

Modeling Object Views in Distributed Query Processing on the Grid

Krzysztof Kaczmarek¹, Piotr Habela², Hanna Kozankiewicz³, Kazimierz Subieta^{2,3}

¹Warsaw University of Technology, Warsaw, Poland
k.kaczmarek@mini.pw.edu.pl

²Polish-Japanese Institute of Information Technology, Warsaw, Poland
habela@pjwstk.edu.pl

³Institute of Computer Science PAS, Warsaw, Poland
{hanka, subieta}@ipipan.waw.pl

Abstract. This paper proposes a method for modeling views in Grid databases. Views are understood as independent data transformation services that may be integrated with other Database Grid Services. We show examples of graphical notation and semi-automated query construction. The solution is supported by a minimal metamodel, which also provides reflection capabilities for dynamic services orchestration.

1 Introduction

The Grid applications, integrating the whole variety of different computer systems recently became extremely complex. Emerging OGSA standard is a significant improvement of distributed systems' integration. With the introduction of Grid Database Services, OGSA-DAI and OGSA-DQP, access to distributed database resources, querying, optimizations, parallelization and analysis has been greatly simplified [10]. In services-based philosophy users formulate queries and send them to dynamically created Grid Data Service, which is responsible for the rest of the job, and which may use other services to achieve results in the best possible way, for example a Distributed Query Service combined with several Grid Query Evaluation Services. However, formulating queries directly over distributed data, having in mind all possible fragmentations is very difficult if not impossible for most users. What we need in distributed systems, especially in heterogeneous environments, are higher-level interfaces, which could transparently integrate fragmented data and transform it to our needs.

Here, we propose updatable object views as a convenient means of defining higher abstraction over tangled and distributed data. OGSA-DAI defines two general kinds of components: *data access components* and *data integration components*. Our updatable views are rather the latter case, but could also be called *data transformation components*, since they do not store any data and are not limited to any particular task like integration. The advantage of our solution is its simplicity and minimalism beneficial for Grid end users, who are not aware of the local databases heterogeneity, neither data fragmentation nor other aspects of data distribution.

1.1 Motivation and Related Work

Our main motivation is to provide a support for development process of such a Grid database, in which certain nodes have a role of data transformation points. We see each view as an object's interface transformation service, which has its semantics, data input and output. Such a data transformation component may be wrapped by a dedicated Data Grid Service and used among other data services, serving high-level integrated data. What is more important is that such a wrapped data transformation component may be used by DQP services in many places according to an optimal query evaluation plan. The only limitation is that our updatable view must be created by a database designer in order to specify the semantics of generic update operations performed through the view. The goal is to propose a modeling tool supporting such object transformations and their compositions. The designer's tasks are describing resources consumed or produced by certain views; and finding and describing dependencies between consumed and produced objects, which in fact describe transformations' logics. To assist these tasks, in the rest of the paper we focus on:

- modeling virtual and real objects' interfaces as descriptions of views' input and output
- describing sequences of transformations, tracking dependencies between interfaces
- recognizing basic patterns of transformations performed by a view
- establishing constraints between views.

One of the recent approaches to visual modeling of database views or virtual objects except for the simple view notation in UML for database design [9]. Our notation is much richer and dedicated to distributed database systems.

There are also many multiparadigmatic database query building proposals based on forms, graphical notations or virtual reality [13]. However, they are focused rather on inexperienced database users, while we support database designers and integrators.

Data transformations (integrations) performed by specialized views, creating virtual resources, are common in many systems. In distributed databases, there are two major, well-known approaches to integrate data: global-as-view [1], where the data in the global schema are defined as views over the data in the sources, and local-as-view, where the data in the sources are defined as views over the global schema [8]. In this paper, we present a mutation of global-as-view approach, which creates virtual data in a form consumed by users, but distributed among nodes and ready not only to serve local but also global users. Piazza Peer Data Management System [12] uses similar way of exchanging information between nodes. In Piazza, schema mediation may be performed dynamically, plus peers may attach and detach dynamically. We consider allowing such a possibility in updatable views, based on the mechanisms located higher in meta-levels' hierarchy and allowing generic programming over metadata. This topic is, however, beyond the scope of this paper.

