Semantic Wrapper:

Concise Semantic Querying of Legacy Relational Databases

Naphtali Rishe, Borko Furht, Malek Adjouadi, Armando Barreto, Debra Davis
NSF Industry-University Cooperative Research Center for Advanced Knowledge Enablement (CAKE.fiu.edu) at Florida International and Florida Atlantic Universities

Ouri Wolfson
Computational Transportation Science Program (CTS.cs.uic.edu) at the University of Illinois at Chicago

Yelena Yesha and Yaacov Yesha
NSF Industry-University Cooperative Research Center for Multicore Productivity Research (CHMPR.umbc.edu) at the University of Maryland Baltimore County

1. Introduction

From business to education to research, one of the most common needs in today's world is the ability to efficiently and effectively store and organize information. In the past several decades, the amount of information required to perform even everyday tasks has dramatically increased. In addition, there has been a substantial increase in the availability of more sophisticated and complex data, such as imagery, multimedia, geospatial and remotely sensed data. As our need for more extensive and complex information has increased, so has the size and complexity of databases used to store this information. To even further complicate the situation, needed data often resides in various independent, distributed databases as well as in unstructured forms such as social media and web sites. This has resulted in a greater need for more flexible, scalable and efficient database technology that can be used to store, query and combine massive amounts of various types of data that are distributed over multiple structured and unstructured data sources [1; 2; 3].

1.1. Relational Database Systems and Structured Query Language (SQL)

The majority of commercial data systems in use to today are relational databases. These database systems have been developed based on the Relational Data Model proposed by Codd in 1970 and subsequently updated in 1990 [4][5]. A relational database is composed of one or more two-dimensional tables, each of which contains data fields, or attributes, in the form of columns, and data records, or tuples, in the form of rows. Relationships between tables are implicit, by cross-referencing data values in fields found on different tables. The Relational Data Model has a number of advantages that have contributed to the prevalence of these systems including: (1) it has rigorous design methodologies (e.g., normalization), alleviating redundancy and inconsistency, (2) the database schema is easily modifiable by the addition of tables and fields, and (3) the database has flexible and powerful well-defined operations, such as join, that is based on the algebraic set theory [6].

One of the biggest advantages of relational databases is the use of Structured Query Language (SQL) as the standard query syntax in most relational database products. In its basic form, the syntax is simple and relatively easy to understand and use: Select *fields* From *tables* Where *conditions*. SQL can be used directly on a database, or as embedded SQL within general-purpose languages, or as an intermediary language via a standard communication protocol such as Open Database Connectivity (ODBC) protocol, Java Database Connectivity (JDBC) protocol or Object Linking and Embedding (OLE).

Relational database systems have become a staple in modern technology. However, the data needs of today's world are increasingly changing at a very rapid pace. As a result, today's technology is becoming overtaxed and obsolete at an increasingly faster rate as more sophisticated solutions are required. As this pressure on technology has increased, the limitations of relational databases have come more to the forefront. For example, relational databases are designed for organizing information that is easily categorized by common characteristics, and described by simple string or number data. Complex data, such as images, spatial and remotely sensed data, and multimedia products, are not easily described or categorized in this manner. Increased storage and utilization of massive amounts of complex data tends to lead to the implementation of more complex database schemas. Due to the difficulty in effectively handling these complex structures, this can lead, in turn, to isolation of complex relational database systems where information cannot be easily shared between those systems [1].

There are also challenges involved in the use of SQL. Although in its basic form, SQL is a relatively simple and easy to understand syntax, rarely will a simple query be sufficient to provide users with the data results that they require. This is particularly true for the more common, complex systems in use today. The effective use of SQL in most real-world relational databases requires technology specialists who have extensive training in the principles of relational databases, and complex and efficient SQL query design. They must also have a thorough understanding of the structure of the database of interest. This is because the relationship between tables in a relational database is implicit rather than explicit, and data of interest of is often found across various tables in the database. When creating a query, users must explicitly identify each table and provide a formula relating the various tables in order to effectively achieve the desired data results. The more abstract and complex the relationships are between tables in the database, the more difficult it becomes to create accurate and efficient queries.

These highly skilled database technology specialists are also needed when designing and implementing a new relational database or making changes to the structure of an existing system. Because relational databases are so easy to modify, some untrained users mistakenly believe that one can simply add new tables and fields without much thought. However, this flexibility makes it easy to create poor, inefficient database designs that do not meet the needs of end users. These complex solutions require thorough analysis and planning, and are quite expensive and time consuming to implement or change.

1.2. Overcoming the Limitations of Relational Databases

In recent years, there have been a number of efforts to overcome the limitations of relational database systems. Semantic and object-oriented databases have been introduced that provide improvement on the structure of the database itself, such as allowing explicit relationships between classes of objects, and the inclusion of super-categories, sub-categories and inheritance. However, to be effective, these improvements to the structure of the database have historically required the addition of enhanced query language [7]. Although this can provide more powerful capabilities, it has also precluded the ability to use standard SQL for effective querying of these databases, as well as the use of standard relational database interface tools.

Approaches that allow semantic and object oriented query interfaces over relational databases have also been proposed, including SQL3, Object query Language (OQL), and various graphical query languages [7][8]. While these provide the advantages of the object approach, changes to the data models and syntax of SQL are required to enable syntax expressiveness. This requires users to study new syntax and does not allow the utilization of the very extensive existing middleware and end-user GUIs that interface in standard SQL. Also, while the new languages work well with new database implementations based on object models, it does not provide improvement to existing relational databases. Further, these approaches require additional training and a change in how users must approach their programming. Standard SQL is a purely declarative language, whereas SQL3 and OQL, in particular, are semi-procedural.

Graphical query languages attempt to make query interfaces more user friendly. Through the use of approaches such as hypertext language, menu-driven queries and Query-By-Example (QBE), users theoretically need not learn SQL syntax. In real world applications, however, these approaches only work well for very simple databases and very simple queries. Attempting to use these types of approaches for complex queries often leads to frustration when trying to follow the required procedures, and produces irrelevant results once the user completes the process. In order to be successful when creating complex queries, users must still have full knowledge of the logical structures of the database of interest, and understand how to express the needed joins correctly.

It is clear that there is a need for a more dynamic solution that overcomes the increasing pressures and limitations of relational databases, while, at the same time, avoids some of the obstacles encountered by other object oriented approaches. To address and overcome these many challenges, Semantic Wrapper technology has been created. The Semantic Wrapper is a middleware system that provides semantic views over legacy relational databases. As middleware, this system provides straight-forward, easy access to legacy relational databases without requiring users to switch from their existing GUI to a new, unfamiliar GUI. It further greatly improves usability by allowing the use of standard SQL syntax to access the semantic layer via more simplified and concise SQL queries than what would be required for direct querying of a standard relational database. This approach is also applicable in a heterogeneous multi-database environment that can include both structured (relational and semantic databases) and unstructured data (social media and related Internet sites).

The remainder of this article presents an overview of the Semantic Wrapper and its major components. Section 2 discusses the Semantic Wrapper's primary features and compares it to other systems. This is followed by discussions of the capabilities and integration of its three major components in Section 3.

2. Semantic Wrapper Overview

The Semantic Wrapper is a set of middleware tools developed to quickly and easily access and query relational databases, semantic databases and unstructured data such as Internet data sources. To accomplish this, the Semantic Wrapper creates a semantic schema view over relational and legacy databases. This schema is then used by the system to provide users with the ability to query the data quickly and easily. This technology allows the use of standard SQL queries to access the semantic schema via simple and concise syntax. It enables the use of 3rd-party interfaces to formulate queries, allowing users continued use of familiar interfaces while still providing them with the Semantic Wrapper's powerful capabilities.

The system is based on the semantic modeling approach [9], and employs the Semantic Binary Object Data Model (Sem-ODM) and Semantic SQL query language (Sem-SQL) for improved data querying and usability[10]. Various aspects and application of this and related technology are discussed in [11; 12; 13; 14; 15]. This platform provides the system with powerful and adaptive capabilities. When the semantic features of these technologies are employed, they have been found to provide a high level of efficiency when compared to relational databases [16].

