Ontology-driven information extraction and integration from heterogeneous distributed autonomous data sources: A federated query centric approach.

by

Jaime A. Reinoso-Castillo

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2002

Table of Contents

List of Figures

Acknowledgements

I would like to give my special thanks to Dr. Vasant Honavar for his invaluable help, guidance, patience, mentoring and support during the preparation of this thesis. Learning to do research in computer science was one of my goals when I decide to pursue graduate studies. He taught me how to do it. Much of this work was carried out in close collaboration with Dr Honavar and with Adrian Silvescu.

I would also like to thank my committee members, Dr. Drena Dobs and Dr. Shashi Gadia , for their help and advice.

I am also indebted to the members of the Artificial Intelligence Research Lab for the discussions and friendship. The discussions in the artificial intelligence seminar help me to refine and make concrete several of the ideas presented in this thesis.

Finally, I like to thank to my wife and my family for their love, patience and support in this endeavor.

Much of this work has been funded in part by grants 9972653 and 0087152 from the National Science Foundation, Pioneer Hi-Bred, the John Deere Foundation, and the Fulbright scholarship.

Abstract

Development of high throughput data acquisition in a number of domains (e.g., biological sciences, space sciences, etc.) along with advances in digital storage, computing, and communication technologies have resulted in unprecedented opportunities in scientific discovery, learning, and decision-making. In practice, the effective use of increasing amounts of data from a variety of sources is complicated by the autonomous and distributed nature of the data sources, and the heterogeneity of structure and semantics of the data. In many applications e.g., scientific discovery, it is necessary for users to be able to access, interpret, and analyze data from diverse sources from different perspectives in different contexts. This thesis presents a novel ontology-driven approach which builds on recent advances in artificial intelligence, databases, and distributed computing to support customizable information extraction and integration in such domains.

The proposed approach has been realized as part of a prototype implementation of INDUS, an environment for data-driven knowledge acquisition from heterogeneous, distributed, autonomous data sources in Bioinformatics and Computational Biology.

1. Introduction

Development of high throughput data acquisition technologies in a number of domains (e.g., biological sciences, social sciences, physical sciences, environmental sciences, space sciences, commerce) together with advances in digital storage, computing, and communications technologies have resulted in unprecedented opportunities for scientists, decision makers, and the public at large, at least in principle, to utilize this wealth of information in learning, scientific discovery, and decision making. This is leading to a radical transformation of several scientific disciplines (e.g., biological sciences) accompanied by the emergence of entirely new disciplines such as Bioinformatics and Computational Biology. In practice, effective use of the growing body of data, information, and knowledge to achieve fundamental advances in scientific understanding and decision making presents several challenges [Honavar et al., 1998; Honavar et al., 2001]:

a) Information sources are physically distributed, and large in size. 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 efficient algorithms (e.g., for data mining) that can operate across multiple data sources without the need to transmit large amounts of data. Consequently, there is significant work on distributed learning algorithms [Caragea et al., 2001]

b) Information sources reside on heterogeneous hardware and software platforms, and may use a variety of different access mechanisms and protocols. Consequently, there is a need for software infrastructure that hides the underlying complexity from users and supports seamless access to information residing on diverse hardware and software platforms. This problem is more or less solved with the advent of platform independent programming languages such as Java, and uniform protocols for communication.

c) 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 required information within the operational constraints imposed by the data source are needed [Honavar et al., 2001; Honavar et al., 1998; Wiederhold et al., 1997; Levy, 2000].

d) Data sources are heterogeneous in structure, content, and capabilities (e.g., the types of interactions or queries supported). The ontologies implicit in the design of autonomous data sources (i.e., assumptions concerning *objects* that exist in the *world* which determine the choice of *terms* and *relationships* among terms that are used to describe the domain of interest and their intended correspondence with objects and properties of the world) often do not match the ontologies of the users of those data sources.

e) In scientific discovery applications, because users often need to examine data in *different contexts from different perspectives*, methods for context-dependent dynamic information extraction from distributed data based on user-specified ontologies are needed to support information extraction and knowledge acquisition from heterogeneous distributed data [Honavar et al., 2001; Levy, 2000].

Hence, there is a need for a modular and extensible approach to support flexible and customizable means for extracting and integrating information from heterogeneous, autonomous, and distributed data sources for a scientific discovery environment.

Against this background, we describe the design and implementation of the data integration component of INDUS, an environment for data driven scientific discovery. INDUS allows users to

- view the set of data sources as if they were located locally and they were using an homogeneous interface (e.g., all data sources are presented as if they were relational tables regardless their real underlying technology);
- interact with data sources (i.e., posting queries) through a provided interface that takes advantage of the functionality offered by each of data source using the query capabilities offered by the data sources to answer queries;
- define their own language for defining queries and receiving answers;
- define new concepts based on others concepts by applying a set of well-defined compositional operations;
- use different definitions for the same concept, facilitating the exploration of new paradigms that explain the world.

The rest of the thesis is organized as follows:

- chapter two introduces the key problems that need to be addressed in designing a data integration system for scientific discovery applications,

- chapter three presents our solutions for those problems. In the first part, a formal definition is given for the principal components of our proposed solution. In the second part, a prototype based on such a model is presented. The principal modules that compose the prototype are described and the design decisions explained,
- chapter four presents a summary of our work, the system is compared with other proposed systems, and a list of possible future works is described.

2. Data Integration

This chapter introduces the principal problems associated with the development of a data integration system. It begins with a general definition of a data integration system followed by a description of some of the problems that such a kind of system must solve. For those problems, possible solutions will be presented and compared.

Data integration systems attempt to provide users with seamless and flexible access to information from multiple autonomous, distributed and heterogeneous data sources through a unified query interface [Levy, 2000; Calvanese et al., 1998; Wiederhold et al., 1997]. Ideally, a data integration system should allow users to focus on *what* information is needed without having to offer detailed instructions on *how* to obtain the information. Thus, in general, a data integration system must provide mechanisms for the following:

- Communications and interaction with each data source as needed,
- Specification of a query, expressed in terms of a common vocabulary, across multiple heterogeneous and autonomous data sources.
- Transformation of such a query into a plan for extracting the needed information by interacting with the relevant data sources, and
- Combination and presentation of the results in terms of a vocabulary known to the user.

Thus, bridging the syntactic and semantic gaps among the individual data sources and the user is a key problem in data integration.

Figure 1 presents the general layers that are involved in the design and operation of a data integration system.

The Physical Layer is in charge of the communication between the data integration system and the data sources.

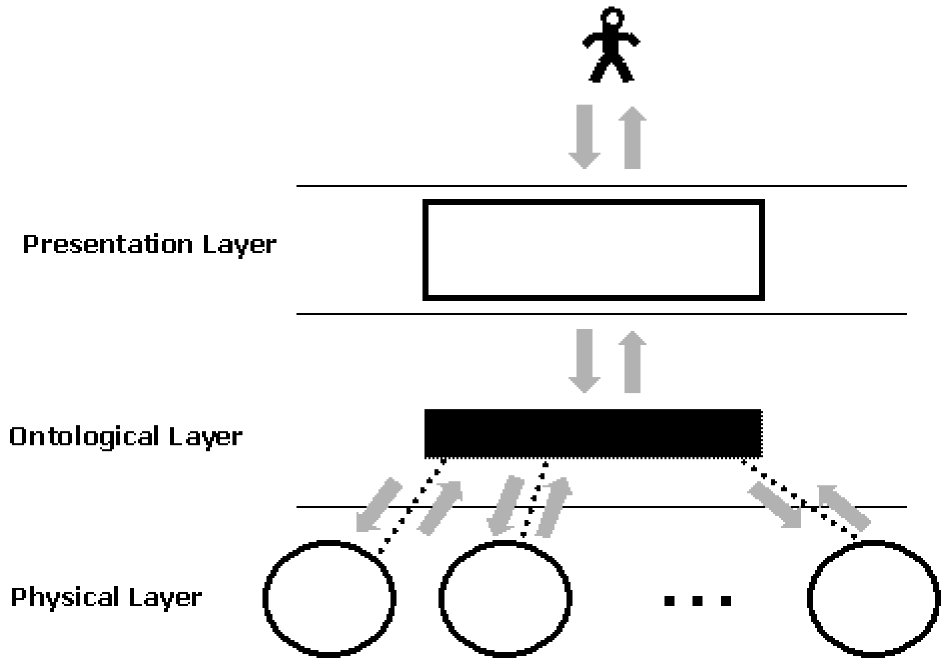


Figure 1. Data Integration Layers.

Layers involved in the design and operation of data integration systems. The Physical layer communicates INDUS with data sources, the ontological layer offers mechanisms for defining ontologies and queries along with a procedure for transforming those queries into execution plans. The Presentation layer allows users to interact with the system.

In the Ontological Layer,

- Mechanisms for defining one or more general ontologies are provided,
- Definitions of queries over general ontologies are supported, and
- Transformations of global queries into execution plans are offered and executed.

In the Presentation Layer, in addition to presenting the answer to the queries, additional functionality may be provided.

The possible problems that can affect one or more layers are presented accordingly in the following sections.

2.1. Infrastructure Definition

Recall that the principal goal of a data integration system is to provide a unified query interface over a set of distributed, heterogeneous and autonomous data sources. Implicitly, the answer for a query is expected to be based on the current content of the data sources.

The distributed nature of data sources calls for mechanisms that are able to gather the needed information and put them together; therefore, a gathering process is required.

The heterogeneity implies that each data source may be characterized as follows:

- Use different technology for supporting its own operation (relational databases, flat files, web pages, etc.),
- Store its information using different syntactic representation, and
- Use different ontological structure (semantic differences).

Heterogeneity of data sources requires mechanisms for transforming the data into a common form. In the Physical Layer, we focus primarily on transformations that deal with syntactic aspect of this problem. However, semantic heterogeneity is the focus of the Ontological Layer.

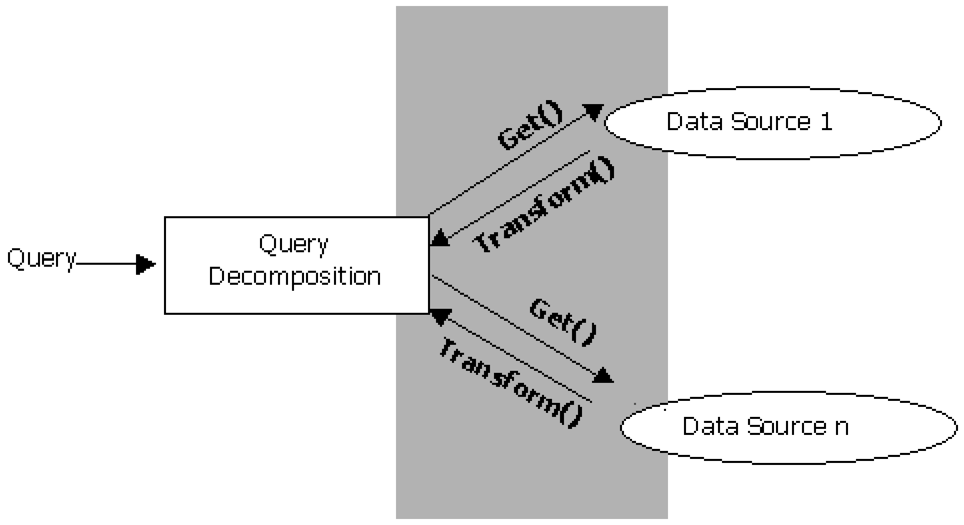


Figure 2. Operations Offered by the Physical Layer.

Operations provided by the infrastructure. The *Get( )* process asks for particular information to the data sources and the *Transform( )* process gathers the information and transform it to a format required by the system.

Figure 2 presents a schematic diagram showing the set of operations expected to be offered by the infrastructure. Thus, when a query is posted, it must be decomposed into a set of operations describing the information that is needed to be gathered from each data source and the form in which this information must be returned to the system. In order to achieve this, the infrastructure must support two basic sets of operations: Get() to query the information sources and Transform() for mapping the results into the desired form.

Two broad classes of solutions are proposed for the infrastructure component of the data integration systems: Data Warehousing, which copies all content of data sources into a central place, and Database Federation, in which the information is accessed directly from the data sources when a query is answered [Haas et al., 2001; Davidson et al., 2001; Convey et al., 2001; Mena et al., 2000]. Another possible architecture proposed by some authors for an infrastructure supporting a data integration system is the Link-driven Federation [Davidson et al., 2001]. Different to Data Warehousing or Federated Database solutions, Link-driven Federation does not move or store information for answering a query. Instead, the set of links where the information is stored is presented to the user. Thus, to recollect, format and unify the data is a user's responsibility. As a result, this infrastructure does not fit in our definition of a data integration system.

In the Data Federation solution, when a query is posted, a set of *get()* and *transform()* processes are triggered in order to gather the required information and return it in the desired form. This ensures that the gathered information is up to date with respect to the content of data sources.

Several alternative architectures exist for the Data Federation solution. One of them, known as a materialized federated system, physically stores the information that is gathered in the course of answering a query in a central place. Federated integration systems that do not materialize the information are known as mediated federated systems [Draper et al., 2001]. Materialized federated systems that use a database system are known as database middleware systems [Haas et al., 2001; Rodriguez-Martinez, 2000b; Haas et al., 1999].

On the other hand, in the data warehousing solution, the get() and transform() processes are not triggered every time that a query is posted. In this case, the system maintains a copy of the content of the data sources in a central place. In order to ensure that the information is up to date, changes in the content of data sources have to be detected and the central repository updated. The answer for a query is then gathered from the central repository instead of directly from the data sources.

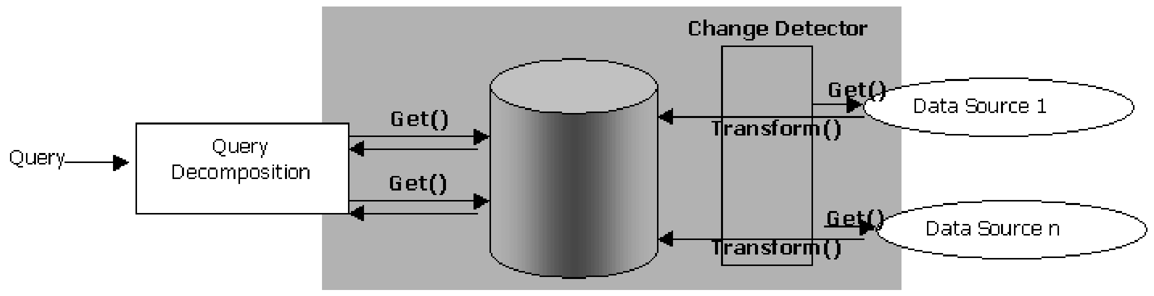


Figure 3. Data Warehousing Infrastructure.

In the Data Warehousing solution, the answer for a posted query is retrieved directly from a common repository. A parallel process keeps up to date the content of the central repository with respect to the content stored in the data sources.

In order to maintain the currency of the central repository, data sources must provide a mechanism for detecting changes in their content. This can be accomplished in two ways: If the data source offers a mechanism for alerting users (or other systems) when their content has changed, the data integration system needs a process for extracting the needed information and introducing those changes in the central repository. If no mechanism for alerting the user about changes in the data source content is provided, the data integration system has to compare the contents of data sources with those of the central repository, find the differences (if any), and update the central repository.

In what follows, we compare the two classes of solutions along several dimensions including some introduced in Davidson et al., [2001].

- **Source of query answers:** Federated Database brings information from data sources on demand when a query is posted. In contrast, Data Warehousing attempts to keep a central repository up to date with respect to the data sources content. Answers to queries are then gathered from the central repository.

- **Initial Cost:** When a Data Warehousing solution is implemented, all relevant information stored in the data source needs to be transformed and copied into the central repository. Additionally, a mechanism for maintaining and synchronizing information stored in the data source and in the central place needs to be implemented. In contrast, there is no need to gather information in a central repository in the federated solution.

- **Maintenance Cost:** As opposed to Data Warehousing which needs to verify periodically that the central repository is up to date relative to the data sources, Federated Database solutions do not need to implement mechanisms for synchronizing information. Note that if the data sources are autonomous and do not provide any method for informing the integration system when their contents have changed, the Data Warehousing system has to implement methods for detecting those changes. In the worst case, this might require repopulating the central repository.

- **Reliability:** Given that Data warehousing solutions create a copy of the information stored on data sources in the central repository, even if data sources are inaccessible, it is possible to return an answer for a posted query. On the other hand, Federated database architecture is unable to respond to queries if the needed data source is inaccessible.

- **Query Performance**: Data Warehousing solution clearly offers a better architecture for improving query responses because all data is located in a central repository. Federated Database is more dependent on network performance or data source accessibility, which are very difficult to predict. Additionally, Data Warehousing transforms and unifies information once per each update process while Federated Database has to apply transformation and unification once for each query.

- **Several General Models**. In the Data Warehousing solution, given that the information is physically stored in a central repository, if several global models are to be offered, each of them has to have its own repository. This represents a very expensive solution in terms of space and makes Federated database solutions more desirable if several global models are required.

- **Query Containment**. If Q and Q′ are queries, Q contains Q′ if every answer for Q′ is also an answer for Q. In a sequence where a query Q is followed by several other queries that are contained in it, the Federated Database solution will bring the required information for each query instead of reusing the information brought previously for answering Q. The Data Warehousing solution on the other hand, naturally supports reuse of information.

It is clear that both techniques have their advantages and disadvantages. The Data Warehousing solution may be a preferred alternative under the following conditions:

a) The content of data sources does not change frequently
b) A mechanism for automatically detecting changes on information stored in data source exists,
c) The frequency of queries is high relative to the frequency of changes in the content of the data sources, and
d) Only one global model is required.

On the contrary, a Federated Database solution may be a better choice under these conditions,

a) Data sources are autonomous,
b) They do not offer mechanisms for detecting changes in their content,
c) The frequency of queries is low relative to the frequency of changes in the content of the data sources , or
d) More than one global model (hence multiple ontologies) need to be supported.

Therefore, given that our primary goal is an environment for scientific discovery, in which data sources are autonomous and may not offer mechanism for detecting changes in the content, large amounts of information may be stored locally in each data source, and supported access to data from different points of view must be provided (hence, multiple global ontologies), Federated Database solution is a more attractive solution. As a result, the rest of the document assumes a Federated Database solution.

2.2. Global Ontology and Query Formulation

As described previously, the physical layer is in charge of providing a mechanism to allow the data integration system to communicate with the data sources. In this communication process, the system is able to gather and, if necessary to transform the required information in order to answer a posted query. So, the primary goal of this layer is to resolve the distribution and syntactical problem. On the other hand, the Ontological layer is primarily concerned with offering a set of mechanisms for resolving the semantic heterogeneity problem.

Recall that the goal of the data integration system is to provide a unified query interface over a set of distributed, heterogeneous and autonomous data sources, hiding any complexity associated with the integration process and also resolving any syntactical and semantic difference. The focus of this section is the Ontological Layer.

Four elements are highlighted in the definition:

a) The idea of a unified model,
b) The possibility to define queries over that model,
c) The idea of hiding the complexity associated with the process for resolving the queries, and
d) The solution of the syntactic and semantic problems.

The Ontological layer helps to resolve the first three elements in addition to the semantic heterogeneity presented in option *d*.

2.2.1 Ontologies

According to the Web Dictionary of Cybernetics and Systems (http://info.babylon.com/cgi-bin/temp.cgi?id=4278&layout=gloss.html) and [Sowa, 2001], *Ontology* (with a capital O) is a branch of philosophy studying what really exists. A particular *ontology* (with a small o) is commonly described as a "specification of a conceptualization" [Gruber, 1993], or as a "formal explicit description of concepts in a domain of discourse" [Noy et al., 2001]. From a practical point of view, *an ontology* [Sowa, 1999; Uschold and Jasper, 1999] specifies terms and relationships among terms (e.g., father and mother are both parents). For example, the first step that a database analyst does when he/she is designing a database [Sowa, 2001] is to recognize the ontological structure, which will be materialized has a database schema. Another common use for ontologies is as hierarchies of terms describing a common (standard) language for a domain as in the Gene Ontology project (http://www.geneontology.org/). In addition to relations between concepts [Gruber, 1993] axioms (or constraints) may also be included [Sowa, 2001].

Therefore, in a data integration system, a global ontology may represent the set of concepts that exists in the domain so queries may be defined in terms of these concepts, and answers may be achieved in terms of instances (or examples) of those concepts. In this way, a unified model can be created, resolving point *a* presented at beginning of section 2.2.

From a user perspective, in conjunction with the infrastructure, the global ontology permits the following:

a) Hiding the complexity of accessing and retrieving information from data sources,
b) Describing the content of each data source in terms of concepts in the global ontology,
c) Imposing a structure on the information stored in data sources in terms of concepts,
d) Allowing the definition of new concepts based on concepts previously defined using a well-defined set of compositional operations, an important tool for resolving semantic heterogeneity problems,
e) Providing a mechanism where queries can be defined based on concepts from a particular ontology, and
f) Presenting answers to a query in a unified and structured form.

Several languages such as KIF [KIF, 2002], OIL [OIL, 2002], RDF [RDF, 2002] and others are used for ontological representation. It is common to find description logics dialects as languages for ontology representation [Description Logics, 2002], as GRAIL [Stevens et al., 2000] and CLASSIC [Kirk et al., 1995]. Some other alternatives include Entity-Relationship Models, Object Oriented Models or XML [XML, 2002]. In general, a language that at least allows users to define concepts, attributes and relationships (roles) between concepts is preferred.

2.2.2. A Unified Query Interface

A unified query interface allows users to post queries over a global model and obtain answers which are structured according to the global model.

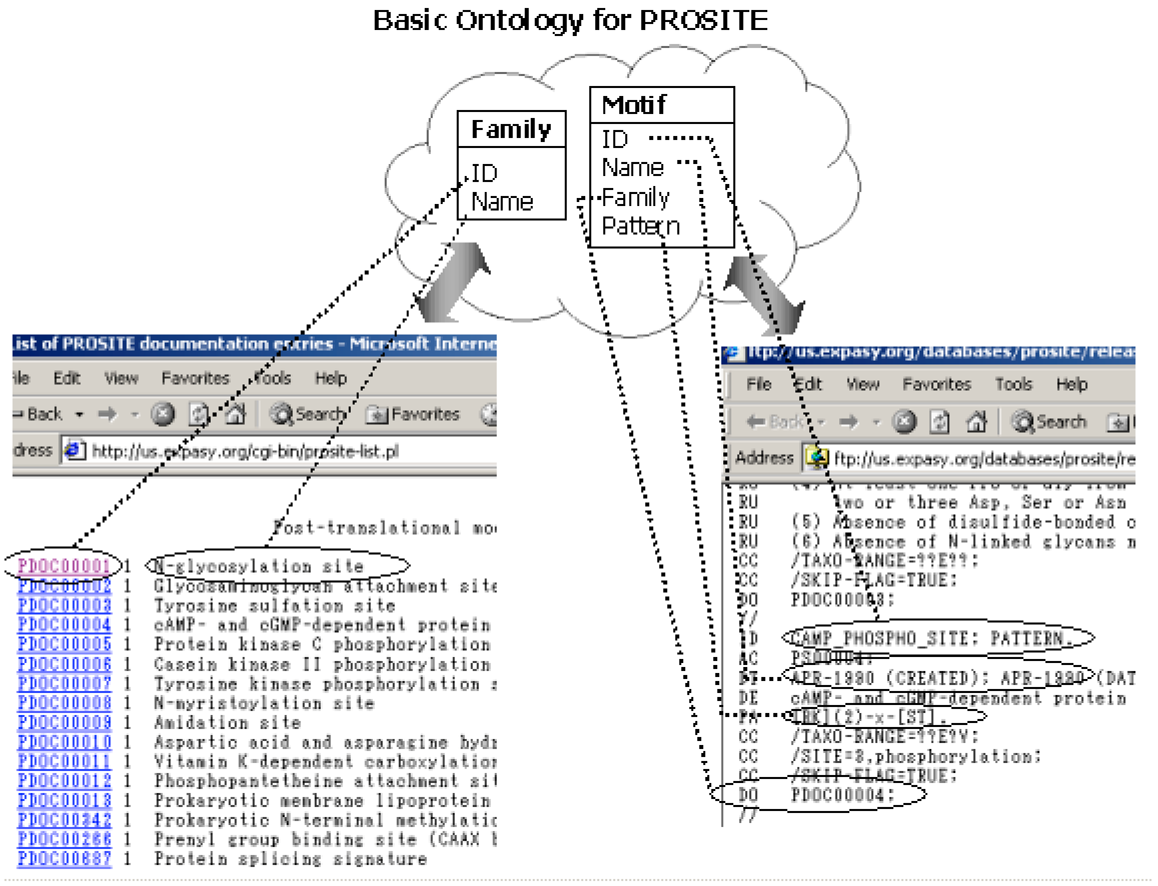
Three elements are necessary in order to provide a unified query interface.

a) The definition of an ontology as a set of basic concepts.
b) Means to associate base concepts with methods that extract instances of the relevant data sources.
c) Operations for creating new concepts in terms of previously defined concepts.

The ontological layer includes the global ontology and also information indicating what exact piece of information must be extracted from each particular data source in order to obtain examples or instances of each basic concept through the physical layer. In general, a relational paradigm is used for describing concepts, so a table is a common representation. Therefore, the set of instances that belong to the concept may be described as rows.

Figure 4 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 attributes, known as the structure of the concept.

A unified interface is provided with these two elements. Nevertheless, no semantic heterogeneity is offered yet. In order to resolve the semantic heterogeneity, it is necessary to introduce some set of compositional operations that allow users to define new concepts in terms of other previously defined concepts.


Basic Ontology for PROSITE
Family
ID
Name
Motif
ID
Name
Family
Pattern
List of PROSITE documentation entries - Microsoft Internet
File Edit View Favorites Tools Help
Back Search Favorites
Address http://us.expasy.org/cgi-bin/prosite-list.pl
ftp://us.expasy.org/databases/prosite/release
File Edit View Favorites Tools Help
Back Search
Address ftp://us.expasy.org/databases/prosite/re

```
                   Post-translational mo
PDOC00001 1  N-glycosylation site
PDOC00002 1  Glycosaminoglycan attachment site
PDOC00003 1  Tyrosine sulfation site
PDOC00004 1  cAMP- and cGMP-dependent protein
PDOC00005 1  Protein kinase C phosphorylation
PDOC00006 1  Casein kinase II phosphorylation
PDOC00007 1  Tyrosine kinase phosphorylation s
PDOC00008 1  N-myristoylation site
PDOC00009 1  Amidation site
PDOC00010 1  Aspartic acid and asparagine hydr
PDOC00011 1  Vitamin K-dependent carboxylation
PDOC00012 1  Phosphopantetheine attachment sit
PDOC00013 1  Prokaryotic membrane lipoprotein
PDOC00342 1  Prokaryotic N-terminal methylatic
PDOC00266 1  Prenyl group binding site (CAAX b
PDOC00887 1  Protein splicing signature
```

```
RU          two or three Asp, Ser or Asn
RU     (5) Absence of disulfide-bonded c
RU     (6) Absence of N-linked glycans n
CC     /TAXO-RANGE=??E??;
CC     /SKIP-FLAG=TRUE;
DO     PDOC00003;
//
ID     CAMP_PHOSPHO_SITE; PATTERN.
AC     PS00004;
DT     APR-1990 (CREATED); APR-1990 (DAT
DE     cAMP- and cGMP-dependent protein
PA     [RK](2)-x-[ST].
CC     /TAXO-RANGE=??E?V;
CC     /SITE=3,phosphorylation;
CC     /SKIP-FLAG=TRUE;
DO     PDOC00004;
//
```

Figure 4. A Basic Ontology and Extended Definitions for Basic Concepts

A simple ontology composed by two concepts. Because those are basic concepts, their instances are stored in data sources. The information indicating how to extract the set of instances for each basic concept, represented by the lines between the attributes and the data sources, is also stored in the ontology.

2.2.3. Resolving Semantic Heterogeneity

Semantic heterogeneity appears in different ways, for example, when the same term is used to describe two different things, or when two different concepts are represented with the same term. This simple case can be resolve by a mapping function. The problem became more complex when the mapping is not one to one. So this problem is reduced to find adequate mechanisms for describing the local ontologies in terms of the global one or vice versa.

Semantic heterogeneity also arises when different units are used by two or more data sources to describe the value of an attribute of a global concept. As an example, assume that two data sources store information about organisms indicating the length of each specimen, one in inches, and the other in centimeters. When a query is posted asking for the biggest value in inches, a transformation must be applied in order to convert the data of one data source from centimeters to inches.

In some cases, the data source can offer this transformation. For example, if you are planning to send a package to another state in the USA, you can access web pages from UPS or FEDEX and ask for pricing based on the size of the package. These web sites permit you to introduce this information in centimeters or in inches. In other cases, this functionality is not offered, thus implementing this transformation at the central repository is needed. In other words, a transformation function is needed, independently of the selected infrastructure.

As a conclusion, in order to bridge semantic gaps between data sources, a set of operators that allow users to define concepts in terms of other concepts is required. Also, a set of transformations or functions may help to resolve the heterogeneity. These functions may be located in the physical layer or in the ontological layer as special compositional operands or as user-defined functions.

Figure 5 presents some possible examples of operations for constructing new concepts in term of previously defined concepts. At left, a new concept C_3 is created based on a union operation between instances from concepts C_1 and C_2. At right, a new concept C_5 is defined as a selection and projection operations applied on concept C_4. C_5 can be seen as a subclass of C_4 using object-oriented terminology.

Therefore, at least two classes of concepts can be defined: those whose instances are directly stored in data sources and those whose instances are obtained through of a set of compositional operations based on instances of other concepts.

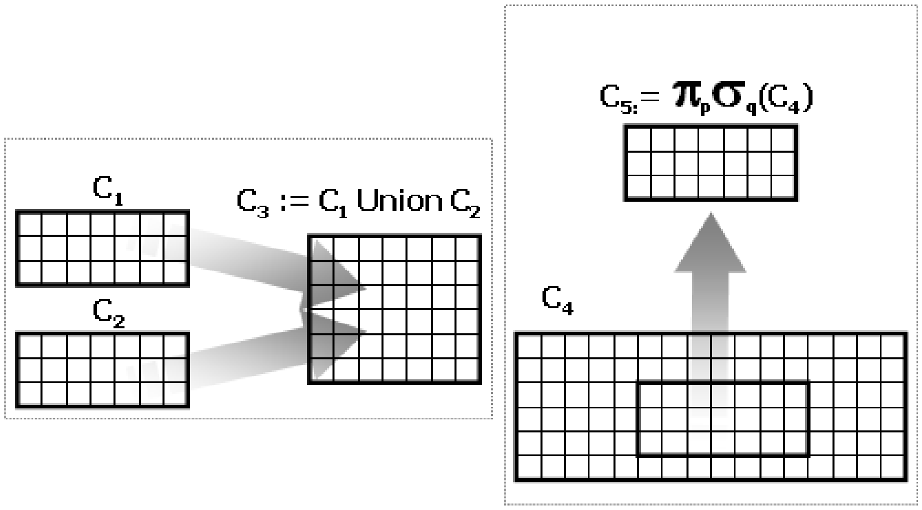


Figure 5. Compound Concepts Examples.

Example of operations that used to define new concepts in terms of others. At left, a union operation is applied over the set of instances of the C_1 and C_2 concepts in order to construct the concept C_3. At right, the concept C_5 is created based on a selection and projection operations applied over the concept C_4.

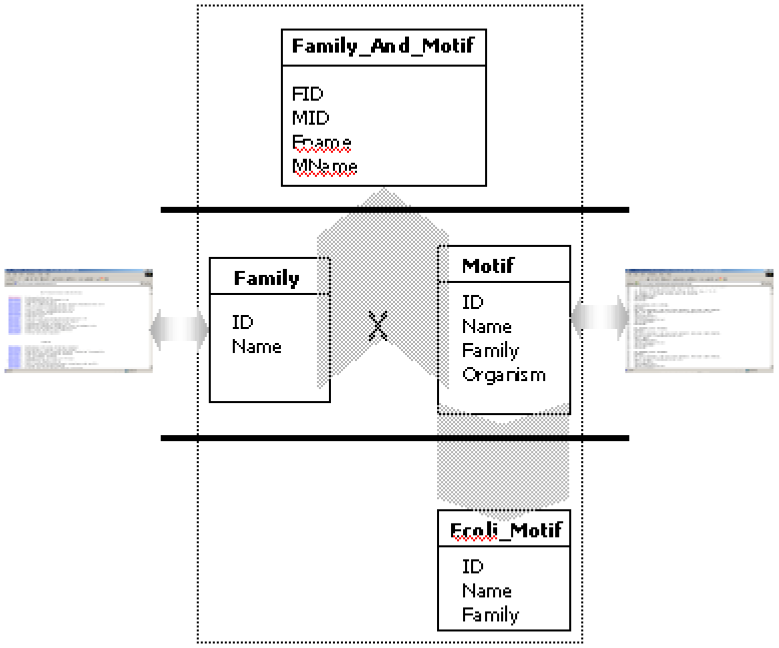


Figure 6. Basic Concepts and Compound Concepts.

Instances of basic concepts are physically stored in data sources. Instances of compound concepts are “obtained” applying compositional operations over the set of instances of other concepts. In this case, the applied operation corresponds to a Cartesian product.

The following section presents two techniques for defining ontologies and for resolving queries. First is an introduction to a formalism called DataLog followed by a description of the Query-centric and Source-centric techniques for query resolution.

2.3 Query Resolution

In general, a query is defined in terms of global concepts and a set of pre-defined allowed operations for combining concepts in the same way that a compound concept is defined. The first step to answering a query involves rewriting the query in terms of a set of basic concepts and a set of compositional operations. This provides the necessary information for the system to retrieve the relevant data from each source and to combine the data to obtain an answer for a query.

There are two principal proposed techniques for resolving the query transformation, also known as reformulation or rewriting problem: Query-centric approach, also known as Global as View, and Source-centric approach, also known as Local as View (LAV) [Li, 2001; Levy, 2000; Lambrecht et al., 1997; Cal`i et al., 2001; Convey et al., 2001; Ullman, 1997].

As is common in the literature, we will treat each concept instantiated in a data source as though it were a relation. A relation is informally defined as a table with a name and a set of attributes implemented as columns. Thus, an employee relation with attributes *name* and *phone* can be represented as *Employee(name, phone)* where *Employee* is a table and *name* and *phone* are columns.

EMPLOYEE

Name	**Phone**
Jaime	445566
John	335799
Peter	110946

Figure 7. The Employee Table.

In the data integration literature, DataLog [Tari, 1995; Ullman, 1988], a subset of first order logic, is a language for representing the available set of data sources and for describing the expressive power of the query language offered by each data source.

Before continuing with Query-centric and Source-centric approaches, a short introduction to DataLog taken from [Ullman, 1988] and [Ramakrishnan, 2000] will be presented.

2.3.1. DataLog

DataLog is a restricted version of Prolog oriented to databases operations. Its principal differences with Prolog are as follows:

- Functions are not allowed as predicate arguments. Thus, only constants and variables are allowed.
- DataLog programs are based on the model theoretic notion of semantics as opposed to the proof-theoretic notion.

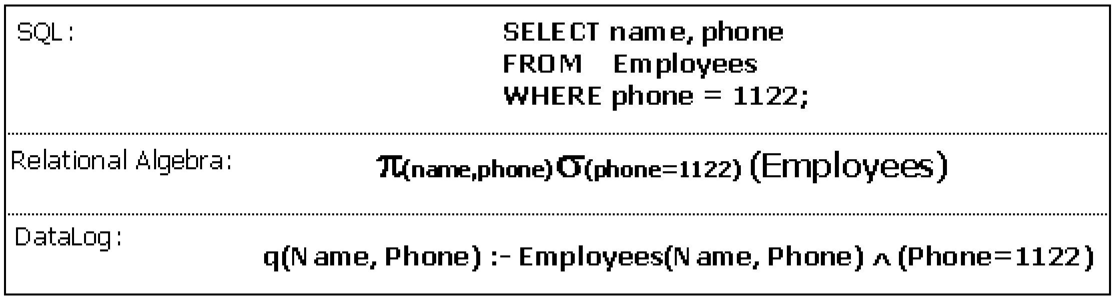
Atomic terms are predicates which are presented as a name followed by parenthesis with variables or constants separated by commas. Thus, a relation of employees storing names and phone numbers can be represented as Employee(Name, Phone) where Name and Phone are variables. A fact about an employee can be represented as employee(james, 445566) where 'james and '334455' are constants.

Thus, relations for relational algebra are equivalent to predicates in DataLog. Attributes in relational algebra are named; therefore, no two attributes can have the same name. In DataLog, the notion of a list of attributes is used. Thus, the first attribute will refer to the same domain independently of the name of the variable.

The rows stored in the table in Figure 7 can be represented in DataLog as follows:

```
Employee(james, 445566).
Employee(john, 335799).
Employee(peter, 110946).
```

In DataLog there are two kinds of predicates: EDB and IDB. EDB stands for Extensional DataBase and indicates that the predicate has a table, or set of facts, that explicitly stores the correspondent set of instances, as shown in the Employee Table above. On the other hand, IDB stands for Intentional DataBase and refers to predicates that are defined in terms of other predicates. The IDBs are similar to the definition of views in the relational algebra, except that recursion is allowed in IDBs while it is not in relational algebra.



SQL:	SELECT name, phone
	FROM Employees
	WHERE phone = 1122;

Relational Algebra:	$\pi_{(name,phone)}\sigma_{(phone=1122)}$ (Employees)
DataLog:	q(Name, Phone) :- Employees(Name, Phone) $\wedge$ (Phone=1122)

Figure 8. Equivalent Query Expressed using SQL, Relational Algebra and DataLog.

IDBs are defined using rules of the form:

$$\mathbf{q(X) :- p_1(X_1) \wedge p_2(X_2) \wedge \ldots \wedge p_n(X_n)}$$

where X and X_i are sets of variables. The left part is called the head and the right part the body. The body may consist of EDB, IDB or built-in predicates (e.g. *salary=1250*).

A DataLog rule is said to be safe if each variable at the left side appears as an argument of a predicated or in a built-in predicate in the right side, and it is said to be Recursive if the head appears at least once at the right side of the rule. A DataLog rule with negations has at least one of the right side predicates negated. A special kind of negation, known as *stratified negation*, appears when a rule of the form **p :- ...** $\wedge$ **not q** $\wedge$ **...** is defined but no rule of the form **q :- ...** $\wedge$ **p** $\wedge$ **...** exists.

