

Measuring and Diffusing Data Quality in a Peer-to-Peer Architecture

Diego Milano, Monica Scannapieco and Tiziana Catarci

Dipartimento di Informatica e Sistemistica,
Università degli Studi di Roma "La Sapienza",
Via Salaria 113, Rome, Italy
{milano,monscan,catarci}@dis.uniroma1.it

Data quality is becoming an increasingly important issue in environments characterized by extensive data replication. Among such environments, this paper focuses on Cooperative Information Systems (CISs), for which it is very important to declare and access quality of data. The paper describes a general methodology for evaluating quality of data, and the design of an architectural component, named Quality Factory, that implements quality evaluation of XML data. The detailed design and implementation of a further service, named Data Quality Broker, are presented. The Data Quality Broker accesses data and related quality distributed in the CIS and improves quality of data by comparing different copies present in the system. The Data Quality Broker has been implemented as a peer-to-peer service and a set of experiments on real data show its effectiveness and performance behavior.

I - Introduction

Data quality is a complex concept defined by various *dimensions* such as accuracy, currency, completeness, consistency (Wang & Strong, 1996). Recent research has highlighted the importance of data quality issues in various contexts. In particular, in some specific environments characterized by extensive data replication high quality of data is a strict requirement. Among such environments, this paper focuses on Cooperative Information Systems.

Cooperative Information Systems (CISs) are all distributed and heterogeneous information systems that cooperate by sharing information, constraints, and goals (Mylopoulos & Papazoglou, 1997). Quality of data is a necessary requirement for a CIS. Indeed, a system in the CIS will not easily exchange data with another system without a knowledge on the quality of data provided by the other system, thus resulting in a reduced cooperation. Also, when the quality of exchanged data is poor, there is a progressive deterioration of the overall data quality in the CIS. On the other hand, the high degree of data replication that characterizes a CIS can be exploited for improving data quality, as different copies of the same data may be compared in order to detect quality problems and possibly solve them.

In (Scannapieco, Virgillito, Marchetti, Mecella, & Baldoni, 2004; Mecella et al., 2003), the DaQuinCIS architecture is described as an architecture managing data quality in cooperative contexts, in order to avoid the spread of low-quality data and to exploit data replication for the improvement of the overall quality of cooperative data.

In this paper we will describe the design of a component of our system named as Quality Factory. The Quality Factory has the purpose of evaluating quality of XML data sources of the cooperative system. While the need for such a component had been previously identified, this paper first presents the design of the Quality Factory and proposes an overall

methodology to evaluate the quality of XML data sources.

Quality values measured by the Quality Factory are used by the *Data Quality Broker*. The Data Quality Broker has two main functionalities: (i) *quality brokering*, that allows users to select data in the CIS according to their quality; (ii) *quality improvement*, that diffuses best quality copies of data in the CIS.

As a further research contribution, this paper will focus on the design and implementation features of the Data Quality Broker as a Peer-to-Peer (P2P) system. More specifically, the Data Quality Broker is implemented as a peer-to-peer distributed service: each organization hosts a copy of the Data Quality Broker that interacts with other copies. While the functional specification of the Data Quality Broker is not a contribution of this paper, and has been presented in (Scannapieco et al., 2004; Mecella et al., 2003), its detailed design and implementation features as a P2P system are a novel contribution of this paper. Moreover, we will present some results from tests made to prove the effectiveness and efficiency of our system. The Data Quality Broker is implemented by a peer-to-peer architecture in order to be as less invasive as possible in introducing quality controls in a cooperative system. Indeed, cooperating organizations need to save their independency and autonomy requirements. Such requirements are well-guaranteed by the P2P paradigm which is able to support the cooperation without necessarily involving consistent re-engineering actions; in Section VII, we will better detail this point, comparing our choice with a system that instead does not adopt a P2P architecture.

The rest of this paper is organized as follows. Section II describes the main features of the Quality Factory and of the Data Quality Broker. Sections III presents the overall methodology and Sections IV details the architectural design of the Quality Factory, by focusing on the case of XML data sources. Section V describes the detailed design and implementation of the Data Quality Broker as a peer-to-peer

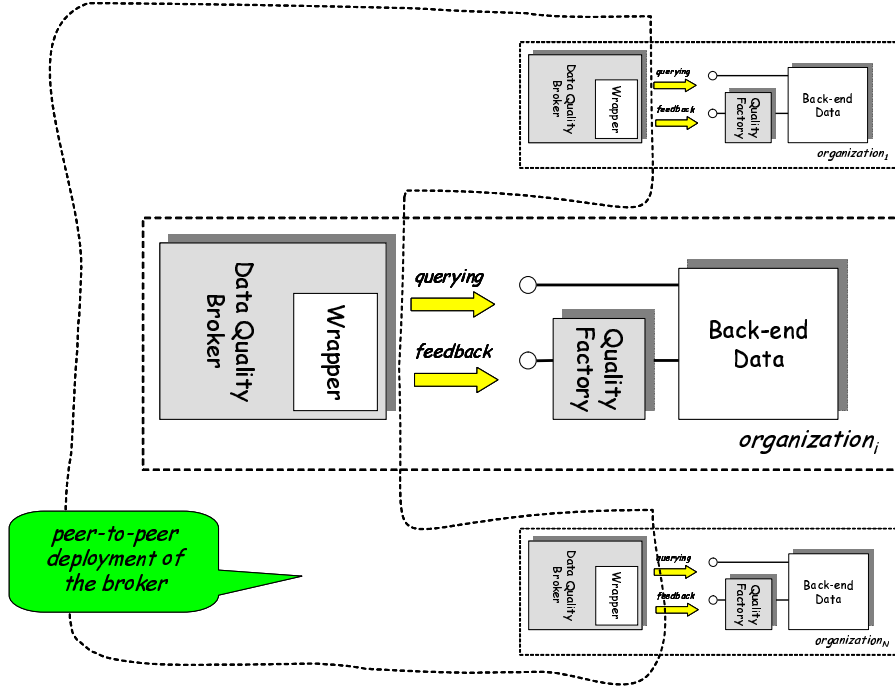


Figure 1. Interaction between the Data Quality Broker and the Quality Factory

system, and of each module of its component architecture. The set of performed experiments is described in Section VI. Finally, related work and conclusions are presented in Section VII and VIII respectively.

II - The Data Quality Broker and Quality Factory: Generalities

In this section, we provide an overview of the main functionality of the Data Quality Broker and we detail the interaction of such module with the Quality Factory, the design of which is provided in Sections III and IV. The component architecture and implementation details of the Data Quality Broker are instead described in Section V.