2.1. Capabilities of the Semantic Wrapper

There are a number of object oriented approaches that are focused on overcoming the problems inherent in the Relational Model's inability to effectively deal with modern data needs. To address these issues, other approaches have implemented improvements to database structures, enhancements to standard query languages, and the use of graphical user interfaces to improve ease of use. However, each of these attempts has created their own challenges and limitations. These challenges include:

- Inability to be implemented with existing relational databases and legacy systems
- Increased complexity and decreased user friendliness due to changes to query languages
- Attempts to improve ease of use that also result in significantly decreased functionality

The Semantic Wrapper has been designed to provide a powerful, yet easy to use, system that improves our ability to work with various types of heterogeneous data without encountering the above challenges.

The Semantic wrapper can be implemented either independently, or, as can be seen in Figure 1, over an existing relational database. A number of the object oriented approaches discussed in Section 1.2 incorporate improvement on the structure of the database itself to provide a greater capability to handle various types of data. While this was a major step towards more efficiently and effectively dealing with the shortcomings of a relational schema, it does not allow for the ability to implement the system with existing relational and legacy database systems whose structures are not be amenable to structural change. The Semantic Wrapper has overcome this obstacle by avoiding any changes to existing database structures. Instead, a separate semantic schema is constructed that allows a compact and logical semantic view while still providing an accurate reflection of the original database. This, in essence, provides the system with the improved database structure sought by other systems, without the same limitations. Thus, the Semantic Wrapper can be used to construct new applications, or as an additional interface to existing relational databases.

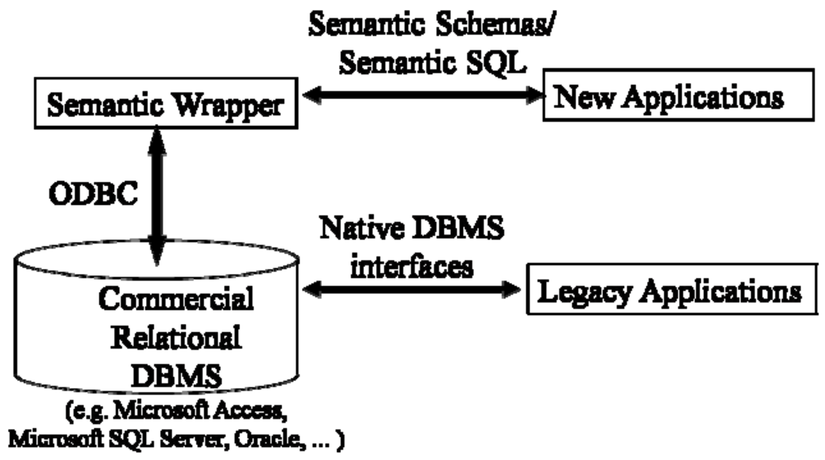


Figure 1. High-level architectural view of the Semantic Wrapper

The Semantic Wrapper uses Semantic SQL, which is syntactically identical to the standard relational SQL, but exploits the semantics of relationships. This, in turn, makes Semantic SQL queries much more concise and clear. Specifically, Semantic SQL queries refer to a virtual relational schema that consists of inferred tables defined as spanning trees containing all reachable relations from a given category. Complications of multiple table references and keys are eliminated. As a result, users have no need to explicitly express joins. Instead, when a user queries the database, it is as if there is a universal table for each category. This significantly reduces the complexity of queries, and significantly increases the user friendliness of the system's querying capabilities [17].

Attempts by other approaches, such as graphical query languages, to improve user friendliness of database systems have succeeded in terms of simplifying the process needed for very basic queries. However, these have also lead to significant reductions in the ability to create even moderately complex queries that are more commonly used in the real world. Even though the Semantic Wrapper's increased user friendliness is partially through the use of more simplified query syntax, because of the use of Semantic SQL in combination with the Semantic Wrapper's structure and the various tools that are part of the Semantic Wrapper, there is no decrease in functionality. Specifically, the mapping and query translation processes that are used to create the semantic schema (see Section 3.1) and subsequently query the database (see Section 3.3) merely hide the complexity from the user without actually limiting functionality. In other words, the system does much of the work for the user, without loss of functionality or an increase in errors.

In sum, the Semantic Wrapper is a solution that provides users with the ability to view multiple data sources as one, centralized virtual database through a semantic database schema, and access needed data via a standard, easy-to-use and flexible interface. This standard interface allows users to interact and query data in a more intelligent and friendlier manner that is based on the stored meaning of the data. Further, 3rd party tools that the user is already familiar with can be deployed with the system. The system can be implemented independently for new applications, or over an existing legacy relational database. Other advantages of using the Semantic Wrapper include comprehensive enforcement of integrity constraints, greater flexibility, and substantially shorter application programs [10].

3. Technical Background

3.1. Semantic Schemas

A key component of Semantic Wrapper technology is the use of a semantic schema that is functionally equivalent to a corresponding relational database. Use of an equivalent semantic schema has numerous advantages over a relational schema, including:

- Knowledge is described at a conceptual level rather than at a table layout level
- Meaning of information in a semantic schema is retained and stored within the schema itself
- Relationships between categories are explicitly represented
- It is significantly easier to formulate queries as any relationship can be queried and joins are not required to be explicitly defined

For example, a subschema of a Hydrology database that was developed for the Everglades National Park can be seen in Figure 2 and its normalized relational counterpart can be seen in Figure 3. As the semantic subschema is a conceptual representation, its structure is more analogous to how the real world is conceptualized by humans. This makes it easier for users to understand the types of information contained in the database, as well as how data within the database is conceptually related.

Figure 2. Semantic sub-schema for Physical Observations

As can be seen in Figure 4, Semantic SQL queries are an order of magnitude shorter than the corresponding relational SQL query. This query is representative of the types of real world information that might be needed by a user. In this case, the user would be a scientist or environmentalist needing data about environmental conditions over time (see Figure 2 and Figure 3 for the semantic and relational subschemas). The question of interest, “Give me all of the observations with all of their attributes since January 1, 1993 and the location of the observing stations”, requires a very short and easy to compose query to the semantic schema. As can be seen, if the same query were written for a relational schema, the query would be substantially longer and more complex.

PHYSICAL-OBSERVATION-STATION
physical-observation-station-id-key:Integer 1:1; comments:String; housing:String; structure:String;
is-part-of--physical-observation-station-id:Integer;

LOCATION
north-UTM-in-key:Number; east-UTM-in-key:Number; elevation-ft:Number; description:String;

ORGANIZATION
name-key:String 1:1; description:String;

PROJECT
name-key:String 1:1; description:String; comments:String; starting-date:Date; ending-date:Date;

MEASUREMENT-TYPE
name-key:String 1:1; measurement-unit:String; upper-limit:Number; lower-limit:Number;

FIXED-STATION
physical-observation-station-id-key:Integer 1:1; platform-height-ft:0..50.000;
located-at--north-UTM:Number;
located-at--east-UTM:Number;

MEASUREMENT
observation-id-key:Integer 1:1; comment:String; time:Date-time; value:Number; of--name:String;
by--physical-observation-station-id:Integer;

IMAGE
observation-id-key:Integer 1:1; comment:String; time:Date-time; image:Raw; subject:String;
direction-of-view:0..360; comments:String; type:Char(3); by--physical-observation-station-id:Integer;

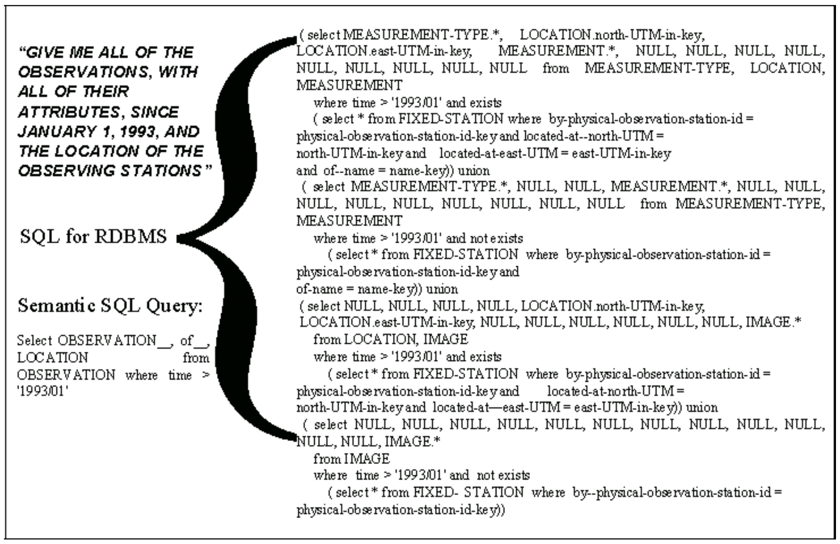
PHYSICAL-OBSERVATION-STATION--BELONGS-TO--ORGANIZATION
physical-observation-station-id-in-key:Integer; organization--name-in-key:String;