2 Views in a Grid Database

A Grid Database, created and used by a consortium [2], consists of independent database systems, and specialized Grid Services, which share data i.e., publish and con-

some objects. Each of them may use two kinds of specialized views to achieve desired purposes:

Contributory view (sometimes called *mediator*) is a view by means of which a node shares its own resources with the others. Its main task is to hide heterogeneity of local database systems (object, relational, etc.) within the consortium, by transforming local data models into unified data model specific for the consortium. Its second task is controlling access rights and hiding data, which should not be published.

Grid view is a view performing a Grid Data Transformation Service for users (local or global). Through this view, a user sees Grid resources adapted to his/her particular needs. All data transformations, like integration, are transparent. Users see only resulting objects created according to a predefined schema. We say that data created by the view is *virtual*, because it does not exist in any concrete place but is created on the fly, when needed upon distributed resources [5].

One of the most important features of our approach to data transformation in Grid is that virtual objects are indistinguishable from real objects. Applications may use data from local repository exactly in the same way as data produced by Grid views. In this way, transparency of data fragmentation and transparency of location are satisfied. However, if a Grid view uses data provided by other views in the system, it never knows where data come from and where exactly a query will be executed. The request may propagate over the network [4]. Thus, a virtual object created by a Grid view indirectly depends on the whole *sequence of transformations*. Describing and analyzing possible views settings is one of the database designer's tasks.

2.1 A Generic Data Transformation Based on an Updatable View

A mature view-based database system has to support not only data retrieval but also data updates. Classical database views (materialized or not) have limited functionality in this field. Our data model and query language allows to specify the intended virtual data update behavior through dedicated view's procedures.

The view mechanism is defined in terms of the Stack-Based Approach (SBA) [11], which assumes that query languages are a special kind of programming languages and can be formalized in a similar manner (Stack-Based Query Language). A database view definition is not a single query (as in SQL), but it is a complex structure. It consists of two parts: the first one determines the so-called *seeds* (the values or references to stored objects that are the basis for building up virtual objects), and the second one redefines the generic operations on virtual objects [5]. The first part of the view definition is an arbitrarily complex functional procedure, and is similar to extraction programs used in [7]. The *seeds* it returns are passed as parameters for the operations on virtual objects. The operations have the form of procedures that override default updating operations. We identified four generic operations that can be performed on virtual objects (which also completely cover CRUD functionality):

Updating, which assigns a new value to the virtual object. A parameter the procedure accepts is the new value to be assigned.

Deletion, which deletes the virtual object.

Insertion, which inserts a new object into the given virtual object.

Dereference, which returns the value of the given virtual object.

For a given view, an arbitrary subset of these operations can be defined. If any operation is not defined, it means it is forbidden (we assume no updating through side effects, e.g. by references returned by a view invocation).

Moreover, a view definition may contain nested views, defined within the containing view's environment. Thus, arbitrarily nested complex objects may be constructed.

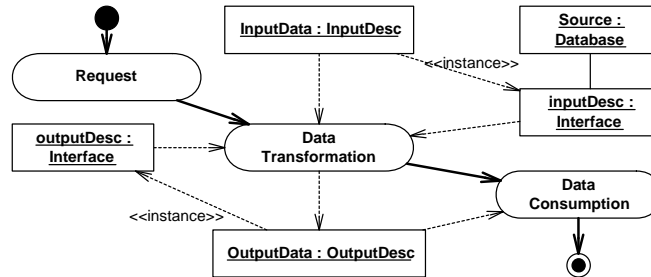


Fig. 1. A generic data transformation performed by a single view uses (meta) description of input and output data (maybe implicit, known only to programmer), collections of input data and produces output data in desired form. Input description is connected to certain location, which is separate information and thus may be easily changed.

When a view is invoked in a query, it returns a set of virtual identifiers (that are counterparts of the identifiers of stored objects). Next, when a system tries to perform update operation with a virtual identifier as an l-value, it recognizes that it deals with the virtual object and calls a proper update operation from the view definition. To enable that, a virtual identifier must contain both a seed and the identifier of the view definition. The whole process of view updating is internal to the proposed mechanism and is invisible to view users, who deal with virtual objects in the same manner as with real objects. This feature is known as view *transparency* and it is a key requirement for the view mechanism in our data integration approach.