The Data Quality Broker Functionality

In the DaQuinCIS architecture, all cooperating organizations export their application data and quality data (i.e., data quality dimension values evaluated for the application data) according to a specific data model. The model for exporting data and quality data is referred to as *Data and Data Quality (D^2Q) model* (Mecella et al., 2003). The Data Quality Broker allows users to access data in the CIS according to their quality. Specifically, the Data Quality Broker performs two tasks, namely *query processing* and *quality improvement*.

The Data Quality Broker performs *query processing* according to a global-as-view (GAV) approach by unfolding queries posed over a global schema, i.e., replacing each atom of the original query with the corresponding view on local data sources (Ullman, 1997; Lenzerini, 2002). Both the

global schema and local schemas exported by cooperating organizations are expressed according to the D^2Q model. The specific way in which the mapping is defined stems from the idea of performing a quality improvement function during the query processing step. Global schema concepts are defined by means of queries over the local sources that retrieve *all* data present in the system that can populate such concepts. When retrieving results, data coming from different sources can be compared and a best quality copy can be constructed. Specifically, in our setting, data sources have distinct copies of the same data with different quality levels, i.e., there are *instance-level conflicts*. We resolve these conflicts at query execution time by relying on quality values associated to data: when a set of different copies of the same data are returned, we look at the associated quality values, and we select the copy to return as a result on the basis of such values. More details on the algorithm implemented for processing queries can be found in (Scannapieco et al., 2004). The best quality copy is also diffused to other organizations in the CIS as a *quality improvement feedback*.

Interaction between the Data Quality Broker and Quality Factory

The Quality Factory has the purpose of evaluating the quality of data stored by the cooperating sources. Such values will be used to populate the D^2Q model with the quality values associated to the integrated data. Therefore, the principal role of the Quality Factory is to measure such quality values in order to make them accessible by the Data Quality Broker. More specifically, at query time the Data Quality Broker accesses quality values in order to solve instance level

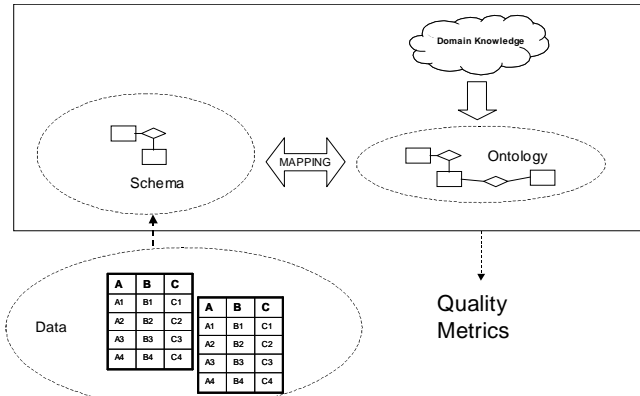


Figure 2. Methodology for quality evaluation

conflicts, and thus returning an answer to the user query. At query time, a further interaction may occur, as the Data Quality Broker can send better data and associated quality to specific organizations in the CIS, while performing the improvement functionality. In Figure 1, the interactions between the Data Quality Broker and the Quality Factory are shown. Notice also the P2P deployment of the Data Quality Broker that will be discussed in Section V.

III - The Data Quality Factory

The Quality Factory has the task of measuring the quality of the data that each organization makes available to the others. In this section we introduce the data quality evaluation methodology which is implemented by the Quality Factory. We then make some considerations about the architectural design of a Quality Factory module.

The definition of a data quality evaluation methodology is dependent on the data models used by organizations for their application data. As an example, the types of integrity constraints that are defined for the relational data model are of course different from the ones defined for a semistructured data model, like the XML data model.

In this paper, we focus on the case of data sources adopting the XML data model, showing how appropriate data quality measures can be devised in this case. We first start from a generic methodology, then we specify the methodology for XML data sources and we describe an example of definition of suitable metrics for quality evaluation.

A Data Quality Evaluation Methodology

The quality evaluation methodology is shown in Figure 2. The main idea is to measure data quality not by relying on the original schema of data sources, but through a comparison with a more constraining schema.

Indeed, many data quality problems are caused by the fact that data models and data management systems are often too weak at expressing several constraints that nonetheless exist in reality. On one hand, data management systems have the possibility to avoid some quality errors, such as the ones that can be introduced during data entry processes, or depend

on applications behavior. As an example, relational DBMSs can perform various checks at data entry or when certain operations are performed; XML documents can be validated against a DTD or other kind of schema, etc. On the other hand, database management systems are currently not able to enforce all the constraints that must hold on the data in order for them to be error-free and consistent; both relational DBMSs and XML data storage systems are examples of the missing enforcement of such a wide range of constraints.

This may be due to failure of data management systems to support enforcement of some constraints, or in limitations of the expressivity of the data models used to actually represent data, that fail to support some of the constraints holding on the domain even if they are known. Good design approaches exist for relational databases. Such approaches usually start with a requirement analysis. The result of this phase of the design is a conceptual model that tries to capture all the details concerning the domain involved, including any possible constraint that should be enforced in order to guarantee the consistency of the database during the lifetime of the application. All the domain and application specific knowledge that is necessary to run the application is usually formalized through a high level, expressive language like the Entity Relationship model. However, the final relational schema cannot enforce some of the constraints identified during this process.

Furthermore, missing constraints can also be due to poor schema design. In the case of XML data, for instance, conceptual XML design is still an open problem that only recently has received attention from the research community (e.g., (Conrad, Scheffner, & Freytag, 2000)).

The methodology we propose aims at identifying data quality problems that can be imputed to inconsistency with “constraints” that should hold but are not actively enforced on data. In order to evaluate data quality a comprehensive schema is first created. Such a schema is built by complementing the knowledge contained in the original data schema with knowledge representing the specific application domain, gathered through a domain analysis activity (e.g. performed by a domain expert).

The language used should be expressive enough to allow representation of more complex constraints than those already holding on the data to be analyzed. Beside being expressive, the modelling language used to represent such knowledge should be formal and have a machine processable format, in order to be used in an automated quality evaluation process, i.e. by the Quality Factory module. We call the resulting representation a *reference ontology*. In the next section we’ll give details about the modelling language used for the reference ontology.

In order to allow evaluation of data based on this “rich” representation, it must be related to the schema describing the data. The correspondence between the original data schema and the ontology is established by a *mapping*, as detailed in the next section.

The main advantage of this approach is that referring to this high-level, formalized representation of the reality of interest provides an homogeneous and effective way to define

metrics for quality evaluation.

Reference Ontology and Mapping

The reference ontology used in the above described methodology must be expressed in a language rich enough to model complex application domains and represent a wide range of constraints. Conceptual data models (Hull & King, 1987) like the Entity-Relationship model have been initially introduced to help the schema designer, and are capable of expressing rich constraints on the modelled reality. Lately, it has been shown that such models can be formalized through appropriate expressive description logics (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2003), thus making available for them the basic logical reasoning services characterizing such description languages. However, a detailed formal description of a full-fledged ontology language to be used for this task is outside the scope of this paper. Therefore, in the following we introduce a simplified language, whose features are indeed sufficient to describe our methodology and to show an example of its application (Section IV).