ORGANIZATION--RUNS--PROJECT
organization--name-in-key:String; project--name-in-key:String;

PHYSICAL-OBSERVATION-STATION--SERVES--PROJECT
physical-observation-station-id-in-key:Integer; project--name-in-key:String;

ORGANIZATION--IS-PART-OF--ORGANIZATION
organization--name-in-key:String; organization-2--name-in-key:String;

Figure 3. Relational sub-schema for Physical Observations



"GIVE ME ALL OF THE OBSERVATIONS, WITH ALL OF THEIR ATTRIBUTES, SINCE JANUARY 1, 1993, AND THE LOCATION OF THE OBSERVING STATIONS"

SQL for RDBMS

```
( select MEASUREMENT-TYPE.*,    LOCATION.north-UTM-in-key,
LOCATION.east-UTM-in-key,     MEASUREMENT.*,   NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL  from  MEASUREMENT-TYPE,  LOCATION,
MEASUREMENT
   where time > '1993/01' and exists
   ( select * from FIXED-STATION where  by-physical-observation-station-id =
physical-observation-station-id-key and located-at--north-UTM =
north-UTM-in-key and   located-at-east-UTM = east-UTM-in-key
and  of--name = name-key)) union
 ( select MEASUREMENT-TYPE.*, NULL, NULL, MEASUREMENT.*, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL  from MEASUREMENT-TYPE,
MEASUREMENT
   where time > '1993/01' and not exists
      ( select * from FIXED-STATION  where  by-physical-observation-station-id =
physical-observation-station-id-key and
of-name = name-key)) union
 ( select NULL, NULL, NULL, NULL, LOCATION.north-UTM-in-key,
 LOCATION.east-UTM-in-key, NULL, NULL, NULL, NULL, NULL, NULL, IMAGE.*
   from LOCATION, IMAGE
   where time > '1993/01' and exists
      ( select * from FIXED-STATION  where  by-physical-observation-station-id =
physical-observation-station-id-key and      located-at-north-UTM =
north-UTM-in-key and  located-at—east-UTM = east-UTM-in-key)) union
 ( select NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, IMAGE.*
   from IMAGE
   where  time > '1993/01' and  not exists
      ( select * from FIXED- STATION  where  by--physical-observation-station-id =
physical-observation-station-id-key))
```

Semantic SQL Query:

```
Select OBSERVATION__, of__, LOCATION from
OBSERVATION where time > '1993/01'
```

Figure 4. Comparison of a Semantic SQL Query to a corresponding Relational SQL Query

3.2. The Semantic Binary Model

The semantic schemas used by the Semantic Wrapper are based on the Semantic Binary Model (SBM) [9]. The SBM is a flexible, new generation data model that is natural, simple, non-redundant, and implementation-independent. Its strength lies in its ability more accurately capture the meaning of information, while, at the same time, providing a succinct, high-level description of that information. Its use of objects, categories and their relationships is very easy for users to conceptualize as they are reflective of the manner in which users already think about the real world. A sample semantic schema can be seen in Figure 5.

Objects - The central notion of semantic models is the concept of an object. Objects are defined as any real world object or entity that we wish to store information about in the database. Examples of objects include a student, department, course and course name. Objects can be further classified as concrete objects, which are printable objects such as course name, or abstract objects, which are non-value objects in the real world such as a course itself.

Categories - Objects that have common properties are considered in the same category in the database. As with objects, categories can be concrete (consisting of only concrete objects) or abstract (consisting of only abstract objects). Objects may also belong to more than one category at a time. For example, an object can be both a student and instructor at the same time (see Figure 5). A schema may also contain subcategories. A category is a *subcategory* of another category if every object in that category is always an object in the latter category. For example, a student is always also person. Therefore, STUDENT is a subcategory of PERSON. On the opposite end of the spectrum, categories can be disjoint. Two categories are *disjoint* if no object can ever be a member of both categories at the same time. For example, a student can never also be a course; therefore, STUDENT and COURSE are disjoint categories.

Relationships - A binary relationship is a connection between two objects indicating that they are related by a certain property. Such a property is called a *binary relation.* At every moment in time, a binary relation *R* is descriptive of a set of pairs of objects *(x,y)* which are related at that time. This is denoted as *xRy.* For example, an instructor WORKS-IN a department (see Figure 5). The relation is WORKS-IN, and is denoted as *i WORKS-IN d*.

Binary relations may be *m:1* (many-to-one), *1:m* (one-to-many), *m:m* (many-to-many) or *1:1* (one-to-one).. A binary relation *R* is *m:1* if there is never a time when *xRy* and *xRz* where *y≠z*. For example, every person has only one birth year, therefore, BIRTH-YEAR is m:1. A binary relation R is *1:m* if there is never a time when *xRy* and *zRy* where *x≠z*. For example, every student can have at most one major. If we had a relation MAJOR-STUDENT (instead of MAJOR-DEPARTMENT, which is m:1), then that relation would be 1:m.). A binary relation *R* is m:m if it is neither m:1 or 1:m. For example, every instructor can work in more than one department, and every department can employ more than one instructor. Thus, WORKS-IN is a m:m relation. A binary relation *R* is 1:1 if it is always both m:1 and 1:m. For example, if every course is uniquely identified by its name (there is no character string that can be the name of two or more courses), then COURSE-NAME is 1:1.

The *domain* of a relation is the smallest category such that for every *xRy*, *x* always belongs to the category. The *range* of a relation is the smallest category such that for every *xRy*, *y* always belongs to the category. For example, the domain of WORKS-IN is INSTRUCTOR and the range is DEPARTMENT.. A binary relation is *total* if for every object *x* in its domain, there always exists an object *y* such that *xRy*. For example, the domain of the relation BIRTH-DATE is PERSON. Although every person has a date of birth, that date of birth is not always known. Therefore, BIRTH-DATE is not total.

