instances that satisfy certain criteria).

## B. Related Work

There has been a large body of related work on data integration. Early work on multi-database systems [21] focused on relational or object-oriented database *views* for integrated access to data from several relational databases. More recently, mediators [24] and wrappers have been developed for information integration from multiple data repositories (including semi-structured and unstructured data). Examples include the TSIMMIS project at Stanford University [7], the SIMS project [17], the Ariadne project [26] at the University of Southern California, the Hermes project at the University of Maryland [23], NIMBLE - a commercial system based on research at the University of Washington [20], and the TAMBIS project in UK [8].

INDUS has been inspired by, and builds on previous work by several groups on data integration, and in particular, logic-based and ontology-based approaches to data integration [5, 6]. A major goal of the INDUS project is to provide modular, extensible, open source software environment for ontology-based information integration and data-driven knowledge acquisition from heterogeneous, distributed, autonomous information sources.

The *Information Manifold System* developed at AT&T Bell Laboratories [3] is a heterogeneous data integration system offering a unified query interface for retrieving structured information stored in the WWW and in internal sources. Unlike INDUS which uses a query-centric approach, Information Manifold uses a source-centric approach for answering queries. Like INDUS, Information Manifold utilizes definition of data source capabilities to perform query decomposition and query resolution. However, unlike Information Manifold, INDUS allows users to define several access points for a concept in a data source, each allowing different binding parameters. Information Manifold allows definition of only one capability record per data source. INDUS offers support for several operands (=, <, > etc.), while in Information Manifold only the equality operator is supported.

The *Stanford-IBM Manager of Multiple Information Sources* (TSIMMIS) is a system that facilitates the rapid integration of heterogeneous data sources [7]. The data integration TSIMMIS architecture is based on the concept of wrappers and mediators. Each wrapper knows how to deal with a particular data source and it is able to receive a query in a common language - Object Exchange Model

(OEM) and to transform it into a particular language understood by the data sources. Both INDUS and TSIMMIS use query-centric approach to data integration. However, unlike TSIMMIS, INDUS maintains a clear separation between ontologies used for data integration (which are supplied by users) and the procedures that use ontologies to perform data integration. This allows INDUS users to replace ontologies used for data integration ‘*on the fly*’. This makes INDUS attractive for data integration tasks that arise in exploratory data analysis wherein scientists might want to experiment with alternative ontologies.

The *Transparent Access to Multiple Bioinformatics Information System (TAMBIS)* is an ontology centered system for evaluating queries that offers access to multiple heterogeneous bioinformatics data sources [8]. TAMBIS is based on three-layer wrapper/mediator architecture. Like INDUS, it uses a query-centric approach to data integration. It includes an ontological layer and a graphical user interface for querying. The ontology allows the creation of new concepts based on compositional operations of previously defined concepts using a restricted grammar based on the description logic language GRAIL [9]. TAMBIS returns the answer for a query as an HTML file. Thus, the size of the main memory may limit the amount of data that may be returned in response to a query. In contrast, INDUS stores the answer for a query in a user private area implemented by a relational database system. Thus, queries that return large amounts of data are manipulated more efficiently in terms of hardware and software resources. INDUS also provides better support for defining multiple ontologies for use in different contexts by different users.

## C. Work in Progress

INDUS is a prototype for a data integration system for a scientific discovery environment. As a prototype, it has helped us to understand and demonstrate elements of a promising approach for design of software environments for information integration and knowledge acquisition from heterogeneous, distributed information sources. Some directions for ongoing and future research include:

- a) Further development of the INDUS prototype into a platform to support exploratory data analysis and knowledge acquisition in representative problems in bioinformatics and computational biology e.g., data-driven construction of classifiers of protein function [11, 12]; and predictors of protein-protein interaction [25].
- b) Extending the information integration framework to support extraction of sufficient statistics (e.g., counts that satisfy certain constraints on attribute values) needed for construction of classifiers. This can be accomplished by utilizing aggregate operators for retrieving such statistics (if such operators are supported by the data sources) or by supplying executable code in the form of mobile agents [13] that can execute in a secure environment where the data and computation resources are available to compute the

desired statistics instead of bringing the raw data to the user. This would allow us to extend the recently developed distributed learning algorithms [14,15] to work with heterogeneous data sources.

- c) Extending recently-developed algorithms for learning from multiple relational databases [16] to work with heterogeneous data sources, taking advantage of the capability of INDUS to view heterogeneous information sources as though they were a collection of relational databases.
- d) Extending recently developed algorithms for learning from attribute value taxonomies (a special type of ontologies) and partially specified data [18] to work with data from heterogeneous sources.
- e) Performance improvements in INDUS through the use of more sophisticated query optimization methods, and data caching methods.
- f) Exploration of the use of emerging frameworks for data and metadata description, ontologies, and data source (or more generally resource description), and registry services, being developed as part of the Semantic Web project and related efforts in INDUS.
- g) Exploration of methods for automatically learning mappings between data sources from examples [19], algorithms for ontology merging, and algorithms for learning specific types of ontologies (e.g., attribute value taxonomies) from data.
- h) Extension of approaches used in INDUS to support user and context-specific information integration in peer-to-peer environments and distributed sensor networks.

## ACKNOWLEDGMENTS

This research is sponsored in part by National Science Foundation ITR (Grant Number IIS – 0219699). The views and conclusions contained in this document are those of author(s) and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of the United States Government or of the sponsoring institution.

## REFERENCES

[1] Honavar, V., Millar, L., and Wong, J. 1998. Distributed Knowledge Networks. Design, Implementation, and Applications In: Proceedings of the IEEE Information Technology Conference. pp. 87-90. IEEE Press.