Reference Ontology In the following, we will denote a reference ontology with Σ .

Syntactically, Σ is a tuple $\langle C, Prop, R \rangle$ where:

- C is a set of *Concepts*
- For each $c \in C$, $Prop(c)$ denotes a set of named properties. We assume that on properties it is possible to express cardinality constraints that must be satisfied by instances of the concept. In particular, properties may be defined as optional or mandatory.
- R is a set of binary relationships of the form $\langle c, c' \rangle$ where c and c' are concepts in C .

For the relationships in R , we require that some constraints can be expressed. In particular, given a relationship $r = \langle c, c' \rangle$ in the ontology, we assume the ontology formalism allows:

- to specify cardinality constraints on both concepts c and c' ;
- to specify a *direction* for the relationship. A relationship $r = \langle c, c' \rangle$ on which a direction is defined is said to be a *parent-child relationship*. The concept c is said to have the *role of parent* and the concept c' is said to have the *role of child*;
- to specify a constraint over two properties p and p' , belonging respectively to $Prop(c)$ and $Prop(c')$, such that related instances of c and c' will have the same value for p and p' . A relationship on which this constraint holds is named as *join relationship*. If $r = \langle c, c' \rangle$ is a join relationship with an equality constraint over the properties p of c and p' of c' , we will also write $r = \langle c : p, c' : p' \rangle$.

Though we have introduced only binary relationships, a generalization to higher arity relationships is straightforward.

Beside creating the reference ontology, it is also necessary to establish a mapping between the original data schema and the ontology itself. This mapping links the original data

to the ontology, thus allowing to evaluate constraints holding on the ontology over the data populating the original schema. Starting from the ontology and the mapping, appropriate quality metrics can be defined. The language used to describe the reference ontology can be the same for each organization; instead, the mapping can be defined in several ways and is dependent on the particular data model and schema language used by each organization. Different mapping formalisms must be devised for example for the relational model, the various XML schema languages and so on.

In the following sections, we first illustrate a general architecture for the a Quality Factory module based on the proposed methodology. Then, we show how the general architecture of the Quality Factory can be tailored for a specific data model, namely the XML data model. We also introduce a specific mapping formalism to map from the DTD schema language to our ontology language, and we show how quality metrics appropriate for such data model can be defined. In particular, we provide an example of the definition of metrics related to the *completeness* quality dimension.

The possibility of defining these metrics is of particular interest, since quality metrics for XML data have not yet been devised. The main motivations for choosing XML are: (i) DTD, which a widely diffused XML schema formalism, is particularly weak at expressing some constraints that are essential to ensure the quality of XML data; (ii) while long-time established good design methodologies exist for the relational case, the problem of defining guidelines for XML schema design has been only recently addressed (Arenas & Libkin, 2004) and is far to be solved.

The approach described so far share some ideas with (Milano, Scannapieco, & Catarci, 2005), where the purpose was a different one, namely cleaning XML data. Instead, here we focus on the evaluation of XML data quality in the context of a comprehensive system for data quality management.

Generic Architecture of the Quality Factory

Some preliminary considerations justify the architecture. In order for a Quality Factory module described in this section. First, the organizations participating to a CIS might employ heterogeneous data models to store their data. Thus, appropriate quality metrics and quality evaluation strategies must be devised, that take into account such different models. Second, a Quality Factory not only has the duty to evaluate quality values, but also to manage such values and maintain a connection between such values and the data from which they were derived. After performing an evaluation of the quality of a data source, the Quality Factory stores the resulting quality values, making them available to a wrapper module which is responsible for presenting both data and quality trough the D^2Q data model. At this stage, however, quality values are not related to data by a D^2Q representation, as such representation is only built by the wrapper at query time. It is thus necessary to solve the problem of how to maintain a connection between the original data values and the evaluated quality values. We describe a possible solution to this problem for the XML case in Section IV.

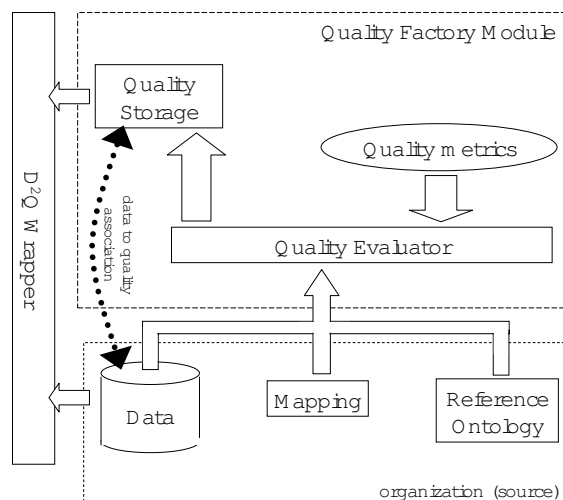


Figure 3. The logical architecture of the Quality Factory

An alternative solution allows the Quality Factory to interact directly with data already represented through the (data part of the) D^2Q model. This choice would indeed simplify the architecture of the system, as the Quality Factory would interact with a single data model and thus, being independent on data and schema heterogeneity, it could be used without changes within different organizations. Unfortunately this option has some serious drawbacks. First, as previously explained, organizations export their data in the D^2Q model simply by implementing a wrapper that allows to access the data in that format. This works coherently with the spirit of CISs, in which organizations cooperate preserving their independence, and can maintain their own data models. Data is never actually stored in the D^2Q model, but it is only translated to this model at query time. If the Quality Factory interacted directly with the model, it should query the data through the wrapper and then store the computed quality values, while preserving the links to original data values. This strategy imposes too strict constraints on the implementation of the wrapper. The choice of having the Quality Factory interact directly with the D^2Q model has a second important drawback, namely quality evaluation would be performed independently from the original data model used by organizations to store their own data. Instead, we believe that structural properties of the underlying data models should be taken into proper account when evaluating quality. If all data are translated to a single model *before* quality evaluation, much information on the data structures and on the constraints would be lost.

A generic logical architecture for a Quality Factory module is presented in Figure 3.