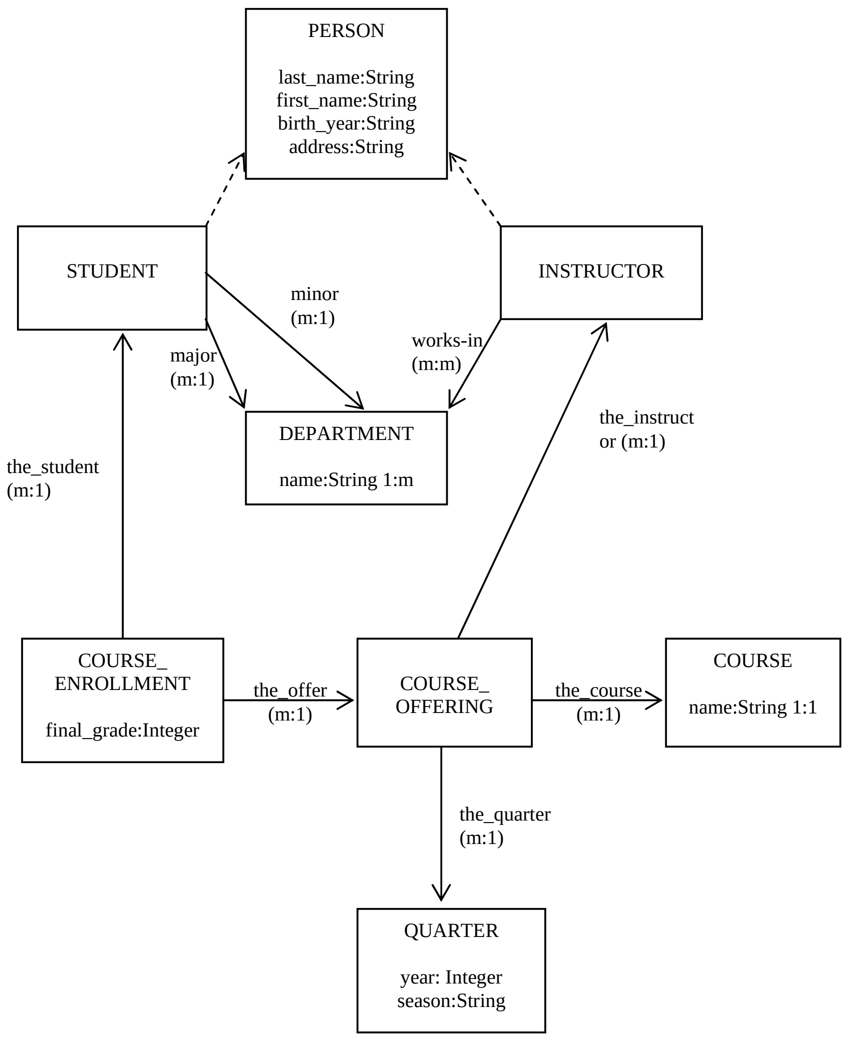


Figure 5. Semantic Schema of a University Application

A binary relation whose range is a concrete category is called an *attribute*. Thus, the phrase "*a* is an attribute of *C*" means that *a* is an attribute and its domain is category *C*. For example, LAST-NAME, FIRST-NAME and BIRTH-YEAR are attributes of PERSON.

3.3. Semantic SQL

The Semantic SQL language paradigm is the core interface of the Semantic Wrapper both to the end user and as a middleware. Semantic SQL was originally designed to query semantic and object oriented databases, and, as part of the Semantic Wrapper, it is used to query a semantic schema that is reflective of the underlying relational schema. Semantic SQL is syntactically identical to Standard SQL but assumes a virtual schema comprised of infinitely-wide tables, one table per semantic category with all the fields reachable from it. In the middleware mode, the fact that the user refers to this virtual schema is technically transparent to third-party tools, thus allowing for standard protocols, such as ODBC and JDBC. Thus, for example, the user can utilize a third-party GUI to create a query. The input is user clicks, the GUI output is SQL. The complexity of the input is proportional to the complexity of the output. Because the formulation of a query in Semantic SQL is very concise in the output, so is the user's input as measured by the number of clicks and the intellectual complexity of the task.

Because Semantic SQL syntax is identical to the relational SQL syntax, it supports standard database access interfaces such as ODBC and JDBC. Thus, Semantic SQL, like standard SQL, is a purely declarative query language. This is particularly advantageous for users as there no need to learn new query syntax or a new programming approach. Where Semantic SQL and relational SQL differ is in the simplicity of Semantic SQL queries as compared to relational queries. Because semantic databases use real world concepts such as objects and categories, Semantic SQL is able to query schemas at the conceptual level instead of the table layout level.

Semantic SQL queries refer to a virtual relational schema that consists of inferred tables defined as spanning trees containing all reachable relations from a given category. A virtual table is implicitly defined for each category in the semantic schema, where all related data is grouped together. Appendix 1 provides a formal definition of this grouping. Referring back to our semantic schema in Figure 5, the following is an example of some of the fields in the virtual table for the category STUDENT.

Full Attribute Name	*Abbreviation*	*Type*	*Sample*
STUDENT	-	*surrogate*	*123235*
last_name	-	*string*	*Smith*
birth-year	-	*integer*	*1990*
the_student__the_offer_the_quarter__year	*year*	*integer*	*2011*
the_student__the_offer_the_quarter__season	*season*	*string*	*Spring*
the_student__final_grade	*final_grade*	*integer*	*75*
Major	-	*surrogate*	*CS*
Minor	-	*surrogate*	*ECE*
major__name	-	*string*	*CompSci*
minor__name	-	*string*	*Electrical*

In the virtual table for STUDENT, the attribute STUDENT__major__name (where STUDENT is the category, MAJOR is the relationship, and NAME is the attribute) refers to "the name of the department in which a student majors."

For ease of use, every attribute in a virtual table has a short semantic name and a long semantic name. A short semantic name is created by removing prefixes in the long attribute name, and can be used for queries so long as there is no ambiguity within that virtual table. For example, in the virtual table STUDENT, the long semantic name for one attribute is *the_student___the_offer__the_quarter__year* and its short semantic name is *year*. As there is no other attribute of the same depth with the name *year* in this virtual table, no ambiguity arises from use of this short semantic name. Conversely, using the abbreviated attribute *name* would lead to ambiguity as there would be two possible attributes of the same depth that this could represent: *the_student__major__name* and *the_student__minor__name*. In this case, it can be disambiguated by using either *major__name* or *minor__name*.

With relational SQL, users usually need to define a join operation to link two tables together. While inner joins are hard enough to define, most realistic queries require an outer join (left outer join), which is very hard to define in relational SQL. As can be seen in the above example, Semantic SQL relieves users of the need to explicitly express joins. Instead, joins are expressed in the names of the attributes found in the virtual table. Because of the semantic information inherent in the schema, there is no ambiguity in the query. This results in simpler query construction for the user. For example, if a user wants to retrieve the first name and last name of a student whose major department is 'computer science', the Semantic SQL query is a follows:

```
select   first_name, last_name  from  STUDENT
where    major__name = 'computer science'
```

In the relational SQL, however, the same query might be composed as follows (depending on the relational schema):

```
select   first_name, last_name       from    STUDENT, DEPARTMENT
where    STUDENT.deptID = DEPARTMENT.deptID and
         DEPARTMENT.name = ‘computer science’
```

Semantic SQL queries requesting the retrieval of more complex combinations of information are still simple. For example, a query to retrieve the student’s last_name, first_name, address, major, name of each course, final_grade for the course, year and semester the course was taken, for every student would be:

```
select   last_name, first_name, address, major__name,
          the_course__name, final_grade, season, year
from     STUDENT
```

Should a user prefer to explicitly express join conditions, they can still do so. As Semantic SQL is completely compatible with relational SQL, the syntax is exactly the same for both query languages.

Queries to update against a virtual relational database are inherently ambiguous. Semantic SQL provides disambiguating semantics on the underlying semantic schema (see Appendix 2 for formal definitions). As with queries for retrieving data, standard SQL syntax, such as insert, delete and update, is used. An example of a simple update is:

Delete students whose final grade is less than 50:
Structure**:** **delete from** *C* **where** *condition*
Example: **delete from** STUDENT **where** FINAL_GRADE<50

4. Components of the Semantic Wrapper

The Semantic Wrapper is primarily comprised of three engineering components that can be used as either a standalone application or as middleware This section provides a high-level description of each of these components, a description of how these components interact with each other to produce the desired results, and an example that illustrates the Semantic Wrapper’s capability to be implemented as middleware.

4.1. The Knowledge Base Tool: Reverse Engineering of a Semantic Schema

In order to interpret Semantic SQL queries to a relational database, a semantic view of the database must be created. This is the primary function of the Knowledge Base Tool (KDB Tool). By using this tool, the user is able to create semantic information for a relational database of interest via the construction of a semantic schema that accurately reflects the information in and structure of data in the relational database. This can include the specification of inheritance of categories and many-to-many relations.

The KDB Tool's capabilities are designed to ensure the integrity of the mapping between the original relational database and the corresponding semantic schema. This is accomplished by enforcing a rule at every step of the creation process that keeps specific mapping information between the relational database tables and semantic schema categories and relations intact. The system will not allow changes to be made to the semantic schema that would damage the integrity of the mapping information.

Use of the KDB Tool on a standard relational database involves a possible total of eight steps, four of which are automated and four of which involve the skills and knowledge of the relational database's administrator [18]. To automatically generate the initial semantic schema, it is assumed that metadata regarding tables, attributes, primary keys and other constraints are available via the relational database management system in use. The first four steps needed to create the semantic schema are as follows:

1. A category in the semantic schema is created for each table in the relational database
2. Within each semantic category, an attribute is created for every field (i.e., column) in the corresponding relational table
3. A semantic relation is created for each functional dependency in the relational database
4. Within each semantic category, attributes that correspond to foreign keys in the relational database are removed (these are reflected in the relations that are created in step 3, rendering these attributes redundant)

A sample relational database structure can be seen in Figure 6, along with its transformation through step 4 to the semantic schema seen in Figure 7. Perusing both figures, it is easy to discern the correspondence between the relational and semantic schemas. For example, the relational table CURRENCY_FOR_COUNTRY corresponds to the semantic category CURRENCY-FOR-COUNTRY. The functional dependency between CURRENCY_FOR_COUNTRY and CURRENCY relational tables transformed into a semantic relationship, 'the-currency', and the foreign key, THE_CURRENCY__CODE_KEY, was removed as it is reflected in the semantic relationship between CURRENCY-FOR-COUNTRY and CURRENCY semantic categories.