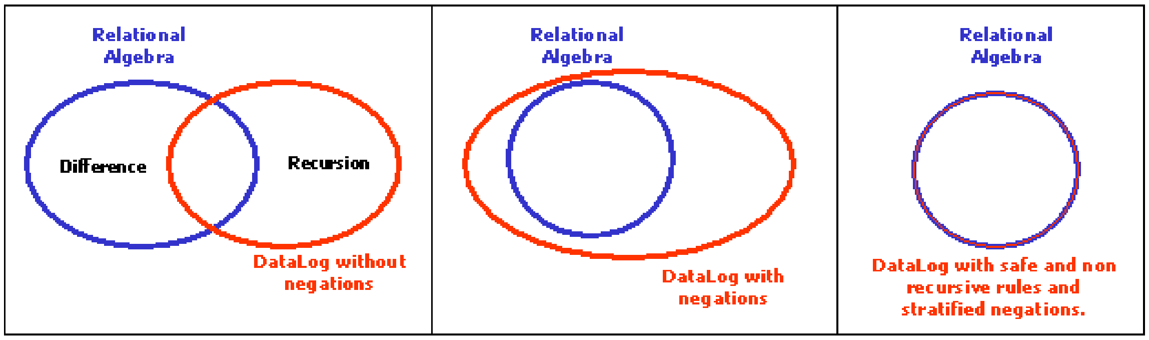


Figure 9. DataLog and Relational Algebra Relationship.
DataLog and Relational Algebra are used to represent basic and compound concepts in a data integration system. Both formalisms have an equivalent expressive power when DataLog is restricted to safe and non recursive rules using stratified negations.

Figure 9 illustrates the differences between DataLog and Relational Algebra. DataLog and Relational Algebra are equivalent when DataLog is restricted to safe and non recursive rules with only stratified negations.

2.3.2. Query-centric Approach

A Query-centric approach defines each global concept in terms of basic concepts, those whose instances are stored in data sources, using a predefined set of compositional operations. Therefore, the system has a description of how to obtain the set of instances of a global concept based on extracting and processing instances from data sources.

Using a relational database paradigm, assume that there is a general model with one table called PROTEIN with columns ID, NAME, TYPE. Assume also that there are two data sources A and B that can provide this information. In a Query-centric approach, the PROTEIN concept is described in terms of A and B.

Following SQL syntax, this can be expressed 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;
```

In DataLog this is equivalent to the following:

```
PROTEIN(Id, Name, Type) :- A_Protein(Id, Name, Type).
PROTEIN(Id, Name, Type) :- B_Protein(Id, Name, Type).
```

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;
```

In DataLog it will appears as follows:

```
PROTEIN(Id, Name, Type) :- A_Protein(Id, Name, Type).
PROTEIN(Id, Name, Type) :- B_Protein(Id, Name, Type).
PROTEIN(Id, Name, 'Enzyme') :- C_Protein(Id, Name).
```

Similarly, if a data source D stores different kind of 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';
```

In DataLog it will appears as depicted below:

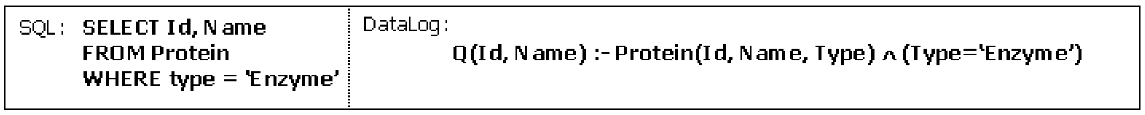
PROTEIN(Id, Name, Type) :- A_Protein(Id, Name, Type).
PROTEIN(Id, Name, Type) :- B_Protein(Id, Name, Type).
PROTEIN(Id, Name, 'Enzyme') :- C_Protein(Id, Name).
PROTEIN(Id, Name, Type) :- D_Chemical_Compound(Id, Name, Type, Kind) ∧ (Kind='Protein').

Thus, the procedure for answering a query over the PROTEIN concept must create a plan that expresses the PROTEIN concept in terms of its definition. The plan can be seen as an expression tree, where each internal node corresponds to a well-defined operation (union, projection, selection, etc.) and each leaf corresponds to a concept. Any non basic concept appearing in the previous plan must be replaced by its definitions recursively just to arrive at a plan where only basic concepts appear as leaves.

After the plan is created, the next step is executed. Given that each leaf corresponds to a basic concept, their instances are extracted from the corresponding data sources. For the internal nodes, a set of instances is created combining the set of instances returned by each child appropriately. The execution process finishes when the set of instances for the root of the tree is obtained.

Because of its simplicity, the algorithm for answering a query is one of the advantages of the Query-centric approach [Levy, 2000].

As an example, Figure 10 presents a posted query over the PROTEIN concept, expressed in SQL and in DataLog.



```
SQL:  SELECT Id, Name          DataLog:
      FROM Protein                 Q(Id, Name) :- Protein(Id, Name, Type) ∧ (Type='Enzyme')
      WHERE type = 'Enzyme'
```

Figure 10. Query Centric Approach Example.
the query Q is expressed in terms of basic concept using SQL (at left) and DataLog (at right).

Thus, the first step is to create an expression tree for the query based on the definition of the PROTEIN concept. This is shown in Figure 11.

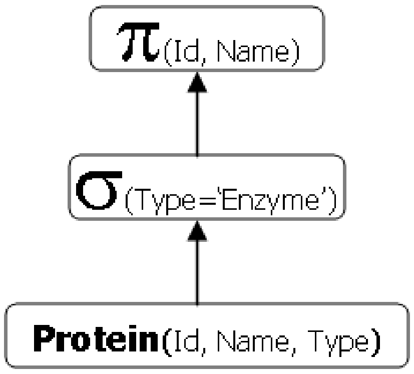


Figure 11. Initial Expression Tree.

The query presented in Figure 10 is transformed into an initial expression tree. This tree applies a selection operation over a projection operation on the set of instances belonging to the *Protein* concept.

The definition for the Protein concept is expanded because it is not a basic concept. Figure 12 presents the expanded expression tree.

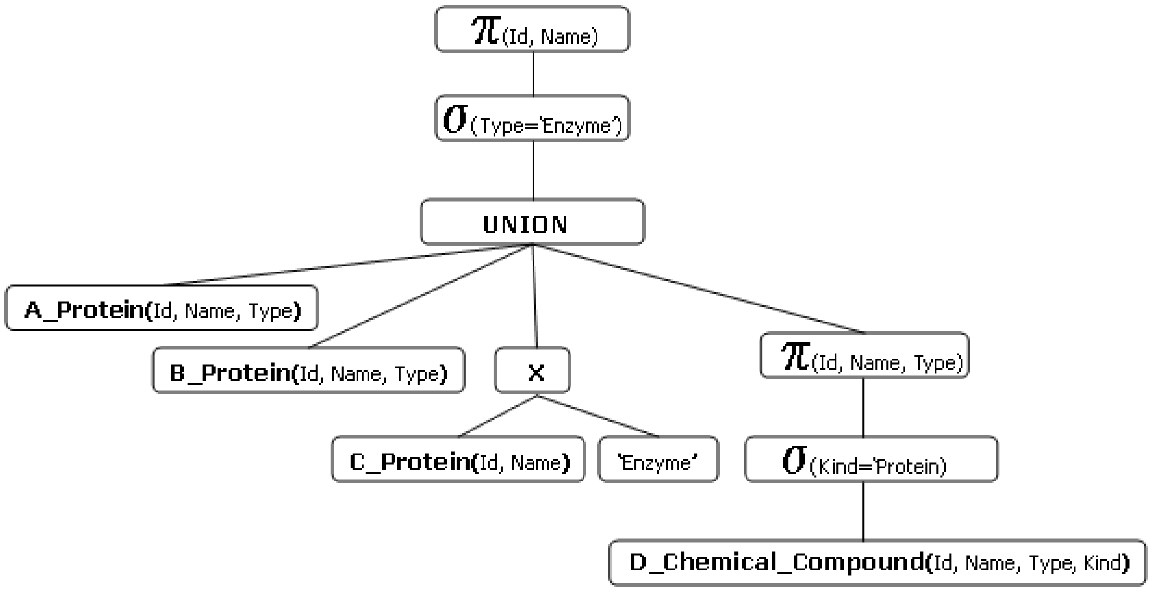


Figure 12. Expanded Expression Tree.

The expression tree presented in Figure 11 is modified replacing the node storing the *Protein* concept with the operational definition of the *Protein* concept, which in this case corresponds to a union over a set of basic concepts.

A final step may include some optimization operations as shown in Figure 13. In general, a common practice for expression tree optimization consists in pushing down the selection and projection operations as much as possible.

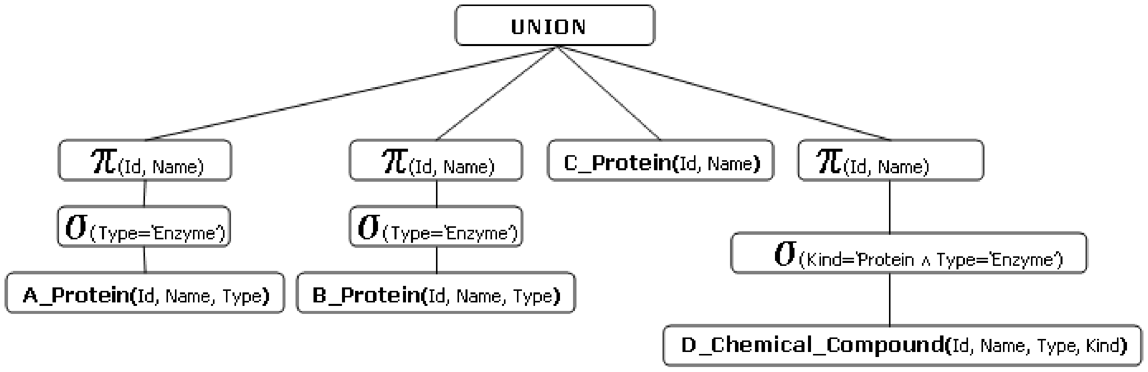

UNION
π(Id, Name)
σ(Type='Enzyme')
A_Protein(Id, Name, Type)
π(Id, Name)
σ(Type='Enzyme')
B_Protein(Id, Name, Type)
C_Protein(Id, Name)
π(Id, Name)
σ(Kind='Protein ∧ Type='Enzyme')
D_Chemical_Compound(Id, Name, Type, Kind)

Figure 13. Optimized Expression Tree.

The expression tree presented in Figure 12 may be optimized minimizing the set of instances to be retrieved from each basic concept. Because the *C_Protein* concept only stores information about enzymes, no selection operations needs to be defined in that data source.

The plan is executed using in-order ordering. Thus, a node is executed only if its children have been executed first as it is shown in Figure 14.

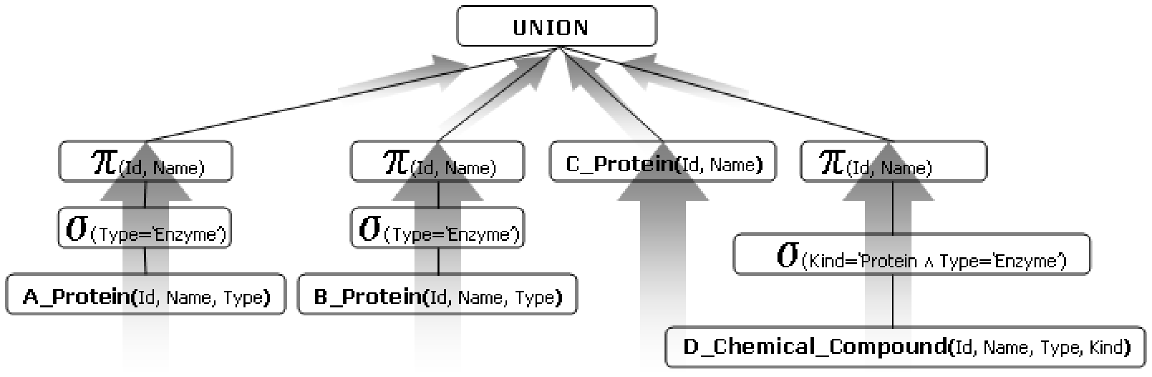

UNION
π(Id, Name)
σ(Type='Enzyme')
A_Protein(Id, Name, Type)
π(Id, Name)
σ(Type='Enzyme')
B_Protein(Id, Name, Type)
C_Protein(Id, Name)
π(Id, Name)
σ(Kind='Protein ∧ Type='Enzyme')
D_Chemical_Compound(Id, Name, Type, Kind)

Figure 14. Execution Order of a Expression Tree.

The expression tree is executed from bottom to top. A post-order ordering may accomplish this task.

2.3.3. SOURCE-CENTRIC APPROACH

A Source-centric approach defines local concepts, those whose instances are stored directly in one or more data sources, in terms of global concepts. In order to illustrate this approach, assume that four global concepts have been defined as follows:

- Student (Id, GPA): Information students Id and GPA.
- CoursesByStudent(CourseId, StudentId): Courses taken by each student.
- PaymentsByStudent(StudentId, Payment): Payments (in dollars) done by each student
- Courses(Id, dept): courses offered by each department.

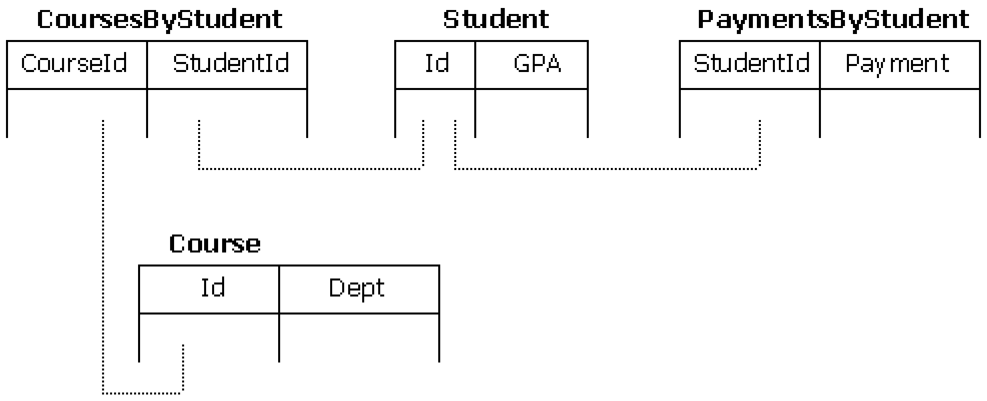


Figure 15. Global Model Example for a Source-centric Approach.

In a Source-centric approach, a set of global concepts are originally defined. When a new data source is added to the system, its content is described in terms of those global concepts. For this example, four global concepts were defined. *Student* and *Course* refer to a set of students and courses offered by a university. *CoursesByStudent* indicates which courses are taking for each student. *PaymentByStudent* indicates the payments made by each student.

Also, assume that three data sources *A*, *B*, and *C* store information about this global model. The *A* data source has information about those students with GPA > 3.0 that have taken courses from the Computer Science Department. Information about those students with a GPA > 2.0 that have made payments for a value higher than 300 is stored in the *B* data source. Finally, the *C* data source stores information about those students that have made a payment higher than 150.

The figures above present a graphical description of each data source and its correspondent DataLog representation.

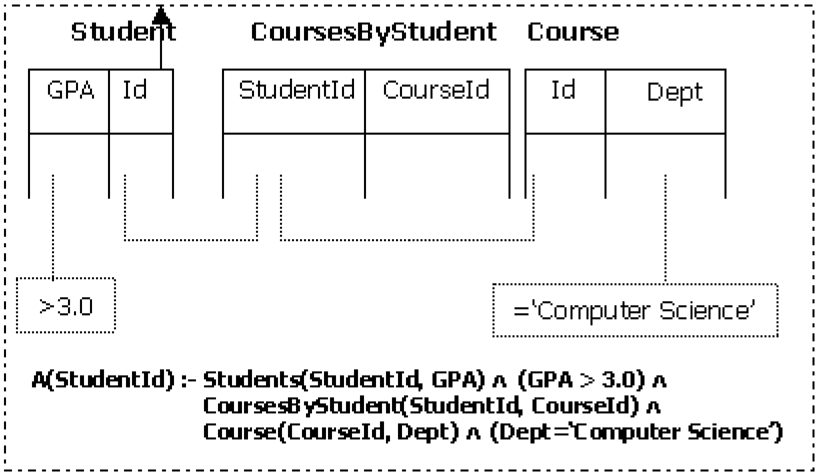

Student
GPA
Id
CoursesByStudent
StudentId
CourseId
Course
Id
Dept
>3.0
='Computer Science'
A(StudentId) :- Students(StudentId, GPA) ∧ (GPA > 3.0) ∧
CoursesByStudent(StudentId, CourseId) ∧
Course(CourseId, Dept) ∧ (Dept='Computer Science')

Figure 16. The *A* Data Source.

The *A* data source offers a list of student ids. Only those ids that belong to a student with a GPA bigger than 3.0 taking at least a course in the Computer Science department are returned.

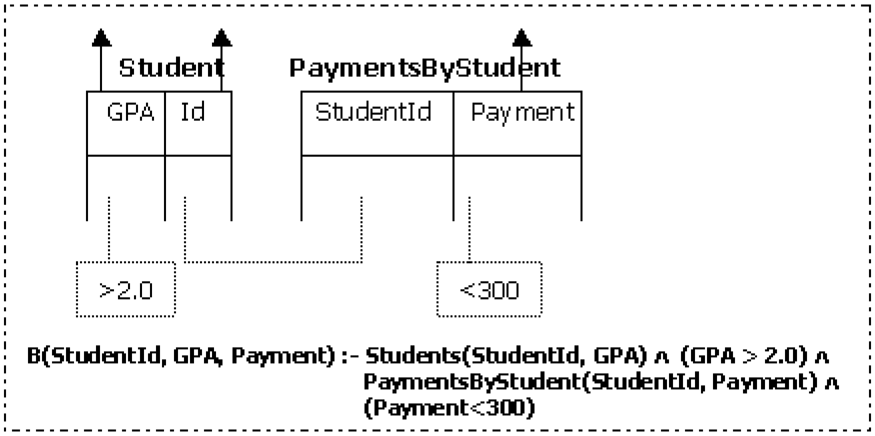

Student
GPA
Id
PaymentsByStudent
StudentId
Payment
>2.0
<300
B(StudentId, GPA, Payment) :- Students(StudentId, GPA) ∧ (GPA > 2.0) ∧
PaymentsByStudent(StudentId, Payment) ∧
(Payment<300)

Figure 17. The *B* Data Source.

The *B* data source returns the Id, the GPA and the payments of students with a GPA bigger than 2.0 and payments lower than 300.

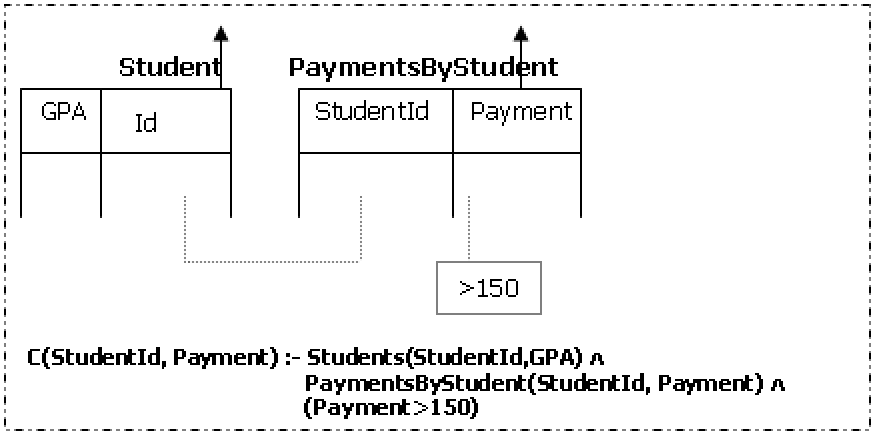

Student
GPA
Id
PaymentsByStudent
StudentId
Payment
>150
C(StudentId, Payment) :- Students(StudentId,GPA) ∧
PaymentsByStudent(StudentId, Payment) ∧
(Payment>150)

Figure 18. The C Data Source.

The *C* data source returns the ids and payments of students with payment bigger than 150.

Assume that a query asking for those students with a GPA > 3.0 that have made a payment of 200 is posted. In DataLog this is expressed as follows:

Q(StudentId) :- Students(StudentId, GPA) $\wedge$ (GPA > 3.0) $\wedge$
Payments(StudentId, Payments) $\wedge$ (Payments = 200)

As opposed to the Query-centric approach where the algorithm for answering a query is straightforward, a more complex process is necessary in order to find an answer in the Source-centric approach. In this case, a solution can be obtained by the following query:

Q′(StudentId) :- (B(StudentId, GPA, Payment) $\wedge$ (Payment = 200) $\wedge$ (GPA > 3.0)) $\vee$
(A(StudentId) $\wedge$ C(StudentId, Payment) $\wedge$ (Payment = 200))

Then, the problem of finding a solution for a query *Q* (expressed in global concepts) is to find a rewriting *Q′* of *Q*, where *Q′* is expressed in terms of local concepts and some valid operations (in this case $\vee$ and $\wedge$) such that any instance satisfying *Q′* also satisfies *Q*. The goal is to find a *Q′* such that no other rewriting of *Q″* of *Q* contains *Q′*. This last problem is known as the containment problem and it is known to be NP-Complete [Ullman, 1997; Duschka et al., 2000, Chaudhury et al., 1995].

Another way to express the containment problem is as follows: Be $I(Q)$ the set of instances satisfying the query *Q*. *Q′* is considered a rewriting of *Q* if $I(Q') \subseteq I(Q)$. *Q′* is an equivalent rewriting of *Q* if also $I(Q) \subseteq I(Q')$. In data integration systems, in addition to the previous definition, the problem of containment also implies that *Q* is defined in term of global concepts and *Q′* in term of local concepts. The goal is to find the maximal rewriting (instead of an equivalent rewriting) which is defined as follows: *Q′* is a maximal rewriting of *Q*, if $I(Q') \subseteq I(Q)$ and no other *Q″* (also expressed in term of local concepts) exists such that $I(Q') \subset I(Q'') \subseteq I(Q)$ [Ullman, 1997; Duschka et al., 2000, Chaudhury et al., 1995].

Similar to the Query-centric approach, the answer can be seen as an expression tree plan.

Levy et al [1996] presents three different algorithms for finding a maximum rewriting *Q′* of a query *Q* when *Q* and *Q′* are restricted to conjunctions of predicates. These three algorithms are: the Bucket Algorithm (developed under the Information Manifold System), the Inverse-rules Algorithm and the MiniCon Algorithm.

In <u>the Bucket Algorithm</u> [Levy, 2000; Levy et al., 1996], a bucket is created for each global concept in the query *Q*. Each local concept definition containing information about a global concept is added to the correspondent bucket. The second part of the bucket algorithm combines one local concept for each bucket forming conjunctions. Then, the rewriting *Q′* of *Q* is the union of all the conjunctions. Levy et al [1995] prove that if *Q* is restricted to the conjunction of predicates without built-in predicates, having *p* predicates, the rewritten *Q′* will have at most *p* predicates too.

Following the example presented above, if Q is defined as follows:

Q(StudentId) :- Students(StudentId, GPA) $\wedge$ PaymentsByStudent(StudentId, Payments)

the bucket algorithm will create two buckets, one for Students and one for Payments. In the Students bucket three predicates will be added: A, B and C. In the Payments bucket two predicates will be added: B and C. Thus, the solution obtained for the bucket algorithm, without including the built-in predicates, will be as follows:

$$(\mathbf{A} \wedge \mathbf{B}) \vee (\mathbf{A} \wedge \mathbf{C}) \vee (\mathbf{B} \wedge \mathbf{B}) \vee (\mathbf{B} \wedge \mathbf{C}) \vee (\mathbf{C} \wedge \mathbf{B}) \vee (\mathbf{C} \wedge \mathbf{C})$$

this can be rewritten as:

$(A \wedge B) \vee (A \wedge C) \vee (B) \vee (B \wedge C) \vee (C)$

In <u>the Inverse-rules algorithm</u> [Levy, 2000], the definition of any predicate is inverted, obtaining a new set of predicates where the body is the global concept. Duschka et al [2000] showed that the inverse-rule algorithm finds the maximally-contained.

So, if a source V_1 is defined as follows,

$V_1(x,z) \text{ :- } B(x,y) \wedge C(y,z)$

it is possible to create two inverse rules:

$B(x,f_1(x,p)) \text{ :- } V_1(x,p).$
$C(f_1(x,p),x) \text{ :- } V_1(p,x).$

The next step is to bring all instances from the data sources, assign values for the variables and for the functions f_x and return only those instances accomplishing the given set of conditions posted in the query.

The bucket algorithm makes an exponential search looking for the set of all possible combinations of predicates. The inverse-rule searches for the space of the instances in the data sources. The MiniCon algorithm tries to address the limitation of these two algorithms restricting the set of views that can be taken into account by verifying the possible values that the set of variables can take [Levy, 2000; Pottinger et al., 2001].

Two additional algorithms, not described here, are presented in [Convey et al.,2001]: The Shared-Variable-Bucket Algorithm and the CoreCover Algorithm.

2.3.4. Plan Properties

A rewriting *Q'* of a query *Q* can be seen as a plan for answering *Q*. Some of plan properties that are of interest include soundness, optimality and minimality [Lambrecht et al., 1997; Levy, 2000].

Let *I(Q)* be the set of instances satisfying *Q*. A plan *Q′* is sound with respect to *Q* if $I(Q') \subseteq I(Q)$. Similarly, a plan is said to be complete with respect to *Q* if $I(Q') \supseteq I(Q)$. At left of Figure 19 is presented a rewriting *Q′* that is sound with respect to *Q*. At write of Figure 19 is presented a rewriting *Q′* that is complete with respect to *Q*.

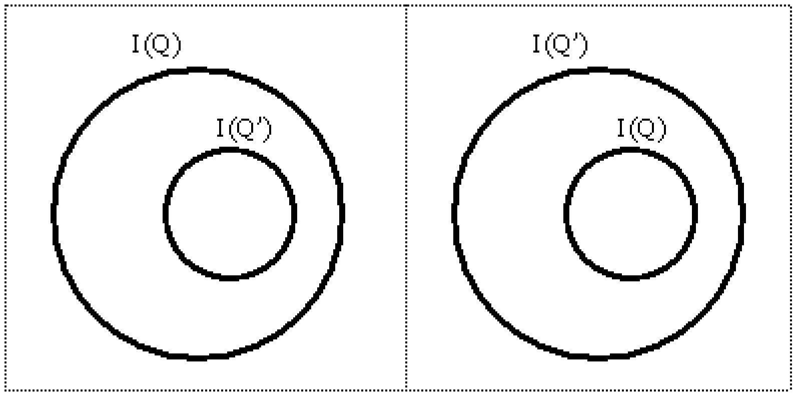


Figure 19. Soundness and Completeness Properties.

At left, the instances of Q' are a subset of the instances of the original query Q. This implies that all instances returned by Q' satisfies also Q (although not all instances satisfying Q are returned by Q'). Therefore, Q' is sound with respect to Q. At right, the set of instances of the rewriting Q' is a super set of the instances of the query Q. This implies that all instances satisfying Q are returned by Q' (although Q' may return instances that no satisfies Q). In this case, Q' is complete with respect to Q.

The completeness property may also be defined relative to a particular language. In that case, the term *operational completeness* is used. When the completeness property is defined for any possible language, then the term conceptual completeness is used [Lambrecht et al., 1997].

A cost can be associated with a plan based on some predefined metrics. Thus, an optimal plan is that which has the lowest cost of all the possible plans for answering a query. When the metric is based on the number of predicates appearing in a plan *Q′*, the optimal plan is known as a minimal plan.

Under a Query-centric approach, as was described previously, a plan *Q′* for answering a query *Q* is created based on the expansion of the definitions associated with each global concept in *Q* and in a set of predefined compositional operations. When the plan *Q′* is completely expanded, all the leaves refer to basic concepts, and the internal nodes refer to compositional operations. Thus, the language *L* for defining *Q′* depends on the set of compositional operations and on the existence of a set of functions, associated with the infrastructure, for retrieving the required instances from the relevant data sources for each basic concept. Therefore, *Q′* is sound and operationally complete with respect to *Q* and *L*. No assertion can be made about conceptual completeness.

Similarly, under the Source-centric approach, the three algorithms shown previously are sound and operationally complete [Levy et al., 1995].

2.3.5. Comparing Query-centric and Source-centric Approaches

A Query-centric approach defines global concepts in terms of basic concepts while a Source-centric approach defines basic or local concepts in terms of global concepts.

This has implications in terms of how new basic concepts are incorporated into the system. Consider a system in which the query language is restricted to set union operations applied over EDI predicates without built-in predicates. Assuming a Query-centric case, Figure 20 shows a family of queries Q_i based on a set of basic concepts q_{ij}. Let $I(Q)$ and $I'(Q)$ be the set of instances satisfying Q respectively before and after adding c to the system. Assume that $\forall_i$ $I(c) \neq I(Q_i)$. Then, $\forall_i$ such that c is added to $G(Q_i)$, $I(Q_i) \neq I'(Q_i)$. In other words, only those queries where c is explicitly added may return a different answer.

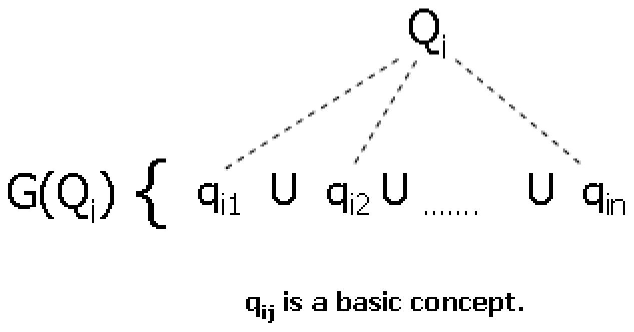

Q_i
$G(Q_i) \{ q_{i1} \cup q_{i2} \cup \ldots \cup q_{in}$
q_{ij} is a basic concept.

Figure 20. Query-centric Approach Example.
The query Qi is defined in terms of basic concepts (qij) and the union operation.

In the Source-centric approach, all queries Q_i are defined in terms of a set of global concepts q_i as is shown in Figure 21. c is now defined as the union of a set of global concepts c_i and also assumes that $\forall_i$ $I(c) \neq I(Q_i)$. Thus, $\forall_i$ such that $\exists_{j,k}$ $c_j = q_{ik}$ implies that $I(Q_i) \neq I'(Q_i)$. Thus, although their definition has not been modified, some queries Q_i may return a different answer when they share some global concepts with the set of global concepts belonging to the new basic concept c.

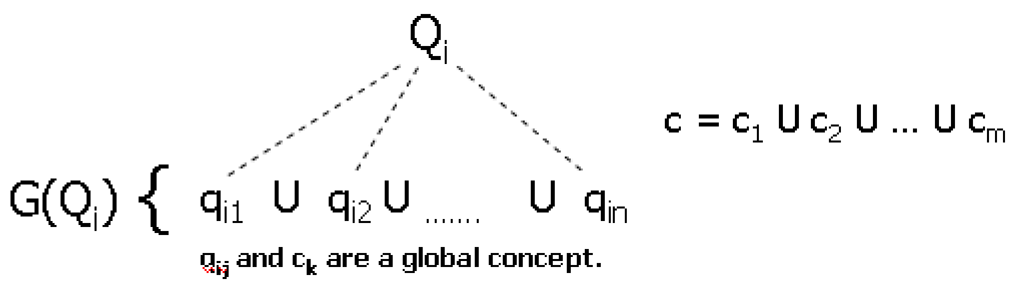

Q_i
$G(Q_i)$ { q_{i1} U q_{i2} U .... U q_{in}
$c = c_1$ U c_2 U ... U c_m
q_{ij} and c_k are a global concept.

Figure 21. Source-centric Example.
In this technique, the query Qi is defined in terms of global concepts (qij).

There are situations in which it is desirable for users to be able to control whether or not a newly added concept must automatically affect the results of previously defined queries. Consider the case of an internet user looking for information about cars. Assume that he/she has defined a query Q returning the set of cars that sell for under $4.000. Suppose that a new data source has become available. Then when Q is posted again, the user will expect that the information stored in the newly added data be included as an answer to Q. Note that the definition of Q is not modified. In this case, a Source-centric approach may be desirable.

Now, consider the case of a biological laboratory where users have defined a set of queries Q returning different information about proteins. Assume that a new data source A is available. In this case, users may want to decide if the new data source must be included in any particular Q or not. The decision for including the newly added data source to the system may be based on the way that the proteins have been classified by the data source. In general, in a scientific discovery environment, users are expected to determine when a new data source must be added to a set of particular queries or not. Therefore, in such an environment, a Query-centric approach may be preferable to a Source-centric approach.

2.3.6. Expressive Power of the Query Language vs. Data Source Autonomy

In a scientific discovery environment, Data sources are typically autonomous and heterogeneous. Consequently, each data source may offer different query facilities. For example, if a data source is based on a relational database and a full query interface is offered to other applications or users, the possible set of queries that this data source is able to answer is limited only by the expressive power of SQL. Compare the previous situation with a data source that uses a web interface where only a particular kind of query can be posted. In this second case, the expressive power of the query language is restricted to only this kind of query.

This situation affects both the Query-centric approach as well as the Source-centric approach. Consider the expression tree presented in Figure 22 for the Query-centric case. Focus on the data source *D*. Assuming that the required attributes for the projection operations are provided, the query shown can be resolved in at least two ways:

a) The *D* data source allows a user to gather the complete set of instances stored in D, in which case the selection operation must be applied at the central repository,
b) An interface that allow users to define a condition (Kind='Protein') is provided in which case the data source returns the set of instances matching the specified condition.

In summary, if the query capabilities of the data sources are not taken into account, a rewriting may not be able to answer the original query because there is not enough power in the query language of the data sources [Rajaraman et al., 1995; Horrocks et al., 2000; Vassalos et al., 1997].

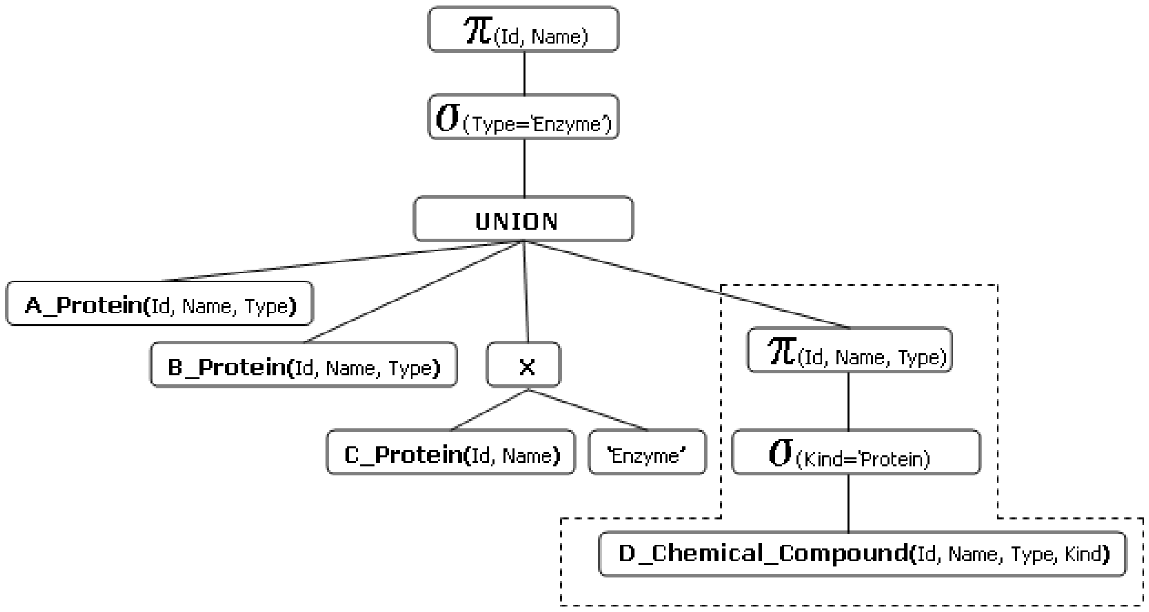


Figure 22. Query Capabilities and Query-centric Approach.

After a query is rewritten in terms of basic concepts using a Query-centric approach, the selection operation applied over a leaf may require the extraction of a set of instances matching a particular condition. If the correspondent data sources storing the instances of the basic concept does not offer a mechanism for extracting that particular set of instances (or any superset), answering the posted query may be not possible.

In general, the goal in an environment where data sources have restricted query capabilities is to describe which variables (attributes) must be bound prior to the execution of a query over a data source. Rajaraman et al [1995] use a template representing this information where the name of each variable that needs to be bound has '$' as prefix. Ullman [1998] uses an *adornment* in a DataLog program to specify which of the variables (of the head) needs to be bound as well as those that are free.

Thus, query rewriting algorithm must take into account the query capabilities to ensure that the rewriting is sound and complete.

Rajaraman et al [1995] prove that the conjunctive rewriting Q' from a conjunctive query Q with *m* variables and *p* sub goals will have at most *m + p* sub goals using at most *m* different variables. Consequently, there is an algorithm for finding the rewriting Q' of Q in NP time.

The restricted query capabilities of data sources also have an impact on the query performance. Sometimes, performance can be improved dramatically if the appropriate access point is chosen in order to answer a query. This is illustrated by the following example:

Let **q** be a global query asking for the names of all proteins discovered by Benson.

q(name) :- proteins(name, author)$\wedge$(author='Benson').

Assume that there is only one data source **A** providing this information. Suppose that **A** offers two access points: one requires downloading the complete file, and the other returns a flat file with all the proteins discovered by a given author.

Assume that there are 100 rows in the complete file, with only 5 matching the condition. So, if the first alternative is chosen, 100 instances are going to be transported from the data source to the central repository. Additionally, a selection operation must be applied in the central repository in order to pick out the instances that satisfying the condition *author='Benson'*. On the other hand, if the second alternative is chosen, only 5 instances are transported without the need for applying a selection operation in the central repository.

At this point, the ontological layer allows the system to resolve several semantic heterogeneities by allowing users to create or define new concepts or queries using concepts of the global ontologies. Also, in conjunction with the physical layer, basic concepts can be defined hiding all the complexity associated with the process for gathering their information. Several issues about how to transform global queries in terms of basic concepts was also discussed. The next chapter will describe our solutions for these and other problems in our INDUS prototype.

3. INDUS System

Data integration within a scientific discovery environment presents several challenges including the following: Data sources are autonomous, distributed, and heterogeneous in structure and content; the complexity associated with accessing the data answering queries must be hidden from the users; the users need to be able to view disparate data sources from their own point of view.

In this chapter we will briefly describe each of these problems (and others) along with our proposed solutions. After that, a formalization of the theoretic framework will be provided. Finally, the INDUS prototype will be presented.

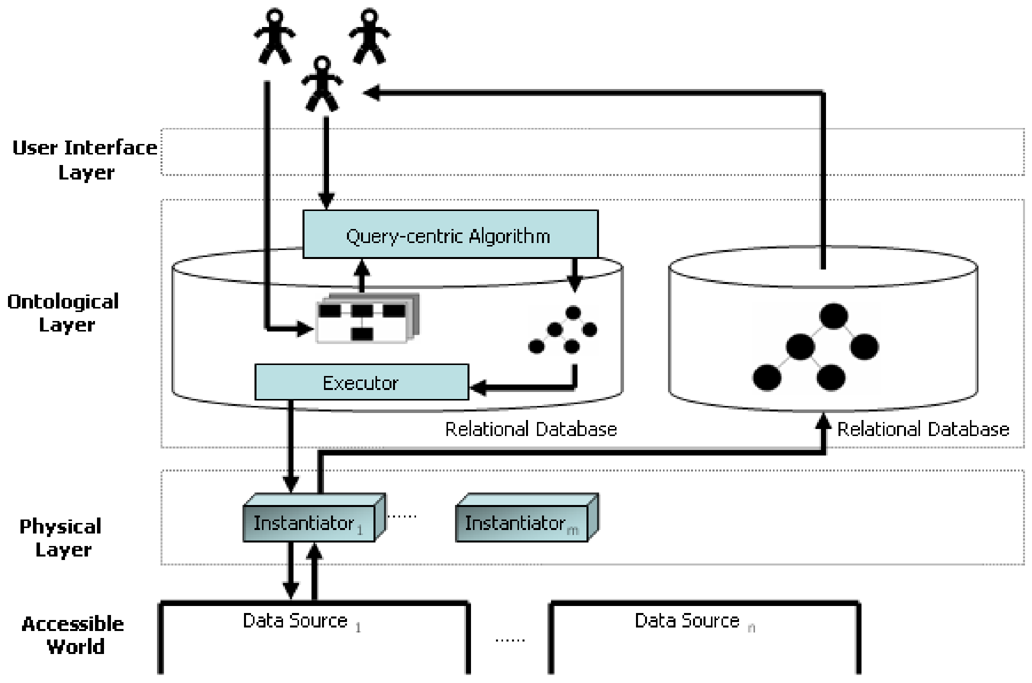


Figure 23. INDUS Schematic Diagram.

INDUS consists of three principal layers. In the lower part, the set of data sources accessible by INDUS are shown. In the physical layer, a set of instantiators enable INDUS to communicate (post queries and extract instances) with the data sources. The ontological layer offers a repository where ontologies are stored. Using these repository syntactical and semantic heterogeneities may be solved. Also, another relational database system is used to implement the user workspace private area where users materialize their queries. The user interface layer enables users to interact with the system.

Figure 23 presents a schematic diagram for INDUS. INDUS consists of three principal layers which together provide a solution to the data integration problem in a scientific discovery environment: The physical layer allows the system to communicate with data sources. This layer is based on a federated database architecture. The ontological layer permits users to define one or more global ontologies and also to automatically transform queries into execution plans using a query-centric approach. The planes describe what information to extract from each data source and how to combine the results. Finally, the user interface layer enable users to interact with the system, define ontologies, post queries and receive answers.

The physical layer implements a set of instantiators, i.e., allows interaction with each data source including constraints imposed by their autonomy and limited query capabilities. These allow the central repository to view disparate data sources as if they were a set of tables (relations). New iterators may be added to the system when needed (e.g. a new kind of data source has become available, the functionality offered by a data source has changed).

The ontological layer consists of the following:

- A meta-model, developed under a relational database, allowing users to define one or more global ontologies of any size. New concepts can be defined in terms of existing concepts using a set of compositional operations. The set of operations is extensible allowing users to add new operations as needed.
- An interface for defining queries based on concepts in a global ontology.
- An algorithm, based on a query-centric approach, for transforming those queries in an executable plan.
- An algorithm that executes a plan by invoking the appropriated set of instantiators and combining the results.
- A repository for materialized (instantiated) plans which allow users to inspect them if necessary.

The user interface layer allows users to work with the ontological layer to define global ontologies or to post queries. Also, the materialization of an executed plan can be inspected.

In what follows, we briefly describe the key problems that arise in data integration within a scientific discovery environment along with the proposed solutions in INDUS.

Data source autonomy: Data sources of interest are autonomously owned and operated. Consequently, the range of operations that can be performed on the data sources (e.g. the types of queries allowed), and the precise mode of allowed interactions can be quite diverse. In order to address this problem, INDUS uses a set of instantiators in the physical layer. These instantiators accomplish two principal functions: They are able to deal with the structural heterogeneity associated with each data source (e.g. relational database, flat file, etc.) and present to the central repository a homogenous structure which is based on the relational paradigm. Also, the instantiators along with the ontological layer are able to handle the limited query capabilities that each data source may have associated. Thus, when a query is posted, this is transformed into a plan by the ontological layer. When the plan is constructed, information is introduced regarding what operations to apply in the data source or in the central repository. The corresponding iterator is executed in order to apply those operations that the data source is able to resolve. The other operations are resolved in the central repository by the execution algorithm.

Data source heterogeneity: There are at least two types of data source heterogeneity: Syntactical heterogeneity arises when the underlying technology supporting data sources differs (e.g. web based interface, ftp files). This kind of heterogeneity is hidden from the users by the instantiators. Thus, each data source is tread as though it has the same structure, i.e. a set of tables. The second kind of heterogeneity arises at semantic level. Semantic heterogeneity may be due to differences in source ontologies, choice of units for attribute values, and nomenclature or vocabulary used. INDUS provides two principal mechanisms for resolving the semantic heterogeneity: An extensible set of compositional operations that allow users to define new concepts in terms of others (e.g. to define a new concepts in terms of the union of other two), and an extensible library of functions (including user-defined functions) that allow transformations of instances or attribute values (e.g. transforming values defined in kilograms into pounds).

Distributed data sources: Data sources are large in size and physically distributed. Consequently, it is neither desirable nor feasible to gather all of the data in a centralized location for analysis. Therefore, a federated database architecture was chosen for the implementation of the physical layer. However, some of the queries that users post can return large amounts of data. Therefore, the federated database architecture was expanded to include a relational database for storing the results of queries allowing further manipulation of those data using SQL.

Multiple Global Ontologies: In scientific discovery applications users often need to examine data in different contexts and from different perspectives. Hence, there is generally no single universal ontology that serves the needs of all users, or for that matter, even a single user, in every context. Standardized domain ontologies e.g. biological ontologies [Ashburner et al., 2000; Karp, 2000], while they are very useful for information sharing using a common vocabulary, do not obviate the users need to construct their own ontologies. In particular, it is useful for users to be able to combine concepts defined in the shared domain ontology or ontologies that are implicit in the design of different data repositories. Once the implicit ontologies associated with the data sources are made explicit, a user can exploit them to dynamically assemble the information that is of interest by establishing correspondence between the concepts in his or her ontology with those of each data source. Thus, making the ontological commitments explicit supports the context-dependent dynamic information integration from distributed data. INDUS offers users the ability to define one or more global ontologies. These global ontologies are stored in a meta-model architecture based on a relational database system. This allows sharing of ontologies among users.

Transparent Query Resolution: One of the principal goals of a data integration system is to hide the complexity associated with the process of integrating results from heterogeneous data sources as an answer for a query. In INDUS, the ontological layer and the physical layer allow users to post queries in terms of familiar concepts in the ontology. The posted queries are resolved by a query-centric algorithm by invoking the iterators and additional needed operations. The execution algorithm follows the plan and obtains the results in the form of a set of populated tables and views.

In what follows, a formal description of the elements that constitute INDUS will be provided.

3.1. INDUS Formalization

INDUS is an ontology driven data integration system which allows users to access information stored in heterogeneous, distributed and autonomous data sources. The system provides an ontological layer for defining multiple global ontologies, each representing a unified and particular view of the information stored in data sources. After a global model has been defined, it is possible to post queries on it. The results are transported, transformed and integrated according to the global ontology structure. All the complexity associated with the process of gathering the information is hidden from the final user.

Queries are posted on concepts that belong to the global ontologies using a particular language. As follows, we introduce formally the notion of a concept in INDUS.

A concept in INDUS is equivalent to the mathematical entity for a relation underlying the relational model [Ullman, 1998]. Thus, a concept is a subset of the Cartesian product of a list of domains. A domain is a set of values, and for this particular environment they are assumed as a finite but unknown. So, if a concept is based on 3 domains $D_1=\{a, b\}$, $D_2=\{b\}$, $D_3=\{d, e\}$, the Cartesian product of D_1 and D_2 and D_3 , written $D_1 \times D_2 \times D_3$, would be the set of all three-tuples (x_1, x_2, x_3) where $x_1 \in D_1$, $x_2 \in D_2$, $x_3 \in D_3$. Thus, $D_1 \times D_2 \times D_3 = \{ (a,b,d), (a,b,e), (b,b,d), (b,b,e)\}$.

Formally, if X is a concept, the structure of a concept, $\mathrm{X}._{\text{atts}}$, is described by the list of domains, from now on referred to as attributes. The elements of this list are drawn from Θ, the set of all domains. The i-th element of $\mathrm{X}._{\text{atts}}$ is represented by $\mathrm{X}._{\text{atts[i]}}$; its name by $\mathrm{X}._{\text{atts[i].name}}$ and the associated domain by $\mathrm{X}._{\text{atts[i].domain}}$.

Figure 24 shows a graphical representation of the structure of the concept X, when $\mathrm{X}._{\text{atts}}$ = (("name", ***S***), ("age", ***N***)) where ***S*** represent the set of strings and ***N*** represent the set of natural numbers.

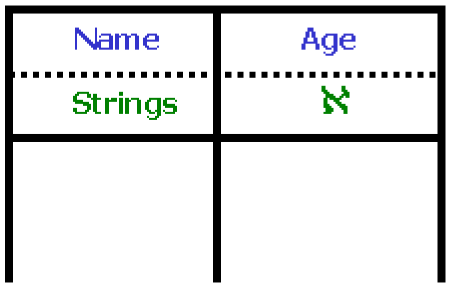


Figure 24. The Structure of a Concept

The extensional definition of a concept X, denoted by $X._{insts}$, is the enumeration of all instances from the Cartesian products of the domains D_i, such that $D_i \in X._{atts}$. Each instance is represented by a list of attribute values. As an example, a concept X, based on $\boldsymbol{D_1 \times D_2 \times D_3}$ *= { (a,b,d), (a,b,e), (b,b,d), (b,b,e)}*, may be extensionally defined as $X._{insts}$ *= { (a,b,d), (a,b,e)}*.

The intentional definition of a concept X, $X._I$, is a description of instances that belongs to the concept. Thus, an intentional definition for the concept X may be: $X._I$ = { $i \in D_1 \times D_2 \times D_3$ / *the first element of the i tuple is an 'a'*}. In general, intentional definitions offer a shorter representation than extensional definitions.

Obtaining the extensional definition of a concept is possible under the following conditions:

- The intentional definition provides a decidable process implementing a membership function (a function that given any instance returns *true* when it belongs to the concept and *false* otherwise).
- Access to the universe of instances is provided.
- The universe of instances is finite (when it is infinite partial results may be returned).

An algorithm may verify, using the membership function, which instances satisfy the conditions provided by the intentional definition. This subset of instances corresponds to the extensional definition of the concept.

An operational definition of a concept X, $X._D$, is a procedure that specifies how to obtain the set of instances of X.

Concepts on INDUS are of two types: ground concepts and compound concepts. Ground concepts are those whose instances can be retrieved from one or more data sources using a set of pre-defined operations. Instances of compound concepts are obtained from the set of instances of other concepts in the system by applying the necessary compositional operations.

3.1.1. Ground Concepts

The operational definition of a ground concept describes a procedure for retrieving $X._{inst}$ from a set of relevant data sources. In order to accomplish this task, the system 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.

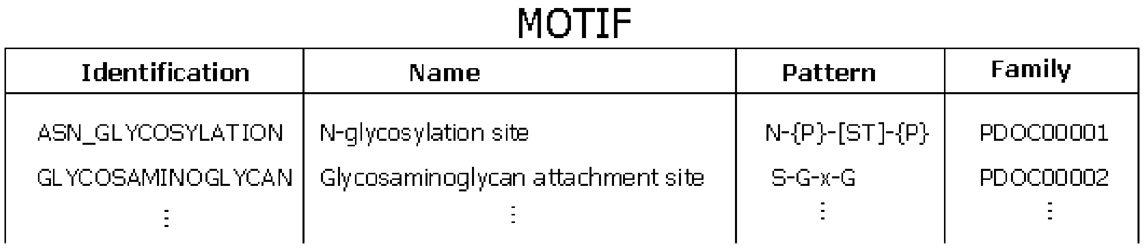
Then, the operational definition of a ground concept X corresponds to a set of *instantiators*, $X._D$.

The core of an instantiator is an *iterator*. *Iterators* are programs that provide the following:

- A set of parameters that allow INDUS to control the behavior of the iterator itself.
- A method that retrieves instances based on the parameters (e.g., the data source where to retrieve instances)
- A structure for representing a returned instance. INDUS is able to assemble $X._{inst}$ using the instances returned by the correspondent instantiator.

In summary, when a ground concept X is to be instantiated, INDUS selects and invokes one instantiator $\iota \in X._D$. Invoking ι entails execution of the correspondent iterator $\iota.\tau$ by supplying the values of the arguments of $\iota.\tau$. The instances of X, i.e., $X._{inst}$, are assembled by mapping each instance returned by the instantiator ι into the structure specified by $X._{atts}$

The following example illustrates how an operational definition of a ground concept is created. . In biology, the concept *motif* corresponds to a pattern that is shared by a set of DNA or protein sequences. Such motifs may be described by an id, a name, a regular expression, and a family. Figure 25 presents a graphical representation of the *motif* concept.



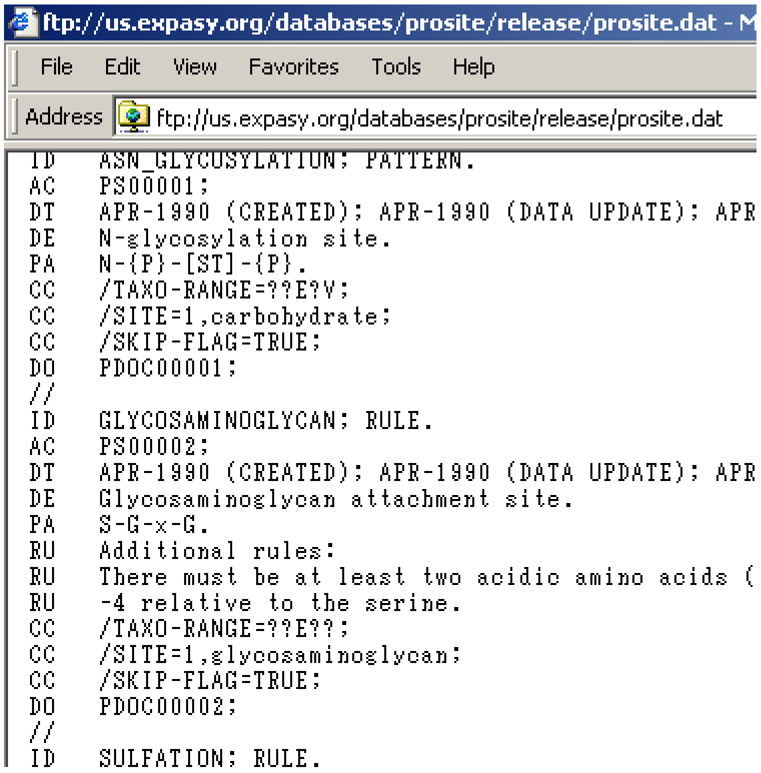
MOTIF

Identification	Name	Pattern	Family
ASN_GLYCOSYLATION	N-glycosylation site	N-{P}-[ST]-{P}	PDOC00001
GLYCOSAMINOGLYCAN	Glycosaminoglycan attachment site	S-G-x-G	PDOC00002
⋮	⋮	⋮	⋮

Figure 25. The *MOTIF* Ground Concept.

The structure of the MOTIF concept is composed by four attributes (Identification, Name, Pattern, and Family). Some examples of instances are presented.

Suppose that two data sources *A* and *B* store instances for this concept. The data source *A* returns all instances of the MOTIF concept as a flat file using an ftp protocol (shown in Figure 26). Consider the iterator I_X which is able to interact with the data sources data source *A*.


ftp://us.expasy.org/databases/prosite/release/prosite.dat - M
File Edit View Favorites Tools Help
Address ftp://us.expasy.org/databases/prosite/release/prosite.dat