The figure shows the main logical components of the Quality Factory module, namely a Quality Storage, a Quality Evaluator and a set of Quality Metrics. The figure also shows how the Quality Factory interacts with the data stored at the source. The interaction is driven by a reference ontology and a mapping. The reference ontology and the mapping are built

by a domain expert and they are organization specific. The quality storage is (logically) linked to the data stored inside the organizations. The details of such connection depend on the specific way the data storage itself is designed. When accessing some source in order to export them in the D^2Q format, the wrapper module also accesses the quality storage and exploits this logical connection to retrieve the relevant quality values. The principal features of each module are briefly described in the following

Quality Storage. The quality storage is a logical component inside the Quality Factory that has the role of storing quality values evaluated by the quality evaluator component. From what described above, when designing a quality storage for a particular data model, the most important aspect is how to maintain the connection between the data values and the quality values. In this paper, we show a possible solution to this problem for an XML data source. A notable difference between the XML model and the relational one is that pieces of data must not only be identified with regard to their values, but also with regard to their position in the XML tree, as we describe in Section IV.

Quality Evaluator. The quality evaluator component has the role of actually accessing the data stored in the organization and assess their quality. This is done on the base of the quality metrics defined for the particular data model used within the source. Beside considering the specific metrics to be evaluated, the quality evaluator must also implement efficient algorithms to “visit” the data at the source. Furthermore it must be able to manage changes in the data due to updates. Whenever possible, incremental evaluation strategies should be devised, in order to avoid the need for a new assessment each time the underlying source changes.

Quality Metrics. As already remarked, quality metrics depend on the particular data model considered. The quality metrics are identified as input to the quality evaluator because it is highly desirable that such component is parametric with respect to the metrics.

IV - The Quality Factory for XML Data Sources

This section gives the architectural details of a Quality Factory module designed for an organization that stores its data as XML documents. As shown in Figure 3, the Quality Factory consists of various logical components. Such components can be better specified when referred to a specific data model, and in this section we specify them for the XML data model. Later in the section, we also describe an appropriate way of defining a mapping from the schema available for the original XML data to the *reference ontology*. Finally, an example of definition of metrics for the completeness data quality dimension is provided.

Quality Storage for XML Data

In order for the Quality Factory to store quality values, it is necessary to decide how quality values are linked to the

related data values.

Our solution consider XML data that satisfy some general requirements. First, if the document is not explicitly modified, then the order of its nodes does not change between two subsequent accesses to it. Second, the Quality Factory and the D^2Q wrapper can access the XML documents stored in a source directly, and not through a query language like XPath or XQuery. For example, the document-tree could be made accessible through a DOM interface. Finally, without loss of generality, we assume that the data at the sources is stored as a single logical XML document.

As we suppose that the XML documents are directly accessible, we can easily assign a unique identifier to each piece of data. This can be done, for instance, by associating to each node in the XML tree a label representing its number in a pre-order depth-first visit of the ordered XML tree. Another way is to describe the position of a node in the tree by means of *node addresses*, as proposed in (Buneman, Davidson, Fan, Hara, & Tan, 2001). Edges in an XML tree going from an element node to another element node or a text node can be labelled with the index of this subnode among the children of its parent. Thus, starting from the root, an element or text node can be uniquely identified from the concatenation of such indexes (e.g. 1#2#1#4) in the path that leads to that node. The XML model is a partially ordered tree, in that an order is not imposed among the attribute children of an element. However, attribute names are unique, and thus a unique attribute node address can be obtained by concatenating the node address of its parent and the name of the attribute itself (e.g. 1#2#2@name).

In order to store quality values, we use the following approach. Each node in the XML tree can be assigned a set of quality values, corresponding to values of quality dimensions; the considered quality dimensions are accuracy, completeness, consistency and currency. These values are stored in an XML tree, called *quality-tree*; we call *data-tree* the XML tree storing application data. The quality-tree conforms to the following rules:

- For each element node e of the data-tree, the quality-tree contains an element node q_e named after e and with the same address as e ;
- For each text node t in the data-tree, the quality tree contains an element node q_t having the same address as t , and named “text”;
- An element node in the quality tree contains four quality attributes that are used to store quality values related to the data-tree element or text node it represents. These attributes are named after the quality dimensions used in the system, i.e. *accuracy*, *completeness*, *consistency* and *currency*;
- For each attribute node a in the data-tree, the element node of the quality-tree which corresponds to the parent of a will contain four additional attributes whose names are obtained by concatenating the name of a with the names of the quality dimensions.

As an example, let us suppose that the data-tree contains the node:

```
<Xelem Xatt="...">...</Xelem/>
```

Then, the quality-tree will contain a node:

```
<Xelem accuracy= $\mathbf{v}_1$  completeness= $\mathbf{v}_2$ 
consistency= $\mathbf{v}_3$  currency= $\mathbf{v}_4$  XattAccuracy= $\mathbf{v}_5$ 
XattCompleteness= $\mathbf{v}_6$  XattConsistency= $\mathbf{v}_7$ 
XattCurrency= $\mathbf{v}_8$ >
...
</Xelem/>
```

where $\mathbf{v}_1, \dots, \mathbf{v}_8$ are appropriate quality values.

Given the unique address of a node, this data structure allows to retrieve its associated quality values. The choice of representing this data structure with an XML document is motivated by two reasons. First, as the structure is a tree, XML is particularly well suited to represent it. Second, as the wrapper used to transform source data from its original model into the D^2Q model must already manipulate XML data (those at the source itself), representing also the quality data in XML format simplifies the wrapper, allowing for the reuse of any XML manipulation facility already present in it.

Notice that the names of the element used to construct this tree are not significant, only their order is. Also notice that we are not making here any assumptions on the fact that a certain node is actually assigned a quality value. This depends on how quality metrics are defined, on the reference ontology and on the defined mapping.

Quality Evaluator for XML Data Sources

In order to evaluate the quality of an XML source, given the storage model described above, the quality evaluator module can perform a pre-order, left-to-right, depth-first visit of the data-tree, evaluate the quality at each node and construct a corresponding node in the quality-tree, with the following steps:

- Given the root of the data-tree r , construct the root q_r of the quality-tree as an element node having the same name as r . Then, for each quality dimension measured on r , add to q_r an attribute named after the quality dimension and having a value equal to the measured value. The addresses of these attribute nodes will be @accuracy, @completeness, @consistency, @currency;
- Given an element or text node d , let $addr(d) = addr(p)\#n$ be its address, where p is the parent of the node. This means that d is the n -th child of p . Let furthermore q_p be the node corresponding to p in the quality-tree. Construct an element node q_d corresponding to d as n -th child of q_p , that is with address $addr(q_d) = addr(q_p)\#n$. If d is an element node, then q_d will be named after d . Otherwise, the name of q_d will be text. Then, for each quality dimension measured on d , add to q_d an attribute named after that quality dimension and having value equal to the measured value. The addresses of such attribute nodes will be $addr(q_p)\#n@accuracy$, $addr(q_p)\#n@completeness$, $addr(q_p)\#n@consistency$, $addr(q_p)\#n@currency$;
- Given an attribute node a let $addr(a) = addr(p)@name$ be its address, where p is the element node containing a and $name$ is the name of a . Let q_p be the