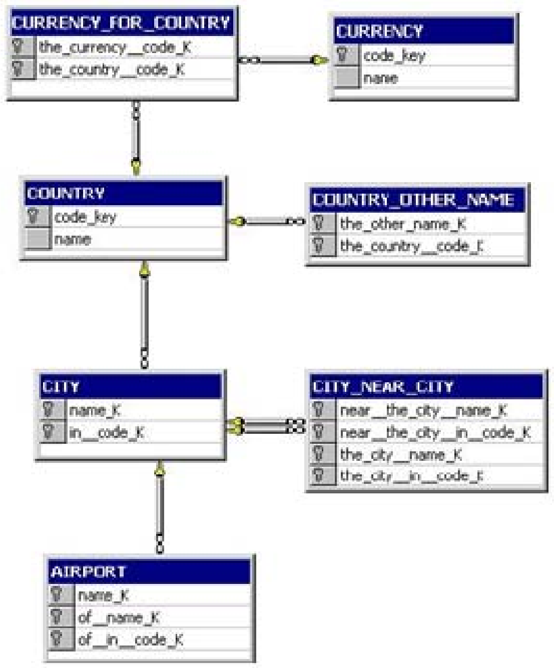


Figure 6. Relational Schema of a Geography Database

Once steps One through Four are completed, a valid semantic view of the database has been created. This view, however, can be further refined to create a more accurate reflection of the application's semantics by completing steps Five through Eight, which require human intervention, i.e. the skills and knowledge of the relational database's administrator (DBA). The reasons for this include the ease at which a database domain expert can understand semantic databases, the DBA's in-depth knowledge of the structure of the relational database, the DBA's intimate understanding of needed userviews and end-user needs, and the DBA's responsibility for the correct functioning of the database tools.

Steps Five through Eight of the semantic schema creation process are as follows:

5. Any semantic categories that correspond to a relational table whose sole purpose is to represent many-to-many relations should be replaced with actual many-to-many relationships in the semantic schema

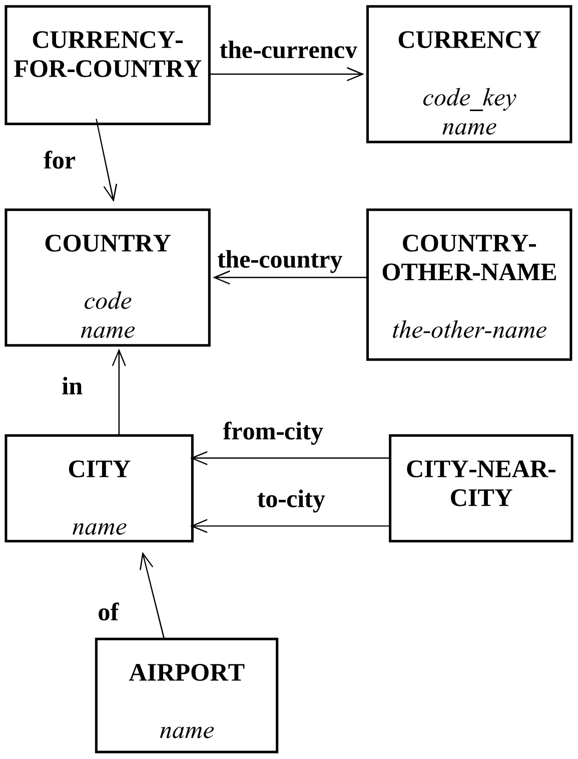


Figure 7. Geography Semantic Schema after Transformation through Step 4

In our sample schema, this type of transformation can be seen in Figure 8. In short, the relational table CURRENCY_FOR_COUNTRY, contains only foreign keys and has two many-to-one relationships. This is reflected in the semantic schema as a category, CURRENCY-FOR-COUNTRY, that has no attributes and two many-to-one relationships. Thus, it is clear that the sole purpose of this category is to represent a many-to-many relationship in the relational schema, which can be directly replaced in the semantic schema as a many-to-relationship.

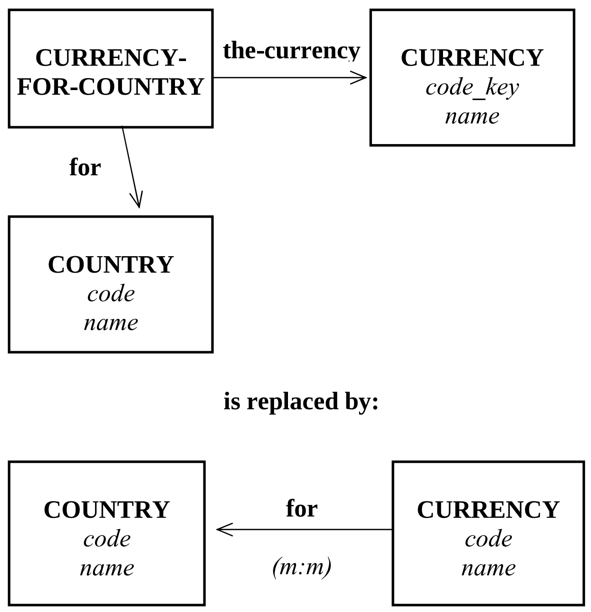


Figure 8. Step 5 of the KDB Tool on the Geography database

6. Any semantic categories that correspond to relational tables whose sole purpose is to represent a recursive reference should be replaced with an is-part-of relationship in the corresponding semantic category. This relationship may have a cardinality of many-to-one or many-to-many.

In our sample schema, this type of transformation can be seen in Figure 9. In short, the relational table CITY_NEAR_CITY, contains only foreign keys and has two many-to-one relationships with the CITY relational table. This is reflected in the semantic schema as a category, CITY-NEAR-CITY, that has no attributes and two many-to-one relationships with the semantic category CITY. Thus, it is clear that the sole purpose of the CITY-NEAR-CITY category is to represent a recursive reference in the relational schema.

7. Any semantic categories that correspond to relational tables whose sole purpose is to represent a one-to-many relationship should be replaced with a one-to-many attribute in the corresponding semantic category.

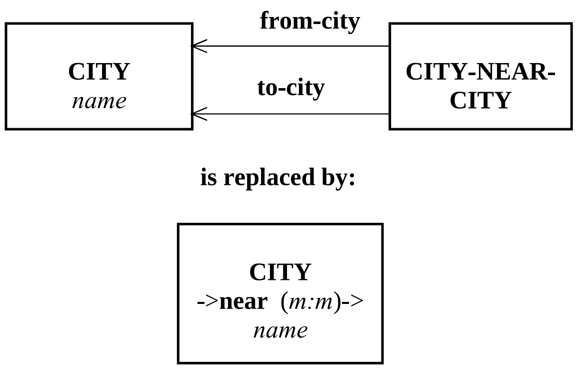


Figure 9. Step 6 of the KDB Tool on the Geography Database

In our sample schema, this type of transformation can be seen in Figure 10. In short, the relational tables COUNTRY_OTHER_NAME and COUNTRY have attributes that contain the same information (the name of a country). Further, the table COUNTRY_OTHER_NAME has no other attributes and only a one-to-many relationship with the COUNTRY relational table. This is similarly reflected in the semantic schema. Thus, it is clear that the sole purpose of the COUNTRY-OTHER-NAME category is to represent a one-to-many relationship in the relational schema and should be replaced with a one-to-many attribute, *other-name*, in the COUNTRY semantic category.

8. Include the relevant category inheritance hierarchy into the semantic schema.

Step Eight introduces an additional level of abstraction to the semantic schema. In our sample schema, this type of transformation can be seen in Figure 11. In short, the relational tables COUNTRY, CITY and AIRPORT all have an attribute in common; that is, they all have a *name*. Because of this commonality, a supercategory, GEOGRAPHICAL-ENTITY, with the attribute *name* can be introduced into the schema, and the *name* attribute can be removed from the aforementioned categories.