In views, data transformations are defined using any procedural and declarative constructs allowed in SBQL. Fig. 1 shows a view performing a generic transformation of *InputData* into *OutputData*. The transformation is described by interfaces of input data (*InputDataDescription*) and interfaces of output data (*OutputDataDescription*). For simplicity in the figure, data transformation consumes and produces only one type of objects, but it is not a requirement.

Each interface transformation has its semantics intended by the designer to produce certain data objects. We may recognize several well-known types of typical data transformations – *transformation patterns*. For example, we may consider hiding some information by selecting only desired object's attributes [3] (*projection pattern*), or aggregate – e.g. a collection of integers to calculate an average value. One of the most important transformation patterns in distributed scenario are merging and joining views.

Please note, that we do not explicitly say here, whether transformation is performed by contributory view, or Grid view. It is not important from conceptual point of view, since the difference is only in source data used. Data distribution is transparent, so in fact, within assumed privileges, a view may access data anywhere. For modeling views and sequences of transformations, their location is less important.

2.2 Sequences of Transformations

In the mentioned Grid database system, each view may offer virtual objects based on source objects, which may be concrete or virtual. This ability is important for flexibility of the system. Usage of virtual objects by another view builds a sequence of transformations. Data objects may travel through several nodes and undergo transformations before their final consumption. In some cases, it may be important to look for optimizations to minimize transfer and unnecessary transformations. In larger Grid systems, automated DQP Service supported by an optimizer and query evaluator may decide to move some parts of a data transformation sequence to nodes that are more powerful or ones closer to the end user, achieving performance improvements. These systems would have to use meta-information concerning involved objects' interfaces and be able to infer about their compatibility. This inference may sometimes require human cooperation.

3 Transformations Modeling

This section introduces notations for modeling various transformations on data that may appear in a Grid. However, please note that the transition from a design to an implementation, by for instance code generators, cannot be fully automated in case of Grid databases. This is because of two reasons: limitations of sensible graphical modeling notation, which is not capable of providing all the necessary details (except perhaps for the simplest patterns) and independence of nodes in distributed systems. A Grid system may always face problems of missing fragments of objects, contradictory or incomplete information [6]. Thus, creation of updatable view requires manual (or at least partially manual) implementation of two basic parts: virtual object creation (seeds) upon source objects, which must handle all exceptional situations, and virtual object changeability implementation methods, which must react to users' operations. We propose assistance for some parts of implementation tasks, but manual programming or at least human control, would always be necessary.

3.1 Modeling Virtual Objects

As it was explained earlier, describing data objects and dependencies between them is crucial for designing data transformations in Grid databases. Here, we propose solutions for extended interfaces and dependencies descriptions.

Objects Interfaces. Comparing to standard UML notation, externally visible interfaces of virtual and concrete objects in Grid database require additional constructs for proper showing the changeability allowed for particular (virtual or concrete – treated uniformly) objects. Distinction of the composition of nested objects and references among objects, which is important for object databases, can be solved with the standard UML notation, if there is an agreement on the semantics of the composition relationship. However, changeability flags would need the following symbols (see Fig. 2): *isUpdatable* (“!”); *isDereferenceable* (“?”); *isRemovable* (“^”); *isInsertable* (“>”). The symbols can appear before a feature name (or before a class name in

the simplified syntax suggested in Fig. 2). The changeability symbols are shown within the curly brackets to allow suppressing changeabilities (by showing no brackets, to distinguish from declaring a feature with none of the changeabilities allowed).

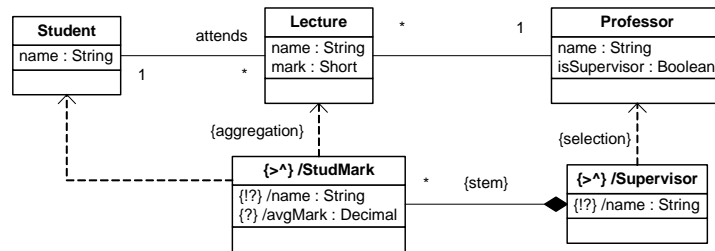


Fig. 2. A sample definition of virtual objects’ interfaces (transformations hidden for clarity). Assume the data are restructured according to the needs of some external system (e.g. a statistical analysis subsystem), which should not have any access to the identities of the students. *StudMark* and *Supervisor* show also the changeability notation. Selection means, that to provide Supervisor virtual objects only certain *Professor* source objects are selected. Aggregation indicates that a number of *Lecture* objects are used to create a single *StudMark* object. ‘Stem’ label indicates preserving the structure (and dependency) of source objects. Here *StudMark* depends on *Supervisor* as *Lecture* is connected to *Professor*.