[2] Calvanese, D., Giacomo, G., Lenzerini, M., et al., 1998. Information integration: Conceptual modeling and reasoning support. In: CoopIS'98

[3] Levy, A., 1998. The Information Manifold approach to data integration. In: IEEE Intelligent Systems, 13 12-16, August 19 1998.

[4] Haas L.M., Schwarz, P.M., Kodali, P., Kotlar, E., Rice, J.E., Swope, W.P., 2001. DiscoveryLink: A system for integrated access to life sciences data sources. In: IBM System Journal . Vol 40. No 2. 2001.

[5] Levy, A., 2000. Logic-based techniques in data integration. Logic Based Artificial Intelligence, Edited by Jack Minker. Kluwer Publishers.

[6] Ullman, J., 1997. Information integration using logical views. In: 6th ICDT. Pages 19-40, Delphi, Greece.

[7] Garcia-Molina , Y. Papakonstantinou , D. Quass , A. Rajaraman , Y. Sagiv , J. Ullman , V. Vassalos , J. Widom (1996). The TSIMMIS approach to mediation: Data models and Languages. *Journal of Intelligent Information Systems*

[8] Paton, N.W., Stevens, R., Baker, P.G., Goble, C.A., Bechhofer, S., 1999. Query processing in the TAMBIS bioinformatics source integration system. In: Proc. 11th Int. Conf. on Scientific and Statistical Databases (SSDBM), IEEE Press, 138-147, 1999.

[9] Stevens, R., Baker, P., Bechhofer, S., Ng, G., Jacoby, A., Paton, N., Goble, C., Brass, A, 2000. TAMBIS: Transparent access to multiple bioinformatics information sources. *Bioinformatics*, 16:2 PP.184-186.

[10] Reinoso Castillo, J 2002. Ontology-Driven Information Extraction and Integration from Autonomous, Heterogeneous, Distributed data sources – A Federated Query-Centric approach. Masters Thesis. Artificial Intelligence Research Laboratory, Department of Computer Science, Iowa State University.

[11] Wang, X., Schroeder, D., Dobbs, D., and Honavar, V. (2003). Data-Driven Discovery of Rules for Protein Function Classification Based on Sequence Motifs. *Information Sciences*. In press.

[12] Andorf, C., Dobbs, D., and Honavar, V. (2003) Reduced Alphabet Representations of Amino Acid Sequences for Protein Function Classification. *Information Sciences*. In press.

[13] Wong, J., Helmer, G., Naganathan, V. Polavarapu, S., Honavar, V., and Miller, L. (2001) SMART Mobile Agent Facility. *Journal of Systems and Software*. Vol. 56. pp. 9-22.

[14] Caragea, D., Silvescu, A., and Honavar, V. (ISDA 2003) Decision Tree Induction from Distributed, Heterogeneous, Autonomous Data Sources. In: Proceedings of the Conference on Intelligent Systems Design and Applications. In press

[15] Caragea, D., Silvescu, A., and Honavar, V. (2001a). Analysis and Synthesis of Agents that Learn from Distributed Dynamic Data Sources. Invited chapter. In: Wermter, S., Willshaw, D., and Austin, J. (Ed.). *Emerging Neural Architectures Based on Neuroscience*. Springer-Verlag. In press.

[16] Atramentov, A., Leiva, H., and Honavar, V. (ILP 2003). Learning Decision Trees from Multi-Relational Data. In: Proceedings of the Conference on Inductive Logic Programming To appear.

[17] Arens, Y., Chee, C., Hsu, C., and Knoblock, C. (1993) Retrieving and Integrating Data from Multiple Information Sources. *International Journal of Intelligent and Cooperative Information Systems*. Vol. 2, No. 2. Pp. 127-158

[18] Zhang, J. and Honavar, V. (2003). Learning Decision Tree Classifiers from Attribute-Value Taxonomies and Partially Specified Data. In: Proceedings of the International Conference on Machine Learning. Washington, DC. In press.

[19] Madhavan, J., Bernstein, P., Halevy, A., and Domingos, P. 2002. Representing and Reasoning about Mappings between Domain Models. Proceedings of the Eighteenth National Conference on Artificial Intelligence (pp.80-86). Edmonton, Canada: AAAI Press.

[20] Draper, D., Halevy, A., and Weld, D. (2001). The NIMBLE XML Data integration System. In: Proceedings of the International Conference on Data Engineering (ICDE 01).

[21] Sheth, A.P. and J. A. Larson (1990) Federated database systems for managing distributed, heterogeneous and autonomous databases, *ACM Computing Surveys* 22 pp. 183--236.

[22] Sowa, J. (1999) Knowledge Representation: Logical, Philosophical, and Computational Foundations. New York: PWS Publishing Co.

[23] Subrahmanian, V.S., Sibel Adali, Anne Brink, James J. Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, Charles Ward (2000). HERMES A Heterogeneous Reasoning and Mediator System.

[24] Wiederhold, G. and M. Genesereth (1997) The Conceptual Basis for Mediation Services, *IEEE Expert*, Vol.12 No.5 pp. 38-47

[25] Yan, C., Dobbs, D., Honavar, V., (2003) Identification of Residues Involved in Protein-Protein Interaction from amino acid sequence – A Support Vector Machine approach. In: Proceedings of Intelligent Systems Design and Application

[26] Knoblock, C.A., Minton, S. Ambite J.L., Ashish, N. Muslea, I. Philpot, A.G. and Tejada, S. The Ariadne Approach to Web-Based Information Integration. *International the Journal on Cooperative Information Systems* 10(1/2), pp145-169, 2001.