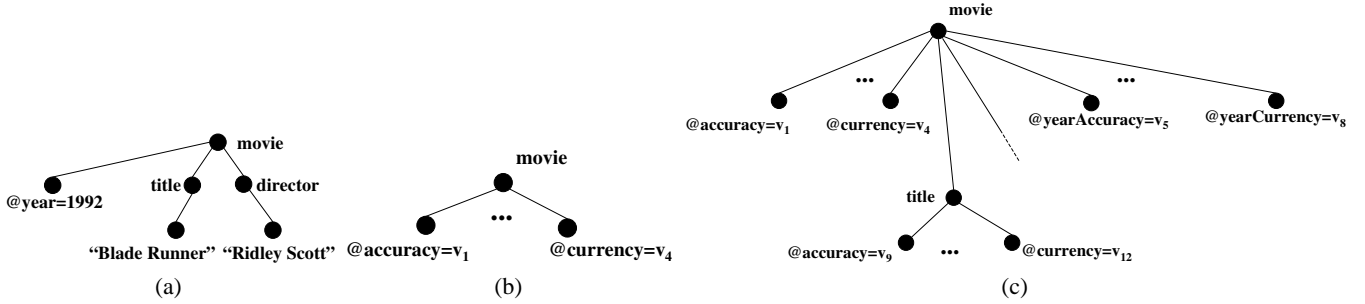


Figure 4. An example of construction of quality-tree nodes

node corresponding to p in the quality-tree. For each quality dimension measured on a , add to q_p an attribute whose name is constructed by concatenating the name of the quality dimension and the name of a , as described before. The value of these attributes will be set to the measured values. The addresses of the attributes nodes added will be:

$addr(q_p)\#n@nameAccuracy$,
 $addr(q_p)\#n@nameCompleteness$,
 $addr(q_p)\#n@nameConsistency$,
 $addr(q_p)\#n@nameCurrency$;

In this way, a new tree is constructed which has the same structure of the data-tree and contains the quality values measured for it. Figure 4 shows an example of how a (portion of) a quality tree is built starting from a data-tree. Figure 4(a) shows a simple data-tree. Attribute nodes have their names prefixed by a '@' symbol. Text nodes are depicted as nodes labelled with strings in double quotes. Figure 4(b) shows a first step in the construction of the quality-tree. An element with the same name of the data-tree root is built, and quality attributes are added. We show only two of such attributes to avoid visual cluttering. The values v_1, v_2 and so on are just placeholders for real quality values. In Figure 4(c), a bigger portion of the quality-tree has been built. Particularly, the tree contains attribute nodes corresponding to the "year" attribute in the data-tree, and nodes corresponding to the "title" node of the data-tree. A child of node "title" named "text" will be the next node to be created. The skewed line indicates that the quality node corresponding to node "director" will be created as second child of the node "movie" in the quality-tree, as it is the second child of the node "movie" in the data-tree.

This approach has the problem of maintaining the quality storage updated with regard to changes in the related XML data. We assume here that this task is performed by triggering a new quality evaluation each time the data is modified. Two issues must be considered. First, whenever a node is inserted, deleted or moved, the structural correspondence between the quality-tree and the data-tree might be partially or completely lost. Maintaining the alignment of the two trees only requires to add or delete a node in the quality-tree to reflect the changes in the data-tree. Second, when a node is inserted, deleted or moved and when text and attributes values are updated, this change might have consequences also on the quality values of nodes that don't take part to this trans-

formation, depending on how the quality metrics are defined and also on the given reference ontology and the mapping. We plan to address these problems in our future work.

Reference Ontology and Mapping for XML Data Sources

Before quality metrics can be introduced for the XML model, it is necessary to detail how a schema for this data model, such as a DTD, can be mapped to a reference ontology. When trying to establish a mapping between a DTD and a conceptual model, there is no generally adopted way to put in correspondence elements of an XML document with conceptual level constructs. Conceptual relationships might be represented in various ways, from simple nesting of elements to attribute-value based joins. XML elements can be used with different intended meaning, including to identify an object-type, a named relationship, or to represent a role name in a n-ary relationship.

In the following we make some assumptions to capture the case of a reasonable representation in which: (i) elements are put in correspondence with types; (ii) relationships are only established by means of nesting elements, namely *parent-child relationship*, and through value-based joins, considering both attribute values and text node values, namely *join relationships*. Let us note that the DTD formalism is expressive enough to capture some constraints over parent-child relationships, but cannot express almost any constraint over join-relationships.

The following definition of *restricted DTD* considers some structural limitations over the full generality of what a DTD can express. It is worthwhile to recall that DTDs were originally conceived to represent (textual) document structure and not data, thus some limitations, similar to those proposed here, occur very often when considering XML as a data model.

restricted DTD A *restricted DTD* is a tuple $D = \langle T_v, T_c, \tau_r, A, def, attlist, req \rangle$ where:

- T_v is a finite set of value-types;
- T_c is a finite set of complex-types;
- τ_r is a separate type called the root type;
- A is a finite set of attribute types;
- for each $\tau \in T_c \cup \{\tau_r\}$, $def(\tau)$ is a regular expression called the element type definition of τ . The language of the

regular expressions used for element type definitions is described by the following grammar:

$$\alpha ::= \tau_v \mid \tau_c \mid \alpha \mid \alpha \mid \alpha^* \mid \varepsilon$$

where ε denotes the empty content, $\tau_v \in T_v, \tau_c \in T_c$ and the symbols “|”, “ $\alpha\alpha$ ” and “ α^* ” denote union, concatenation and Kleene closure;

- for each $\tau \in T_c \cup \{\tau_r\}$, $attlist(\tau) \subseteq A$ is a set of attribute types.

Notice that in this simplified model we explicitly disallow mixed content, i.e. elements having both element and text children. Also, value-typed elements, that is elements containing only one text child, cannot have attributes. With these assumptions, it is quite straightforward to interpret elements containing a text child as representing “named values”, as it is for attributes.

Based on this simplified version of DTD, and on the ontology language previously introduced, we can introduce the following way of establishing mappings between schemas and reference ontologies.

Mapping Let $D = \langle T_v, T_c, \tau_r, A, def, attlist, req \rangle$ be a simplified DTD and $\Sigma = \langle D, C, Prop, R \rangle$ an ontology. We define a mapping M between D and Σ as a set of correspondences between types of D and elements of Σ such that:

- $\forall \tau \in T_c \cup \{\tau_r\}, M(\tau) = c \in C$
- $\forall \tau \in T_c, \forall \tau' \in T_v$ such that τ' appears in $def(\tau)$, if $M(\tau) = c$ then $M(\tau, \tau')$ is a property $p \in Prop(c)$;
- $\forall \tau \in T_c, \forall \tau' \in A$ such that $\tau' \in attlist(\tau)$, if $M(\tau) = c$ then $M(\tau, \tau')$ is a property $p \in Prop(c)$;

Notice that, given an ontology and a simplified DTD, multiple mappings could be established between them.