Dependency Illustration. Here, we suggest notation based on the generic UML dependency relationship and using the same graphical notation (labeled «view dependency» if necessary). Notice that for pragmatic reasons we simplify the notation. Although the view dependencies span between structural features the dependency arrows are drawn rather between their classes. In contrast to the regular dependency arrow, view dependency can additionally indicate (using keywords within curly brackets, as shown in Fig. 2), the selectivity and aggregation property (*selection* and *aggregation* keywords respectively). To indicate that particular complex view (that is, a view containing other views) preserves the structure of its source object (mapping the features of the latter), we use the *stem* keyword in the properties representing sub-views.

3.2 Modeling Sequences of Transformations

Simple dependency arrows as shown in the previous section are enough to indicate necessary interface transformation when a single virtual object depends on one source object. A more complicated case, especially data integration, may introduce additional dependencies between sources and results. Fig. 3 shows an example of several data objects correlated by a common join operation. We may notice following properties of this joining transformation:

- All source objects must have a composition key used in join (here, ID)
- Resulting virtual object have at most data from input objects
- If some corresponding part of an object (say scholarship) may be missing it should be reflected in output interface as an optional property. Otherwise output virtual object cannot be created

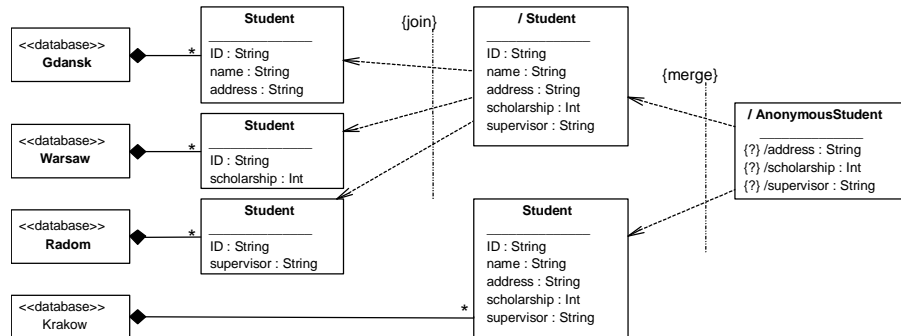


Fig. 3. Example integration of distributed data. Two transformations realize patterns named 'join' and 'merge'. For clarity, we do not show explicitly nodes offering virtual objects. Objects' composition is performed using certain keys. Here, it would be student's ID. Please note, that we do not consider expressions defining keys in this paper.

However, the diagram says nothing about constraints or situations in which some data are missing. Programmers of updatable views must overcome these problems. In the same way, extraction programs from [7] solve the problem of uncertainty.

Transformations' compositions may be done only under constraints on input and output data types. In the Fig. 3, merging transformation uses results of the previous join. It unions two collections of data: from Krakow database and the newly joined objects. Union of collections of objects is sensible if all objects have the same semantics, which is expressed by their interfaces. Because interface of Student objects in Krakow is known and fixed, it constrains output interface of the join transformation.

An important feature of diagram in Fig. 3 is that objects' interfaces and thus views, which produce them, are separated from certain locations. As it was quoted earlier, for transformation's semantics data origin is not important. However, it becomes important for system's implementation. Thus, to support transition from the design to the implementation we must supply information about concrete roles of database nodes in data exchange. It means that each interface used in the Grid should be explicitly assigned to a node or be somehow connected (specialization, reference, etc.) to an interface, which is already assigned. If objects implementing same interface are located in many nodes each location must be shown explicitly (like two identical Student objects before merging transformation in Fig. 3).