In addition, prior to defining a virtual table, the name of every category and relation is “cleaned” as follows:

1. replace all non-alphanumeric characters with "_"
2. if the name begins with a digit or "_", prepend "A"
3. if the name ends with "_" append "Z"
4. collapse multiple "_" into a single "_"

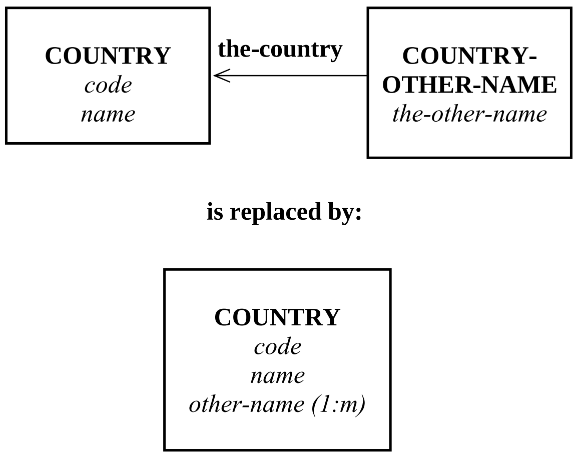


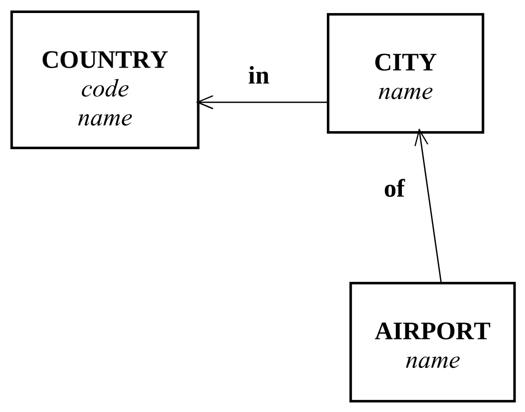
Figure 10. Step 7 of the KDB Tool on the Geography Database

Once this is completed, the systems checks to ensure that no ambiguity has been introduced. If this process does introduce any ambiguity, the schema is rejected.

4.2. Knowledge Base (KB)

The Semantic Wrapper's Knowledge Base (KB) is the interface between the KDB Tool and the Query Translator. All of the mapping information that is generated by the KDB Tool during semantic schema creation is saved in the KB. The KB is an Extensible Markup Language (XML) [19] file that will subsequently be used to translate Semantic SQL queries into standard SQL queries by the Query Translator (see Section 3.3). Specifically, the KB stores all the needed information for both the relational and semantic database schemas, as well all derivation rules for query translation. Its XML format, in particular, provides a flexible, robust and easy to use avenue for capturing complex semantic information in conjunction with the relational and semantic schemas.

Along with the KDB Tool, the KB includes sets of inference rules that can be used to generate new knowledge that is needed during query translation. This is particularly useful when there is not enough information to complete the transformation of the semantic query, such as when the Semantic Wrapper is being used to integrate data from heterogeneous multi-database environments.



is replaced by:

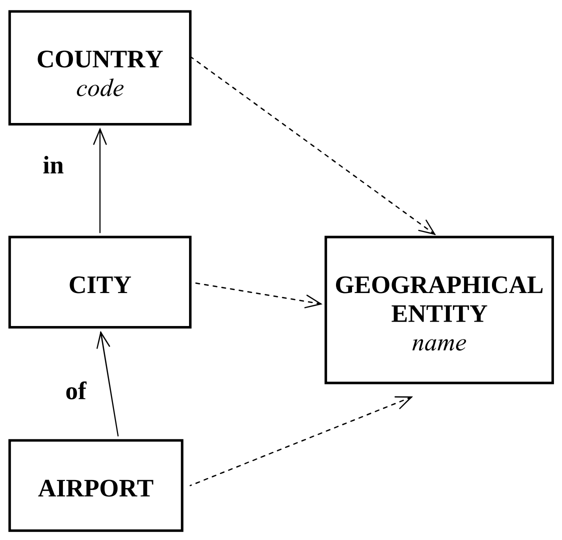


Figure 11. Step 8 of the KDB Tool on the Geography database

4.3. Query Translator: Automatic Query Conversion

The Query Translator is the central processor of the Semantic Wrapper. This component is responsible for transforming the easy to use Semantic SQL queries that are based on the semantic schema into the more complex corresponding relational SQL queries on the relational database. While this component translates Semantic SQL queries into relational SQL queries that are significantly more complex, they are still semantically equivalent to the original query posed by the user.

To accomplish this, the Query Translator interfaces with the KB to retrieve and use the semantic and relational schema information recorded during the semantic schema creation (see Section 4.1), as well as any needed derivation rules. Once the Query Translator retrieves needed information from the KB, it generates the appropriate projections on virtual tables via the use of temporary views. Query results are generated through the application of outer joins or sub-queries explicitly in the *Where* clause between these temporary views. The constructed relational SQL queries are subsequently transmitted to the relational database management system via a standard interface.

Semantic SQL queries are often an order of magnitude shorter than the corresponding relational SQL queries. The complexity involved in transforming a Semantic SQL query into a relational SQL query is best seen by a direct comparison of SQL statements on a real world semantic schema and its relational counterpart. Referring back to Figure 2 and Figure 3, some examples of real world Semantic SQL queries and their functionally equivalent relational SQL queries on these subschemas include:

Example 1: List of the time and housing of temperature measurements over 50 degrees

Semantic SQL Statement	***Relational SQL Statement***
select *housing,time* from *MEASUREMENT* where *of__name='Temperature'* and *value>50*	select *housing, time* from *PHYSICAL_OBSERVATION_STATION,* *MEASUREMENT* where exists *(*select * from *MEASUREMENT-TYPE* *where name_key = of__name and* *name_key = 'Temperature' and* *by_physical_observation_station_id =* *physical_observation_station_id_key* and *value > 50)*

Example 2: The descriptions of organizations and locations of their fixed stations

Semantic SQL Statement	***Relational SQL Statement***
select *description, LOCATION* from *ORGANIZATION*	select *description, LOCATION.north_UTM_in_key, LOCATION.east_UTM_in_key* from *ORGANIZATION, LOCATION* where exists (select * from *FIXED_STATION* where exists (select * from *PHYSICAL_OBSERVATION_STATION__BELONGS_TO__ORGANIZATION* where *name_key = organization__name_in_key* and *PHYSICAL_OBSERVATION_STATION__BELONGS_TO__ORGANIZATION. physical_observation_station_id_in_yy = FIXED_STATION. physical_observation_station_id_key* and *located_at__north_UTM = north_UTM_in_key and located_at__east_UTM = east_UTM_in_key))*

Example 3: The observations since January 1, 1993 (including images, measurements and their types) with location of the stations

Semantic SQL Statement	***Relational SQL Statement***
select *OBSERVATION__, of__, LOCATION* from *OBSERVATION* where *time>'1993/01'*	*(*select *MEASUREMENT_TYPE.*, LOCATION.north_UTM_in_key, LOCATION.east_UTM_in_key, MEASUREMENT.*, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL* from *MEASUREMENT_TYPE, LOCATION, MEASUREMENT where time > '1993/01'* and exists (select * from *FIXED_STATION where by__physical_observation_station_id = physical_observation_station_id_key and located_at__north_UTM = north_UTM_in_key* and *located_at__east_UTM =east_UTM_in_key* and *of__name = name_key))* union (select *MEASUREMENT_TYPE.*, NULL, NULL, MEASUREMENT.*, NULL,NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULLfrom MEASUREMENT_TYPE, MEASUREMENT* where

```
time > '1993/01' and not exists (select * from
FIXED_STATION
whereby__physical_observation_station_id =
physical_observation_station_id_key and
of__name = name_key )) union (select NULL,
NULL, NULL, NULL,
LOCATION.north_UTM_in_key,
LOCATION.east_UTM_in_key, NULL, NULL,
NULL, NULL, NULL, NULL, IMAGE.* from
LOCATION, IMAGE where time > '1993/01'
and exists (select * from FIXED_STATION
where by__physical_observation_station_id =
physical_observation_station_id_key and
located_at__north_UTM = north_UTM_in_key
and located_at__east_UTM =
east_UTM_in_key )) union (select NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, IMAGE.* from IMAGE
where time > '1993/01' and not exists (select
* from FIXED-STATION where
by__physical_observation_station_id =
physical_observation_station_id_key))
```

As can be seen in theses three examples, the relational SQL queries that are constructed by the Query Translator are often substantially larger and more complex than the semantic SQL queries created by users.