```
ID   ASN_GLYCOSYLATION; PATTERN.
AC   PS00001;
DT   APR-1990 (CREATED); APR-1990 (DATA UPDATE); APR
DE   N-glycosylation site.
PA   N-{P}-[ST]-{P}.
CC   /TAXO-RANGE=??E?V;
CC   /SITE=1,carbohydrate;
CC   /SKIP-FLAG=TRUE;
DO   PDOC00001;
//
ID   GLYCOSAMINOGLYCAN; RULE.
AC   PS00002;
DT   APR-1990 (CREATED); APR-1990 (DATA UPDATE); APR
DE   Glycosaminoglycan attachment site.
PA   S-G-x-G.
RU   Additional rules:
RU   There must be at least two acidic amino acids (
RU   -4 relative to the serine.
CC   /TAXO-RANGE=??E??;
CC   /SITE=1,glycosaminoglycan;
CC   /SKIP-FLAG=TRUE;
DO   PDOC00002;
//
ID   SULFATION; RULE.
```

Figure 26. Data Source Accessed by *Ix*(<u>*http://A*</u>).

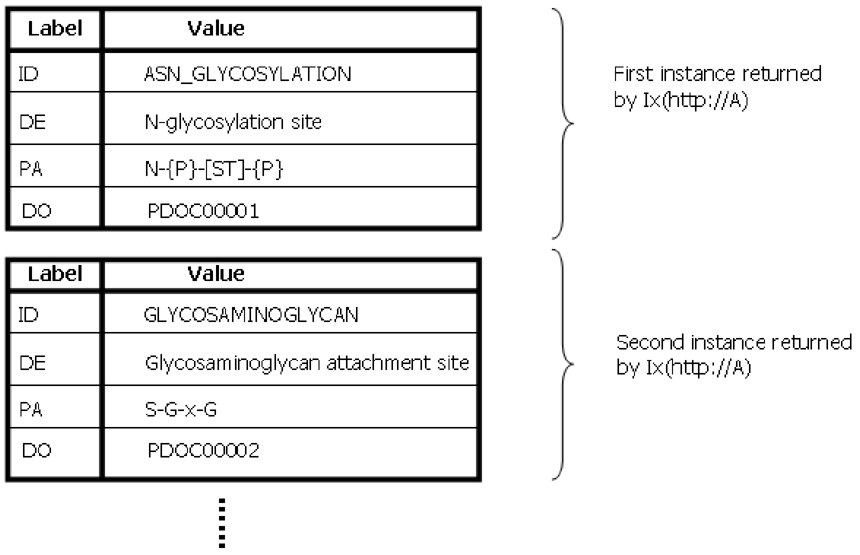
When the *Ix* iterator is called assigning a value of <u>*http://A*</u> to the URL parameter, this file is returned. Each instance begins with an *ID* label and ends with a // label.

Suppose I_X needs a single parameter indicating the *URL* of the data source to be accessed. Figure 27 presents a couple of examples of the format used by the iterator I_X for returning instances from *A* (e.g., I_X*(http://A)*).

The structure of the concept MOTIF and the structure used by Ix(http://A) for storing instances do not match. Hence, a mapping between the two structures must be specified. Such a mapping is shown in Figure 28.

The instantiator ι_A of the concept MOTIF corresponds to the iterator I_X called with the parameter *http://A* using the mapping shown in Figure 28 for transforming the instances returned by *Ix* into instances of the concept MOTIF.

Consider a data source *B* which can be accessed though the web interface shown in Figure 29. Unlike data source *A*, *B* requires that the user provide the identification number of the instance to retrieve. The I_Y iterator is able to interact with B through such an interface.



Label	Value
ID	ASN_GLYCOSYLATION
DE	N-glycosylation site
PA	N-{P}-[ST]-{P}
DO	PDOC00001

Label	Value
ID	GLYCOSAMINOGLYCAN
DE	Glycosaminoglycan attachment site
PA	S-G-x-G
DO	PDOC00002

First instance returned by Ix(http://A)
Second instance returned by Ix(http://A)

Figure 27. Examples of Instances Returned by Ix(http://A).

Each instances found by the iterator is returned in an independent vector. Each vector has two columns: *Label* and *Value*. *Label* indicates the kind of information that is begin returned while *Value* indicates the information itself. INDUS will call a method associated with the iterator for returning each instance.

One of the parameter needed by I_Y is the URL of the data source to be accessed. Figure 30 shows the structure returned by the data source when I_Y is executed as I_Y*(http://B)* and the value ASN_GLYCOSYLATION is provided.

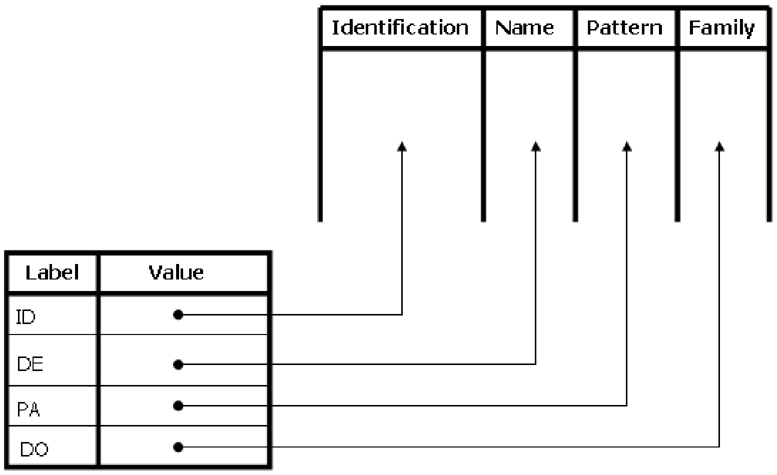
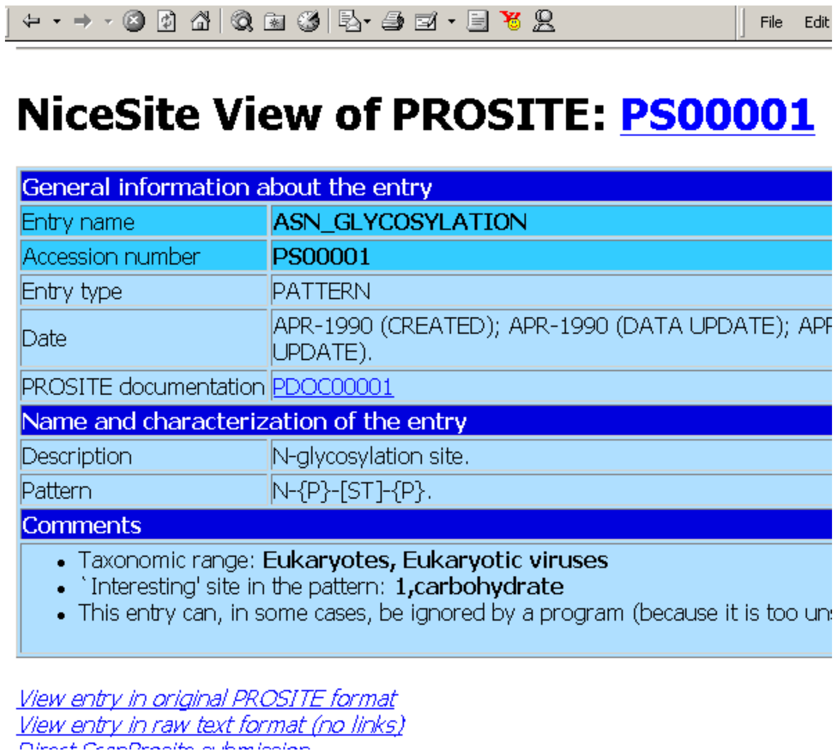


Figure 28. Mapping Between Ix(http://A) and *MOTIF* Concept.

The mapping information allows INDUS to assign correctly a returned vector (i.e., an instance) to the set of instances of the correspondent concept.



NiceSite View of PROSITE: PS00001

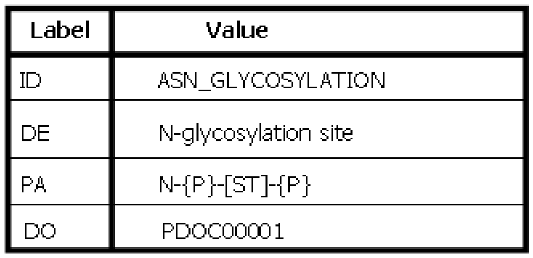
General information about the entry	
Entry name	ASN_GLYCOSYLATION
Accession number	PS00001
Entry type	PATTERN
Date	APR-1990 (CREATED); APR-1990 (DATA UPDATE); APF UPDATE).
PROSITE documentation	PDOC00001
Name and characterization of the entry	
Description	N-glycosylation site.
Pattern	N-{P}-[ST]-{P}.
Comments	

- Taxonomic range: **Eukaryotes, Eukaryotic viruses**
- `Interesting' site in the pattern: **1,carbohydrate**
- This entry can, in some cases, be ignored by a program (because it is too un

View entry in original PROSITE format
View entry in raw text format (no links)

Figure 29. The *B* Data Source.

The *B* data source interacts with a web page that returns a particular instance that match a given ID.



Label	Value
ID	ASN_GLYCOSYLATION
DE	N-glycosylation site
PA	N-{P}-[ST]-{P}
DO	PDOC00001

Figure 30. Instance Returned by I_Y(http://B).

Similar to the previous case, the structure of the concept MOTIF and the structure used by I_Y(http://B) do not match. The required mapping is shown in Figure 28.

The interface for the *B* data source requires that the user supplies the ID of the instances to be retrieved.

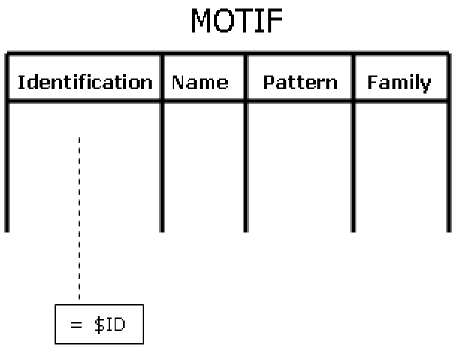


Figure 31. Graphical Representation of the Queries Accepted by *B*.
The *B* data source returns instances that match a provided ID. The '=' indicates that only those instances storing an exact match to the given ID are returned.

Figure 31 presents a graphical representation of the kind of queries that the data source *B* can answer. In this particular case, the query capabilities of *B* can be described indicating the attribute where a value is required (identification) and the operation (=) is involved in the condition. This means that this data source requires a value for the Identification attribute, and it will return all instances that match the specified Identification value. The matching is performed using the equality (=) operator. Similarly, note that the query capabilities of the data source A are equivalent to queries over the concept MOTIF without conditions (it always returns all instances).

In our current INDUS prototype, the query capabilities associated with instantiators are described by a list of conditions, where each condition indicates the attribute and the operation to be applied. The set of instances to be retrieved must match all conditions associated with the query capabilities of the data source. Therefore, the set of conditions are treated as a conjunction of conditions.

The instantiator ι_B of the concept MOTIF corresponds to the following:

- the iterator I_Y called with the parameter *http://B*,
- using the mapping shown in Figure 28 for transforming the instances returned by I_Y into instances of the concept MOTIF, and
- the condition (ID =) indicating that this data source requires a value for the attribute ID and returns all instances that match the provided value.

The operational definition of the MOTIF concept, MOTIF.$_D$, corresponds to the set of instantiators $\{\iota._A, \iota._B\}$.

Now, we are going to proceed to formalize the notions of iterator, instantiator and the operational definition of a ground concept.

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 T be the set of iterators provided by INDUS. If $\tau \in T$, then $\tau._{name}$ corresponds to the name of the program to be executed and $\tau._{param}$ specifies the list of parameters associated with τ. 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 iterator's 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 though this instantiator.

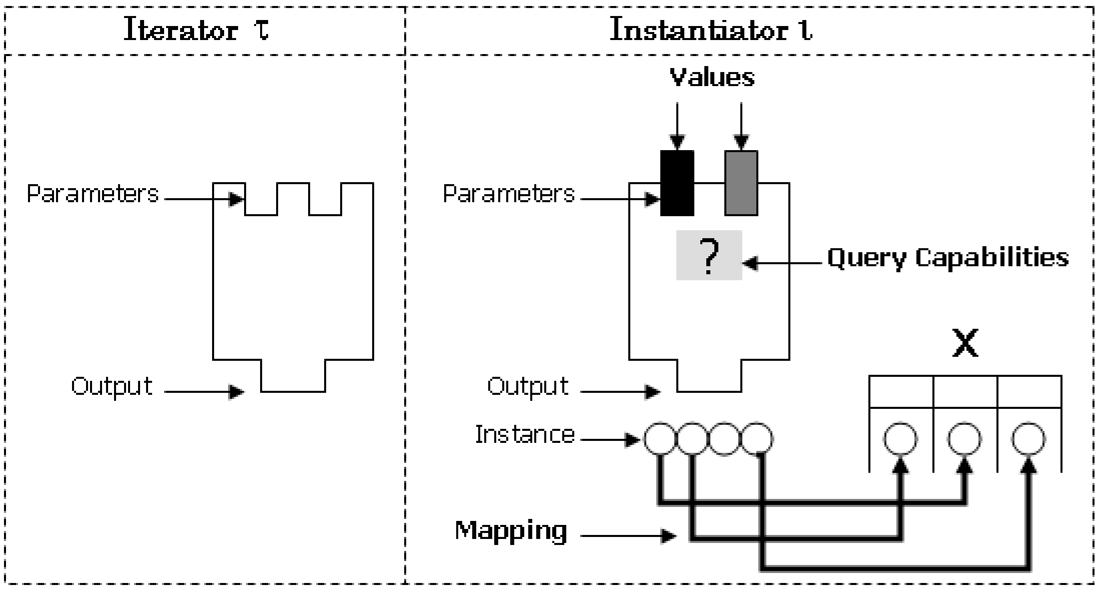


Figure 32. Iterators and Instantiators.

Iterators are java classes that are able to interact with data sources. They have a set of parameters that controls their behavior. All *iterators* return instances following a pre-defined format (based on a vector). An *instantiator* is a component based on an *iterator*. The *instantiator* provides values for the required parameters of the correspondent *iterator* in order to fix its behavior. In addition to that, information indicating the query capabilities offered by the data source (base on the values assigned to the parameters of the iterator) are described by the *instantiator*. Finally, the *instantiator* describes how to assign a returned instance to the related ground concept.

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 $\iota \in X._D$ then:

- The concept associated with the instantiator must be X ($\iota._{concept} \equiv X$).
- $\iota._{iterator} \in T$.
- $\iota._{values}$ 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 ith-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 correspondent $X._{atts[i]}$.

- $\iota_{\text{.queryCapabilities}}$ is the list of conditions associated with the instantiator ι where the ith-element of the list is $\iota_{\text{.queryCapaibilities[i]}}$. If $b \in \iota_{\text{.queryCapabilities}}$, then $b_{\text{.attribute}} \in X_{\text{.atts}}$ and $b_{\text{.operator}} \in O$, where O is the set of operators supplied by INDUS.

For the example shown above, if the concept X represents the MOTIF concept shown previously, the following is the operational definition for X, $X_{\text{.D}}$.

Let MOTIF be a ground concept such that:

- $\text{MOTIF}_{\text{.atts[1].name}} \leftarrow$ Identification,
- $\text{MOTIF}_{\text{.atts[2].name}} \leftarrow$ Name,
- $\text{MOTIF}_{\text{.atts[3].name}} \leftarrow$ Pattern,
- $\text{MOTIF}_{\text{.atts[4].name}} \leftarrow$ Family, and
- $\forall_i\ \text{MOTIF}_{\text{.atts[i].domain}} \leftarrow \mathbf{S}$, the set of strings.

Let τ_1 and τ_2 iterators of T, such that:

- $\tau_{1\text{.name}} \leftarrow I_X$, $\tau_{1\text{.param}} \leftarrow$ ("URL") and
- $\tau_{2\text{.name}} \leftarrow I_Y$, $\tau_{1\text{.param}} \leftarrow$ ("URL")

Then, the operational definition of MOTIF, $\text{MOTIF}_{\text{.D}}$ is formally defined as follows:
$\text{MOTIF}_{\text{.D}} \leftarrow \{\iota_1, \iota_2\}$

Where the first instantiator is defined as follows:

- $\iota_{1\text{.concept}} \leftarrow$ MOTIF
- $\iota_{1\text{.iterator}} \leftarrow \tau_1$
- $\iota_{1\text{.values}} \leftarrow$ ("http://A")
- $\iota_{1\text{.mapping[1]}} \leftarrow$ ID, $\iota_{1\text{.mapping[2]}} \leftarrow$ DE, $\iota_{1\text{.mapping[1]}} \leftarrow$ PA, $\iota_{1\text{.mapping[1]}} \leftarrow$ DO,
- $\iota_{1\text{.queryCapabilities}} \leftarrow \perp$, where $\perp$ indicates a empty list.

The second instantiator is defined as follows:

- $\iota_{2\text{.concept}} \leftarrow$ MOTIF
- $\iota_{2\text{.iterator}} \leftarrow \tau_2$
- $\iota_{2\text{.values}} \leftarrow$ ("http://B")
- $\iota_{2\text{.mapping[1]}} \leftarrow$ ID, $\iota_{2\text{.mapping[2]}} \leftarrow$ DE, $\iota_{2\text{.mapping[1]}} \leftarrow$ PA, $\iota_{2\text{.mapping[1]}} \leftarrow$ DO,
- $\iota_{2\text{.queryCapabilities[1].attribute}} \leftarrow$ Identification and $\iota_{2\text{.queryCapabilities[1].attribute}} \leftarrow =$

3.1.2. Compound Concepts

The operational definition of a compound concept X specifies the set of operations that must be applied over the set of instances of other previously defined concepts in order to determine the set of instances of the concept X.

Four operations in INDUS are provided to facilitate the operational definition of compound concept: selection, projection, vertical integration, horizontal integration.

Selection

Selection operation (denoted by σ) defines the set of instances of a concept based on a subset of instances of a previously defined concept as is shown in Figure 33. The structure of the two concepts must be equivalent, that is, the list of domains of both concepts must be the same.

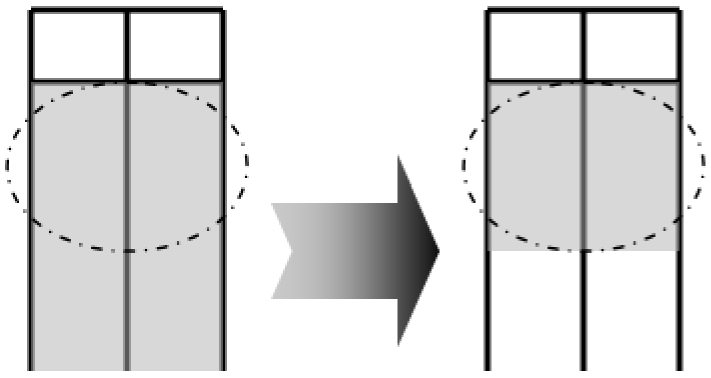


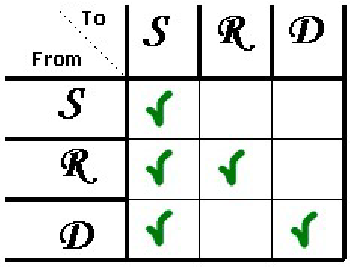
Figure 33. Selection Example.

Formally, if X and Y are concepts, X <u>has an equivalent structure with</u> Y if:

- $|X_{.atts}| = |Y_{.atts}|$, and

- $\forall_{1 \leq i \leq |X.A|}$ the ith-element of both lists has the same domain or there is a natural transformation between the corresponding domain of X and Y.

A natural domain transformation is a predefined procedure to convert any value belonging to a particular domain to another domain. Then, a table where valid transformations are described must be provided to the system. An example for this table is shown in Figure 34. In this table, a transformation from the $\mathcal{S}$ to the $\mathcal{R}$ domains is allowed. On the other hand, no transformation is allowed from the $\mathcal{R}$ to the $\mathcal{D}$ domains.



From \ To	$\mathcal{S}$	$\mathcal{R}$	$\mathcal{D}$
$\mathcal{S}$	✓		
$\mathcal{R}$	✓	✓	
$\mathcal{D}$	✓		✓

Figure 34. Domain Transformation Example.

So, the selection operation is defined as follows:

Let X and Y be two concepts such that X has an equivalent structure with Y. 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 $function_1(attribute_1, attribute_2, \ldots)$. For example, Upper(Salary) = 33 is a valid built in predicate where the second argument corresponds to a constant.

Thus, the previous definition implies that the set of instances of Y is the set of instances of X that accomplish the condition expressed by *s*.

Consider a new concept Y defined in terms of the concept X:

$Y._D := \sigma_{Age=33} (X)$

Figure 35 presents the set of instances of the concepts X and Y using the operational definition provided above.

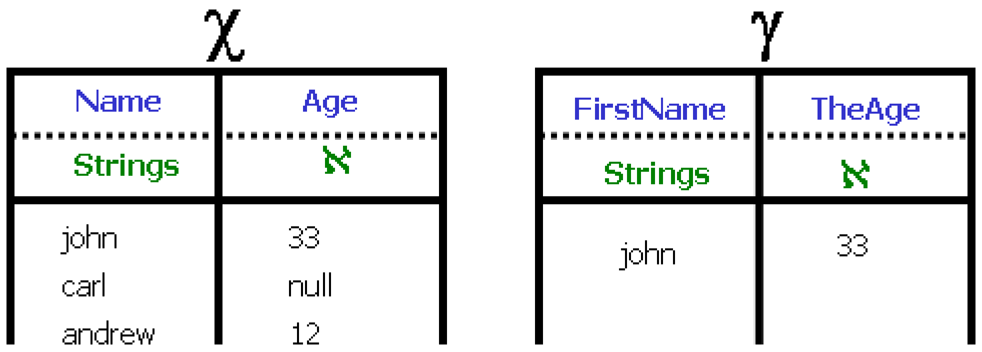


Figure 35. Selection Operation.

Projection

Projection operation, denoted by π, defines the set of instances of a concept in terms of the set of instances of other concept taking into account only a subset of the attributes.

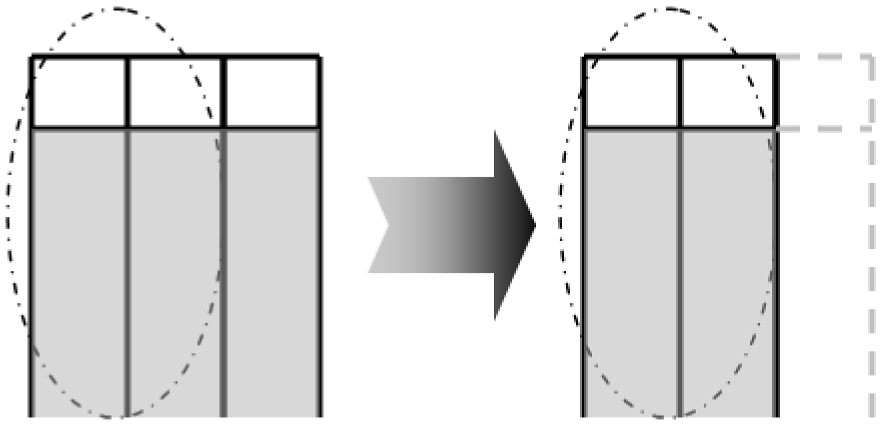


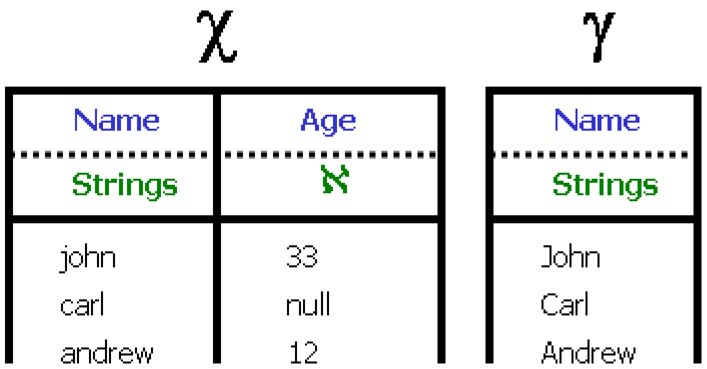
Figure 36. Projection Example.

Let *X* and *Y* be two concepts and let *q* be a list with equivalent structure with *Y*. The operational definition of Y defined in terms of a projection over the concept X is as follows:

$Y_{.D} := \pi_p(X)$ where **p** is a list of functions applied over X attributes.

This formula implies that, 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.

Then, $Y_{.D} := \pi_{(\text{InitCap(name)})}(X)$ will result in the set of instances shown in Figure 37.



Name	Age
Strings	ℵ
john	33
carl	null
andrew	12

Name
Strings
John
Carl
Andrew

Figure 37. Projection Operation.

Vertical Integration

Vertical integration of two concepts *A* and *B* in a new concept *AB* involves combining the each instance of *A* with each instance of *B* and maybe applying selection and projection operations. Figure 38 presents a graphical example of combining the concepts *A* and *B*, such that all *A*-instances that share the same value with the *B*-instances for the attribute *a1* are merged to form a new instance that will have only one column *a1* but all other columns of *A* and *B*.

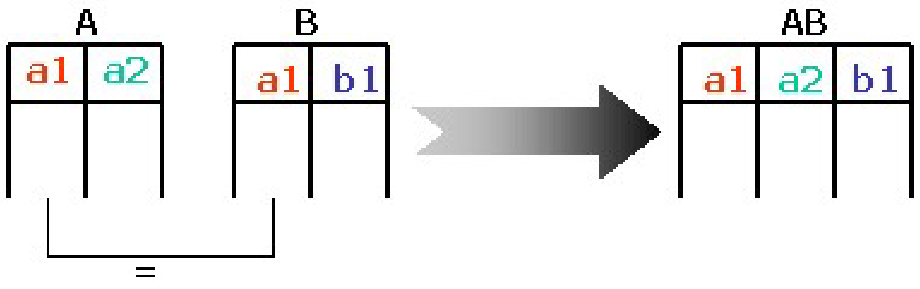


Figure 38. Vertical Integration and Selection Example.

This operation accomplishes an important role in a data integration environment because it provides a method for reversing a vertical fragmentation of data. Vertical fragmentation occurs when the concept has been split into two or more distributed nodes. The split operation normally creates a disjointed set of attributes (except some particular attributes that store enough information for recovering the original concept). Each of these sets in addition to the set of common attributes is located in a separated node.

This operation can be defined in terms of selection, projection and Cartesian product operations. The Cartesian product of the set of instances of concepts X and Y is obtained concatenating every instance of X with every instance of Y.

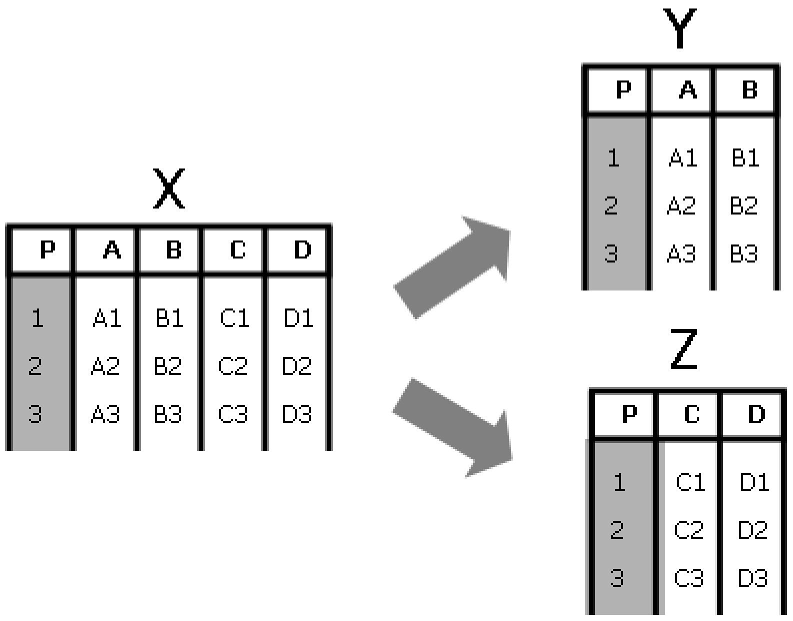


Figure 39. Vertical Fragmentation.

Recall that instances are represented as lists, so the concatenation of two lists *A* and *B* is only a new list with head *A* and tail *B* and the size of the concatenated list is equal to size of *A* plus size of *B*.

Formally it can be expressed as follows:

Let X, Y and Z be three concepts such that $|X._{atts}| = |Y._{atts}| \cdot |Z._{atts}|$ where "$\cdot$" represent the list concatenation operation. The operational definition of X is defined in terms of the Cartesian product of Y and Z, is as follows:

$X._D := Y \times Z$

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 $Y_1, Y_2, \ldots, Y_n$ in its intentional definition using a vertical integration operation, as follows:

$X._D := \pi_p (\sigma_s (Y_1 \times Y_2 \times \ldots \times Y_n))$

Note that the selection operation is a special case of a vertical integration operation with $n = 1$, and *p* retrieving all attributes of Y_1. Similarly, a projection operation can be defined in terms of a vertical integration operation with $n \leftarrow 1$ and *s* retrieving all rows (i.e., $p \leftarrow \perp$). This same situation happens with the Cartesian product.

Horizontal Integration

Horizontal fragmentation is another common situation that happens in a distributed environment. In this case, a set of instances of a concept X is distributed across several geographically dispersed servers or nodes as shown in Figure 40.

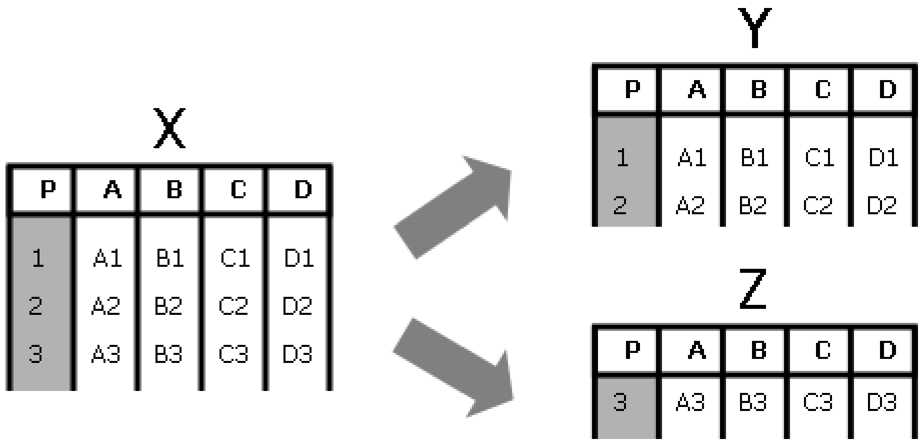


Figure 40. Horizontal Fragmentation.

The horizontal integration operation provides a solution for the horizontal fragmentation.