Separation of a data source, transformation's semantics and interfaces simplifies insertion of a new transformation or a data source inside a previously modeled sequence. A new interface or a transformation has to agree with established standardized data description, plus it must be known to the rest of the transformations in the chain, which stay unmodified. Let us consider introducing a new node offering Student objects similar to those published by Krakow database from Fig. 3. Merging view has only to get the information about the new data source. That could be supplied in the runtime by a dynamical list of data sources or statically by modifying the transformation chain diagram and regenerating code for particular views.

Example Sequence of Transformations. If the semantics of a transformation is of known pattern, then even if updatable view's code generation is not completely auto-

mated, a CASE tool could suggest partial solution. Below we present transformations from Fig. 3 implemented in a single query. It uses input and output interfaces definitions and source objects location. Combination of many transformations in a single query is possible because of powerful capabilities of query composition in SBQL.

1. Performing ‘join’ (transformation’s semantics) and shaping the output for the next transformation (output–input constraint):

```
((Gdansk.Student as g) join (Warsaw.Student as w where w.ID=g.ID)
join (Radom.Student as r where r.ID=g.ID)).(g.ID as ID, g.name as
name, g.address as address, w.scholarship as scholarship,
r.supervisor as supervisor)) as Student
```

2. Adding ‘merge’ (transformation’s semantics) and creating final output required by output constraints (output constraint):

```
((((Gdansk.Student as g) join (Warsaw.Student as w where w.ID=g.ID)
join Radom.Student as r where r.ID=g.ID)).(g.ID as ID, g.name as
name, g.address as address, w.scholarship as scholarship,
r.supervisor as supervisor)) as Student) union Cracow.Student as Stu-
dent)).
(Student.address, Student.scholarship, Student.supervisor))
as AnonymousStudent
```

Presented query is only the updatable view’s data retrieving declaration. We may see the model’s influence on its certain parts. The rest of the view definition, which supports updating, deleting, inserting and dereferencing procedures, has to be programmed separately.

3.3 Metamodel Supporting Transformations

At the metamodel level, we decided to treat all objects’ interfaces in the same way, regardless of their virtual or concrete nature. Notions, which are in fact view-related but have influence on how we see resulting objects, are kept in *StructuralFeature* class. The only change into existing UML notions is the replacement of the changeability attribute from that class by four boolean attributes. We use the following names: *isUpdatable*, *isRemovable*, *isInsertable* and *isDereferenceable*. We assume that those structural features, which possess (standard-defined) tagged value “derived” represent virtual objects and may be the subject of data dependency specifications.

Database class represents a source node offering objects. Transformation is attached to certain location though *StructuralFeature*, which describes its output. In other words, a view must be located in a node, which is to offer certain objects.

Transformation class is responsible for describing transformation performed by an updatable view to produce virtual objects, pointed by *produces*. Interfaces consumed by a view are enumerated by multiple *uses* reference. Although it is not possible to precisely describe visually how a given virtual object is computed, some information can be easily provided concerning the characteristics of a view dependencies and relations between them, which are in fact data integration patterns. Kind of a transformation may be set in *IntPaternKind* (we follow UML style here). The basic transformation patterns concerning data integration are *projection*, *merge* and *join*, but metamodel is open for any other custom defined and named patterns. The implemen-

tation phase uses this description to create certain template for query matching desired semantics.

Dependency between transformation and interfaces it consumes may be additionally described by mutually orthogonal flags: *isSelective* (Source data are used to select only the objects meeting a given criteria), *isAggregating* (This property indicates that a given virtual object realizes a many-to-one mapping of the source data). This description affects the kind of query that is to be used to create seeds of virtual objects.

The proposed transformation properties are not exhaustive, as it is not possible to cover with such notions the whole expressiveness of even the most typical queries that may be used as view definitions. However, it provides some hint concerning the intent of a given view, with the level of detail that is feasible to show on a diagram and enough to constrain and control implementation of a modeled view.