Defining Quality Metrics: the Case of Completeness

The Quality Factory evaluates the quality of the data inside the XML document following some *quality metrics* definitions. These metrics must only be defined once. They are specifically tailored for the XML data model, but they do not directly depend on the specific domain to which the data belongs (neither, of course, on the specific ontology which is used to describe such domain). The methodology we propose in this paper can be extended to take into account ad-hoc, domain-specific quality dimensions and metrics. This only requires that the Quality Factory module allows for the addition of other metrics defined over a generic reference-ontology apart from those common to all the quality factories for XML documents.

In previous sections we have formalized the concept of simplified DTD and XML document valid with respect to a simplified DTD. Furthermore, we have defined how to establish a mapping between a simplified DTD and a given ontology. In this section we show how to define quality metrics for XML documents based on a reference ontology and a mapping. Specifically, we describe an example of quality metrics definition focusing on a specific quality dimension,

namely *completeness*. Completeness is generically defined as “the extent to which data are of sufficient breadth, depth and scope for the task at hand” (Wang & Strong, 1996). We characterize completeness of XML data in a specific way by introducing a set of metrics that capture various forms of incompleteness of XML data. In the following definitions we consider a node n of type $\tau \in T_c$ and we consider $M(\tau) = c$ as the corresponding concept in the ontology Σ .

value-completeness Let l be a leaf node of type $\tau' \in T_v$ such that $l \in subel(\tau)$ and $M(\tau, \tau') = p \in Prop(c)$. If p is a mandatory property, the leaf l is said to be *value-complete* if $value(l) \neq \varepsilon$. Notice that leaves corresponding to non-mandatory properties are always considered to be value-complete.

leaf-completeness Let p be a mandatory property of c . The node n is said to be *leaf-complete* w.r.t. p if it has at least one leaf child l such that $M(\tau, type(l)) = p$. Let $P = \{p_1, \dots, p_n\} \subseteq Prop(c)$ be all the mandatory properties of c . The *degree of leaf-completeness* of n , written $\delta_l(n)$ is defined as the number of properties w.r.t. which n is leaf-complete divided by the cardinality of P .

parent-child completeness Let $r = \langle c, c' \rangle$ be a directed (parent-child) relationship to which c participates with cardinality $1 \dots *$ with the role of parent. We say that n is parent-child complete with respect to r iff \exists at least one children n' of n , such that $M(type(n')) = c'$.

Let now $R_{pc} = \{r_1, \dots, r_k\}$ be all the parent-child relationships to which c participates with cardinality $1 \dots *$. Let $C_{pc} = \{cr_1, \dots, cr_k\}$ be the concepts having role of child in the relationships of R_{pc} . The *degree of parent child completeness* of n , written $\delta_{pc}(n)$, is defined as the number of relationships in R_{pc} with respect to which n is parent-child complete, divided by the cardinality of R_{pc} . More formally, let us suppose that $\overline{R_{pc}} = \{r_1 = \langle c, c_{r_1} \rangle, \dots, r_s = \langle c, c_{r_s} \rangle\} \subseteq R_{pc}$ is the set of relationships such that $\forall r_i \in \overline{R_{pc}} \exists n_{r_i} \in subel(n)$ such that $M(type(n_{r_i})) = c_i$. Then:

$$\delta_{pc}(n) = |\overline{R_{pc}}| / |R_{pc}|$$

join completeness Let n be a node of type τ and $M(\tau) = c$ the corresponding concept in Σ . Let $r = \langle c : p, c' : p' \rangle$ be a join relationship to which c participates with cardinality $1 \dots *$. We say that n is *join-complete* with respect to r iff the following conditions hold:

- the node n has a leaf child l such that $M(\tau, type(l)) = p$ and l is leaf complete, that is $value(l) \neq \varepsilon$;
- there exist at least one node $n' \in N$ of type τ' such that $M(\tau') = c'$ and n' has a leaf child l' such that $M(\tau', type(l')) = p'$ and l' is leaf complete, that is $value(l') \neq \varepsilon$;
- $value(l) = value(l')$.

Let now $R_j = \{r_1 = \langle c : p_1, c_{r_1} : p_{r_1} \rangle, \dots, r_k = \langle c : p_k, c_{r_k} : p_{r_k} \rangle\}$ be all the join relationships to which c participates

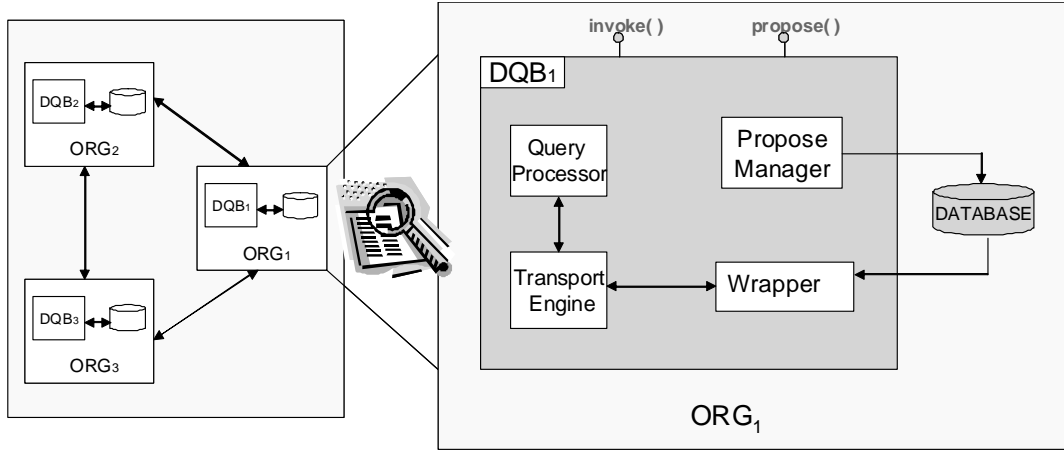


Figure 5. The Data Quality Broker as a P2P system and its internal architecture

with cardinality $1 \dots *$. The *degree of join completeness* of n , written $\delta_j(n)$ is defined as the number of relationships in R_j with respect to which n is join complete, divided by the cardinality of R_j . More formally, let us suppose that $\overline{R}_j = \{r_1, \dots, r_s\} \subseteq R_j$ is the set of relationships such that $\forall r_i \in \overline{R}_j$ n is join complete w.r.t. r_i . Then:

$$\delta_j(n) = |\overline{R}_j|/|R_j|$$

r-completeness Given a node n , let R_{pc} and R_j be respectively the set of parent-child and join relationships to which n participates with minimum cardinality one, and $\overline{R}_{pc} \subseteq R_{pc}, \overline{R}_j \subseteq R_j$ the above defined sets of relations with respect to which n is parent-child complete and join-complete. Then the *degree of parent-child completeness* of n , written $\delta_r(n)$, is defined as:

$$\delta_r(n) = |\overline{R}_j \cup \overline{R}_{pc}|/|R_{pc} \cup R_j|$$

All the definitions provided above have the purpose of showing that quality metrics can be defined on the basis of the reference ontology and the mapping with the original schema (a restricted DTD in our case) according to the quality evaluation methodology described in Section III.