Formally, let X, Y, and Z be three concepts with equivalent structure. $X._I$, the operational definition of X, is defined in terms of the concepts Y and Z using a horizontal integration operation, as follows:

$X._D := Y \cup Z$

This formula states that the set of instances of X is obtained by the union operations of the set of instances of Y and Z.

A more general definition of a horizontal integration operation may include a selection, a projection and the union or more than two concepts, following the same format applied for a vertical integration operation. Therefore, if X is a concept, its operational definition can be based on a horizontal integration operation as follows:

$X._D := \pi_p (\sigma_s (Y_1 \cup Y_2 \cup \ldots \cup Y_n))$

We assume the union operation is applied over bags (not over sets) in our current prototype. This means that duplicated instances are not eliminated.

Using this definition, selection and projection operations are special cases of a horizontal integration operation, as in the case of the vertical integration operation.

Now, we are ready to introduce a general operational definition of a compound concept. Firstly, recall 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:

$X._D := \pi_p (\sigma_s (Y_1 \bullet Y_2 \bullet \ldots \bullet Y_n))$,

where X stands for a compound concept, *s* is the selection criteria, *p* is the projection criteria, 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 this first prototype).

Then, the operational definition of a concept X can be expressed in the following terms:

- the selection criteria *s*,
- the projection criteria *p*,
- the set of concepts where X is based on, Y_1 .. Y_n, and
- the compositional operation applied ($\bullet$).

If X is a compound concept, its operational definition is described by the following attributes:

- $X._{selection}$ is a list of selection criteria. The ith-selection criterion is detonated by $X._{selection[i]}$. If $s \in X._{selection}$ then:
    - $s._{function1} \in \Phi$, where Φ is the set of functions provided by INDUS,
    - $s._{attributes1}$ is a list of attributes that serves as arguments to $s._{function1}$. The ith-element of $s._{attributes1}$, $s._{attributes1[i]}$, corresponds to the ith-argument of $s._{function1}$. $s._{attributes1[i]} \in [\cup_j X._{concepts[j].atts}]$.
    - $s._{operator} \in O$, the set of operators defined in INDUS (=,>,<, etc.),
    - $s._{function2} \in \Phi$,
    - $s._{attributes2}$ is a list of attributes that serves as arguments to $s._{function2}$. The ith-element of $s._{attributes2}$, $s._{attributes2[i]}$, corresponds to the ith-argument of $s._{function2}$. $s._{attributes2[i]} \in [\cup_j X._{concepts[j].atts}]$.
- $X._{projection}$ is a list of projection criteria. The ith-projection criteria is detonated by $X._{projection[i]}$. If $p \in X._{projection}$ then:

  - $p._{\text{function}} \in \Phi$,
  - $p._{\text{attributes}}$ is a list of attributes that serves as arguments to $p._{\text{function}}$. The ith-element of $p._{\text{attributes}}$, $p._{\text{attributes[i]}}$, corresponds to the ith-argument of $p._{\text{function}}$. $p._{\text{attributes[i]}} \in [\cup_j \mathrm{X}._{\text{concepts[j].atts}}]$.
- $\mathrm{X}._{\text{concepts}}$ is the list of concepts where X obtains its set of instances (after applying the correspondent selection, projection, and concept operation). The ith-concept is denoted by $\mathrm{X}._{\text{concepts[i]}}$ and $\mathrm{X}._{\text{concepts[i]}} \in \mathrm{K}$, the set of concepts defined in INDUS.
- $\mathrm{X}._{\text{operation}} \in \Omega$, where Ω is the set of concept operation defined in INDUS. In this first prototype Ω has two elements: the cross product ($\times$) the union operation ($\cup$).

The operational definition of X indicates to INDUS how to retrieve the set of instances of X gathering instances of the concepts in $\mathrm{X}._{\text{concepts}}$ and applying the correspondent selection and projection operations correctly.

3.1.3. Why functions are Important in a Data Integration System?

Semantic and syntactic heterogeneity are one of the most important issues to be resolved by data integration systems. They appear naturally in a data integration environment, especially when data sources are autonomous and heterogeneous.

Some syntactic differences can be quit simple. As an example consider the creation of a new concept based on a vertical integration operation combining the classification for a protein based on the SCOP database and the information about proteins stored in Swissprot and Enzymes data bases, as shown in Figure 41. In order to combine the SCOPClassification and SwissprotProteinReference concepts, a vertical integration must be applied. This operation will require unifying the SwissprotProteinReferences.Reference and the SCOPClassifiaction.pdb attributes using an equality operation. Nevertheless, the information about pdb references in SCOP is stored in lower case while in Swissprot it is stored in uppercase. Therefore, no match will be found between any instances from both concepts using these two attributes. Although the meaning of both columns (reference and pdb) is the same, a syntactic transformation must be applied for transforming any data stored in pdb to its correspondent uppercase version. Hence, a function must be applied over the SCOPClassification.pdb attribute.

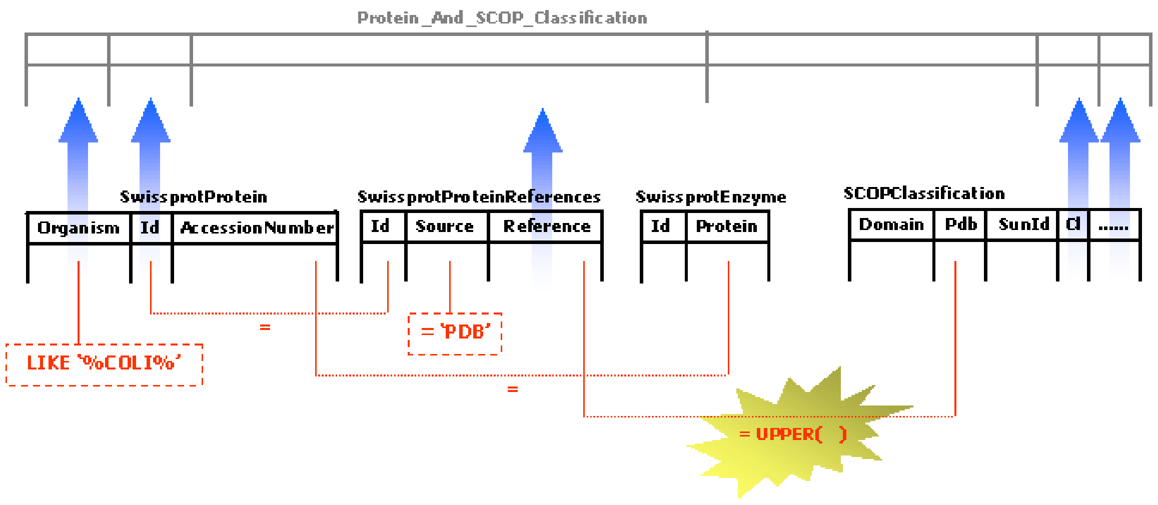


Figure 41. Functional Transformation for Vertical Integration.

In order to relate the classification provided by the Enzyme data source (represented by the SwissprotEnzyme ground concept) with the classification provided by the SCOP data source (represented by the SCOPClassification ground concept), a *pdb* id is required. Given an Enzyme, a pdb id is found in the references associated with the correspondent protein. The reference value stores the required pdb id used to find the relevant SCOP classification. In the SCOP database, the pdb ids are stored in lower case while in the reference field of the proteins it is stored in upper case. Therefore, a function is required to resolve this syntactic difference.

As another example, assume that two labs wants to unify information recollected from an experiment obtaining two measures: Mass and Temperature. Thus, two ground concepts (one for each data source) are available. Therefore, a compound concept may be defined in terms of the ground concepts using a union operation, as shown in Figure 42.

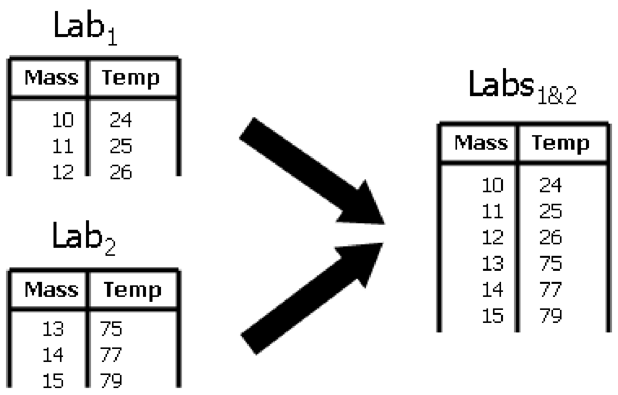
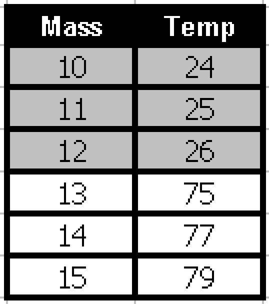


Figure 42. A Compound Concept Based on a Union Operation.



Mass	Temp
10	24
11	25
12	26
13	75
14	77
15	79

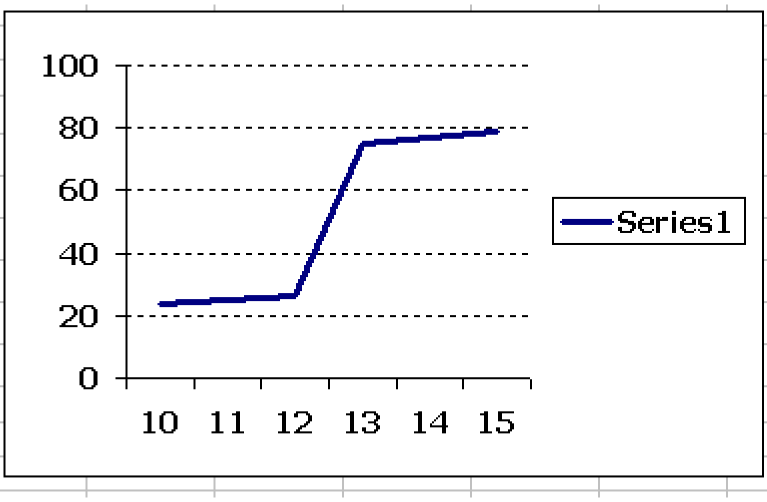


Figure 43. A problem Originated by a Semantic Heterogeneity.

If the information returned by two different laboratories (the gray values correspond to one laboratory while the white values to the other) is unified without verifying the units used by both laboratories (they should be the same), erroneous patterns may be found.

A close revision of the data reported by the two labs reveals that different units were used for representing the temperature information in both labs. While lab1 used Celsius degrees, the lab2 used Fahrenheit degrees.

Therefore, before combining instances from both labs, a transformation must be applied in the information reported by one of them changing the temperature unit to that used by the other lab as shown in Figure 44. The function "f" provides the needed transformation.

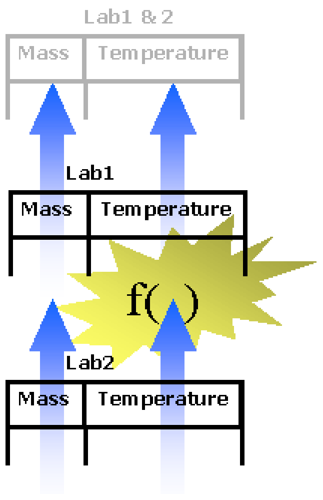


Figure 44. Functional Transformation for Horizontal Integration.

3.1.4. 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 Q := π_p (σ_s (X)).

INDUS provides a query-centric algorithm that takes as input a query *Q* and returns the set of instances satisfying *Q*. This algorithm 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 leaf corresponds to selection and projection operations applied over ground concepts. Any other node in the tree represents compositional operations applied over the set of instances returned by its direct descendants.

Therefore, each non leaf represents an operational definition described by π_p (σ_s ($node_1 \bullet node_2 \bullet \ldots \bullet node_n$)), where $node_j$ represents a descendant node of $node_i$. Here, • corresponds to a cross product operations (×) if vertical integration operation is required. Similarly, a union operation (∪) is specified if an horizontal integration is needed.

In the case of leaf a node *$node_l$*, the associated operational definition of *$node_l$* corresponds to the formula $\pi_p(\sigma_s\,(X))$ where X is a ground concept.

The algorithm for finding an expression tree for a query *Q* has the following goals:

(1) Find an equivalent rewriting *Q′* of *Q* in terms of ground concepts and compositional operations. *Q′* is represented by an expression tree.

(2) Obtain an operational definition of each descendant that retrieves as few instances as possible. This means that the lists *s* and *p* are defined in such a way that the minimum set of instances is retrieved.

We present an example to illustrate the algorithm for finding a rewriting Q′ of a query Q.

Suppose that the ground concepts *A*,*B*,*D* (Figure 45) and the compound concepts *C* and *E* have been defined in INDUS as follows:

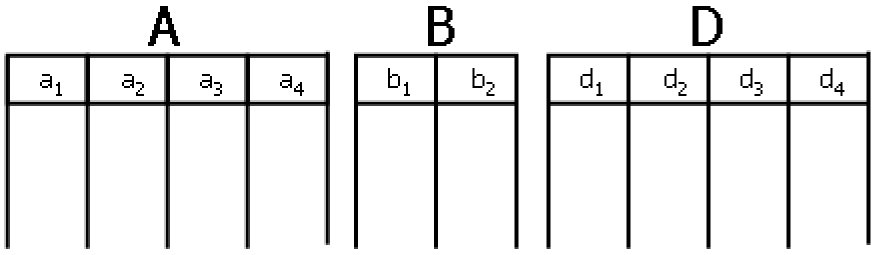


Figure 45. Ground Concepts *A*, *B* and *D*.

The compound concept **C** has the structure presented in Figure 46. Its operational definition is:

$$C._D := \pi_{(c_{11} \leftarrow f(a_1),\ c_2 \leftarrow b_2,\ c_3 \leftarrow a_3)}\ \sigma_{(f(a_2)=f(b_1))}\ (A \times B).$$

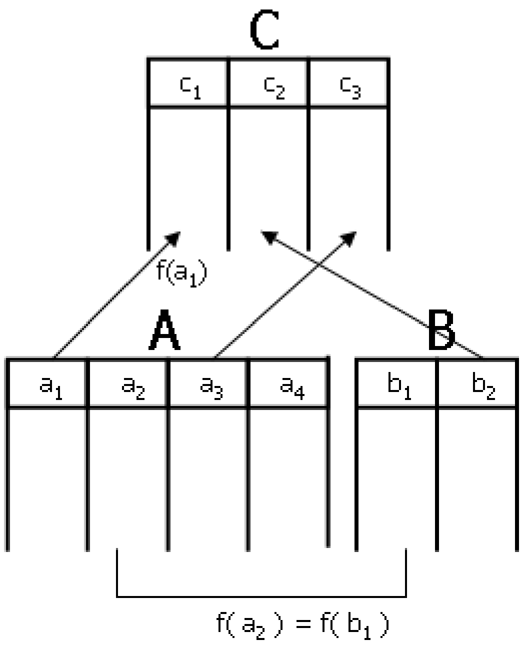


Figure 46. Compositional Operation of *C* and Its Structure.

The compound concept *E* has the structure presented in Figure 47. Its operational definition is:

$$E.D := \pi_{(e_1 \leftarrow f(c_1), e_2 \leftarrow d_2, e_3 \leftarrow d_3, e_4 \leftarrow c_3, e_5 \leftarrow d_4)} \sigma_{(f(c_2)=f(d_1))} (C \times D)$$

Now, assume that a query *Q* is defined as follows (see Figure 48):

$$Q := \sigma_{(e_1=4,\ f(e_2)=3,\ e_4=8,\ e_5=10)} \pi_{(e_2)} (E).$$

In the first step, a root ($node_1$) of the tree is created based on the query definition Q. Also a descendant ($node_2$) is created for the concept E present in the query. Only those conditions that involve the identity functions are added to the *s* list of *node*$_2$ (e_1=4, e_4=8 and e_5=10). The remaining conditions are handled at node *node*$_1$. All attributes present in the selection or projection criteria of *node*$_1$ are included in the list of projection criteria of *node*$_2$.

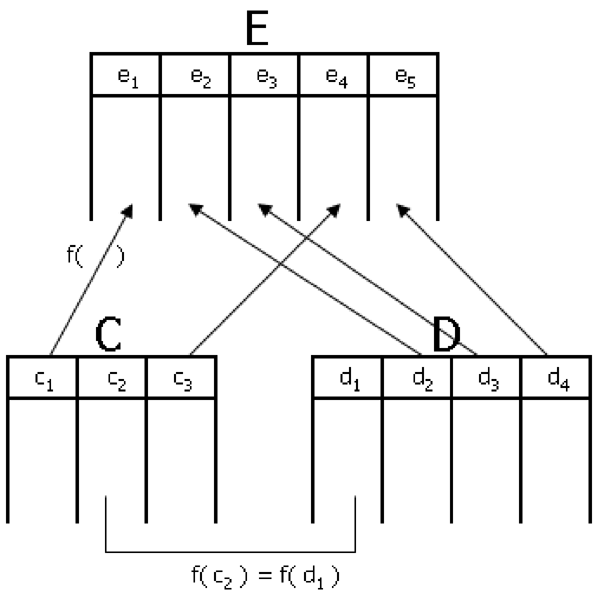


Figure 47. Compositional Operation of Concept E.

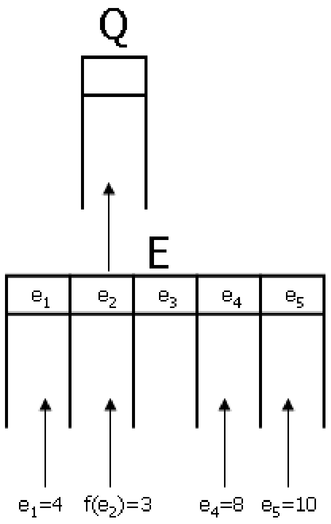
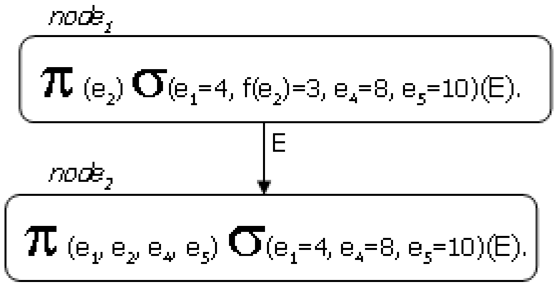


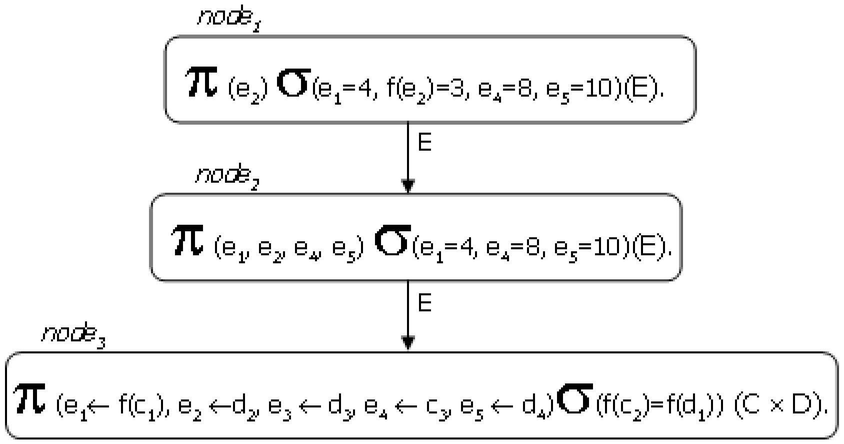
Figure 48. The Operational Definition of *Q*.



node1
π (e2) σ(e1=4, f(e2)=3, e4=8, e5=10)(E).
E
node2
π (e1, e2, e4, e5) σ(e1=4, e4=8, e5=10)(E).

Figure 49. Inserting the Query and Expanding the First Concept.

Given that E is a compound concept, the next step is to create a descendant node of *node*$_3$ based on the operational definition of E as shown in Figure 50.



node1
π (e2) σ(e1=4, f(e2)=3, e4=8, e5=10)(E).
E
node2
π (e1, e2, e4, e5) σ(e1=4, e4=8, e5=10)(E).
E
node3
π (e1← f(c1), e2 ←d2, e3 ← d3, e4 ← c3, e5 ← d4)σ(f(c2)=f(d1)) (C × D).

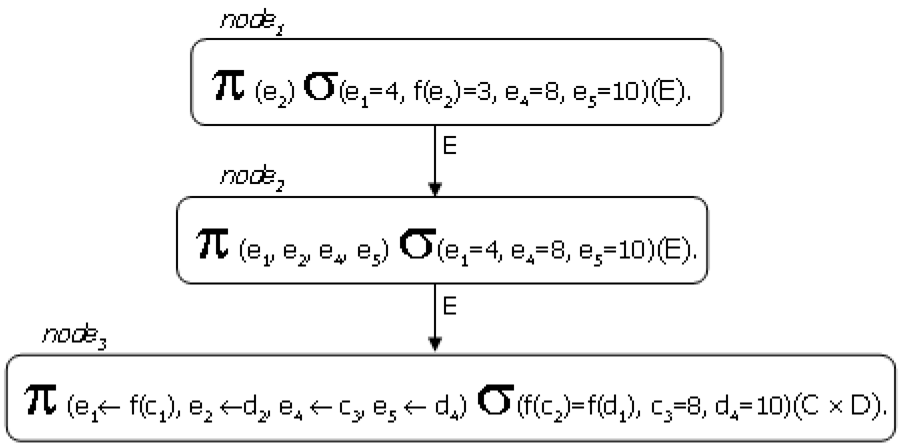
Figure 50. Adding *node3* to the Tree.

The operational definition of *node*$_3$ is transformed as follows:

- Each attribute in the list *p* of *node*$_3$ is inspected to verify if it is used in list *s* or *p* of the father node (*node*$_2$). If the attribute is not used in either *s* or *p* of the father node, it is eliminated from the *s* list of *node*$_3$. In this case, e_3 may be eliminated because it is not required by the father node (*node*$_2$).
- All selection criteria in *node*$_2$ of the form "a_i *operator constant*" where a_i is mapped to an attribute b_j in the form "$a_i \leftarrow b_j$" are added to the list of selection criteria of *node*$_3$ as "b_j *operator constant*". Thus, in our example, "$e_4=8$" and "$e_5=10$" are added to the selection criteria of *node*$_2$ as "$c_3=8$" and "$d_4=10$". If conditions of the form $f_i(a_i)$

operator$_i$ *constant*$_i$ were transported to the child node, we would be required to know the inverse function of f_i (f_i^{-1}), because the father node needs those instances of the child node such that when a function f_i is applied, a value *constant*$_i$ is returned. Therefore, conditions that involve a function are handled by the father node directly.

The transformed tree is presented in Figure 51.



$node_1$
$\pi\ (e_2)\ \sigma(e_1=4,\ f(e_2)=3,\ e_4=8,\ e_5=10)(E).$
E
$node_2$
$\pi\ (e_1,\ e_2,\ e_4,\ e_5)\ \sigma(e_1=4,\ e_4=8,\ e_5=10)(E).$
E
$node_3$
$\pi\ (e_1 \leftarrow f(c_1),\ e_2 \leftarrow d_2,\ e_4 \leftarrow c_3,\ e_5 \leftarrow d_4)\ \sigma(f(c_2)=f(d_1),\ c_3=8,\ d_4=10)(C \times D).$

Figure 51. Transformed Operational Definition of *node*$_3$.

The process is repeated until a complete expression tree is obtained. The final tree for our example is presented in Figure 52.

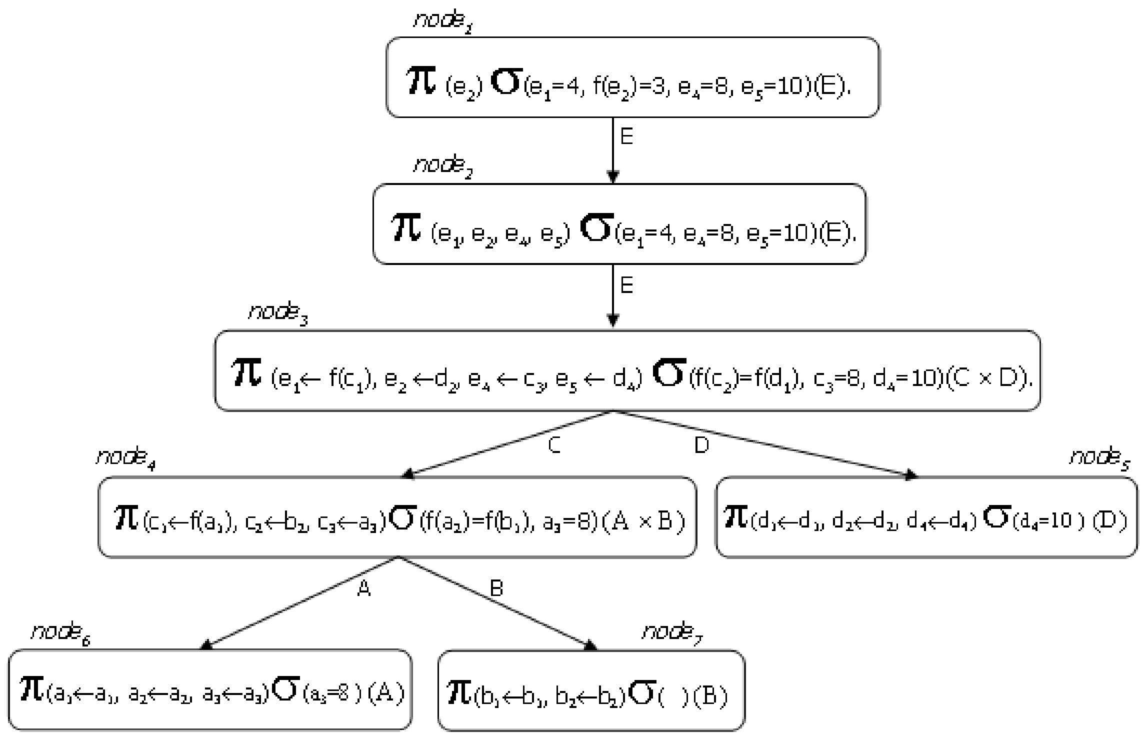

node1
π (e2) σ(e1=4, f(e2)=3, e4=8, e5=10)(E).
E
node2
π (e1, e2, e4, e5) σ(e1=4, e4=8, e5=10)(E).
E
node3
π (e1← f(c1), e2 ←d2, e4 ← c3, e5 ← d4) σ (f(c2)=f(d1), c3=8, d4=10)(C × D).
C
D
node4
π (c1←f(a1), c2←b2, c3←a3) σ (f(a2)=f(b1), a3=8) (A × B)
node5
π (d1←d1, d2←d2, d4←d4) σ (d4=10) (D)
A
B
node6
π (a1←a1, a2←a2, a3←a3) σ (a3=8) (A)
node7
π (b1←b1, b2←b2) σ( ) (B)

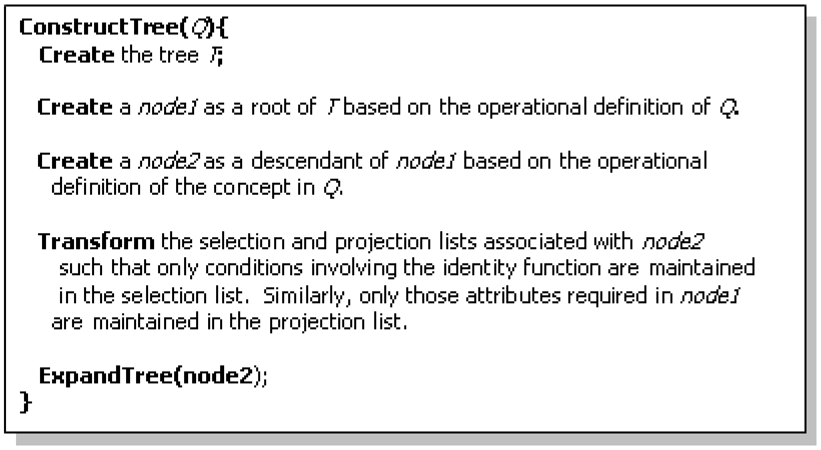
Figure 52. Final Expression Tree.

Before we proceed to describe the algorithm, a few definitions are introduced. If *node* is a node in the tree, then:

- *node*.$_{\text{father}}$ is a reference to the node father. For the root, a null ($\perp$) value is assigned.
- *node*.$_{\text{concept}}$ is the concept that this node is expanding. *node*.$_{\text{concept}} \in \mathrm{K}$.
- *node*.$_{\text{selection}}$ is a list of selection criteria. The ith-selection criterion is detonated by *node*.$_{\text{selection[i]}}$. If $s \in$ *node*.$_{\text{selection}}$ then:
  - *s*.$_{\text{function1}} \in \Phi$, where Φ is the set of functions provided by INDUS,
  - *s*.$_{\text{attributes1}}$ is a list of attributes that serves as arguments to *s*.$_{\text{function1}}$. The ith-element of *s*.$_{\text{attributes1}}$, *s*.$_{\text{attributes1[i]}}$, corresponds to the ith-argument of *s*.$_{\text{function1}}$. *s*.$_{\text{attributes1[i]}} \in [\mathrm{U}_j$ *node*.$_{\text{concepts[j].atts}}]$.
  - *s*.$_{\text{operator}} \in \mathrm{O}$, the set of operators defined in INDUS (=,>,<, etc.),
  - *s*.$_{\text{function2}} \in \Phi$,
  - *s*.$_{\text{attributes2}}$ is a list of attributes that serves as arguments to *s*.$_{\text{function2}}$. The ith-element of *s*.$_{\text{attributes2}}$, *s*.$_{\text{attributes2[i]}}$, corresponds to the ith-argument of *s*.$_{\text{function2}}$. *s*.$_{\text{attributes2[i]}} \in [\mathrm{U}_j$ *node*.$_{\text{concepts[j].atts}}]$.

- $node._{\text{projection}}$ is a list of projection criteria. The ith-projection criteria is detonated by $node._{\text{projection[i].}}$ If $p \in node._{\text{projection}}$ then:
    - $p._{\text{function}} \in \Phi$,
    - $p._{\text{attributes}}$ is a list of attributes that serves as arguments to $p._{\text{function}}$. The ith-element of $p._{\text{attributes}}$, $p._{\text{attributes[i]}}$, corresponds to the ith-argument of $p._{\text{function}}$. $p._{\text{attributes[i]}} \in [\text{U}_j\ node._{\text{concepts[j].atts}}]$.
    - $p._{\text{projectedAttribute}} \in node._{\text{concept.atts}}$
- $node._{\text{concepts}}$ is the list of concepts where the set of instance are obtained (after applying the correspondent selection, projection, and concept operation). The ith-concept is denoted by $node._{\text{concepts[i]}}$ and $node._{\text{concepts[i]}} \in \mathrm{K}$, the set of concepts defined in INDUS.
- $node._{\text{operation}} \in \Omega$, where Ω is the set of concept operation defined in INDUS. In this first prototype Ω has two elements: the cross product ($\times$) the union operation ($\cup$).

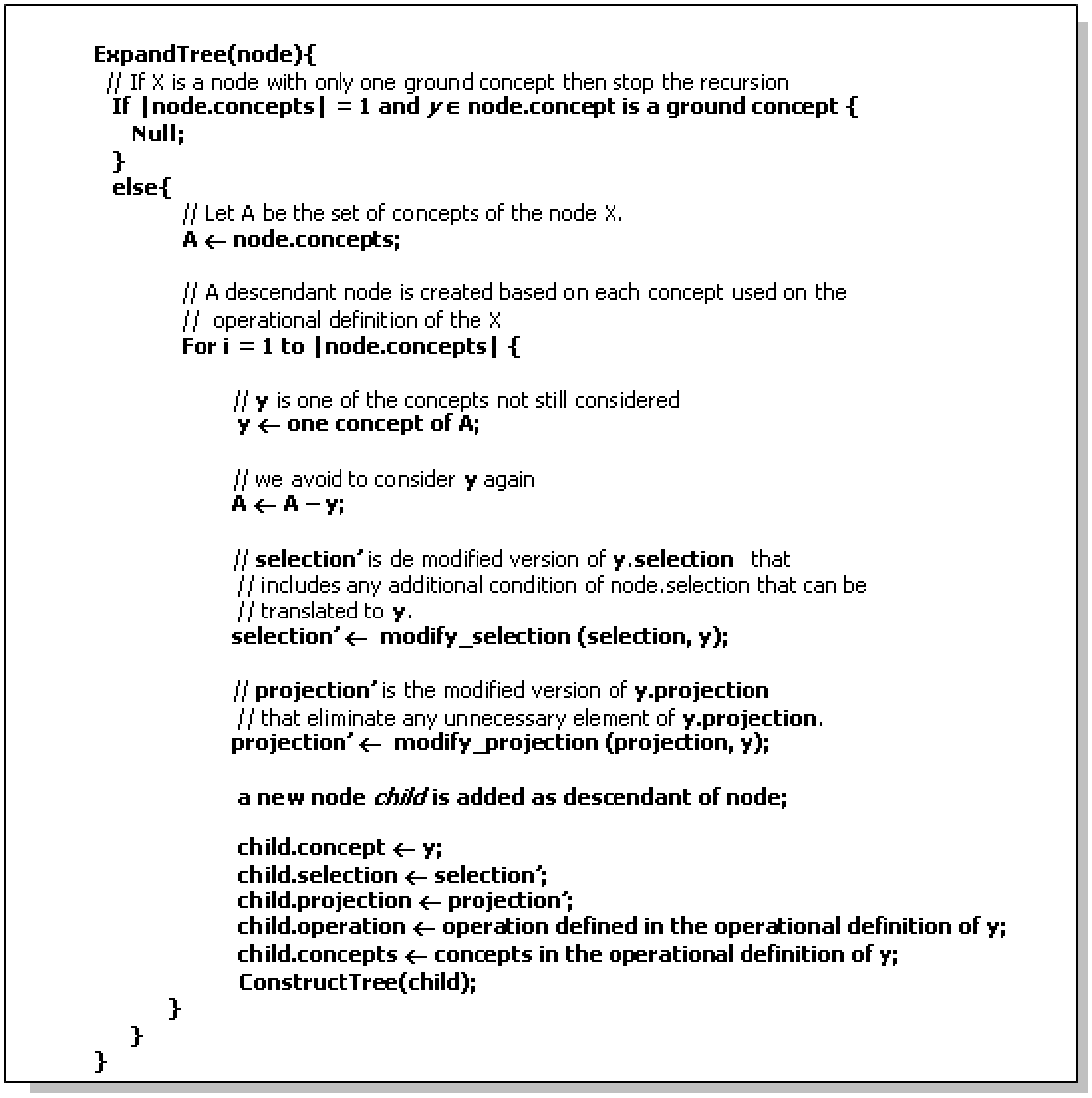
The algorithm for finding the expression tree of a query Q is presented Figure 53