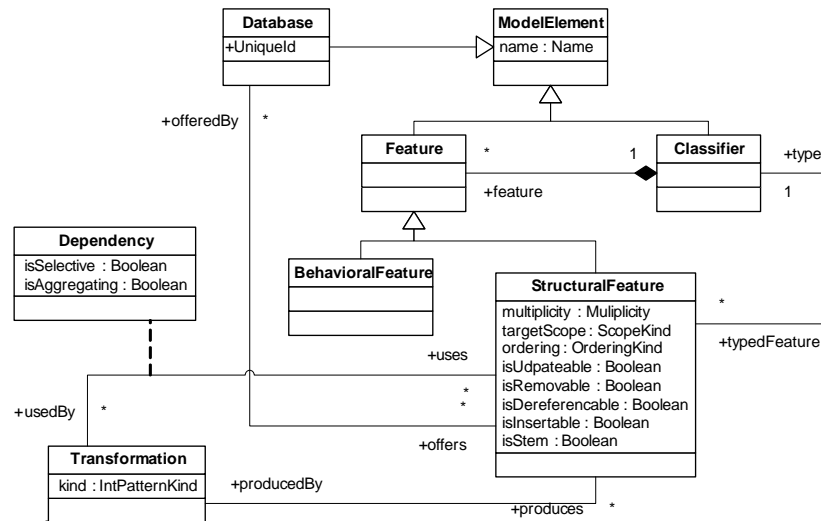


Fig. 4. A fragment of UML Metamodel extended by features necessary for proper dependency tracking between transformations.

4 Conclusions and Future Work

In this paper, we have presented a feasible method supporting a user in modeling views in Grid Databases. A Grid designer is provided with tools to model data transformation and to constrain transformation's chains. Our notation allows to describe visually various data operations like projection, aggregation and join, plus ordered sequences of these operations. Additionally, the notation is flexible and can be easily extended with other operations on data if they would be helpful for the view modeler.

The advantages of our solution are: its simplicity and flexibility for the Grid designer; metamodel that supports semi-automatic generation of queries over and is well suited for accompanying CASE modeling tools; the structure of meta-information database, which may be queried by DQP services and optimizers to compose views or

move them in order to achieve additional query evaluation improvements. Grid users benefit from an abstract middle layer, which hides data integration and heterogeneity and is not limited to querying, but it supports all operations, which can be performed on virtual data shared in the Grid. The described views may be easily incorporated in OGSA-DAI based systems and similar solutions.

The future work will focus on visual modeling of scenarios that are more dynamic, and which could support usage of generic integration patterns and transformation templates based on dynamical data ontology discovering.

References

1. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman and J. Widom, The {TSIMMIS} Project: Integration of heterogeneous information sources. 16th Meeting of the Information Processing Society of Japan, Tokyo, 1994.
2. I. Foster, C. Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 2003
3. W. Heijenga. View definition in OODBS without queries: a concept to support schema-like views. In *Doct. Cons. 2nd Intl. Baltic Wg on Databases and Information Systems*, Tallinn (Estonia), 1996
4. K. Kaczmarek, P. Habela, K. Subieta. Metadata in a Data Grid Construction. Proc. of the 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), Modena, Italy, 2004
5. H. Kozankiewicz, J. Leszczyński, K. Subieta. Updateable XML Views. Proc. of *Advances in Databases and Information Systems (ADBIS)*, Springer LNCS 2798, pp. 385-399, Dresden, Germany, 2003.
6. M. Lenzerini. Data integration: a theoretical perspective. Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. Madison, Wisconsin, 2002
7. D. Calvanese, E. Damaggio, G. De Giacomo, M. Lenzerini, R. Rosati. Semantic Data integration in P2P systems. Proceedings of the International Workshop on Databases, Information Systems, and P2P Computing, Berlin, Germany, September 2003.
8. A.Y. Levy, A. Rajaraman and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. Proceedings of the 22nd VLDB Conference, Mumbai (Bombay), India, 1996
9. E. Naiburg, R. A. Maksimchuk. UML for Database Design. Addison-Wesley, 2001
10. Open Grid Services Architecture Data Access and Integration Documentation <http://www.ogsadai.org.uk/dqp/>
11. K. Subieta, C. Beeri, F. Matthes, and J. W. Schmidt. A Stack Based Approach to Query Languages. Proc. of 2nd Intl. East-West Database Workshop, Klagenfurt, Austria, September 1994, Springer Workshops in Computing, 1995.
12. I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyaska, G. Miklau, and P. Mork. The Piazza Peer Data Management Project. *ACM SIGMOD Record*, 32(3), 2003.
13. T. Catarci, S. Chang, et al. A Graph-Based Framework for Multiparadigmatic Visual Access to Databases, *IEEE Transactions on Knowledge and Data Engineering* 8(3), 1996