4.4. Semantic Wrapper as Middleware

The Semantic Wrapper is a middleware system that provides semantic views over legacy relational databases. As middleware, this system provides straight-forward, easy access to legacy relational databases without requiring users to switch from their existing interfaces to a new, unfamiliar interface. As is illustrated in Figure 12, the Semantic Wrapper can be employed in many environments and for numerous applications, including as middleware for web applications.

The Semantic Wrapper greatly improves usability by allowing the use of standard SQL syntax to access the semantic layer via more simplified and concise SQL queries than what would be required for direct querying of a standard relational database. This approach is also applicable in a heterogeneous multi-database environment that can include both structured (relational and semantic databases) and unstructured data (social media and related Internet sites).

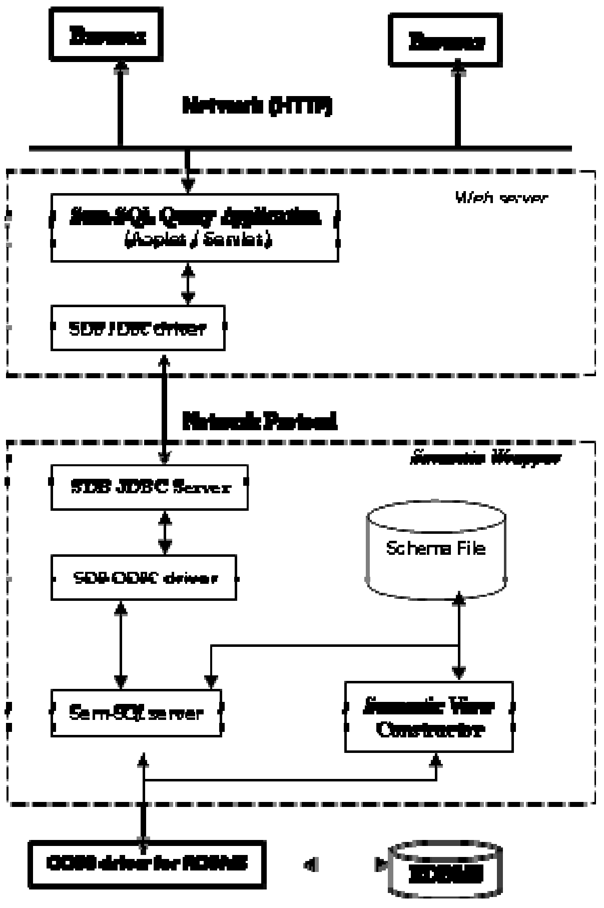


Figure 12. Overall Architecture of the Semantic Wrapper as Middleware for a Web Application

5. Conclusion

The Semantic Wrapper is a middleware system that can be used to greatly improve the ability to meet the intense and ever-changing data management needs of today's world. The Semantic Wrapper provides an easy to use method for accessing legacy and relational databases, while still maintaining the ability to be implemented as a standalone solution. It allows users to continue to use familiar

GUIs and greatly decreases the complexity of SQL syntax needed from users to fulfill their data requests. Finally, the system can be used over both structured and unstructured, heterogeneous data sources, providing a set of tools that can easily incorporate new and diverse sources of data.

Acknowledgement

This work was supported in part by NSF grants CNS-0821345, HRD-0833093, IIP-0829576, IIP-0931517, CNS-1057661, IIS-1052625, CNS-0959985, CNS-0837556, CNS-0426125, DGE-0549489, IIS-0957394, and IIS-0847680**.**

References

1. M. Egenhofer. “Why not SQL!” International Journal on Geographical Information Systems, vol. 6, no.2, 1992, p. 71-85.
2. H.V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. “Making Database Systems Usable”. ACM's Special Interest Group on Management of Data (SIGMOD), Beijing, China, June 11–14, 2007.
3. A. E. Wade. “Hitting the Relational Wall”. Objectivity Inc. White Paper. [Online] 2005. http://www.objectivity.com/pages/object-oriented-database-vs-relational-database/default.html.
4. E.F. Codd. “A Relational Model of Data for Large Shared Data Banks”. Communications of the ACM, vol. 13, no. 6, 1970, pp. 377–387.
5. E.F. Codd. “The Relational Model for Database Management”. Reading, MA, Addison-Wesley, 1990.
6. G. Russell. “Database eLearning”. [Online] http://db.grussell.org/index.html.
7. M. Berler, J. Eastmen, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez. “The Object Data Standard: ODGM 3.0”. [ed.] K.D. Barry, R.G.G. Cattell. San Francisco, CA, Morgan Kaufmann, 2000.
8. A. Eisenberg, J. Melton. “SQL: 1999, formerly known as SQL3”. ACM SIGMOD Record, vol.28, no.1, 1999.
9. N. Rishe. “Database Design: The Semantic Modeling Approach”. New York, NY, McGraw-Hill, 1992.
10. N. Rishe, J. Yuan, R. Athauda, S.C. Chen, X. Lu, X. Ma, A. Vaschillo, A. Shaposhnikov, D. Vasilevsky. “SemanticAccess: Semantic Interface for Querying Databases”. Proceedings of the 26th International Conference On Very Large Data. Cairo, Egypt, September 10-14, 2000. pp. 591-594.
11. A. Cary, Y. Yesha, M. Adjouadi, N. Rishe. “Leveraging Cloud Computing in Geodatabase Management”. Proceedings of the 2010 IEEE Conference on Granular Computing GrC-2010. Silicon Valley, CA, August 14-16, 2010. pp. 73-78.

12. L. Yang, N. Rishe. “Formal Representation and Transformation of DTDs to Sem-ODM”. The 2006 International Conference on Foundations of Computer Science, FCS06. Las Vegas, NV, June 26-29, 2006, pp.182~188.
13. L. Yang, N. Rishe. “Transforming Sem-ODM Semantic Schemas to DTDs”. Proceedings of the 43rd ACM Southeast Conference, ACMSE 2005. Kennesaw, GA, vol. 1, March 18-20, 2005, pp. 237-242.
14. N. Rishe, A. Barreto, M. Chekmasov, D. Vasilevsky, S. Graham, S. Sood. “Semantic Database Engine Design”. Proceedings of the 7th International Conference on Enterprise Information Systems, ICEIS 2005. Miami, FL, May 24-28, 2005, pp. 433-436.
15. N. Rishe, A. Barreto, M. Chekmasov, D. Vasilevsky, S. Graham, S. Sood. “Object ID Distribution and Encoding in the Semantic Binary Engine”. Proceedings of the 7th International Conference on Enterprise Information Systems, ICEIS 2005. Miami, FL, May 24-28, 2005, pp. 279-284.
16. G. Aslan, D. McLeod. “Semantic Heterogeneity Resolution in Federated Databases by Metadata Implantation and Stepwise Evolution”. The VLDB Journal, vol. 8, no. 2, 1999, pp.120-132.
17. N. Rishe. “Semantic SQL”. Internal Document, High Performance Database Research Center, School of Computer Science, Florida International University, 1998.
18. N. Rishe, T. Huang, M. Chekmasov, S. Graham, L. Yang, S. Himelsback. “Semantic Wrapping Tool for Internet”. Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics SCI-2002, Orlando, FL, July 14-18, 2002, pp. 441-445.
19. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. [Online] November 26, 2008. http://www.w3.org/TR/REC-xml/.

Appendix 1: Semantic SQL Virtual Tables Formal Definition

The virtual table *T(C)* for a category *C*, recursive definition:

1. The first attribute of T:

    C — attribute of *T*, range: *C* (*m:1*)

2. For every attribute *A* of *T*, for every relation *r* applicable to the range of *A*:

    *A*__*r* — attribute of *T*, range: *range(r)* (*m:1*)

Note: this virtual table is infinite. When interpreting a specific query, a finite projection of this table is assumed as further explained in Technical Notes.

The name of *T* is the same as of *C*.

Note: *to-many* original relations are reduced to *to-one* attributes of the virtual table.