```
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);
}
```

Figure 53. The *ConstructTree* algorithm.

After the rewriting Q' is created, it is executed traversing the tree using a post-order ordering. For each visited node, a set of instances is retrieved following the operational definition associated with the node. If the node is a leaf, instances matching the selection and projection operations associated with the leaf are retrieved from data sources and stored in the user workspace as a relational table. If the node is not a leaf, a relational view is created based on views or tables created by its descendant. The set of instances associated with the view created for the root node corresponds to the answer to the query Q′, and therefore to the query Q. Figure 55 shows the algorithm.



```
ExpandTree(node){
 // If X is a node with only one ground concept then stop the recursion
  If |node.concepts| = 1 and y ∈ node.concept is a ground concept {
     Null;
  }
  else{
         // Let A be the set of concepts of the node X.
         A ← node.concepts;

         // A descendant node is created based on each concept used on the
         //  operational definition of the X
         For i = 1 to |node.concepts| {

               // y is one of the concepts not still considered
                y ← one concept of A;

               // we avoid to consider y again
               A ← A – y;

               // selection' is de modified version of y.selection  that
                // includes any additional condition of node.selection that can be
                // translated to y.
               selection' ← modify_selection (selection, y);

               // projection' is the modified version of y.projection
                // that eliminate any unnecessary element of y.projection.
               projection' ← modify_projection (projection, y);

                a new node child is added as descendant of node;

                child.concept ← y;
                child.selection ← selection';
                child.projection ← projection';
                child.operation ← operation defined in the operational definition of y;
                child.concepts ← concepts in the operational definition of y;
                ConstructTree(child);
         }
     }
}
```

Figure 54. The *ExpandTree* algorithm.

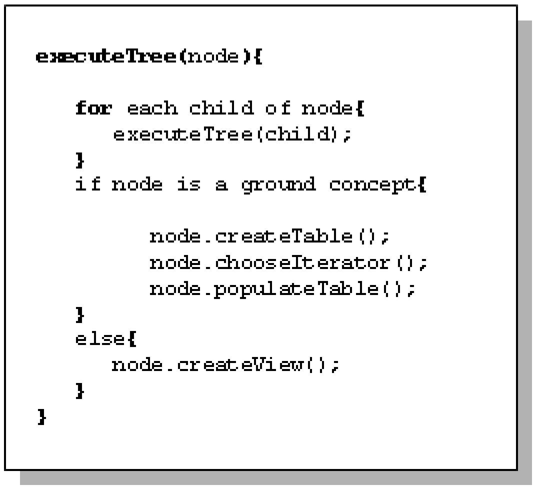
The process of retrieving instances of a leaf (*node*) of the tree requires the following steps:

(1) The creation of a relational table. The set of columns associated with the table corresponds to the elements in the list of projection criteria, *node*$_{\text{projection}}$.

(2) The selection of an instantiator $\iota \in$ node$_{\text{concepts[1].D}}$,

(3) The invocation of ι, and

(4) The insertion of the retrieved instances as rows of the table created in the first step after applying the selection and projection operations defined in the node.



```
executeTree(node){

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

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

Figure 55. Pseudo-algorithm for Executing the Expression Tree.

Recall that the operational definition of a leaf node in the tree has the form $\pi_p(\sigma_s(Y))$, where Y corresponds to a ground concept in INDUS. One important result of applying the algorithm shown in Figure 54 for creating the expression tree Q' is that the selection criteria of each leaf node, *node.*$_{selection}$, follows the form "a_{i1} opt_1 k_1, a_{i2} opt k_2, …" where $a_{i1} \in Y._{atts}$, $opt_i \in O$, and k_i is a constant. Similarly node.$_{projection}$ follows the form "a_{i1}, a_{i2}, …". This is because:

- Each time that a ground concept is expanded, the *node.*$_{selection}$ list associated with the ground concept is null (all instances are retrieved). When the transformation step is applied, only those conditions of the node's father that are of the form specified above are added to *node.*$_{selection}$.
- The list *node.*$_{projection}$ associated with the ground concept originally includes all attributes of the ground concept. The transformation step only left those attributes used by the node's father.

Thus, the instantiator to be selected for Y need not deal with execution of functions. This task is left to the INDUS system through the mechanism for materializing compound concepts. Therefore, the instantiator needs to focus on the following:

(1) Execution of the corresponding iterator using the correct set of parameters such that a minimal (with respect to the query capabilities provided by the data source) superset of instances is returned from the data sources.

(2) Application of the selection and projection conditions described in *node.*$_{\text{selection}}$ and *node.*$_{projection}$ list associated with the node to the set of instances retrieved by the iterator in the previous step. This step eliminates any instance returned in the previous step that does not match the conditions expressed in *node.*$_{\text{selection}}$. This is required because the set of instances returned in (1) are based on the query capabilities offered by the data source and those may not match completely the conditions defined in *node.*$_{\text{selection}}$.

In other words, the algorithm will select the instantiator that retrieves as few instances as possible from the data source given the selection criteria and the query capabilities of the instantiator. From this set, all instances that do not match the selection criteria are eliminated. This ensures that:

- The set of instances transported from the data sources to INDUS is minimized with respect to the query capabilities of the data sources and the selection criteria.
- The query capabilities offered by the data sources are exploited by INDUS in the resolution of the query as much as possible.

An *optimal* instantiator is that which minimize the set of instances to be retrieved from a data source based on a given set of conditions.

Formally, an instantiator ι_i of a ground concept Υ ($\iota_i \in \Upsilon._D$) is optimal with respect to a set of conditions *s* if i = argmax $M(s, \iota_i)$ where $M(s, \iota_i) > 0$. The measure function *M* is defined as follows:

$$\text{M}(\text{s}, t_i) = \sum_{j=1}^{|t_{i \cdot queryCapabilities}|} m(s, t_{i \cdot queryCapabilities[j]})$$

where $m(s, \iota_{\text{i.queryCapabilites[j]}})$ is 1 if exists $s_k \in$ s such that $s_{k\text{.attributes1[1]}} = \iota_{\text{i.queryCapabilites[j].attribute}}$ and $s_{k\text{.operator}} = \iota_{\text{i.queryCapabilites[j].operator}}$. Otherwise, $m(s, \iota_{\text{i.queryCapabilites[j]}})$ is $-\infty$.

Note that a condition $\iota_{\text{i.queryCapabilities[j]}} \in \iota_i$ does not match any selection criteria in *s* resulting in the instantiator ι_i is not taken into account as a solution for *s*.

When *i* is undefined, the query Q is not solvable under the query capabilities offered by the data sources. As an example, assume that a query *Q* is defined over a ground concept *A* as follows: Q = $\pi_{(a1)}\sigma_{(a1=3)}$(A). Consider an operational definition of *A* as *A*.$_{D}$ = {ι}. Assume that the query capability of ι is defined as ι.$_{queryCapabilities}$ = {(a2, =)}, meaning that ι only accepts queries over a2 (using the equality operator). Clearly, the query Q can not be answered under the query capabilities offered by the instantiator (therefore the data source).

3.2. INDUS: The Prototype

This section describes the implementation of INDUS, a prototype of a data integration system for a scientific discovery environment. Several design decision were taken to offer a practical implementation of the formalisms presented in the previous section.

A module is a piece of software that implements a particular functionality into the system. INDUS is composed by five principal modules (shown in Figure 56): The graphical user interface, the iterator's library, the common global ontology area, the query resolution algorithm and the private workspace user area.

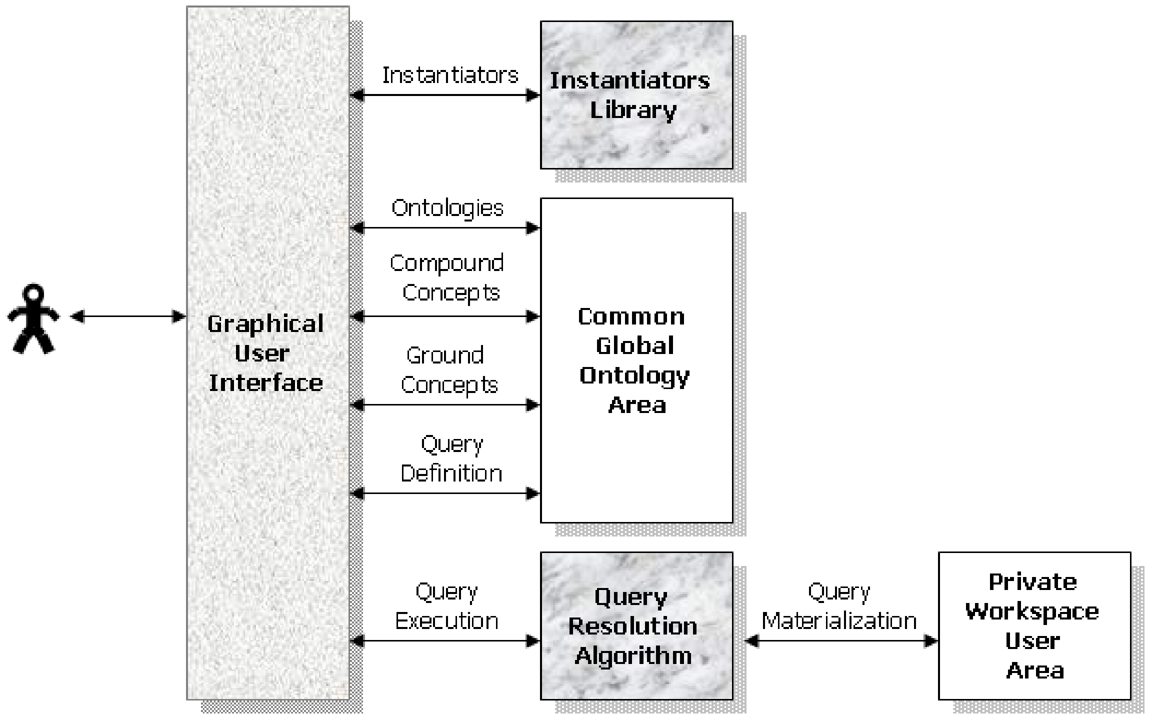


Figure 56. *INDUS* Modules Diagram.

INDUS is based on five modules. The graphical user interface enables users to interact with INDUS. This module is developed under Oracle Developer 6i. The common global ontology area, implemented through a relational database system, stores all information about ontologies, concepts and queries. Any information stored in this repository is shared for all users. The private workspace user area is also implemented through a relational database system. Each INDUS user has a private area where queries (an expression trees) are materialized. The query resolution algorithm is a java class that takes as input a query and materialize it (i.e., gets the set of instances satisfying the query) in the private workspace user area. The library of instantiators is based on java classes that implements iterators.

The graphical user interface enables users to interact with the system to define and query:

- *Ontologies*,
- Operational definitions of *compound concepts*,
- Operational definitions of *ground concepts*, and
- Operational definition of *queries*.
- Registration of *Iterators* signature.
- Execute *queries*.

The common global ontology module manages the repository where definitions of ontologies, ground concepts, compound concepts, queries, and iterator signatures are stored. Each of these definitions is accessible to all users and can be used in the construction of user defined ontologies. In INDUS, an *ontology* allows users to group a set of concepts (a concept may belong to more than one ontology). Recall also that *compound concepts* are those whose operational definition specifies a procedure to retrieve their instances from instances of other concepts. *Queries* are very similar to compound concepts in the sense whose operational definition describes a procedure to retrieve instances (get an answers) from a set of instances of other concepts. *Ground concepts* are those that their operational definition describes a procedure to retrieve their instances directly from data sources using a set of instantiators. Each instantiator is based on a particular iterator. The *iterator signatures* specify the name of the iterator and the set of parameters that control the behavior of each iterator. Thus, when a set of instances must be retrieved from a ground concept, INDUS knows how to correctly invoke the required *iterator* for the correspondent *instantiator*.

The instantiator library is a set of components that INDUS may invoke in order to gather a set of instances of ground concepts from particular data sources. For this first prototype, instantiators use iterators implemented through java classes.

The query resolution module takes as input a query and obtains the set of instances that correspond to the answer of the query. In the process of gathering the answer, an expression tree, described in terms of ground concepts and compositional operations equivalent to the definition of the query, is created. The expression tree then is executed allowing the system to gather the instances from each relevant data source, and combine them to obtain the answer for the posted query.

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. Because each user of the system has his/her own private workspace area, conflicts are avoided when two similar queries are executed by different users. However, several mechanisms are implemented allowing users to share their answers (set of instances) with others, if desired.

The modular design of INDUS ensures that each module can be updated and alternative implementation easily explored. Modularization also enables INDUS to use different network architectures. For example, INDUS may be implemented in a centralized architecture, locating all the modules in a unique server, or in an architecture where modules are distributed through several servers. This feature may be used for improving the overall performance of the system. Consider a case where contention is found in the graphical user interface. Thus, this module may be installed in an independent server. Another possibility may be to deploy two of those modules in different servers. Thus, the population of users may be divided into two disjointed sets, each attended adequately by a different server.

In the first INDUS version, the graphical user interface is implemented using Oracle Developer 6i, the common global ontology area and the user workspace are developed using a relational database system (for this case Oracle 8i™ was selected, but in general any relational database system accessible using JDBC may be used), and the query resolution algorithm, along with the set of iterators are developed as java classes.

The recommended way to deploy INDUS modules is presented in Figure 57. Using this distribution, 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.

Developer 6i was selected as a tool for developing the graphical user interface because it provides an environment for rapid and efficient development of user interfaces when data is stored using relational database systems. Additionally, it offers an automatic way to use the graphical user interface through the web.

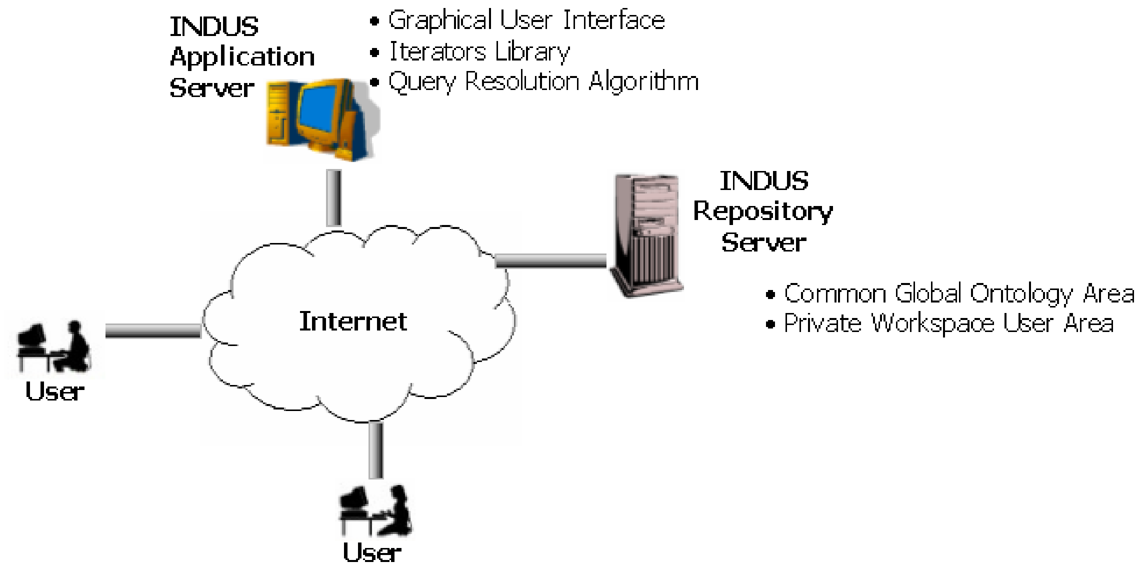

INDUS Application Server
• Graphical User Interface
• Iterators Library
• Query Resolution Algorithm
INDUS Repository Server
• Common Global Ontology Area
• Private Workspace User Area
Internet
User
User

Figure 57. *INDUS* Module Distribution.

INDUS is based on a three-tier architecture. The application servers store the functional implementation of INDUS (screens, procedures, and others). The repository server stores the ontological repository and the workspace user areas. An INDUS user can interact with the system accessing any of the application servers and establishing a database connection with one of the repository servers.

The common global ontology area was developed under a relational database in order to provide a robust environment in which to store and manipulate ontologies of large size. Relational database technology also offers us an efficient way to support multiple concurrent users. Also, standard protocols such as ODBC and JDBC facilitate sharing of the ontology with other applications as needed.

The user workspace was also developed using a relational database. This enables users to manipulate the results of queries using relational database operations or other application programs. Additionally, although the workspaces are private for each user, using *grant* *privileges* mechanisms provided by the RDBMS enables users to share their results with others easily, if required.

The iterators and the resolution algorithm were developed using java, because we wanted the prototype to be able to run in as many as possible hardware and software platforms.

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, queries. Also, a user may execute queries and manipulate his/her answers. Therefore, knowledge on the scientific domain area of interest is expected for users playing this role.
- As an ontology engineer, a user may expand the iterator library by programming new iterators. Also, new ground concepts may be defined in INDUS. This implies that a user playing a role of ontology engineer must be a proficient programmer.
- As an administrator, a user may be able to install INDUS, that is, to install the graphical user interface and the query resolution module, and also to operate the databases supporting the common global ontology module and the user workspace, which includes adding new users to this (those) database(s). Knowledge in databases is required for users playing this role.
- As a developer, a user may add new compositional operations to INDUS, modifying the graphical user interface module and the query resolution module. A proficiency in programming is required.

Of course, in some cases, a user may play more than one role.

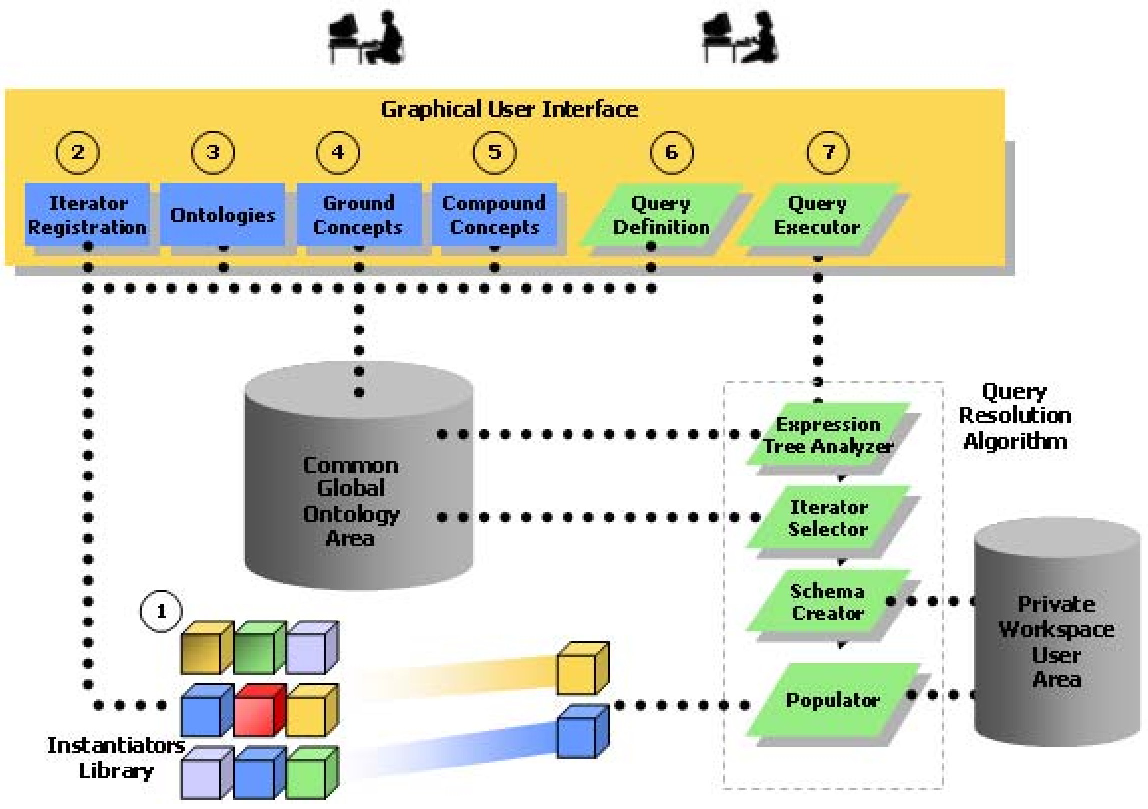


Figure 58. Flow of Operation of *INDUS*.

An ontological engineer defines and registers one or more iterators and defines instantiators and ground concepts under ontologies defined by the domain users. After that, the domain user may define new concepts based on previously defined concepts. With this information the domain user may post queries and execute them, obtaining answers in his/her private workspace user area.

Figure 58 shows a possible flow of operation of INDUS, after an administrator user has installed INDUS.

(1) An ontology engineer creates one or more iterators that are able to deal with specific types of data sources. These iterators are java classes located in a particular directory.
(2) The iterators are registered into INDUS indicating the class name and the set of parameters that control the iterator's behavior. Because only the signature of the iterator is required to be known, this step may be executed by an ontology engineer or by a domain scientific.
(3) One or more ontologies are defined in INDUS. This step, in general, is execute by a domain scientist because he/she has the knowledge to determine why and how to organize (or to group) concepts.

(4) Ground concepts are defined in INDUS describing its structure along with its operational definition, i.e., the set of instantiators. Given that the operational definition of a ground concept involves the use instantiators, iterators must be selected, values assigned to their parameters, mappings defined and query capabilities described. Therefore, this step should be executed by an ontology engineer.

(5) Domain scientists define compound concepts. Compound concepts are described in INDUS by providing their structure along with their operational definition. Recall that the operational definition of compound concepts (and of a query) involves describing how to apply a set of compositional operations (horizontal and vertical integration, selection and projection) to a set of concepts.

(6) Queries are defined. As will be shown later, the definition of queries is very similar to the definition of compound concepts. Therefore, domain scientist often defines queries too.

(7) Queries are executed. Several actions are automatically trigger by INDUS in this step which are described as follows:

   a) The definition of the query is transformed into an expression tree.
   b) The expression tree is recursively expanded to a tree in which every leaf is a set of compositional operations applied over a ground concept.
   c) The tree is traversed in post-order ordering processing each node as follows:
      - If the node is a leaf:
         - An instantiator is selected (from its operational a ground concept),
         - a table with the required set of columns is created in the workspace user, and
         - the table is populated invoking the selected instantiator.
      - If the node is not a leaf, a relational view is created in the workspace user area using an SQL command.
   d) The final step is to create a view in the workspace area representing the selection and projection operations defined in the query.

The following scenario (based on a biological example) illustrates some of the operation in INDUS. As was described previously, a user may play several roles with INDUS. Three operations will be described: How to create a compound concept, how to create a ground concept and how to post and execute queries. The first and last are common tasks executed by domain scientists while the definition of ground concepts is often executed by an ontology engineer.

3.2.1. Defining Compound Concepts

Available information about molecular biology is currently offered by multiple biological databases. In general, these databases specialize in a particular kind of information. For example, SWISSPROT [2002] is a curated and annotated protein sequence database. The annotation includes information describing the function associated with the protein, its domain structure, post-translational modifications and others. On the other hand, PROSITE [2002] is a protein family and domain database, where patterns and profiles are offered to users to facilitate the identification of the family that a protein belongs to. Although the word *database* is frequently used describing those data sources, a restricted set of queries are presented to users (typically based on a set of web forms) instead of offering a full query interface based on SQL or similar languages.

Thus, assume that an ontology engineer has successfully defined three ground concepts, providing the information presented in Figure 59 under a particular ontology. From SWISSPROT, a concept PROTEIN is described in terms of its identification (Id), the accession number, a description (name) of the protein, the organism where the protein is present, a text describing the dates when the protein was discovery, the organelle where the protein is present, a classification of the organism where the protein is present, a cross reference of the organism and the amino acid sequence of the protein. From PROSITE two ground concepts are defined. The instances of the MOTIF concept corresponds to all the motifs present in PROSITE. Each motif is described in terms of the internal id used by PROSITE to identify the motif (Id), the accession number of the motif, the type of motif (pattern, rule, and matrix), the pattern describing the regular expression of the motif, its domain, and its description. The second ground concept defined from MOTIF is a concept representing the set of proteins that have a particular motif. We have called as MOTIFPROTEIN. Two attributes are used to describe each instance, the id of a motif and the accession number of a protein (referencing the accession number used to describe each protein in SWISSPROT).

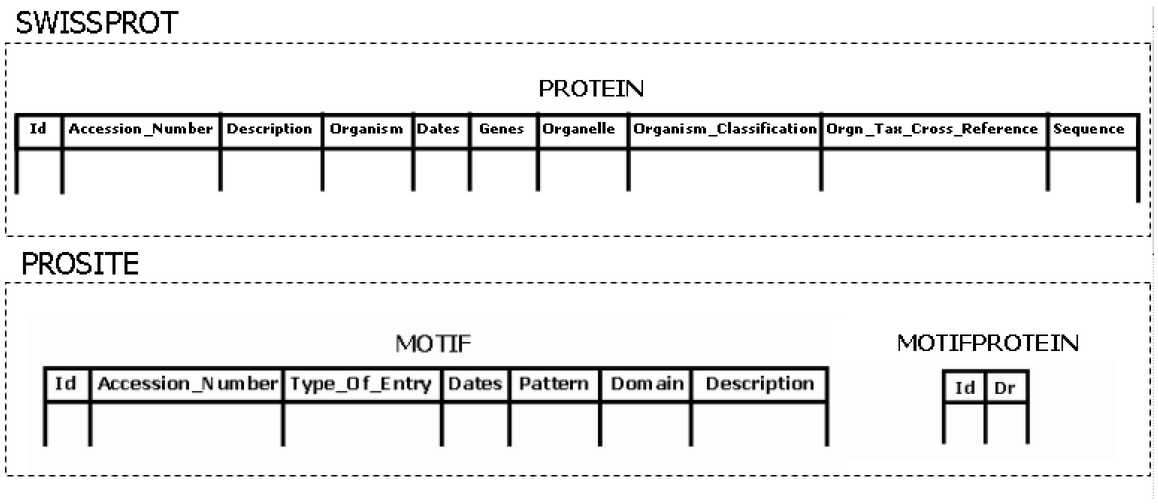


Figure 59. Ground Concepts from *SWISSPROT* and *PROSITE* data sources.

Suppose that a final user wants to define a new concept named MOTIF_AND_PROTEIN (see Figure 60). This new concept will enable the domain scientist to see summary information relating each motif with its correspondent set of proteins. Figure 60 shows a possible structure of such kind of concept containing the motif id, the pattern of the motif, the accession number of one protein that has the motif and the amino acid sequence of the protein.

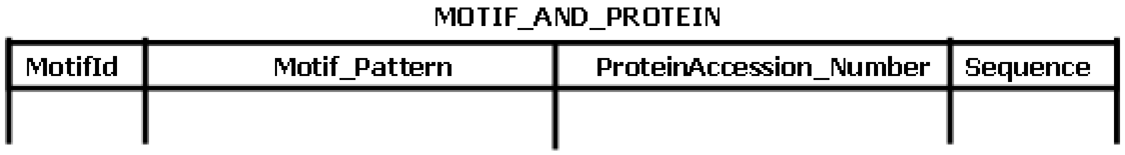


Figure 60. A New Concept *MOTIF_AND_PROTEIN*.

After a brief study of the structure presented by the previously defined ground concepts, the domain scientist determines how to define the MOTIF_AND_PROTEIN concept in terms of the three ground concepts and a set of compositional operations. Recall that INDUS offers four compositional operations: vertical and horizontal integration, selection and projection. The domain scientist finds that the way to define MOTIF_AND_PROTEIN corresponds to a vertical integration and selection and projection operations defined in Figure 61.

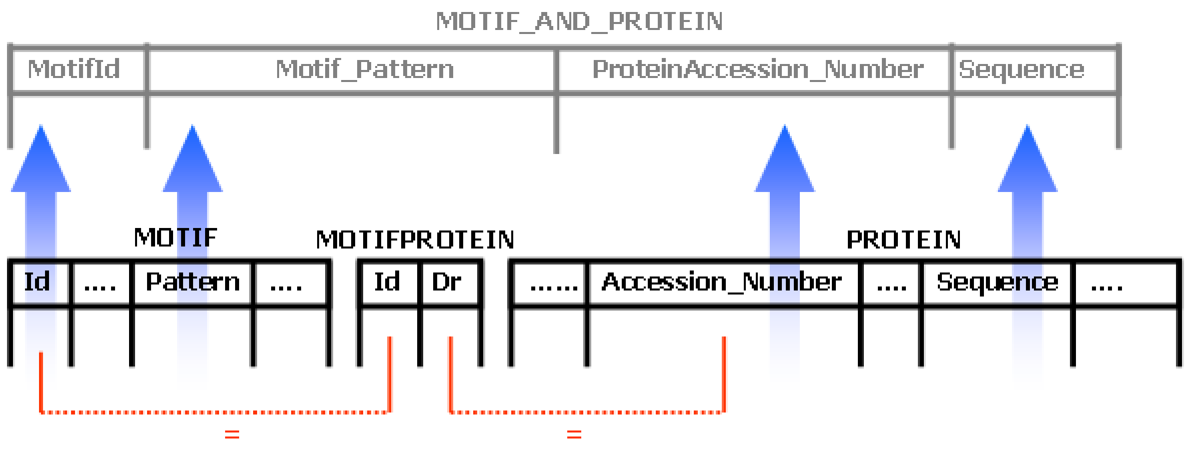
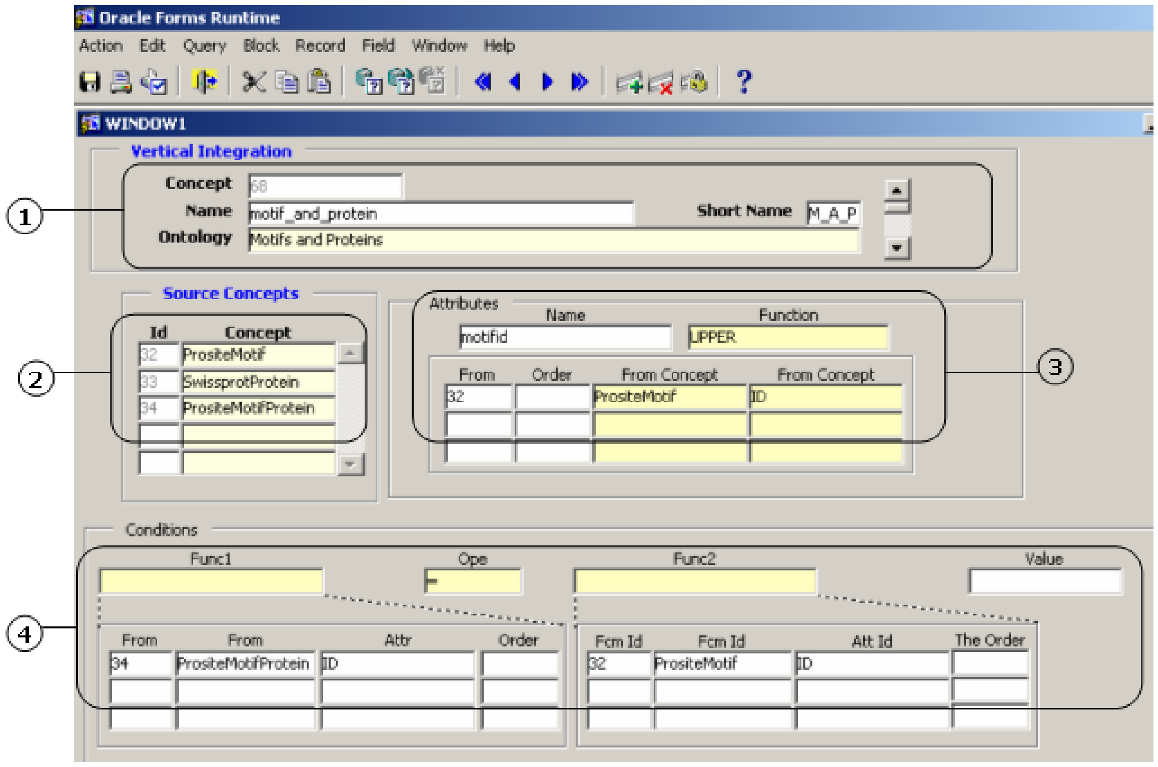


Figure 61. Compound Concept Example.

The MOTIF_AND_PROTEIN concept is defined as a vertical integration operation applied over the MOTIF, MOTIFPROTEIN and PROTEIN concepts. The source of each attribute is represented as a blue row. The lines relating the MOTIF.ID attribute with the MOTIFPROTEIN.ID attribute and the MOTIFPROTEIN.DR attribute with the PROTEIN.ACCESION_NUMBER attribute selects the set of instances to be returned from the Cartesian product of MOTIF, MOTIFPROTEIN and PROTEIN concepts.

Using one of the options offered by the graphical user interface module, our domain scientist uses the screen presented in Figure 62 to define the new concept MOTIF_AND_PROTEIN using a vertical integration operation. Recall that a vertical integration operation is defined as a projection and selection operations applied over a cross product of concepts.



Vertical Integration

Concept 68

Name motif_and_protein

Short Name M_A_P

Ontology Motifs and Proteins

Source Concepts

Id	Concept
32	PrositeMotif
33	SwissprotProtein
34	PrositeMotifProtein

Attributes

Name	Function
motifid	UPPER

From	Order	From Concept	From Concept
32		PrositeMotif	ID

Conditions

Func1	Ope	Func2	Value
	=		

From	From	Attr	Order
34	PrositeMotifProtein	ID	

Fcm Id	Fcm Id	Att Id	The Order
32	PrositeMotif	ID	

Figure 62. Vertical Integration Screen.

In the section 1 is defined the new concept. In section 2 the set of concepts where the Cartesian product is applied are defined. The section 3 describes the source for each attribute of the new concept. The section 4 describes the selection operation to be applied over the set of instances resulting of the Cartesian product over the source concepts.

Figure 62 presents the main steps involved defining a compound concept using a vertical integration operation:

(1) Definition of the structural information of the new concept, i.e., its name, its short name (which will be used as the table name) and the set of ontologies which includes this concept.

(2) Selection of the concepts where the Cartesian product is applied. For this example, this corresponds to the Protein, Motif, and MotifProtein concepts.

(3) Mapping between the attributes returned by the Cartesian product and the attributes of the concept. For this example, the Motif.id attribute is mapped to the motif_and_protein.motifid attribute.

(4) Conditions that specify which instances returned by the Cartesian product belongs to the set of instances of the new concept. All conditions are of the form $f_1(a_{i1},a_{i2},\ldots a_{in}) = f_2(a_{j1},$

a_{j2}, ... a_{jm}) or f1(a_{i1},a_{i2},...a_{in}) = value. For our example, the information shown in the screen corresponds to Identity(MotifProtein.Id) = Identity(Motif.Id).

Thus, in general, this screen allows user to define new concepts of the form:

(using SQL)

Create or Replace view NewConcept as
  Select $f_1(a_{i1}, a_{i2}, \ldots, a_{in})$ attribute$_1$, $f_2(a_{j1}, a_{j2}, \ldots, a_{jp})$ attribute$_2$, ...
  From concept$_1$, concept$_2$ .... Concept$_m$
  Where $g_1(a_{k1}, a_{k2}, \ldots a_{kq}) = g_2(a_{l1}, a_{l2}, \ldots, a_{lr})$
  And $g_3(a_{s1}, a_{s2}, \ldots a_{st}) = value_1$
  ...

Using DataLog with functions:

NewConcept(X_1, X_2, ...) :- Concept1($XC1_1$, $XC1_2$, ..)$\wedge$
Concept2($XC2_1$, $XC2_2$, ...) $\wedge$ ... $\wedge$
Conceptm(XCM_1, XCM_2, ...) $\wedge$
($X_1 = f_1$($XC?_?$, ...)) $\wedge$
($X_2 = f_2$($XC?_?$, ...)) $\wedge$
($g_1(XC?_?,..) = g_2(XC?_?,...)$ $\wedge$ ...