V - The Data Quality Broker

In Section II, we have described the main functionality of the Data Quality Broker that allows query processing and quality improvement in cooperative systems. In this section we provide the detailed design and implementation of this component.

The Data Quality Broker is implemented as a peer-to-peer distributed service: each organization hosts a copy of the Data Quality Broker that interacts with other copies (see Figure 5, left side). Each copy of the Data Quality Broker is internally composed by four interacting modules (see Figure 5, right side). The modules Query Processor and Transport Engine are general and can be installed without modifications in each organization. We have implemented both the

Query Processor and the Transport Engine; details on their implementation will be provided in the next sections.

The Wrapper has to be customized for the specific data management system and translates the query from the language used by the broker to that of the specific data source; it is a read-only module that accesses data and associated quality stored inside organizations without modifying them.

The Propose Manager receives feedbacks sent to organizations in order to improve their data. This module can be customized by each organization according to the policy internally chosen for quality improvement. As an example, if an organization chooses to trust quality improvement feedbacks, an automatic update of databases can be performed on the basis of the better data provided by improvement notifications.

The Query Processor is responsible for query execution. The copy of the Query Processor local to the user query, receives the query and splits it into queries local to the sources, on the basis of the defined GAV mapping. Then, the local Query Processor also interacts with the local Transport Engine in order to send local queries to other copies of the Query Processor and receive the answers.

The Transport Engine provides general connectivity among all Data Quality Broker instances in the CIS. Copies of the Transport Engine interact with each other in two different scenarios:

- Query execution: the requesting Transport Engine sends a query to the local Transport Engine of the target data source by executing the `invoke()` operation (see 5, right side) and asynchronously collects the answers.
- Quality feedback: when a requesting Transport Engine has selected the best quality result of a query, it contacts the local Transport Engines to enact quality feedback propagation. The `propose()` operation (see Figure 5, right side) is executed as a callback on each organization, with the best quality selected data as a parameter. The `propose()` can be differently implemented by each organization: a remote Transport Engine simply invokes this operation.

Another function performed by the Transport Engine is the evaluation of the *availability* of data sources that are go-

ing to be queried for data. This feature is encapsulated into the Transport Engine as it can be easily implemented exploiting Transport Engine's communication capabilities.

The Data Quality Broker has been implemented by web services technologies. To implement web services, we have chosen the J2EE 1.4 Java Platform, specifically the Java API for XML-based Remote Procedure Call (JAX-RPC) (JSR-101-Expert-Group, 2003). In JAX-RPC, request/response of remote methods is performed through the exchange of SOAP messages over an HTTP connection.

The implementation of the Query Processor and of the Transport Engine are better detailed in the next sections.

Query Processor: Design and Implementation Issues

The Query Processor module of the Data Quality Broker implements the *mediation* function of a data integration architecture (Wiederhold, 1992). It performs query processing according to a GAV approach, by unfolding queries posed over a global schema. Both the global schema and local schemas exported by cooperating organizations are expressed according to the D^2Q model. The D^2Q model is a semistructured model that enhances the semantics of the XML data model (Fernandez, Malhotra, Marsh, Nagy, & Walshand, 2002) in order to represent quality data. The schemas and instances of the D^2Q model are almost directly translated respectively into XML Schemas and XML documents. Such XML-based representations are then easily and intuitively queried with the XQuery language (Boag et al., 2003). The unfolding of an XQuery query issued on the global schema can be performed on the basis of well-defined mappings with local sources. The exact definition of the mapping is described in (Milano, Scannapieco, & Catarci, 2004).

Query Processing Steps

Query processing is performed according to the sequence of steps described in Figure 6.

The entire process may be logically divided into two phases: an *Unfolding* phase, which involves a global query and produces a set of sub-queries to be sent to local organizations, and a *Refolding* phase, which collects the results of local sub-queries execution, rewrites the global query and finally executes the global query. In the following, we briefly revise the steps of these two phases.

The Unfolding phase starts by receiving a global query and analyzing it in order to extract those path expressions that access data from the integrated view. Only these parts of the query are actually translated and sent to wrappers for evaluation. During the path expression extraction phase, the Query Processor looks for path expressions. The extraction is straightforward most of the times¹. The result of the path expression extraction phase is a number of identified path expressions that need to be translated. Before the translation phase, they are submitted to a preprocessing step.

The preprocessing step decomposes each path expression into a set of path expressions whose concatenation produces

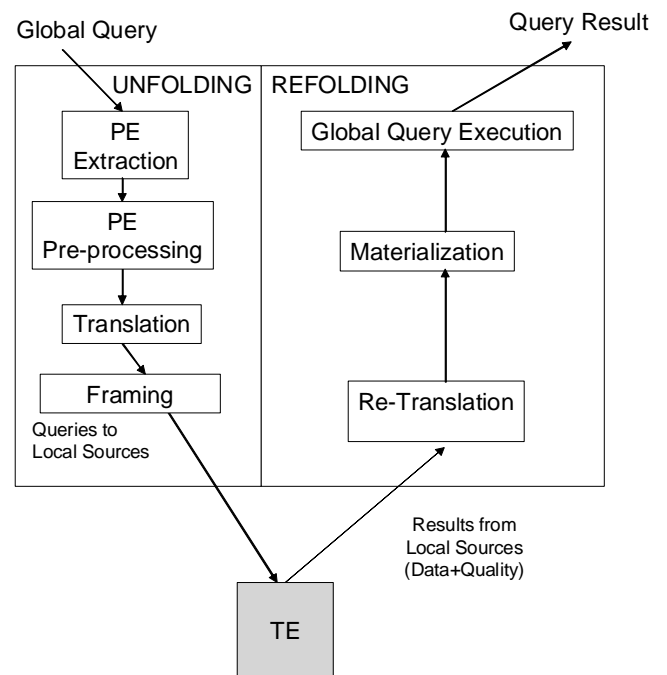


Figure 6. Sequence of steps followed in the query processing phase

a result equivalent to that of the original expression. The elements of this set are still expressed over the global schema alphabet, and are therefore translated into local organizations alphabets, according to the mapping specification.