If the semantic relation *r* is many-to-many or one-to-many, the new attribute would be many-to-one, but many virtual rows would exist in the table *T*, one for each instance of the tree. If *r* has no value for an object, a null value will appear in the virtual relational table.

The relation *r* may be inferred. The range of a virtual attribute may be of multi-media types: numbers with unlimited varying precision and magnitude, texts of unlimited size, images, etc.

Abbreviation of prefixes

- Every component relation *r* in the virtual attribute name may be named by its full semantic name or, if no ambiguity arises, by its short semantic name.
- The attribute names of *T* contain long prefixes. These prefixes can be omitted when no ambiguity arises, i.e.: attribute *y* is an abbreviated synonym of the unabbreviated attribute *x*__*y* of *T* if *T* has no other unabbreviated attribute z__y where *depth (z)* ≤ *depth (x)*.

depth (x) is the number of relations present in *x*.

Surrogates

All attributes of *T(C)* of type Abstract are replaced by their surrogates of type String.

Definition of the extension of a table

The virtual table *T* for a category *C* is logically generated as follows:

1. Initially, *T[C]=C*, i.e. *T* contains one column called *C,* whose values are the objects of the category.

2. For every attribute *A* of *T*, for every schema relation or attribute *r* whose domain may intersect *range(A)*, let *R* be the relation *r* with its domain renamed *A* and range renamed *A__r*, let *T* be the natural right-outer-join of *T* with *R*. (Unlike a regular join, the outer join creates *A__r*=null when there is no match.)
3. For a given query *q* the virtual table against which *q* is interpreted, *T [C ,q],* is a projection of *T [C]* on the following virtual attributes:
    - the virtual attributes that appear in the query,
    - the unabbreviated prefixes of said attributes (including the surrogate attribute *C*),
    - and the attributes *p__r* where *p* is any of said prefixes and *r* is an original printable-type to-one attribute of the semantic schema.

Note: the projection operation here is a *set* operation with duplicated tuples eliminated.

User-control of table depth

(Used only by sophisticated users trying to outsmart $MAXDEPTH defined by a graphical user interface; not needed by users posing direct SQL queries without a GUI.)

- For each category *C*, in addition to the default table named *C*, of depth limited by $MAXDEPTH, there are also tables called *C_i* for any positive integer *i*, with the depth limited by *i* rather than $MAXDEPTH. Also, there is a table *C_0* which includes only the original to-one attributes and relations whose domain is *C* or a supercategory of *C* and the surrogate attribute of *C* .

ODBC schema queries

- The ODBC request for the names of all tables is interpreted as: for every category get the primary virtual table *C* and the tables *C__0* and *C__1*.
- The ODBC request for the names of all attributes of a given virtual table T returns all attributes maximally abbreviated. If the request is for the virtual table corresponding to a category *C*, only attributes of *C__2* are returned
- The ODBC request to browse the virtual table is denied. (Browsing of *C__0* is permitted. Browsability of *C__1* is not guaranteed)

Appendix 2: Disambiguation of Arbitrary Semantic SQL Updates

Let *C* be a category against which an update operation is performed.

Notation:

T = T(C) — the virtual table of *C* .

A — the list of full names of attributes of *T* that are assigned values in the operation.

$R_1,...,R_n$— the set of relations of C such that for some suffix *s* , R_i__*s* is in *A*. (That is, R_i__*s* is a two-step or deeper attribute.)

$C_1,...,C_n$— the ranges of $R_1,...,R_n$.

S_i— list (*s* | R_i__*s* in *A*) in the order of appearance in *A*.

V (a) — For every attribute *a* in *A* let *V(a)* be the value being assigned to the attribute *a* . For every *s* in S_i let *V(s)* be the value assigned to R_i__*s*. Let *V(Si)* be the list of *V(s)* where *s* in S_i.

E_i— the list of assignments s = *V(s)* for *s* in S_i.

1) **delete from** *C* **where** *condition*
   a) perform: select *C* from *C* where condition
   b) for every resultant object *o* in *C*: remove *o* from *C* .

   Example:

   delete from STUDENT **where** FINAL_GRADE<50

2) **insert into** *C* (*attributes*) **values** (*assignments*)
   a) Create a new object in *C* . Let this object be denoted *o*. Its one-step relationships are assigned values from the assignments. If a one-step relationship is m:m or 1:m then only one value may be assigned.
   b) For every category C_i in $C_1...C_n$ do:

```
(1) if $R_i$__$C_i$ is in $A$ and $V(R_i\_C_i)$="new"
     then recursively perform:
          insert into $C_i$ ( $S_i$) values ( $V(S_i)$ );
          let $v$ be the object inserted above
    else do:
          (2.1) perform: select $C_i$ from $C_i$ where $E_i$
          (2.2) if the above select results in exactly one object,
            then denote that object $v$
            else abort with an error message
        (2) relate: $o\ R_i\ v$
```

Example: create a new student James in the department in which Johnson works and enroll Jim in the only existing offering of "Magnetism":

```
insert into STUDENT
        ( FirstName,  Major__WorksIn___LastName,  Enrollment,
        The_Course )
        values ('James', 'Johnson',   'new', 'Magnetism')
```

```
3)  insert into C ( attributes ) query
    a)  Evaluate the query, resulting in a set of rows.
    b)  For each row r perform: insert into C ( A ) values ( r )
```

Example: For every instructor create a department named after him and make him work there:

```
insert into  DEPARTMENT
             ( Name, WorksIn_ )
select       LastName, Instructor
from         Instructor
```

```
4)  update C set assignments where condition
    a)  perform:
        select C from C where condition
    b)  for every object o in the result of the above query perform:
    (1) The object's one-step relationships are assigned values from the
        assignments, i.e.: for every one-step attribute $A_i$ in $A$ perform: $o.A^i := V(A_i)$
    (2) For every category $C_i$ in $C_1 \ldots C_n$ do:
        (2a)  if $R_i$__$C_i$ is in $A$  and $V(R_i\_C_i)$="new"
              then recursively perform:
                  (2a1)  insert into $C_i$ ( $S_i$ ) values ( $V(S_i)$ );
                  (2a2)  let v be the object inserted above
        (2b)  else do:
                  (2b1) perform: select $C_i$ from $C_i$ where $E_i$
                  (2b2) if the above select results in exactly one object,
                        then denote that object $v$
                        else abort with an error message
```

(2c) $o.R_i := v$

5) **insert into** *C__R* ...

Allows creation of multiple relationships *R*. This cannot be accomplished with previous commands when *R* is many-to-many and many values need to be assigned. Note: *C__R* has been defined as a virtual table.

Example: let Johnson work in Physics

> **insert into** INSTRUCTOR__WorksIn (INSTRUCTOR, DEPARTMENT)
> **select distinct** INSTRUCTOR, DEPARTMENT
> **from** INSTRUCTOR, DEPARTMENT
> **where** INSTRUCTOR.LastName='Johnson' **and** DEPARTMENT.Name='Physics'

Example: let Johnson work in every department

> **insert into** INSTRUCTOR__WorksIn (INSTRUCTOR, DEPARTMENT)
> **select distinct** INSTRUCTOR, DEPARTMENT
> **from** INSTRUCTOR, DEPARTMENT
> **where** INSTRUCTOR.LastName='Johnson'

6) delete from *C__R* **where** *condition*

Allows deletion of multiple relationships *R*.

Example: do not let Johnson work in any department Smith works in.

> **delete from** INSTRUCTOR__WorksIn
> **where** LastName='Johnson' **and** WorksIn (
> **selec**t WorksIn **from** INSTRUCTOR **where** LastName='Smith')

7. Object surrogate assignment: if in an insert statement there is an assignment of a user-supplied value to an object being created, that value becomes the object's surrogate, overriding surrogates generated by other algorithms. In the database it is entered into the attribute UserSuppliedSurrogate, which is enforced to be 1:1. Further, if this value begins with the character "#" the database will derive the internal object id from this value — it may have effect only on efficiency. If this value begins with a "$" it will be automatically erased at the end of the session.

Example:

> **insert into** INSTRUCTOR (Instructor, FirstName)
> **values** ('John', 'John')

Note: any expression producing an abstract object is automatically converted into that object's surrogate.

Index terms (alphabetically):