For our example, the associated SQL command will be equivalent to the following:

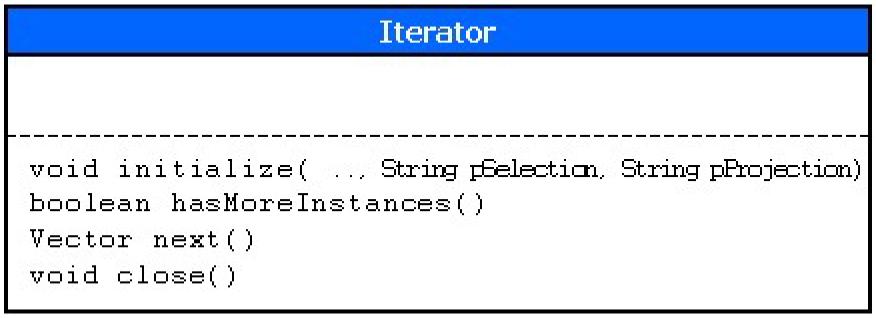
```
Create or Replace view MOTI_AND_PROTEIN (
                              MotifId,
                              Motif_Pattern,
                              ProteinAccesion_Number,
                              Sequence) As
Select UPPER(Motif.Id), Motif.Pattern, Protein.Accession_Number, Protein.Sequence
From Motif, MotifProtein, Protein
Where Motif.Id = MotifProtein.Id
And Protein.Accession_Number = MotifProtein.Dr;
```

3.2.2. Ground Concepts

Continuing with our example, now we turn to the task of defining ground concepts. Recall that the operational definition of ground concepts is based on instantiators. Therefore, one or more iterators must be written and registered into INDUS first.

Following this flow, we will describe what an iterator is. Later, we will present how to define and register iterators. Finally, instantiators are going to be defined based on those iterators.

In INDUS, Iterators are programs that are able to retrieve instances from data sources. The name "iterator" is taken from the database community where they are referred to a piece of software which brings data from the physical layer and present it in the logical layer as rows from a table in a relational database [Garcia-Molina et al., 2000].



```
void initialize( ..., String pSelection, String pProjection)
boolean hasMoreInstances()
Vector next()
void close()
```

Figure 63. Iterator.

Iterators are java classes that are the core of the instantiators. INDUS assumes that each iterator responds to four methods that are called in a predefined order when an instantiator needs to be executed. The *initialize* method is called by INDUS at beginning. Each time that a new instance is required, INDUS verify if there are more available instances (using the hasMoreInstance method) and invokes the *next* method. At the end of the execution, INDUS invokes the *close* method.

In INDUS, an iterator is a java class that implements (at least) four methods: initialize( ), hasMoreInstance( ), next( ), and close( ).

initialize( ): As shown in the algorithm, initialize is the first method to be invoked. Initialize may have two or more arguments and INDUS uses the last two to inform the iterator of the selection and projection operations that need to be applied over the set of instances that the iterator is going to return. In some cases, the iterator may use the query capabilities offered by the data sources to responds to the selection and projection operations. In others, a super set of instances may be retrieved from the data sources and the selection and projection operations applied in INDUS. The value of the selection represents conditions that the set of instances retrieved must match. These conditions are conjunctions of predicates of the forms <attribute><operator><value>. The syntax used by INDUS to represent these conditions match the following grammar:

S -> label * operator * value | S * S | ⊥.

As an example, assume that only those MOTIFS with accession_number equal to PS00010 and ID equal to ASX_HYDROXYL have to be retrieved. Then, the selection will be as follows:

accession_number*=*PS00010*ID=ASX_HYDROXYL

The returned type of this method is *void*. Developers may use the initialize method to define connections with the data sources, or to prepare any previous information before beginning to retrieve instances from data sources.

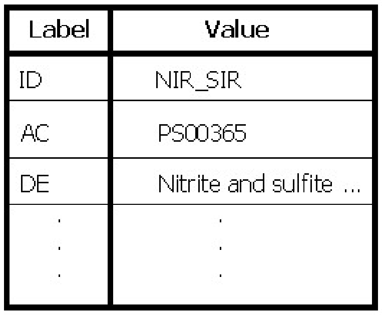
hasMoreInstances( ): this is a boolean method with no arguments. When this method return *true* INDUS assume that it is safe to invoke the **next( )** method, i.e., when invoking **next( )** a new instance matching the selection operation in the **initialize( )** method will be returned. If hasMoreInstances returns *false*, INDUS understand that there are no more instances in the data source that match the selection operation defined in the **initialize( )** method.

next( ): This is a method that returns a java Vector. Similar to the previous method, **next( )** has no arguments. When INDUS invoke **next( )**, a java vector storing a new instance matching the selection operation define in the **initialize( )** argument is returned. Figure 64 presents an example of such as vector. Each element of the vector is composed by two variables. The first indicates the name of an internal attribute and the other a value for that attribute.

close( ): This is the last method invoked by INDUS each time that an iterator is executed. Similar to the previous method, no arguments are allowed here. It returns a java void type.

Developer normally use this method for closing connection with data sources or releasing any resource used during the execution of the iterator.

Iterators support different executions when they are integrated with the query resolution algorithm. For example, if instances from two different ground concepts need to be retrieved (and joined), at least two different plans can be applied: to retrieve all instances of each concept and combine them in INDUS or to implement a two-nested-loop algorithm, using the external loop to retrieving instance from one concept and the internal loop to retrieve instances of the other concept that match with the current instance from the external loop. Note that a "pipeline" architecture, where instances are passed to other iterators as needed, is easily implemented [Garcia-Molina et al., 2000].



Label	Value
ID	NIR_SIR
AC	PS00365
DE	Nitrite and sulfite ...
. . .	. . .

Figure 64. A Sample of a Vector Returned by *next*().

Instances, stored in the vector returned by the **next ( )** method, are easily transformed to rows. Recall that when ground concepts are instantiated, a table is created and populated. Thus, each vector returned by **next( )** is converted as rows and inserted in the corresponding table. Next INDUS implementations will explore multiple values for an attribute in an instance. The current iterator's design will not need to be modified to support that.

Note that iterators encapsulate the underlying technology supporting data sources. Thus, data sources can be relational databases, web pages, or flat files. Given that parameters (bind variables) are also supported, iterators can interact with programs that receive as input a set of values and return one or more instances (e.g., MEME, BLAST, others).

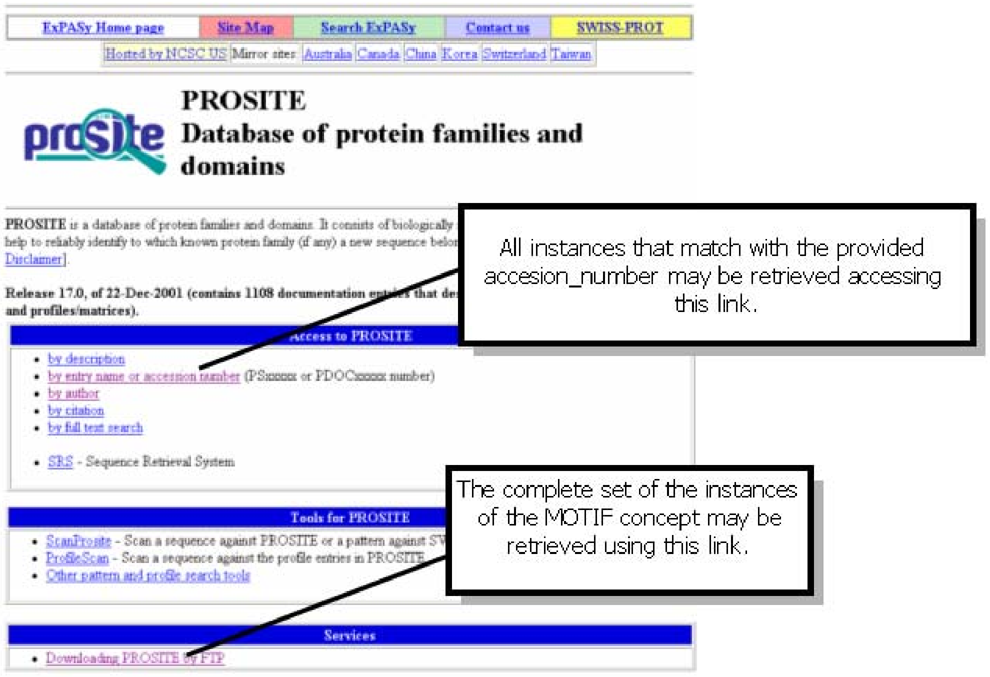
Finally, note that iterators do not need to extend either other classes or interfaces. Instead, the reflection mechanism provided by java was chosen. Thus, iterators are linked to INDUS at run-time.

INDUS uses the algorithm presented below for invoking an iterator:

```
Execution(Iterator)
{
          Iterator.initialize(...., σ, π);
          while(Iterator.hasMoreInstances( )) {
                    theInstance = Iterator.next( )
          }
          Iterator.close( );
}
```

Returning to our example, iterators will be created for retrieving instances for the MOTIF concept from the PROSITE data source. Figure 65 shows two access points for retrieving instances of MOTIF concept provided by the data source.

In the first access point, a particular accession number is supplied by the user through a web interface. With that information, PROSITE returns a set of instances matching the given accession_number, as presented in Figure 66. The second option allows user to retrieve all instances of the MOTIF concept reading the file presented in Figure 67.



ExPASy Home page | Site Map | Search ExPASy | Contact us | SWISS-PROT

Hosted by NCSC US Mirror sites: Australia Canada China Korea Switzerland Taiwan

PROSITE
Database of protein families and domains

PROSITE is a database of protein families and domains. It consists of biologically
help to reliably identify to which known protein family (if any) a new sequence belon
Disclaimer].

**Release 17.0, of 22-Dec-2001 (contains 1108 documentation entries that de
and profiles/matrices).**

Access to PROSITE

- by description
- by entry name or accession number (PSxxxxx or PDOCxxxxx number)
- by author
- by citation
- by full text search

- SRS - Sequence Retrieval System

Tools for PROSITE

- ScanProsite - Scan a sequence against PROSITE or a pattern against SW
- ProfileScan - Scan a sequence against the profile entries in PROSITE
- Other pattern and profile search tools

Services

- Downloading PROSITE by FTP

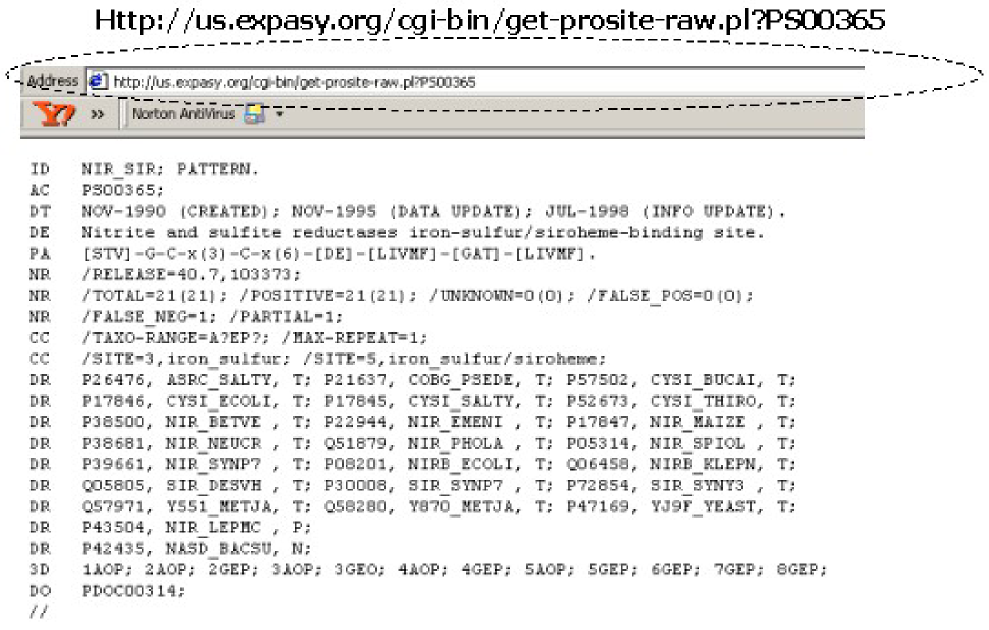
All instances that match with the provided accesion_number may be retrieved accessing this link.

The complete set of the instances of the MOTIF concept may be retrieved using this link.

Figure 65. ***PROSITE*** **First Page.**

Using the second link, PROSITE provides access to a file that stores the complete set of instances of the MOTIF. Figure 67 presents a fragment of the complete file.

Therefore, two iterators will be created (PrositeMotifAC_ID and PrositeMotif). *PrositeMotifAC_ID* will interact with the first accession_number presented in Figure 66 while *PrositeMotif* will retrieve all instances stored in the file presented in Figure 67.


Http://us.expasy.org/cgi-bin/get-prosite-raw.pl?PS00365
Address
http://us.expasy.org/cgi-bin/get-prosite-raw.pl?PS00365
Norton AntiVirus

```
ID   NIR_SIR; PATTERN.
AC   PS00365;
DT   NOV-1990 (CREATED); NOV-1995 (DATA UPDATE); JUL-1998 (INFO UPDATE).
DE   Nitrite and sulfite reductases iron-sulfur/siroheme-binding site.
PA   [STV]-G-C-x(3)-C-x(6)-[DE]-[LIVMF]-[GAT]-[LIVMF].
NR   /RELEASE=40.7,103373;
NR   /TOTAL=21(21); /POSITIVE=21(21); /UNKNOWN=0(0); /FALSE_POS=0(0);
NR   /FALSE_NEG=1; /PARTIAL=1;
CC   /TAXO-RANGE=A?EP?; /MAX-REPEAT=1;
CC   /SITE=3,iron_sulfur; /SITE=5,iron_sulfur/siroheme;
DR   P26476, ASRC_SALTY, T; P21637, COBG_PSEDE, T; P57502, CYSI_BUCAI, T;
DR   P17846, CYSI_ECOLI, T; P17845, CYSI_SALTY, T; P52673, CYSI_THIRO, T;
DR   P38500, NIR_BETVE , T; P22944, NIR_EMENI , T; P17847, NIR_MAIZE , T;
DR   P38681, NIR_NEUCR , T; Q51879, NIR_PHOLA , T; P05314, NIR_SPIOL , T;
DR   P39661, NIR_SYNP7 , T; P08201, NIRB_ECOLI, T; Q06458, NIRB_KLEPN, T;
DR   Q05805, SIR_DESVH , T; P30008, SIR_SYNP7 , T; P72854, SIR_SYNY3 , T;
DR   Q57971, Y551_METJA, T; Q58280, Y870_METJA, T; P47169, YJ9F_YEAST, T;
DR   P43504, NIR_LEPMC , P;
DR   P42435, NASD_BACSU, N;
3D   1AOP; 2AOP; 2GEP; 3AOP; 3GEO; 4AOP; 4GEP; 5AOP; 5GEP; 6GEP; 7GEP; 8GEP;
DO   PDOC00314;
//
```

Figure 66. *PROSITE URL S*howing Instances Matching a Given *Accession_Number*.

```
CC   /TAXO-RANGE=??E??;
CC   /SKIP-FLAG=TRUE;
DO   PDOC00003;
//
ID   CAMP_PHOSPHO_SITE; PATTERN.
AC   PS00004;
DT   APR-1990 (CREATED); APR-1990 (DATA UPDATE); APR-1990 (INFO UPDATE).
DE   cAMP- and cGMP-dependent protein kinase phosphorylation site.
PA   [RK](2)-x-[ST].
CC   /TAXO-RANGE=??E?V;
CC   /SITE=3,phosphorylation;
CC   /SKIP-FLAG=TRUE;
DO   PDOC00004;
//
ID   PKC_PHOSPHO_SITE; PATTERN.
AC   PS00005;
DT   APR-1990 (CREATED); APR-1990 (DATA UPDATE); APR-1990 (INFO UPDATE).
DE   Protein kinase C phosphorylation site.
PA   [ST]-x-[RK].
CC   /TAXO-RANGE=??E??;
CC   /SITE=1,phosphorylation;
CC   /SKIP-FLAG=TRUE;
DO   PDOC00005;
//
ID   CK2_PHOSPHO_SITE; PATTERN.
AC   PS00006;
DT   APR-1990 (CREATED); APR-1990 (DATA UPDATE); APR-1990 (INFO UPDATE).
```

Figure 67. Partial View of the *PROSITE MOTIF* File.

Consider the following query over the MOTIF concept:

Select Id, accession_number, pattern
From MOTIF
Where accession_number = 'PS0001'

This query provides a particular accession number such that the set of instances to be returned match with it.

If the iterator *PrositeMotifAC_ID* is used to answer this query, the search capabilities offered by the data source will be used. Also, only one instance will be transported from the data source to the user workspace. On the other hand, if the iterator *PrositeMotif* is used to retrieve instances of the query, all instances of the MOTIF concept will be retrieved from the data source to the workspace area, and INDUS will have to apply the selection operation by itself looking for those instances that match the given condition (accession_number='PS0001').

Now, consider this other query:

Select Id, accession_number, pattern
From MOTIF

Here, the query asks for all instances of the MOTIF concept.

The *PrositeMotifAC_ID* may be used for answering this query only if all possible accession numbers are provided. For each accession number, INDUS will ask for a search operation in the data source and only one row will be retrieved. On the other hand, *PrositeMotif* may answer this query only asking the data source once for all the set of instances for the MOTIF concept.

In conclusion, *PrositeMotifAC_ID* is a better choice for answering the first query while *PrositeMotif* is a better choice for answering the second query.

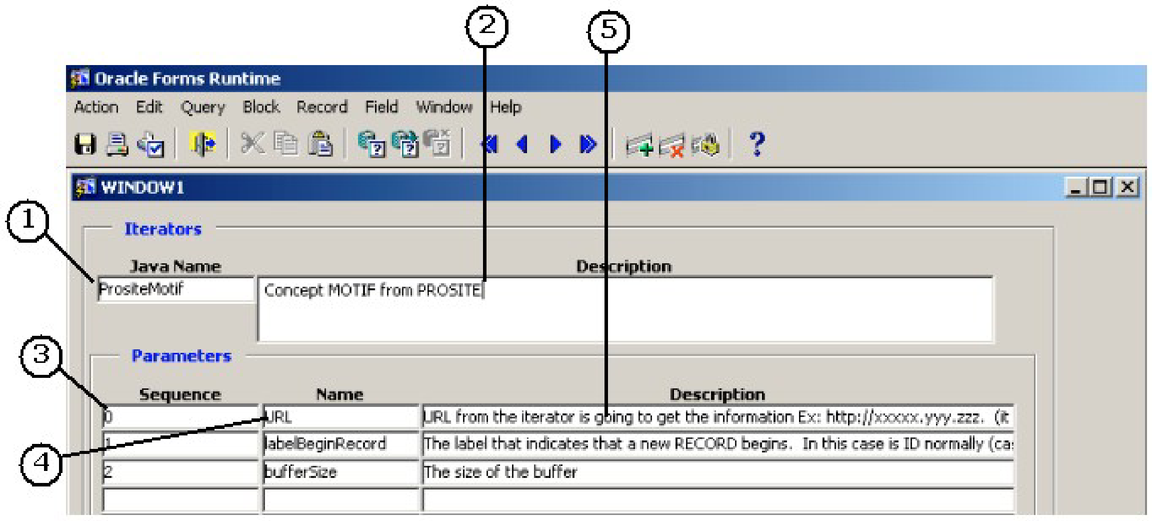
In general, iterators that are able to work with all the query functionality offered by data sources are desirable to be provided by ontology engineers.

Registering Iterators

Recall that in order to be able to define ground concepts, iterators must be created and registered in INDUS. Now we focus on the second step: Registering iterators in INDUS.

When an iterator is registered into the system, INDUS knows what java class must be executed and the set of parameters that controls the behavior of the iterator. This information is stored in the Common Global Ontology. Figure 68 presents the screen where this information is registered. The main steps are as follows:

(1) The java class name associated with the iterator is registered.
(2) A description of the functionality offered by the iterator is provided.
(3) Each parameter is specified. The sequence (corresponding to the *initialize* method) of the parameter is provided.
(4) A name is assigned to the parameter, and
(5) A description of the functionality offered by the parameter is registered.



Java Name	Description
PrositeMotif	Concept MOTIF from PROSITE

Sequence	Name	Description
0	URL	URL from the iterator is going to get the information Ex: http://xxxxx.yyy.zzz. (it
1	labelBeginRecord	The label that indicates that a new RECORD begins. In this case is ID normally (ca
2	bufferSize	The size of the buffer

Figure 68. Iterator Registration Screen.

Registering an iterator involves two tasks: to specify the java class to be invoked and to describes the set of parameters controlling the behavior of the iterator.

Defining Ground Concepts

After the iterators have been registered in INDUS, the ground concept can be created. Its structure and its operational definitions must be defined using the screen presented in Figure 69. The main steps are as follows:

(1) The basic information of the concept is created. This involves assigning an id, a name, a short name (used as table name when needed). Also, the set of ontologies to which the concept belongs is defined.

(2) The structure of the concept is defined.

(3) The set of instantiators that can be invoked in order to retrieve instances from the data sources are described. This is registered iterator by iterator.

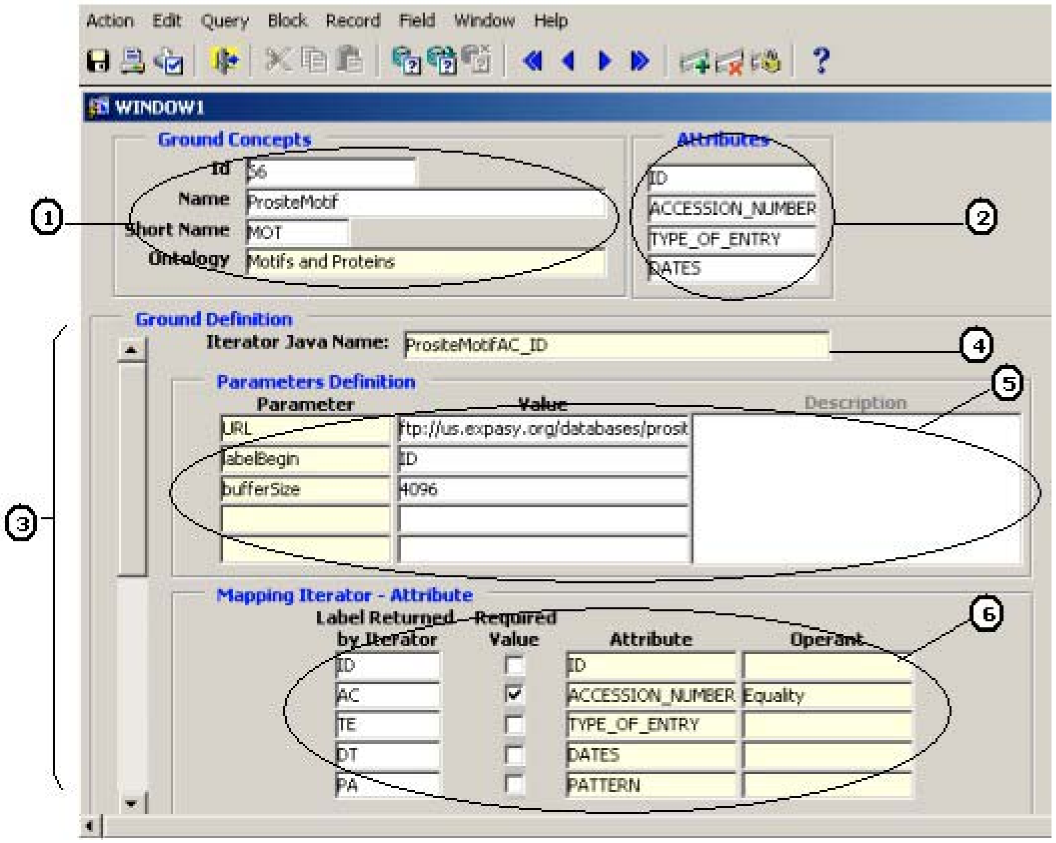
(4) One iterator is selected as the core of the instantiator from the set of registered iterators.

(5) A value must be assigned for each parameter defined for the iterator.

(6) Each label returned by the iterator is mapped to one of the attributes that belongs to the concept structure. The query capabilities associated with the instantiator is also registered here.

The example presented in Figure 69 corresponds to the *PrositeMotifAC_ID* iterator defined in our example previously. Recall that this instantiator retrieves the set of instances that match with a given accession number. In (6) this information is also registered. Note that the flag "Required Value" is on for the accession_number attribute. Similarly, the equality operation was defined for the same attribute. This means that this iterator requires a particular value of the accession number and that the set of instances that are going to be returned correspond to those that have a value equal to that provided to the iterator. In the literature, the kinds of variables associated with iterators are known as *bind variables*.

Figure 70 shows the operational definition of MOTIF for the *PrositeMotif* iterator. Recall that this instantiator returns all instances presented in the PROSITE data source. Therefore, no *bind variables* were defined in (6).



Ground Concepts

Id	56
Name	PrositeMotif
Short Name	MOT
Ontology	Motifs and Proteins

Attributes

ID
ACCESSION_NUMBER
TYPE_OF_ENTRY
DATES

Ground Definition

Iterator Java Name: PrositeMotifAC_ID

Parameters Definition

Parameter	Value	Description
URL	ftp://us.expasy.org/databases/prosit	
labelBegin	ID	
bufferSize	4096	

Mapping Iterator - Attribute

Label Returned by Iterator	Required Value	Attribute	Operant
ID	☐	ID	
AC	☑	ACCESSION_NUMBER	Equality
TE	☐	TYPE_OF_ENTRY	
DT	☐	DATES	
PA	☐	PATTERN	

Figure 69. Ground Concept Definition Screen. Iterator With a Condition.

When a ground concept is defined, the general information of the concept is introduced in section 1. In section 2 the structure of the concept (i.e., the attributes) is specified. In section 3 the instantiators that are able to retrieve instances for this concept are described. This last step involves assigning values for the required parameters for each instantiator, to specify the mapping between the information returned by each instantiator and the attributes of the concept and to specify the query capabilities.

3.2.3. Queries

Continuing with our example, after iterators have been created and registered in INDUS, and ground and compound concepts have been defined, the next step is to define and execute queries.

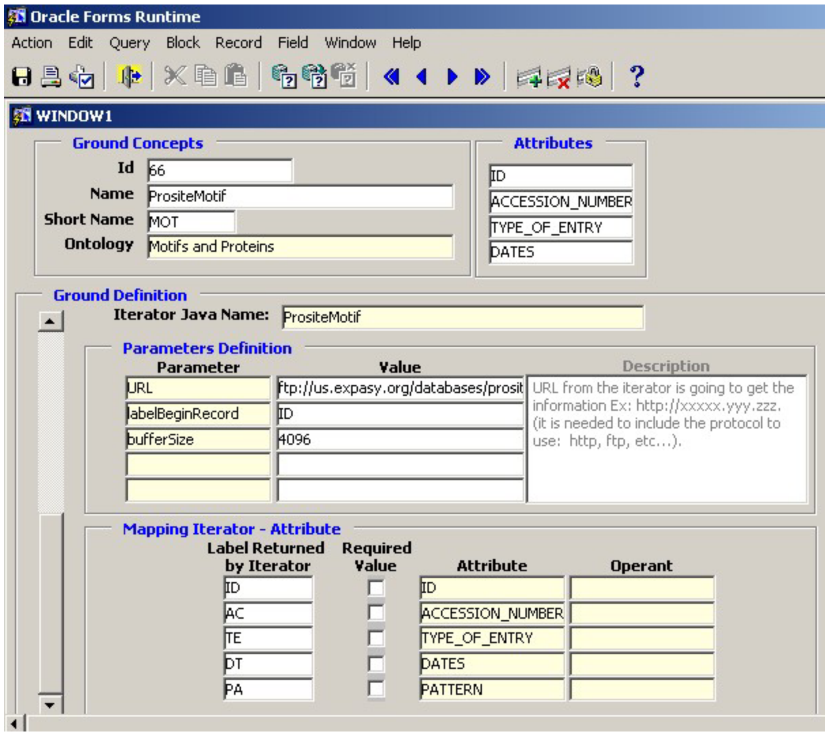
In our current prototype, queries are applied over a concept. In the definition of a query, a concept is selected, and a selection and projection operations are defined. The execution of a query results in retrieval of a set of instances from the concept that satisfies the query.

Query definitions are stored in the common global ontology area. Therefore, they can be re-used (e.g. the same query can be executed several times) as well as shared by different users. Figure 71 shows the screen provided by the graphical user interface for defining queries. The main steps defining queries are the following:

(1) An ontology and a concept are selected. An identification number is assigned automatically to the query.
(2) The projection operation is defined (i.e., the set of attributes to be projected).
(3) The set of conditions that the instances to be returned need to match are defined (i.e., the selection criteria).

The query defined in Figure 71 can be represented in SQL as follows:

Select motifid, proteinaccession_number, sequence, motif_protein
From Motifs_and_Proteins
Where motifid = 'PROKAR_LIPOPROTEIN'



Ground Concepts

Id	66
Name	PrositeMotif
Short Name	MOT
Ontology	Motifs and Proteins

Attributes

ID
ACCESSION_NUMBER
TYPE_OF_ENTRY
DATES

Ground Definition

Iterator Java Name: PrositeMotif

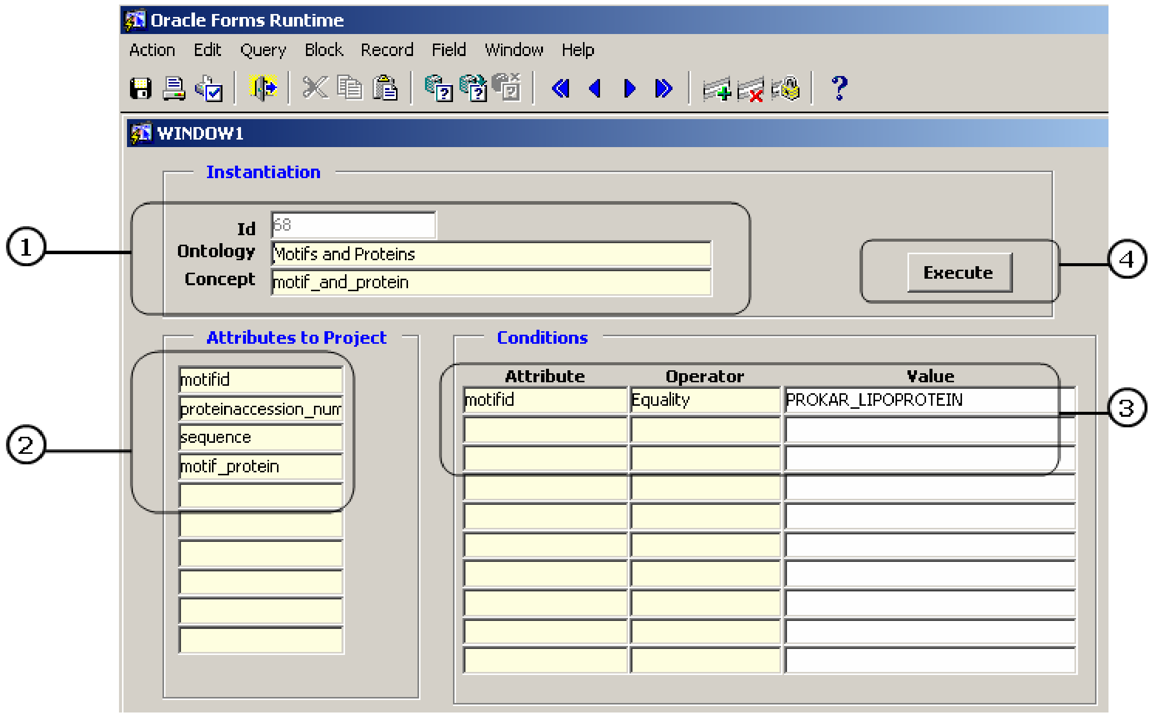
Parameters Definition

Parameter	Value	Description
URL	ftp://us.expasy.org/databases/prosit	URL from the iterator is going to get the information Ex: http://xxxxx.yyy.zzz. (it is needed to include the protocol to use: http, ftp, etc...).
labelBeginRecord	ID	
bufferSize	4096	

Mapping Iterator - Attribute

Label Returned by Iterator	Required Value	Attribute	Operant
ID	☐	ID	
AC	☐	ACCESSION_NUMBER	
TE	☐	TYPE_OF_ENTRY	
DT	☐	DATES	
PA	☐	PATTERN	

Figure 70. Iterator Without Conditions.



Id	68
Ontology	Motifs and Proteins
Concept	motif_and_protein

Attributes to Project:

motifid
proteinaccession_num
sequence
motif_protein

Attribute	Operator	Value
motifid	Equality	PROKAR_LIPOPROTEIN

Figure 71. Query Definition and Execution Screen.

3.2.4. Executing a Query

After a query has been defined, the execution of the query is triggered using the button named "Execution" presented in Figure 71 (4). Once the execution of the query is initiated , INDUS invokes a java class (called INDUS) that finds an equivalent rewriting of the query, expressed in terms of ground concepts and compositional operations, and proceeds to execute it.

Figure 72 presents the expression tree created for the example presented in the previous section.

When the tree has been created, the next step is to execute it. For the execution of the expression tree, the tree is traversed in post-order ordering. For each visited node, a set of actions are executed depending the whether or not it is a leaf node:

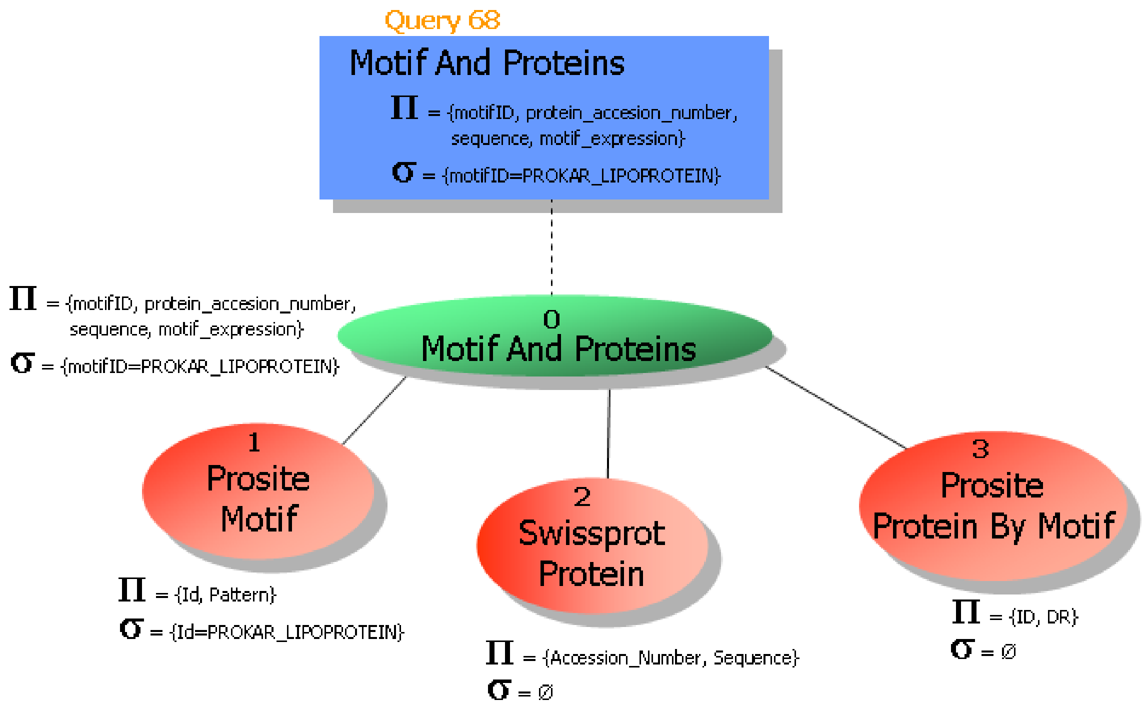

Query 68
Motif And Proteins
Π = {motifID, protein_accesion_number, sequence, motif_expression}
σ = {motifID=PROKAR_LIPOPROTEIN}
Π = {motifID, protein_accesion_number, sequence, motif_expression}
σ = {motifID=PROKAR_LIPOPROTEIN}
0 Motif And Proteins
1 Prosite Motif
2 Swissprot Protein
3 Prosite Protein By Motif
Π = {Id, Pattern}
σ = {Id=PROKAR_LIPOPROTEIN}
Π = {Accession_Number, Sequence}
σ = Ø
Π = {ID, DR}
σ = Ø

Figure 72. Expression Tree for the Example.

If the node is a leaf node, the following will occur:

- A table is created in the user workspace. This table contains all columns that appear in the list of attributes to be projected for the node.
- An instantiator is selected to populate the table. From those instantiator that have a subset of conditions assigned for the node, the instantiator that maximizes the covering of the set of conditions for the concept is selected.
- The selected instantiator is invoked and the retrieved rows are used to populate the table. Only those instances satisfying the set of conditions appearing in the list of conditions of the node are taken into account. Similarly, only information associated with elements in the list of attributes to be projected is retrieved.

If the node contains a compound concept, a relational view is created in the user workspace using SQL. Because the tree has been traversed using post-order ordering, all descendants would have been created before. Therefore, the view can be created safely.

```
executeTree(node){

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

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

Figure 73. Pseudo-algorithm for Executing the Expression Tree.

Note that a ground concept may appear in more than one leaf in the expression tree as shown in Figure 74. In this case, two tables will be created and populate independently. The reason for doing this (instead of creating only one table and two different views) is described as follows using the example presented in Figure 74.

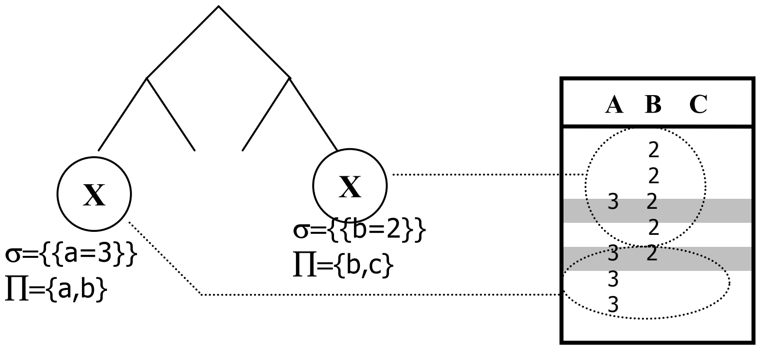


Figure 74. The Concept X Appearing in two Different Nodes of an Expression Tree.

Assume that both nodes were populated using different instantiator I1 and I2. I1 is able to resolve conditions over ((a=)) and I2 conditions over ((b=)). Note that the left node will select I1 and the right node will select I2. Assume that there is an instance in the data source with values 3 and 2 in A and B attributes respectively. This instance will be returned by both I1 and I2 instantiators. If only one table were used for representing this ground concept, this instance will appear twice in the table. Note that although two views V1 and V2 were created over the table X, they will return twice the same instance. The reason for that is because no information about primary key is included in the data source. Therefore, there is no way to determine that both instances are referring to the same "object". Thus, a better solution for this problem is to avoid this kind of conflict storing the information associated with each node in separates tables.

4. Summary and Discussion

This chapter presents a summary of the features of the system. After that, several related projects are presented and compared with INDUS using a set of characteristics. Finally, a list of possible future works is described.

4.1. Summary

The INDUS prototype provides a solution for the data integration problem in a scientific discovery environment. Therefore, data sources are assumed as distributed, autonomous and heterogeneous in structure and semantics. Also, multiple ontologies may be defined over a set of ground and compound concepts allowing users to represent different point of views for the scientific domain.

Among the principal characteristics of INDUS are the following:

- From a user's perspective, concepts stored in data source accessible through INDUS are represented as tables in which the rows correspond to instances and columns correspond to attributes, regardless of the structure of the underlying data sources.
- Each data source has associated with it, one or more data source specific sets of possible *ground instances*. Formally, each set of possible ground instances corresponds to the Cartesian product of a list of *domains*. A *ground co*ncept is simply a subset of the set of ground instances. Formally, a ground concept is analogous to a *relation* within a relational model of data. Each *domain* corresponds to the possible values for the corresponding *attribute* of the concept. Thus, specifying a ground concept amounts to associating the *name* of the concept with the corresponding attributes and (implicitly) the domains of the corresponding attributes. Since we assume autonomy of the data sources, we do not require that the complete domain of each attribute is necessarily known at the time a query is constructed (although specific values from some of the domains may be used to specify the constraints to be satisfied by the results of the query).

- A set of *instantiators* implements the operational definition of ground concepts. Those instantiators can retrieve instances of the concept by executing *iterators* (available set of abstract programs) supplying the appropriate parameters (e.g., given a protein identification number, it is possible to obtain *motifs* associated with that protein from PROSITE). The *instantiators* also define mechanism that transforms the structure in which instances are returned by iterators into the table supporting the concept. The query capabilities of ground concept in the data source are also described by the *instantiator*. This information is used by the query resolution algorithm to select appropriates data sources that guarantee a minimization of the number of instances to be transported from data sources to INDUS.
- The data sources need not all be data repositories that explicitly store the data. For example, they can be programs that accept parameters and return results. Each data repository may have associated with it, a set of operations or function calls on data that are directly supported by the repository. Examples of such *operations* include counts of instances that satisfy certain constraints on their attribute values. Thus, from outside, regardless of its internal structure (whether it is a data repository or a program), each data source appears as a set of ground concepts (and the associated iterators).
- *Compound concepts* are defined by the user using a set of compositional operations over ground concepts or previously defined compound concepts. The compound concepts that are currently supported include *selection* (which allows the user to define a new concept by selecting a subset of instances from a previously defined concept by specifying a set of constraints on the values of some of the attributes), *projection* (which allows the user to define a new concept in terms of the attributes of a previously defined concept), *vertical integration* (which corresponds to defining a new concept whose instances are constructed by combining the instances of previously defined concepts -- analogous to a join operation on the corresponding relations) and *horizontal integration* (which corresponds to taking the union of the corresponding relations). The last two operations help to resolve the problem of vertical partitioning and horizontal partitioning, often found in data integration systems. Since all data sources (including user-supplied programs) are viewed as relations, concepts can be associated with the results of an executable procedure or a black box.
- The system supports a library of functions including functions that can be data repository specific (e.g., a particular aggregate operation on the data). The application programs can utilize a set of functions from the library to process the information obtained from

the data repositories. Compositional operations and functions are two methods supplied by INDUS to help users to resolve syntactical and semantic heterogeneity.

- The system can include several *ontologies* at any time. With the exception of concepts defined as part of a data source specific ontology (which are associated with specific instantiators), the rest of the system is entirely open. In other words, individual users (e.g., biologists) can introduce new ontologies as needed. Each ontology consists of a set of ground and/or compound concepts and relations among them (which are defined by the operations used to combine concepts). The user can generate new ontologies by combining parts of existing ontologies. The ontologies are currently stored as form of meta-model schema in a database that is accessible to the users. Consequently, they can be easily retrieved and manipulated by users or application programs.
- The user or an application program can generate queries within an SQL-like query language. The queries are expressed in terms of the attributes associated with concepts defined within the chosen ontology. The system recursively rewrites the query into an execution tree whose leaves correspond to compositional operations applied on a ground concept. Any parameters passed to the query are passed down to the instantiator associated with the ground concept. When there are multiple instantiators associated with a ground concept, the system selects an optimal iterator based on a set of criteria. In the current prototype, the choice of the iterator is based on the parameters provided as part of the query and on the query capabilities associated with the instantiator.
- The system also allows inspection of the execution of a query by examining the state of nodes in the execution tree. Thus, for example, if the results of a specific query are null, it is possible to determine whether this was caused by absence of data that satisfy the query or by other problems (e.g., network failure, insolvable conditions).
- The results returned by a user query are *temporarily* stored in a local relational database. The concepts (relations) stored in this database are based on the user specified ontology. This in effect, represents a materialized relational view (modulo transformations that map concepts in the user ontology into concepts in data source specific ontologies) across distributed, heterogeneous (and not necessarily relational) data repositories. Unlike a data warehouse approaches, INDUS, 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).

- New data sources can be incorporated into INDUS by specifying the data source descriptions including the corresponding data-source specific ontology and the set of instantiators).
- INDUS has been developed using existent and common technology as java classes, relational databases, and http protocols. Several implementations of those technologies are open and free. Iterators (and hence instantiators) are implemented through a java class which are linked to the system at runtime using the reflection mechanism provided by java itself. Similarly, the algorithm for query transformation and execution was implemented completely using java. The meta-model where INDUS stores the repository of ontologies was implemented using standard relational databases. Similar technology was used to implement the user workspace area (where queries are materialized). Relational database technology is easily integrated to other technologies through *jdbc* and *odbc* protocols and proprietary APIs. They allow INDUS to support ontologies of any size. Similarly, answers to queries are managed and stored in a efficient way exploiting as much as possible the underlying technology.
- The modular design of INDUS ensures that each module can be updated and alternative implementation easily explored. Modularization also enables INDUS to use different network architectures. For example, INDUS may be implemented in a centralized architecture, locating all the modules in a unique server, or in an architecture where modules are distributed through several servers. This feature may be used for improving the overall performance of the system.

4.2. Comparison of INDUS with Other Systems

In what follows, several related projects are analyzed and compared with INDUS. A framework for this comparison is defined in terms of a set of characteristics that may be present in a data integration system. Such characteristics include the implemented infrastructure, the expressive power of the ontological layer, the heterogeneity associated with data sources, the query resolution algorithm and others.

4.2.1. The Framework

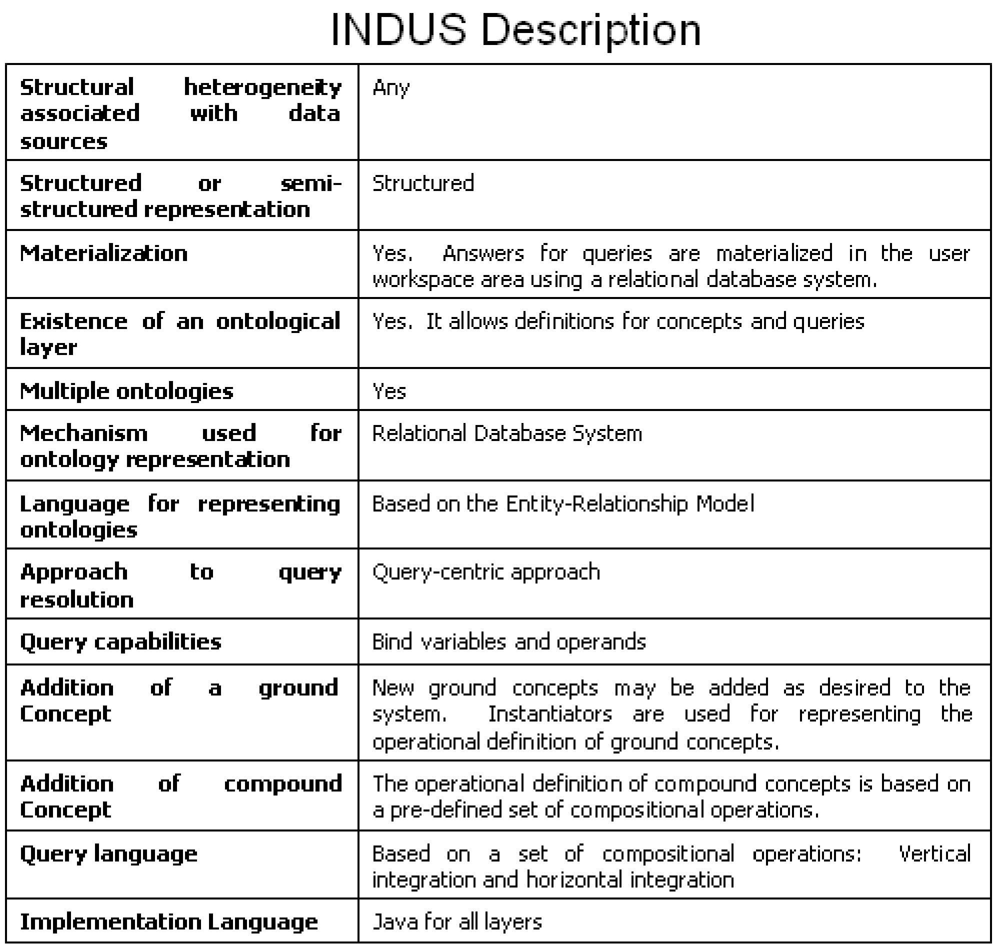
We can identify several key characteristics of data integration systems based on the general framework for data integration described in previous chapters. These are:

- **Structural heterogeneity associated with data sources:** Some systems are open to a broad range of data source regardless the underlying technology, including web pages, flat files, relational database systems. Other work focuses on specific types of data sources, e.g., relational databases in the case of multi-database systems.
- **Structured versus semi-structured representation:** In general, a data integration system implements a physical layer that receives a query in some language and returns the required information. Structured representation (e.g., relational) typically describes data in terms of a fixed set of attributes. On the other hand, semi-structured representation (e.g., XML) provides more flexible ways to describe data.
- **Materialization:** Depending on the physical layer architecture chosen, a data integration system may or may not materialize the answers for a posted query.
- **Existence of an ontological layer:** The ontological layer may or may not be present in a data integration system. When a system does not offer an ontological layer, queries must be defined directly over data sources. When an ontological layer is offered, users may define queries using their own language instead of use the terms specified in the data sources. The ontological layer also provides mechanism for resolving syntactical and semantic heterogeneity.
- **Multiple ontologies:** For some environments, the definition of multiple ontologies enables users to explore alternative concept representations over the same set of data sources. Thus, different point of views can be implemented.
- **Mechanism used for ontology representation:** Ontologies can be stored in a variety of methods or mechanism as flat files, relational database systems and others. Some mechanisms, as relational databases, enable users to post queries over the ontology. Similarly, some mechanisms are more suitable when large ontologies needs to be managed.
- **Language for representing ontologies:** A range of languages may be used to represent ontologies, concepts, relationships, and attributes. They may differs in the expressive power for representing ontologies, or for querying them. Some others are commonly used. Examples of such kind of languages are CLASSIC, GRAIL, ER, XML.
- **Approach to query resolution:** Two broad kind of query resolutions algorithms are commonly used in data integration systems: the query centric approach and the source centric approach. In chapter two, a comparison was presented between them.
- **Query capabilities:** Data sources may have different query capabilities. Some data integration systems describe those capabilities such that the query resolution algorithm

may take advantage of that information to find a better query rewriting or for avoid finding unrealizable plans.

- **Addition of a ground concept:** Some systems enable special kind of users to add new data sources by defining a wrapper or an instantiator. Some others do not permit the addition of new data sources.
- **Addition of compound concept:** A data integration system may offers some special mechanism enabling end users to define new concepts by themselves based on a set of predefined compositional operations. In other systems, the introduction of a new compound concept may require to create a computational program by a specialist.
- **Query language:** The expressive power of the query language may differ from one data integration system to another. The expressive power of the query language may be equivalent to the expressive power of relational algebra in a data integration system. In other, that expressive power may be equivalent to DataLog with recursion.
- **Implementation language:** Portability (e.g., platform independence) is an aspect related with the implementation language chosen for a data integration system. For example, java-based systems may run over a broad set of different platforms.

Figure 75 presents the description of INDUS based on the defined framework.



INDUS Description

Structural heterogeneity associated with data sources	Any
Structured or semi-structured representation	Structured
Materialization	Yes. Answers for queries are materialized in the user workspace area using a relational database system.
Existence of an ontological layer	Yes. It allows definitions for concepts and queries
Multiple ontologies	Yes
Mechanism used for ontology representation	Relational Database System
Language for representing ontologies	Based on the Entity-Relationship Model
Approach to query resolution	Query-centric approach
Query capabilities	Bind variables and operands
Addition of a ground Concept	New ground concepts may be added as desired to the system. Instantiators are used for representing the operational definition of ground concepts.
Addition of compound Concept	The operational definition of compound concepts is based on a pre-defined set of compositional operations.
Query language	Based on a set of compositional operations: Vertical integration and horizontal integration
Implementation Language	Java for all layers

Figure 75. INDUS Description.

4.2.1. TSIMMIS

The Stanford-IBM Manager of Multiple Information Sources (TSIMMIS) is a system that facilitates the rapid integration of heterogeneous data sources [TSIMMIS, 2002; Paton et al., 1999]. TSIMMIS is a federated system based on mediated-wrapper architecture that does not offer an ontological layer [Chawathe et al., 1994]. It implements the query resolution algorithm based on a Query-centric approach [Garcia-Molina et al., 1995].

The data integration TSIMMIS architecture is based on the concept of wrappers (sometimes called transformers) and mediators. Each wrapper is a piece of code that deals with a particular data source (called info-source), receives a query expressed in Object Exchange Model Query Language (OEM-QL), transforms the query into a language understandable for the data source, and retrieves the required information using a Object Exchange Model (OEM) [Quass et al., 1995; Garcia-Molina et al., 1995]. A mediator [Wiederhold,1992] is a piece of code that is able to deal with other mediators or wrappers in order to make some processing before and after sending information to the application or user [Garcia-Molina et al., 1995; Papakonstantinou et Al, 1995b]. A user (or an application) can interact with a wrapper or a mediator using the OEM-QL for querying.

Wrappers and mediators are able to handle structured and semi-structured data coming from the data source. Mediators or wrappers may use a local proprietary database for improving performance when the amount of information to be returned is large (instead of using principal memory). This data repository is known as the Lightweight Object Repository (LORE) and represents the information using a semi-structure architecture [Quass et al., 1995].

TSIMMIS does not offer a global schema or a general ontological layer. Each wrapper and mediator is an independent entity which uses OEM for representing the returned information. OEM uses a semi-structure architecture where each instance is stored along with a description of the information stored there. This description is based on a set of labels. In that sense, if a mediator makes use of two or more mediators or wrappers, similar labels used by sources may have different meaning, or similar information may be represented through different labels in different sources. The user is responsible for resolving these semantic heterogeneities [Papakonstantinou et al., 1995].

A toolkit for rapid development of mediators and wrappers is one of the principal design objectives of the TSIMMIS project [Yernani et al., 2000]. This work has also goals and results in other related areas as automatic generation of wrappers [Papakonstantinou et al., 1995b; Hammer et al., 1997] and mediators [Garcia-Molina et al., 1997], inter database constraint management [Chawathe et al., 1994b] and web browser interfaces for displaying objects.

TSIMMIS runs on Ultrix, AIX, and OSF operating systems. Developed under C and C++, it has around 9000 lines and 4000 lines for Client/Server support libraries [Yerneni et al., 2000].

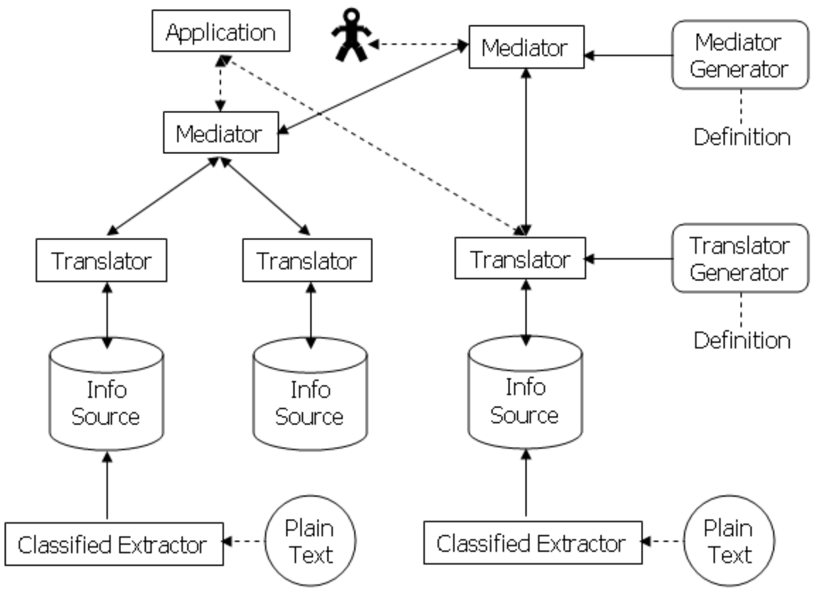
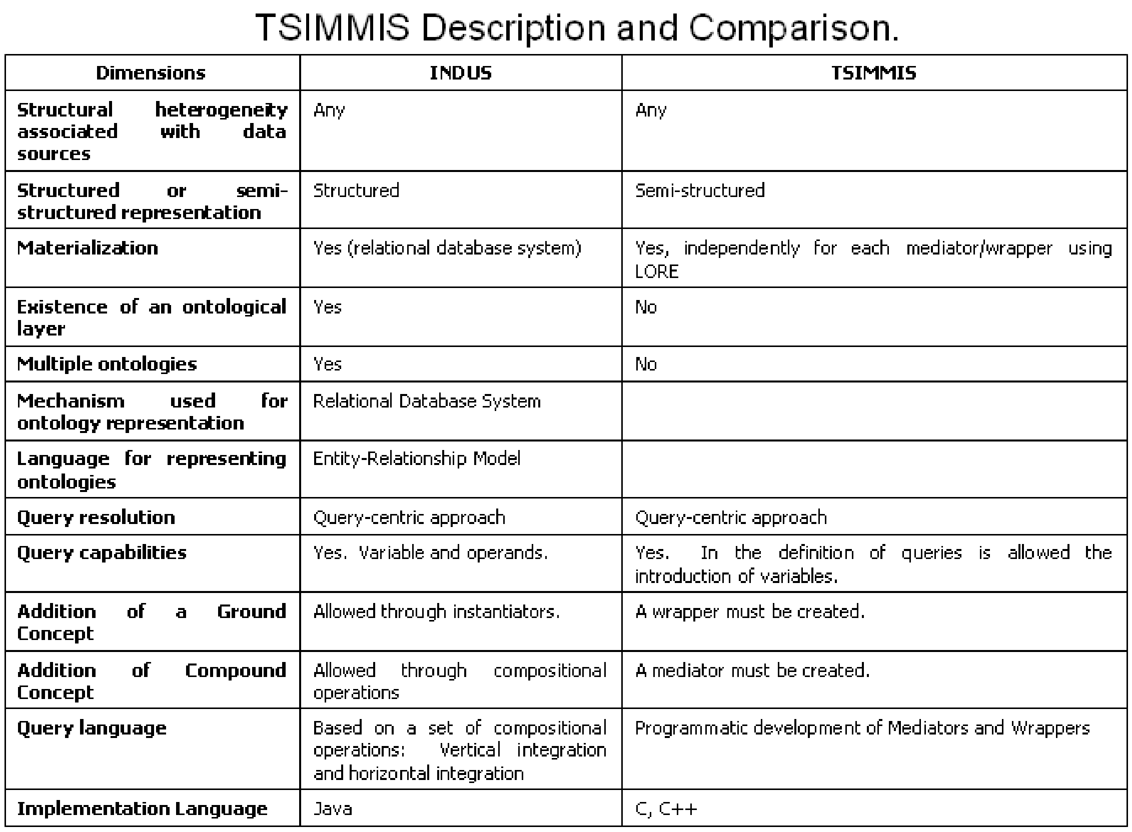


Figure 76. TSIMMIS Architecture (Taken From Garcia-Molina et al, [1995]).



TSIMMIS Description and Comparison.

Dimensions	INDUS	TSIMMIS
Structural heterogeneity associated with data sources	Any	Any
Structured or semi-structured representation	Structured	Semi-structured
Materialization	Yes (relational database system)	Yes, independently for each mediator/wrapper using LORE
Existence of an ontological layer	Yes	No
Multiple ontologies	Yes	No
Mechanism used for ontology representation	Relational Database System	
Language for representing ontologies	Entity-Relationship Model	
Query resolution	Query-centric approach	Query-centric approach
Query capabilities	Yes. Variable and operands.	Yes. In the definition of queries is allowed the introduction of variables.
Addition of a Ground Concept	Allowed through instantiators.	A wrapper must be created.
Addition of Compound Concept	Allowed through compositional operations	A mediator must be created.
Query language	Based on a set of compositional operations: Vertical integration and horizontal integration	Programmatic development of Mediators and Wrappers
Implementation Language	Java	C, C++

Figure 77. TSIMMIS Description and Comparison.

Figure 77 presents summarize the main similarities and differences between INDUS and TSIMMIS. The principal differences between INDUS and TSIMMIS are:

- INDUS uses a structured model (relational tables) for representing instances whereas TSIMMIS use a semi-structured model (OEM);
- INDUS materializes answers in a relational database system whereas TSIMMIS use a proprietary database system called LORE. Relational database system is mature technology and several implementations (some free) are available;
- An ontological layer is present in INDUS but not in TSIMMIS. Therefore, semantic heterogeneities must be resolved in TSIMMIS by the user when two or more sources (mediators or wrappers) use the same label for representing different information or if the same information is represented using different labels by different sources. In TSIMMIS, If a user wants to use a mediator, he/she must known OEM-QL;
- In INDUS, bind variables and operands can be specified when describing the query capabilities of data sources. In TSIMMIS, only bind variables can be specified;
- One or more instantiators must be defined (or reused) when a new data source is added to INDUS. Similarly, TSIMMIS requires a wrapper for adding a new data source. INDUS provides a language that enables end users to define new compound concepts. In TSIMMIS, a mediator must be created (for an expert user) if a new compound concept is required.
- INDUS is implemented using java and relational database systems, both open technologies. TSIMMIS uses C and C++ for defining mediators and wrappers and LORE for materializing queries.

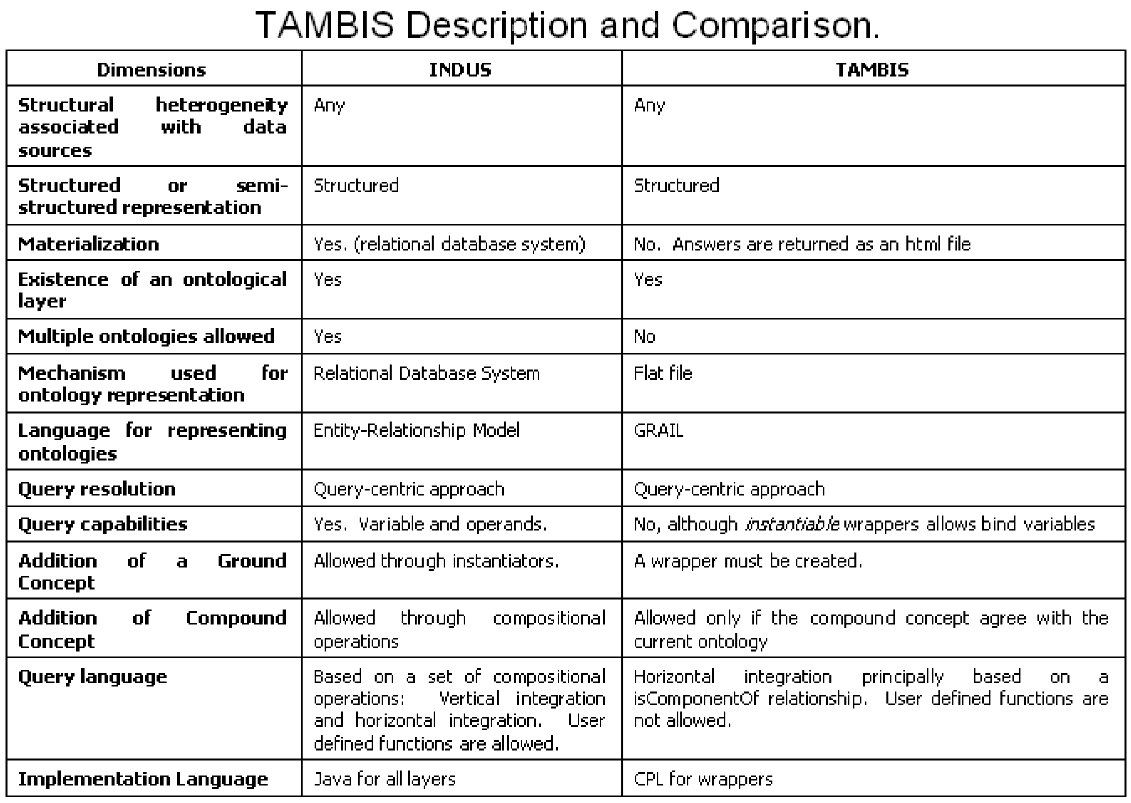
4.2.2. TAMBIS

The Transparent Access to Multiple Bioinformatics Information Sources (TAMBIS) is an ontology centered system querying multiple heterogeneous bioinformatics sources [Paton et al., 1999]. It can be classified as a federated data integration system implemented through a wrapper/mediator architecture which does not materialize answers for queries. It includes a general ontological layer and implements a query-centric approach algorithm for query resolution [Baker et al., 1998; Paton et al., 1999; Stevens et al., 2000; Baker et al., 1999; Stevens et al., 1999].

The ontological layer is known as the Biological Concept Model and it is used to represent biological concepts and their relationships. The ontology allows the creation of new concepts based on compositional operations of previously defined concepts guided by the ontology using a restricted grammar based on the description logic language GRAIL [Stevens et al., 2000]. A file representing the ontology is associated with the system. Therefore, only one general ontology is allowed at any given time.

Queries are defined through a ontological driven graphical user interface. This interface verifies that any defined query agree with the ontology (e.g., only allowed relationships between concepts can be used) [Paton et al., 1999]). Similar situation occurs when new compound concepts are defined. Both, the ontology and queries are transformed to GRAIL internally.

The wrapper/mediator system is based on a data integration system called CPL/BIOKLIESLY [Baker et al., 1998]) which is responsible for executing the needed wrappers to get the data. This data are returned to the system in the form of an html text file.



TAMBIS Description and Comparison.

Dimensions	INDUS	TAMBIS
Structural heterogeneity associated with data sources	Any	Any
Structured or semi-structured representation	Structured	Structured
Materialization	Yes. (relational database system)	No. Answers are returned as an html file
Existence of an ontological layer	Yes	Yes
Multiple ontologies allowed	Yes	No
Mechanism used for ontology representation	Relational Database System	Flat file
Language for representing ontologies	Entity-Relationship Model	GRAIL
Query resolution	Query-centric approach	Query-centric approach
Query capabilities	Yes. Variable and operands.	No, although *instantiable* wrappers allows bind variables
Addition of a Ground Concept	Allowed through instantiators.	A wrapper must be created.
Addition of Compound Concept	Allowed through compositional operations	Allowed only if the compound concept agree with the current ontology
Query language	Based on a set of compositional operations: Vertical integration and horizontal integration. User defined functions are allowed.	Horizontal integration principally based on a isComponentOf relationship. User defined functions are not allowed.
Implementation Language	Java for all layers	CPL for wrappers

Figure 78. TAMBIS Description and Comparison.

Figure 78 presents summarize a comparison between INDUS and TAMBIS. The principal differences between INDUS and TAMBIS are:

- INDUS materializes answers for queries using a relational database system in the user workspace area. TAMBIS, returns the answers as a html file. Relational databases have several advantages with respect to html files. For example, the information can be easily queried, modified, and shared. Also, they provides reliable and optimal mechanisms for manipulating large amount of data and they are easily accessible through several open protocols as jdbc and odbc.
- INDUS provides a mechanism for defining multiple ontologies. In TAMBIS, only one ontology is allowed at any given time.
- INDUS supports bind variables and operands in the description of the query capabilities of data sources. TAMBIS uses a specific mechanism called *instantiated* wrappers for bind variables.

- In INDUS, users may define compound concept applying a set of compositional operations. In TAMBIS, the composition of concepts is restricted to those that agree with the ontology previously defined.

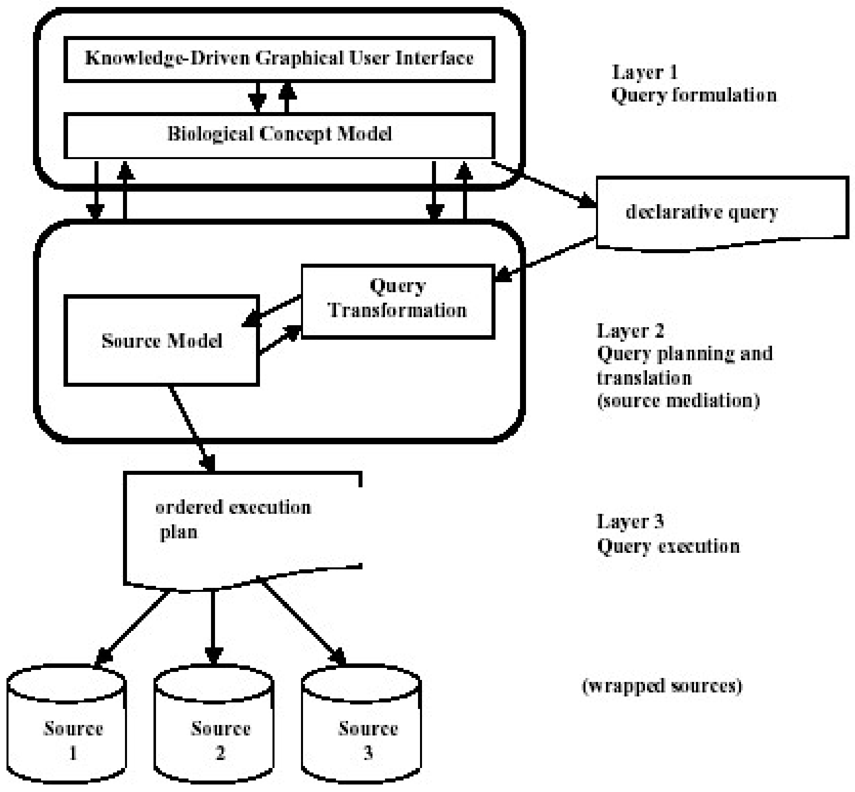


Figure 79: TAMBIS Architecture (Taken From Stevens et al., [2000]).

4.2.3. Information Manifold System

The Information Manifold System is a heterogeneous data integration system offering a unified query interface for retrieving structured information stored in the World Wide Web and in internal sources at AT&T Bell laboratories [Levy et al., 1996]. It uses a Source-centric approach for query resolution implemented through The Bucket algorithm.

Its goals are to provide a scalable mechanism for introducing new data source in a flexible way and to offer a query mechanism for retrieving all the relevant information from data sources

[Levy, 1998]. A Source-centric approach is used in the implementation of the query rewriting algorithm.

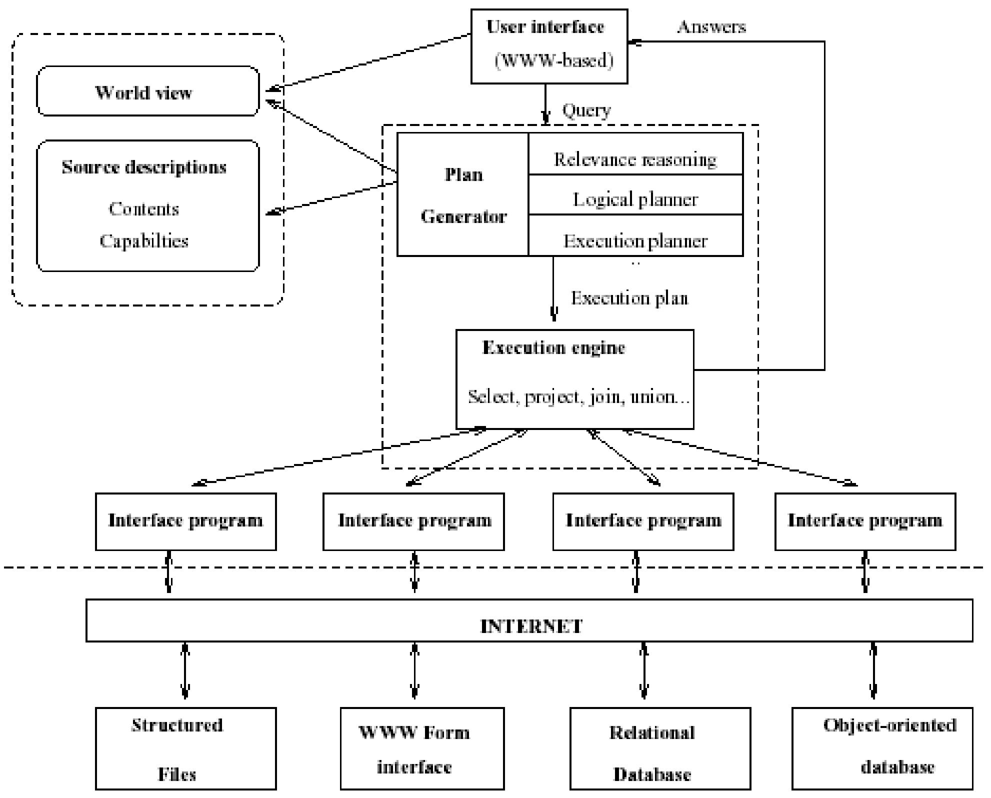


Figure 80. Information Manifold Architecture (Taken From Levy et al., [1996]).

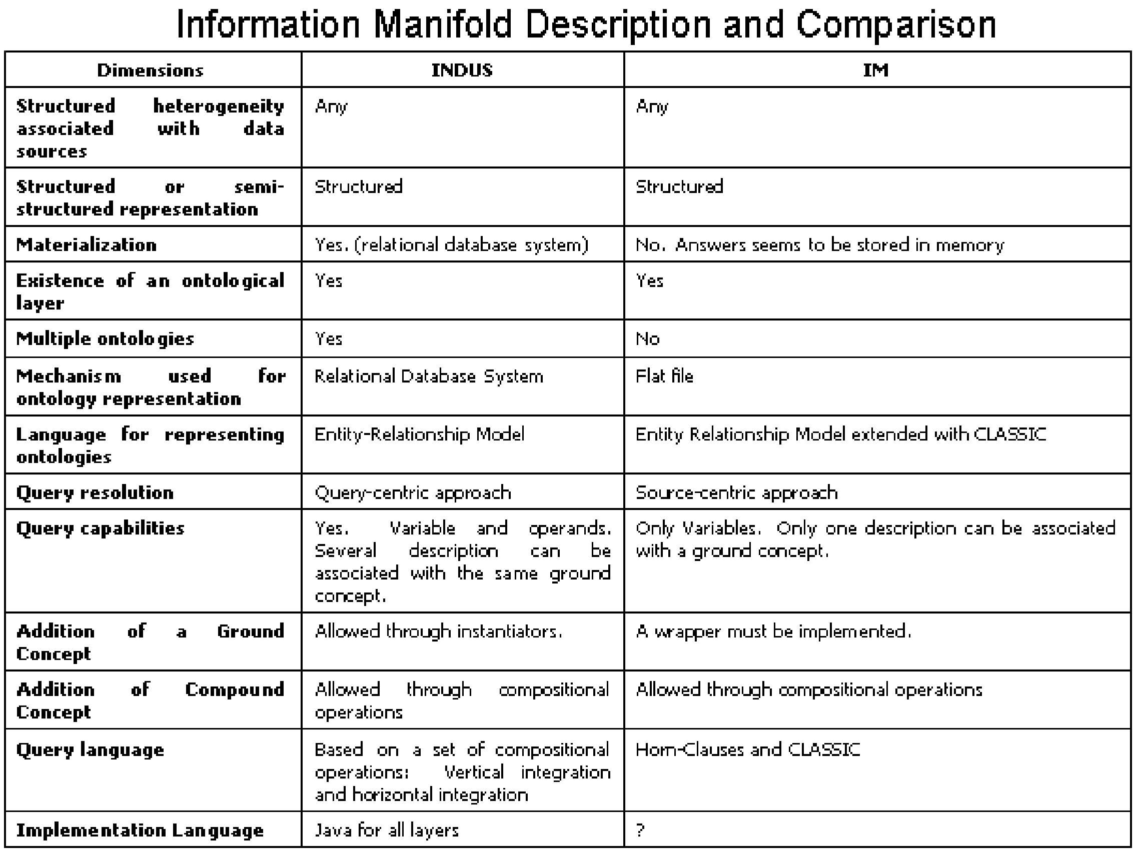
Information Manifold uses an entity relationship technique (concepts and attributes) augmented with object-oriented features (for defining class-subclass relationships) to represent ontologies. The resulting ontologies are encoded using Horn clauses and description logic CLASSIC [Kirk et al., 1995]

Information Manifold provides a web interface where categories (concepts) can be browsed by users. This interface allows users to post queries over the global model. [Levy et al., 1996], and add new data sources to the knowledge base module [Kirk et al., 1995].

Unlike INDUS, where each data source may have associated with it multiple query capability definitions (one for each instantiator), Information Manifold allows only one such description per data source.

The literature on Information Manifold does not explicitly discuses the mechanism for storing the results of a query. However, the mentioning of "stream of answers" on Levy et al. [1996] suggest that main memory is probably used for storing results. In INDUS, results are materialized in the user workspace are using a relational database system.

Information Manifold attempt to minimize the time required for obtaining some instances that satisfy the user query as opposed to minimizing the total time needed to answer the query. This enables a user to terminate execution of a query if results do not match the user expectation.



Information Manifold Description and Comparison

Dimensions	INDUS	IM
Structured heterogeneity associated with data sources	Any	Any
Structured or semi-structured representation	Structured	Structured
Materialization	Yes. (relational database system)	No. Answers seems to be stored in memory
Existence of an ontological layer	Yes	Yes
Multiple ontologies	Yes	No
Mechanism used for ontology representation	Relational Database System	Flat file
Language for representing ontologies	Entity-Relationship Model	Entity Relationship Model extended with CLASSIC
Query resolution	Query-centric approach	Source-centric approach
Query capabilities	Yes. Variable and operands. Several description can be associated with the same ground concept.	Only Variables. Only one description can be associated with a ground concept.
Addition of a Ground Concept	Allowed through instantiators.	A wrapper must be implemented.
Addition of Compound Concept	Allowed through compositional operations	Allowed through compositional operations
Query language	Based on a set of compositional operations: Vertical integration and horizontal integration	Horn-Clauses and CLASSIC
Implementation Language	Java for all layers	?

Figure 81. Information Manifold Description and Comparison.

As is shown in Figure 81, the principal difference between INDUS and Information Manifold are:

- INDUS uses a Query-centric approach whereas Information Manifold uses a Source-centric approach;
- Multiple ontologies are allowed in INDUS but not in Information Manifold;
- Unlike Information Manifold, where not materialization is provided for answers, INDUS uses a relational database system for materializing answers for queries;
- The description of the query capabilities in INDUS includes bind variables and operand whereas in Information Manifold only variables are allowed.

4.2.4. SIMS

The Services and Information Management for Decision Systems (SIMS) is a multi-database integration system that also integrates knowledgebase systems [Arens et al., 1993b], flat files, some types of programs [Arens et al., 1996], and Web Pages [Ambite et al., 1998]. The data integration infrastructure is implemented using a wrapper/mediator architecture based on a KQML-CORBA technology and a Source-centric approach is used in the query resolution algorithm.

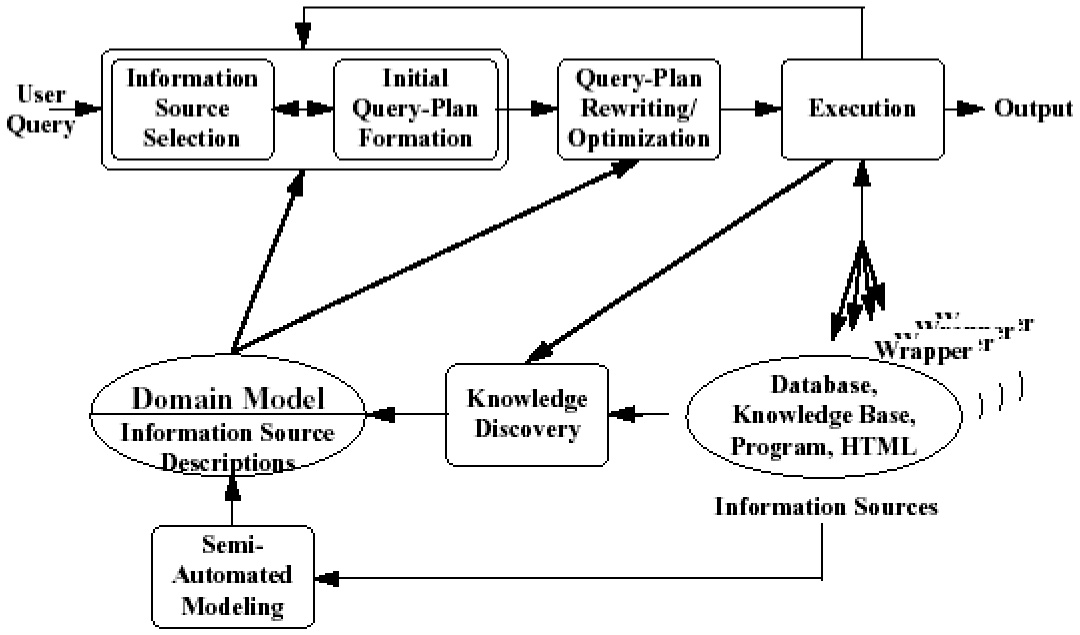

User Query
Information Source Selection
Initial Query-Plan Formation
Query-Plan Rewriting/ Optimization
Execution
Output
Wrapper
Domain Model
Information Source Descriptions
Knowledge Discovery
Database, Knowledge Base, Program, HTML
Information Sources
Semi-Automated Modeling

Figure 82. SIMS overview diagram (Taken From Ambite et al., [1998]).

Like INDUS, SIMS provides a global ontology (called domain model) where a set of terms (concepts), attributes and relationships are defined. Nevertheless, INDUS and SIMS differ in the mechanism used to represent the ontologies. The former uses an ER model implemented in a relational database whereas the later uses description logic LOOM to represent the ontologies in a flat file.

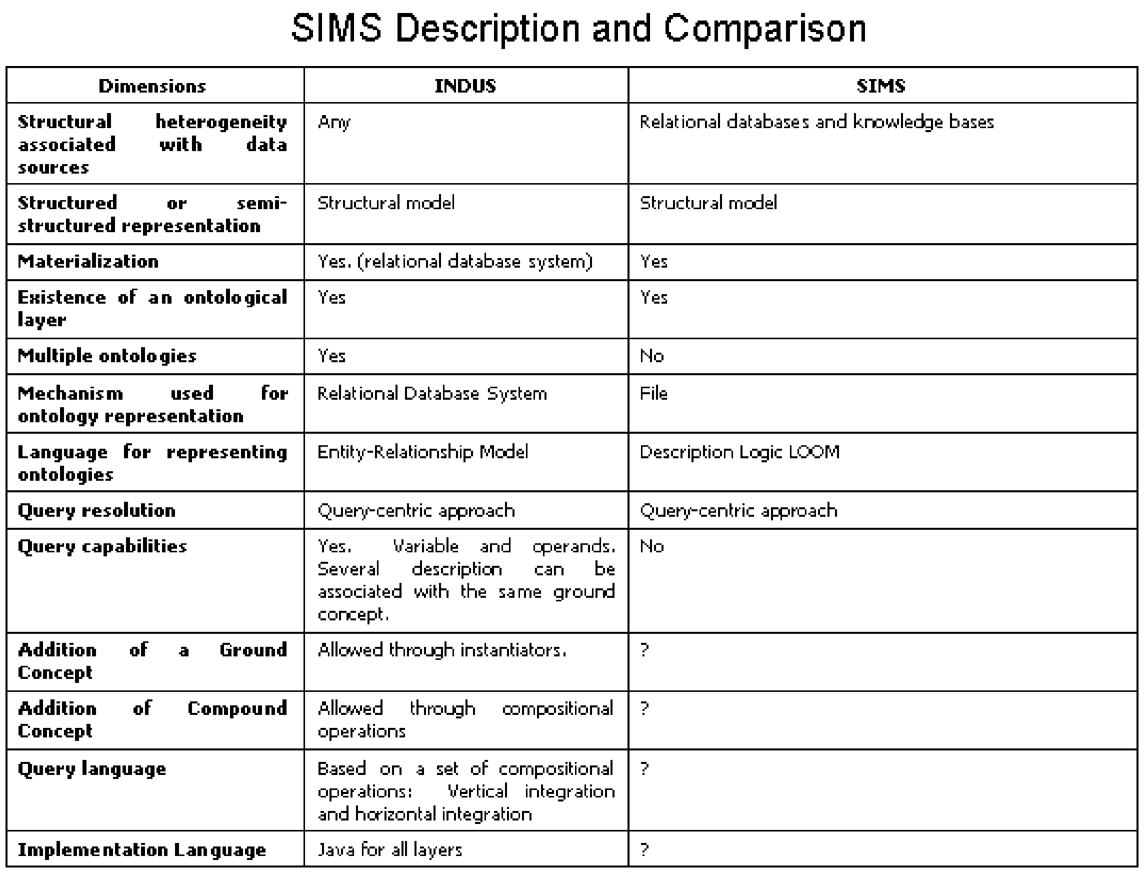
SIMS allows several kinds of relationships including subclass, superclass, covering and join in the ontology. However, each concept is restricted to belong to a unique super-class [Ambite et al., 1998]. Thus, contrary to the statement made in [Arens et al., 1996], query resolution in SIMS appears to be closer to a Query-centric approach as opposed to a Source-centric approach. INDUS implements relationships indirectly when compound concepts are defined. Subclass and superclass relationships are defined in INDUS through a horizontal integration operation and. the join relationship is implemented through the vertical integration operation in INDUS.

SIMS provides two languages for defining queries: the description logic LOOM, and a subset of SQL which includes join, selection, projection (union is not supported). Independently of the language used for queries, queries that are based in more than one concept restricted combine the concepts using relationships defined in the ontology. When a query is based on more than one concept, INDUS enable users to combine them applying any compositional operations valid in the language.

SIMS assumes that data sources offer full query capabilities relative to the query language. INDUS has the ability to interact with data sources with restricted query capabilities.

The planning phase offered by SIMS includes several optimization steps, including semantic optimization, where multiple equivalent resolution plans are evaluated.

SIMS was developed by the Information Agents Research Group of the Information Science Institute at the University of Southern California [Arens et al., 1993b] using Common Lisp running over Unix and later migrated to Allegro Common List 4.3 under Solaris on SUN workstations [Ambite et Al, 1998].



SIMS Description and Comparison

Dimensions	INDUS	SIMS
Structural heterogeneity associated with data sources	Any	Relational databases and knowledge bases
Structured or semi-structured representation	Structural model	Structural model
Materialization	Yes. (relational database system)	Yes
Existence of an ontological layer	Yes	Yes
Multiple ontologies	Yes	No
Mechanism used for ontology representation	Relational Database System	File
Language for representing ontologies	Entity-Relationship Model	Description Logic LOOM
Query resolution	Query-centric approach	Query-centric approach
Query capabilities	Yes. Variable and operands. Several description can be associated with the same ground concept.	No
Addition of a Ground Concept	Allowed through instantiators.	?
Addition of Compound Concept	Allowed through compositional operations	?
Query language	Based on a set of compositional operations: Vertical integration and horizontal integration	?
Implementation Language	Java for all layers	?

Figure 83. Sims Description and Comparison.

4.2.5. Observer

The Ontology Based System Enhanced with Relationships for Vocabulary hEterogeneity Resolution (OBSERVER) [Mena et al., 2000; OBSERVER, 2002] is an ontology integrator system offering a unified platform for querying multiple ontologies. The ontologies are combined using inter-ontology relationships such as synonyms, hypernyms, hyponyms and non-exact relationships (where the relationship between two concepts cannot be exactly captures by any of the previously mentioned relationships).

Unlike other systems described in this chapter, OBSERVER is focused on supporting multiple ontologies instead of offering a unique global ontology because: (a) maintaining and supporting a set of small ontologies is easier and (b) when multiple ontologies are available, it may be more efficient to integrate them as needed as opposed to combining them into a global ontology

Ontologies are defined using the description logic CLASSIC where concepts are defined by two possible predicates: *Primitive* and *Defined*. *Primitive* concepts are defined by set necessary conditions and *defined* concepts are specified in terms of sufficient conditions

The system is able to determine and maintain the hierarchical organization of the concepts (called terms) and incorporates concepts defined by users (in the process of generating a query) into the hierarchy.

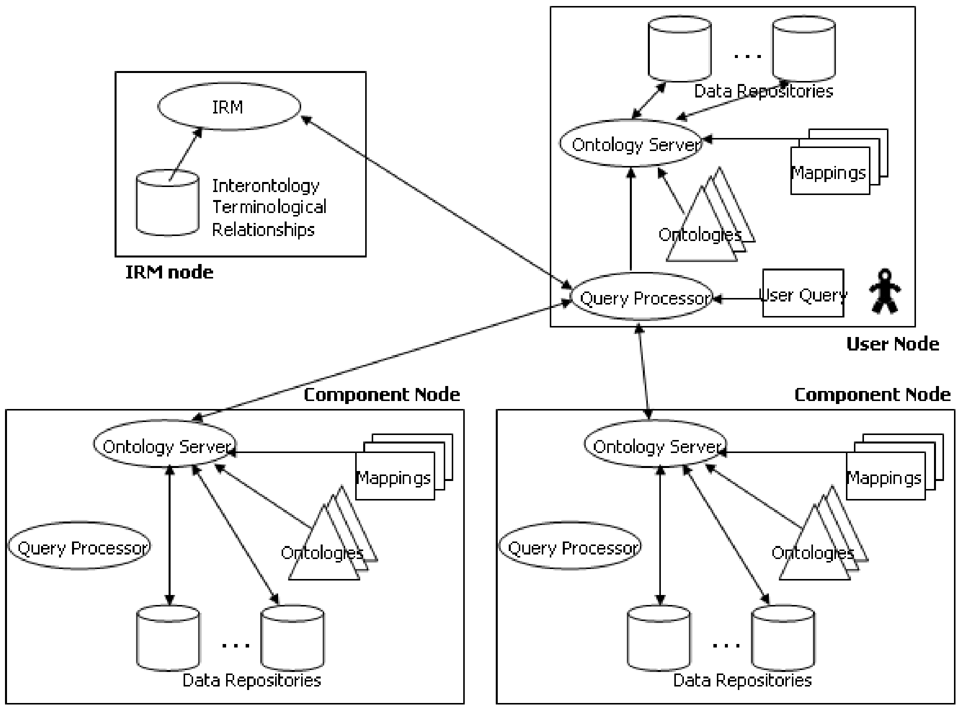


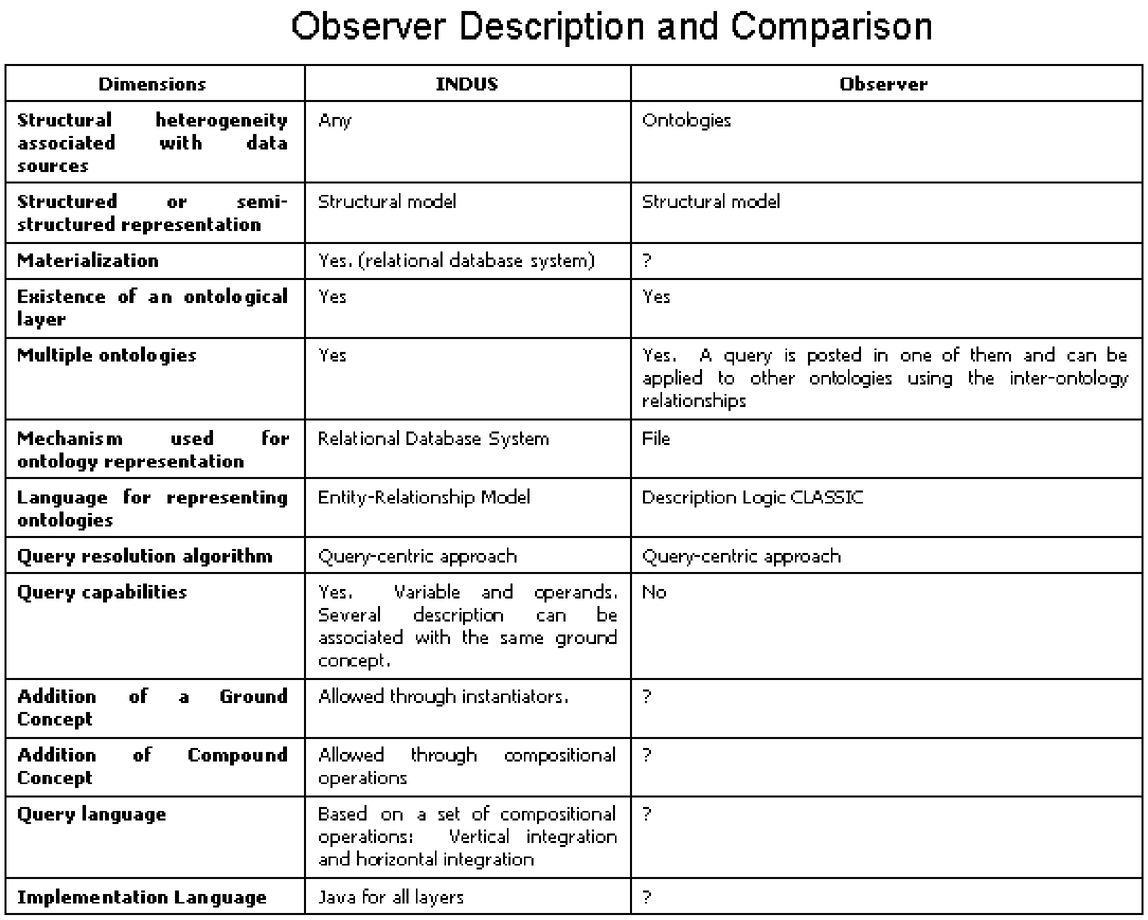
Figure 84. OBSERVER Architecture (Taken From Mena et al., [2000]).

In order to post queries, a user selects one of the available ontologies, which are organized by clusters, and introduces the query using description logic. The system translates this query into a set of basic terms that are stored in data sources following the information defined in the ontology using the relationships. A plan is created for resolving the query (using LQL and Extended ER) and executed in each data source. The resulting information is stored in a (temporal) database where additional operations (defined in the plan) are applied. Finally, results are returned to the systems to be shown to the final user. If the user is not satisfied with the answer, he/she can add new ontologies to the query and the system translates the posted query using terms defined in other ontologies, using the inter-ontology relationships (which also includes data about "loss of information"), and the process is repeated.

In conclusion, OBSERVER uses a Query-centric approach for query resolution, using a wrapper architecture for interacting with data sources. It is unclear whether the system provides the means to describe the query capabilities of data sources.

Like INDUS, OBSERVER supports multiple global ontologies simultaneously. Nevertheless, OBSERVER introduces mechanisms for translation between terms of different ontologies including data about "losing information". In OBSERVER, modification of the ontologies is done by users through the definition of queries

OBSERVER uses an extended relational algebra (including counts). In our current prototype, counts can be implemented through iterators or by creating a user-defined function.



Observer Description and Comparison

Dimensions	INDUS	Observer
Structural heterogeneity associated with data sources	Any	Ontologies
Structured or semi-structured representation	Structural model	Structural model
Materialization	Yes. (relational database system)	?
Existence of an ontological layer	Yes	Yes
Multiple ontologies	Yes	Yes. A query is posted in one of them and can be applied to other ontologies using the inter-ontology relationships
Mechanism used for ontology representation	Relational Database System	File
Language for representing ontologies	Entity-Relationship Model	Description Logic CLASSIC
Query resolution algorithm	Query-centric approach	Query-centric approach
Query capabilities	Yes. Variable and operands. Several description can be associated with the same ground concept.	No
Addition of a Ground Concept	Allowed through instantiators.	?
Addition of Compound Concept	Allowed through compositional operations	?
Query language	Based on a set of compositional operations: Vertical integration and horizontal integration	?
Implementation Language	Java for all layers	?

Figure 85. Observer Description and Comparison.

4.3. Future Work

In this thesis we have presented a data integration model for a scientific discovery environment which is based on a federated database architecture, a query-centric approach for query resolution, an ontological layer for resolving syntactical and semantic heterogeneities, and a graphical user interface that enable users to post queries and receive answers over a set of distributed, autonomous and heterogeneous data sources. Based on such a model, a fully operational prototype was developed implementing the data integration component of INDUS, a scientific discovery environment.

In this section we outline two broad areas of future work: (a) Extension and improvements to the data integration module of INDUS and (b) The development of the other modules of INDUS (e.g., data mining algorithms) .

Some promising directions for work on other modules of INDUS include:

- Development of machine learning algorithms that can operate across multiple distributed heterogeneous data sources [Caragea et al., 2000]. There are two approaches to this problem: (a) Use of the data integration component of INDUS to supply the data needed by the learning algorithm, and (b) Development of distributed learning algorithm based on a decomposition of the learning task into an information extraction and hypothesis generation component. In the later case, the instantiators in INDUS have to be adapted to perform the necessary information extraction task (e.g., obtaining the statistics needed from distributed data sources).
- Implementation of relational learning algorithms [Leiva et al., 2002]. In order to implement this module, the ontological layer along with the query resolution algorithm supported by the data integration module can provide an infrastructure where relational algorithms may be applied over non-relational autonomous, distributed and heterogeneous data sources.
- Development of ontology driven learning algorithms [Zhang et al., 2002]. The ontological layer implemented in our current prototype may provide a knowledge base where those kinds of algorithms may inspect different ontologies to drive the learning process.
- Applications of INDUS to data driven knowledge discovery in biological sciences (e.g., exploration of macro molecular structure-function relationships and interactions).

The current data integration module can be improved in several aspects that includes:

- Optimization of the query resolution algorithm:
    - The mechanism for selecting the instantiator for a leaf in an expression tree is based on the selection and projection conditions associated with the leaf. Additional information may be used to improve this selection including statistics related with performance of accessing the correspondent data source.
    - Sometimes a data source may be inaccessible. In that case, alternative instantiators can be selected based on provided criteria. Another related situation occurs when the same ground concept appears in more than one leaf in

an expression tree. If the set of conditions (and the query capabilities) of the available data sources are able to provide instances satisfying simultaneously both leaves, the selection of the instantiator should take into account the global requirements instead of the local requirements of each leaf.

  - Our current algorithm for creating an expression tree for a posted query, migrates as much as possible conditions from a parent-node to a child-node. Nevertheless, some conditions can be inferred based on the set of known conditions. As an example, if the following condition is given (A1=3, A1=B1) where A1 and A2 are attributes from child A and B1 is an attribute from the child B, in the current implementation the condition (A1=3) will be copied to the child A but no conditions are migrated to the child B. Nevertheless, the condition (B1=3) should be migrated to the child B. Thus, there is room for further optimizations at this step.
  - In our current implementation, the execution of the expression tree is carried out node by node. Parallel executions (e.g., using threads) may be used to reduce the total time for retrieving the required set of instances and creating the needed tables and views.
  - In our current prototype, when a query is posted, a plan is created and executed. If the set of instances required to answer a posted query were previously retrieved (and stored in the user workspace area) answering another query, the current implementation is going to retrieve the required set of instances from the data sources instead of gathering the required information from the user workspace area.

- Instantiators and instance representation:
  - We have not provided any tool for automatic generation of instantiators. Any new instantiator must be programmed. Our experience indicates that instantiators that interact with particular types of data sources (e.g., web pages, ftp flat files, relational databases) may be at least partially automated. The same situation hodls when the interface provided by a data source is changed. In that case, a re-engineering tool for developing instantiators may be desirable.
  - Sometimes, data sources may introduce changes over the instance representation (a new attribute, elimination of an attribute, changes in the cardinality of an attribute). Our current prototype uses a structured representation. Because of that, user workspace area is area is easily

implemented through a relational database system. Nevertheless, in order to facilitate the accommodation of those changes, a more dynamic representation, based on an semi-structured representation, may be useful.

- Graphical user interface:
    - Our current prototype stores in the database the expression tree followed for answering a query. Nevertheless, if a user wants to inspect such a plan, he/she must use SQL to interact with each created table or view. A graphical user interface that facilitates this task to users may be desirable.
    - Although in this first prototype we have offered a graphical user interface to enable users to define ontologies and queries, our experience with users has shown that a even more easy-to-use user interface may be desirable. Some examples of how this new interface should like were presented in Figure 61.

- Null values: The concept of null values [Biskup et al., 1983; Imielinsky et al., 1984]] may have to be revisited in order to determine their effect on the soundness and completeness properties of the query rewriting algorithm.

5. Bibliography

Ambite, J., Arens, Y., Naveen, A., Knoblock, C., Minton, S., Modi, J., Muslea, M., Philpot, A., Shen, W., Tejada, S., Zhang, W., 1998. The SIMS manual version 2.0, working draft. Information Sciences Institute and Department of Computer Science. University of Southern California. http://www.isi.edu/sims/knoblock/overview.html (Date Accessed: April 27, 2002).

Arens, Y., Chin, C., Hsu, C., Knoblock, C., 1993b. Retrieving and integrating data from multiple information sources. In: *International Journal on Intelligent and Cooperative Information Systems*, Vol. 2, No. 2, 127--158, 1993.

Arens, Y., Knoblock, C., 1993. Sims: Retrieving and integrating information from multiple sources. In: *SIGMOD Conference*, 562--563, 1993.

Arens, Y., Knoblock, C., Shen, W., 1996. Query reformulation for dynamic information integration. In: *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, Vol. 6, No. 2/3, 99--130, 1996.

Baker, P.G., Brass, A., Bechhofer, S., Goble, C., Paton, N., Stevens, R. 1998. TAMBIS: Transparent access to multiple bioinformatics information sources. An overview. In: *Proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology*, ISMB98, Montreal, 1998.

Baker, P.G., Goble, C.A., Bechhofer, S., Paton, N.W., Stevens, R., Brass, A., 1999. An ontology for bioinformatics applications. In: *Bioinformatics*, Vol. 15, No. 6, 510—520, 1999.

Biskup, Joachim., 1983. A foundation of Codd's relational maybe-operations. In: *ACM Transactions on Database Systems*, Vol. 8, No. 4, 1983.

Bright, M.W., Hurson, A.R., Pakzad, H., 1994. Taxonomy and current issues in multidatabase systems. In: *Multidatabase Systems: An advanced solution for global information sharing*, 1994.

Cal`i, A., Calvanese, D., De Giacomo, G., Lenzerini, M., 2001. Accessing data integration systems through conceptual schemas. In: *Proc. of the 20th Int. Conf. on Conceptual Modeling ER 2001*, 270-284, 2001.

Calvanese, D., Giacomo, G., Lenzerini, M., 1998. Information integration: Conceptual modeling and reasoning support. In: *CoopIS'98*. New York, 1998.

Caragea, D., Silvescu, A., Honavar, V., 2000. Agents that learn from distributed dynamic data sources. In: *Proceedings of the ECML 2000/Agents 2000 Workshop on Learning Agents*, Barcelona, Spain, 2000.

Chartier, R., 2001. Application architecture: An N-tier approach – Part 1. In: *Internet.com*. http://www.15seconds.com/issue/011023.htm (Date Accessed: April 4, 2002). October 23, 2001.

Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K., 1995. Optimizing queries with materialized views. In: *International Conference on Data Engineering*, 1995.

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J., 1994. The TSIMMIS Project: Integration of heterogeneous information sources. In*:* *Proceedings of the 100th IPSJ Anniversary Meeting*, Tokyo, Japan, October 1994.

Chawathe, S., Garcia-Molina, H., Widom, J., 1994b. Flexible constraint management for autonomous distributed databases. In: *IEEE Data Engineering Bulletin*, Vol. 17, No. 2, 23-27, June 1994.

Convey, C., Karpenko, O., Tatbul, N., Yan, J., 2001. Data integration services. Chapter 1. Brown University.http://www.cs.brown.edu/courses/cs227/groups/dataint/first_draft_Apr20/chapter.pdf. (Date Accessed: April 1, 2002) April 20 2001.

Davidson, S., Crabtree, J., Brunk, B., Schug, J., Tannen, V., Overton, G., Stoeckert, C., 2001. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources. In: *IBM Journal*, Vol. 40, No 2, 2001.

Description Logics, 2002. Principles of Knowledge Representation and Reasoning, Incorporated. http://dl.kr.org/ (Date Accessed: March 8, 2002). March 12, 2002.

Draper, D., Halevy, A., Weld, D.S., 2001. The Nimble XML data integration system. ICDE'01, 2001.

Duschka, O. M., Genesereth M.R., 1997. Answering recursive queries using views. In: *Conference on Principles of Database Systems*, PODS, 109-116, Tucson, AZ, May 1997.

Duschka, O., Genesereth, M., Levy, A., 2000. Recursive query plans for data integration. In: *Journal of Logic Programming, special issue on Logic Based Heterogeneous Information Systems*, 2000.

EcoCyc, 2002. EcoCyc: Encyclopedia of *E.* coli Genes and Metabolism. http://ecocyc.org/ (Date Accessed: March 11, 2002).

Etzold, T., Ulyanov, A., Nardi, D., Nutt, W., 1996. SRS: Information retrieval system for molecular biology data banks. Methods Enzymol, Vol. 266, 114--128, 1996.

ExPASy, 2002. The ExPASy Proteomics Center from the Swiss Institute of Bioinformatics. http://www.expasy.org/contact.html (Date Accessed: March 13, 2002).

Ganter, B., Wille, R., 1999. Formal concept analysis: Mathematical foundations. ISBN 3-540-62771-5.

Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J., 1995. Integrating and accessing heterogeneous information sources in TSIMMIS. In: *Proceedings of the AAAI Symposium on Information Gathering*, pp. 61-64, Stanford, California, March 1995.

Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V., Widom, J., 1997. The TSIMMIS approach to mediation: Data models and languages. In: *Journal of Intelligent Information Systems*, Special Issue on Next Generation Information Technologies and Systems, Vol. 8, No. 2, March-April 1997.

Garcia-Molina, H., Ullman, J., Widom, J., 2000. Database system implementation. Prentice Hall, ISBN 0-13-040264-8.

Gene Ontology, 2002. Gene Ontology Consortium. http://www.geneontology.org/ (Date Accessed: March 12, 2002).

Goñi, A., Illarramendi, A., Mena, E., Blanco, J., 1998, An ontology connected to several data repositories: Query processing steps. In: *Journal of Computing and Information* (JCI), ISSN 1201-8511, Vol. 3, No. 1, 1998, Selected paper from the 9th International Conference on Computing and Information (ICCI'98). 1998.

Gruber, T., 1993. A translation approach to portable ontologies. In: *Knowledge Acquisition*, Vol. 5, No. 2, 199-220, 1993. http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html (Date Accessed: March 4, 2002).

Guarino, N., 1998. Formal ontology in information systems. In: *N. Guarino (ed.) Formal Ontology in Information Systems. Proceedings of FOIS'98*, Trento, Italy, 6-8 June 1998. IOS Press, Amsterdam: 3-15.

Haas, L.M., Miller, R.J., Niswonger, B., Roth, M.T., Schwarz, P.M., Wimmers, E.L., 1999. Transforming heterogeneous data with database middleware: Beyond integration. In: *IEEE Data Engineering Bulletin*, Vol. 22, No. 1, 31--36, March 1999.

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.

Hammer, J., Breunig, M., Garcia-Molina, H., Nestorov, S., Vassalos, V., Yerneni, R., 1997. Template-based wrappers in the TSIMMIS system. In: *Proceedings of the Twenty-Sixth SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 12-15, 1997.

Hammer, J., Garcia-Molina, H., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J., 1995. Information translation, mediation, and mosaic-based browsing in the TSIMMIS system. In: *Exhibits Program of ACM SIGMOD,* San Jose, California, June 1995.

Horrocks, I., Sattler, U., Tessaris, S., Tobies, S., 2000. How to decide query containment under constraints using a description logic. In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning* (LPAR'00), 2000.

Imielinski, T. and W. Lipski, Jr., 1984. Incomplete information in relational databases. In: *JACM,* Vol. 31, No. 4, 1984.

Ke, M., Ali, M., 1990. Knowledge-directed induction in a DB environment. IEA/AIE, Vol. 1, 325-332, 1990.

KIF. Stanford Logic Group. Stanford University. http://logic.stanford.edu/kif/kif.html (Date Accessed: March 12, 2002).

Kirk, T., Levy, A., Sagiv, Y., Srivastava, D., 1995. The Information Manifold. In: *Working Notes of the AAAI String Symposium: Information Gathering from Heterogeneous, Distributed Environments,* AAAI Press, 1995.

Köhler, J., Lange, M., Hofestädt, R., Schulze-Kremer, S, 2000. Logical and semantic database integration. In: *BIBE 2000: IEEE International Symposium on Bio-Informatic & Biomedical Engineering*, Washington, D.C., U.S.A., November 8-10*,* S. 77-80, IEEE Computer Society, 2000.

Lambrecht, E. Kambhampati, S., 1997. Planning for information gathering: A tutorial survey. ASU CSE Technical Report 96-017, May 1997.

Leiva H., A. Atramentov and Honavar V., 2002. Experiments with MRDTL: A multi-relational decision tree learning algorithm. In: *Proceedings of the KDD-02 Workshop on Multi-Relational Learning.* ACM SIGMOD Conference on Knowledge Discovery and Data Mining, 2002.

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

Levy, A., 2002. Answering queries using views: A survey. Technical report, University of Washington, 2002.

Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava., 1995. Answering queries using views. In: *Proc. PODS Conf.*, 95-104, 1995.

Levy, A., Rajaraman, A., Ordille, J., 1996. Querying heterogeneous information sources using source descriptions. In: *Proceedings of the Twenty-second International Conference on Very Large Databases*, 251–262, 1996.

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

Li, Ch., 2001. Query processing and optimization in information integration systems. Ph.D. Thesis, Computer Science Department, Stanford University, August, 2001. http://www1.ics.uci.edu/~chenli/thesis/thesis.html (Date Accessed: April 20, 2002)
and http://murl.microsoft.com/LectureDetails.asp?784 (Date Accessed: April 20, 2002).

Li, Ch., Yerneni, R., Vassalos, V., Garcia-Molina, H., Papakonstantinou, Y., Ullman, J., 1998. Capability-based mediation in TSIMMIS. In: *Exhibits Program of SIGMOD'98*, 1998.

Ludascher, B., Papakonstantinou, Y., Velikhov, P., 2000. Navigation-driven evaluation of virtual mediated views. In: *Extending DataBase Technology (EDBT)*, 2000.

E. Mena, A. Illarramendi, V. Kashyap and A. Sheth, *OBSERVER:* An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In: *International journal on Distributed And Parallel Databases* (DAPD), ISSN 0926-8782, Vol. 8, No. 2, 223-272, April 2000.

MOCHA, 2002. Mocha: Middleware based on a code shipping architecture. The Mocha Project. Department of Computer Science. University of Maryland.
http://www.cs.umd.edu/projects/mocha/ (Date Accessed: May 10, 2002).

NCSC, 2002. North Carolina Supercomputing Center. MCNC. http://www.ncsc.org/ (Date Accessed: February 12, 2002).

Noy N., McGuinness, D., 2001. Ontology Development 101: A Guide to Creating Your First Ontology. Knowledge Systems Laboratory, Knowledge Systems Lab, Stanford University. KSL Report Number KSL-01-05. March, 2001. http://ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html (Date Accessed: March 4,2002).

OIL, 2002. Ontology Inference Layer. Information Society Technologies. http://www.ontoknowledge.org/oil/ (Date Accessed: March 12, 2002).

OBSERVER, 2002. Ontology based system enhanced with relationships for vocabulary heterogeneity resolution. Centro Politécnico Superior. Universidad de Zaragoza. http://siul02.si.ehu.es/~jirgbdat/OBSERVER/ (Date Accessed: March 9, 2002).

Özsu, M.T., Valduriez, P., 1991. Principles of distributed database systems. Prentice-Hall. http://www.cs.byu.edu/courses/cs501r.2/notes.html (Date Accessed: April 20, 2002).

Papakonstantinou, Y., Garcia-Molina, H., Widom, J., 1995. Object exchange across heterogeneous information sources. In: *IEEE International Conference on Data Engineering*, 251-260, Taipei, Taiwan, March 1995.

Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., Ullman, J., 1995b. A query translation scheme for rapid implementation of wrappers. In: *International Conference on Deductive and Object-Oriented Databases*, 1995.

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.

PDB, 2002. PDB: Protein Database Bank. Research Collaboratory for Structural Bioinformatics (RCSB). http://www.rcsb.org/pdb/ (Date Accessed: February 14, 2002).

PROSITE, 2002. PROSITE: Database of Protein Families and Domains. The ExPASy Proteomics Center from the Swiss Institute of Bioinformatics. http://www.expasy.ch/prosite/ (Date Accessed: February 10, 2002).

Pottinger, R., Halevy, A., 2001. MiniCon: A scalable algorithm for answering queries using views. In: *VLDB Journal* , Vol. 10, No. 2/3, 182-198, 2001.

Quass, D., Rajaraman, D., Sagiv, Y., Ullman, J., Widom, J., 1995. Querying semi structured heterogeneous information. In: *International Conference on Deductive and Object-Oriented Databases*, 1995.

Rajaraman A., Sagiv Y., Ullman J., 1995. Answering queries using templates with binding patterns. In: *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)*, San Jose, CA, May 22-24, 1995.

Ramakrishnan, R., Gehrke, J., 2000. Database Management Systems. Second Edition. McGraw Hill. ISBM 0-07-232206-3.

Rodriguez-Martinez, M., Roussopoulos, N., 2000. Automatic deployment of application-specific metadata and code in MOCHA. In: *Proc. of the 7th Conference on Extending Database Technology (EDBT)* Konstanz, Germany, March 2000.

Rodriguez-Martinez, M., Roussopoulos, N., 2000b. MOCHA: A self-extensible database middleware system for distributed data sources. SIGMOD 2000.

RDF, 2002. Resource Description Framework. World Wide Web Consortium. http://www.w3.org/RDF/ (Date Accessed: March 12, 2002).

Silvescu, A., Reinoso-Castillo, J., Andorf, C., Honavar, V., and Dobbs, D., 2001. Ontology-driven information extraction and knowledge acquisition from heterogeneous, distributed biological data sources. In: *Proceedings of the IJCAI-2001 Workshop on Knowledge Discovery from Heterogeneous, Distributed, Autonomous, Dynamic Data and Knowledge Sources*.

Sowa, J.F., 2000. Knowledge Representation: Logical, Philosophical and Computational Foundations. ISBN 0 534-94965-7. 2000.

Stevens, R., Goble, C., Paton, N., Bechhofer, S., Ng, G., Baker, P., Brass, A., 1999. Complex query formulation over diverse information sources using an ontology. In: *Workshop on*

Computation of Biochemical Pathways and Genetic Networks, pages 83-88. European Media Lab (EML), August 1999.

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, Vol. 16, No. 2, 184-186, 2000.

Stoffel, K., Taylor, M., Hendler, J., 1996. Parka-DB: Integrating knowledge and data-based technologies. Department of Computer Science. University of Maryland. http:// citeseer.nj.nec.com/54974.html. (Date Accessed: March 12, 2002).

Stoffel, K., Taylor, M., Hendler, J., 1997. Efficient management of very large ontologies. In: *Proceedings of American Association for Artificial Intelligence Conference (AAAI-97)*, AAAI/MIT Press.

Swiss-Prot, 2002. SWISSPROT Protein Knowledgebase – TrEMBL Computer-annotated. The ExPASy Proteomics Center from the Swiss Institute of Bioinformatics. http://www.expasy.org/sprot/ (Date Accessed: February 10, 2002).

Tari, Z., 1995. Deductive databases – DataLog approach. School of Information Systems, Queensland University of Technology for postgraduate students during the period of 1993-1995. http://goanna.cs.rmit.edu.au/~zahirt/Teaching/subj-datalog.html (Date Accessed: April 4, 2002).

TSIMMIS, 2002. The TSIMMIS project. Stanford University Database Group. http://www-db.stanford.edu/tsimmis/ (Date Accessed: March 15, 2002).

Ullman, J., 1988. Principles of database and knowledge-base systems. Volume 1 & 2. Computer Science, 1998.

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

Vargun, A., 1999. Semantic aspects of heterogeneous databases. University of Colorado at Boulder. CS5817 Project, 1999.

http://www.cs.colorado.edu/~getrich/Classes/csci5817/Term_Papers/vargun/ (Date Accessed: March 10, 2002). 1999.

Vassalos, V., Papakonstantinou, Y., 1997. Describing and using query capabilities of heterogeneous sources. In: *Proceedings of the VLDB Conference*, 1997.

XML. Extensible Markup Language. World Wide Web Consortium. http://www.w3.org/XML/ (Date Accessed: March 12, 2002).

Yerneni, R., Papakonstantinou, Y., et al., 2000. The TSIMMIS project at Stanford. Seminar Course on Intelligent Information Systems. Ecole Polytechnique Federale de Lausanne, Laboratoire de Bases de Donnees. May 5, 2000. http://lbdsun.epfl.ch/e/Gio/may4/sld026.htm (Date Accessed: March 10, 2002).

Zhang, J., Silvescu, A., and Honavar, V., 2002. Ontology-driven induction of decision trees at multple levels of abstraction. In: *Proceedings of the Symposium on Abstraction, Reformulation, and Approximation.* SARA-2002. Kananaskis, Alberta, Canada.