After translation, sub-queries are ready to be executed at local sources. A further preliminary step is needed to make possible to re-translate their results. Usually, results of a query contain nodes and their descendants. Any information regarding their ancestors is lost. We adopt a framing mechanism in order to keep trace of ancestors and thus simplifying the retranslation phase. After retranslation, framing elements may be discarded and result fragments may be safely concatenated to form a single document.

After all the steps of the Unfolding phase have been completed, sub-queries may be passed to a Transport Engine module, which is in charge of redirecting them to local sources for execution and to subsequently collect results.

The Refolding phase starts with a step in which the received results are re-translated according to the global schema specification. Results coming from different organizations answering the same global path expression are then concatenated into a single temporary file.

Each occurrence of a path expression previously extracted from the global query is replaced with a special path expres-

¹ In some cases, a nested expression may contain direct or indirect references to data in the global view. Such cases must be handled in a slightly different way. Our current approach is to split any path expression containing a problematic step and to treat the two parts separately. Specifically, when reverse steps are involved, they must be taken into account to perform the splitting properly.

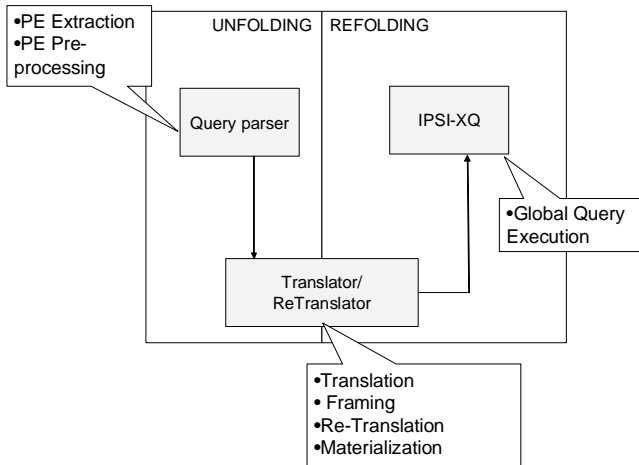


Figure 7. Internal Modules of the Query Processor

sion that accesses one of the temporary files built during the previous step. In this way, the global query is changed into a query that only uses local files, and can then be executed. The execution of a query produces a result that may contain duplicate copies of the same objects coming from different sources. For each object, a best quality representative must be chosen or constructed. For this purpose, results undergo a record matching phase that identifies semantically equivalent objects and groups them into clusters. Copies in each cluster are compared and a best quality object is either selected or constructed; more details on this process can be found in (Scannapieco et al., 2004). Finally, the results best fitting with the user query requirements are sent back to the user. Moreover quality feedbacks are sent to the Transport Engine that is in charge of propagating them throughout the system.

The Query Processor has been implemented as a Java application. Figure 7 shows the main components; the phases of query processing that are executed by each component module are also represented.

The *Query Parser* performs the first query processing steps. To implement it, a parser for the XQuery language has been generated with the help of the JavaCC tools. The *Translation/Retranslator* module is in charge of the translation and retranslation of queries and their results. For query execution a third-party query engine may be used. The engine used in our implementation is *IPSI-XQ* (*IPSI-XQ*, n.d.). Let us note that we made *IPSI-XQ* quality-aware by adding some *quality functions* to it. These functions are written in XQuery, and allow to access quality data; they are simply added to the query prolog of each query submitted to the engine.

Transport Engine: Design and Implementation Issues

The Transport Engine component of the Data Quality Broker provides the connectivity and communication infrastructure of the DaQuinCIS system. In Figure 8 the internal components of the TE are shown; the sequence of interactions among such modules is also depicted. The *Availability Tester*

module works in background continuously executing connectivity tests with servers from other organizations. It executes a ping function on the servers in the cooperative system opening HTTP connections on them. The *Transport Engine Interface* is the module that interfaces the Query Processor and the Transport Engine. Specifically, it uses a data structure to store queries and query results, once the latter have been gathered from each organization. The data structure is organized as an array: each element is representative of a single query execution plan and is composed by a list of queries that are specific of such a plan. Such queries are passed by the Query Processor (step 1). Then, the *Transport Engine Interface* activates the *Execute-Query* module with plans as input parameters (step 2). The *Execute-Query* interacts with the *Availability Tester* module that performs an availability check of the sources involved in the query execution (step 3). Then, the *Execute-Query* activates the *Web Service Invoker* module that carries out the calls to the involved organizations (step 4). The call is performed in an asynchronous way by means of suitable proxy SOAP client. Before invoking data management web services, an availability check is performed by the *Availability Tester* module. When the result of the different plans are sent back, the *Execute-Query* module stores them in a specific data structure and gives it to the *Transport Engine Interface* (step 5) that, in turn, gives it back to the Query Processor (step 6). The data structure is very similar to the input one; the main difference is the substitution of the query field with a special record containing data and associated quality provided as query answers.

Notice that the same interaction among modules shown in Figure 8 occurs when quality feedbacks need to be propagated. The Query Processor selects the best quality copies among the ones provided as query answers and then sends back the result to the *Transport Engine Interface* that activates the *Execute-Query* module with the best quality copies and the organizations to be notified about them as input parameters. The best quality copies are then sent by the *Web Service Invoker*. On the receiver organization side, the

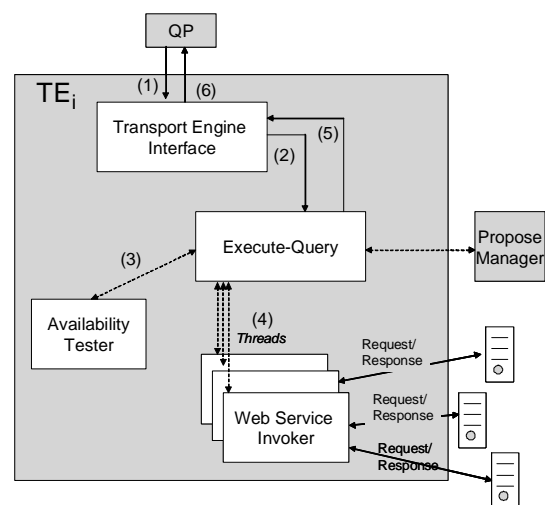


Figure 8. Internal Modules of the TE of organization i

Execute-Query module notifies the *Propose Manager* modules of involved organizations about the better quality data available in the system, thus implementing the quality feedback functionality that the Data Quality Broker provides at query processing time. Notice also that the *Execute-Query* module, on the sender organization side, also interacts with the *Availability Tester* modules: this makes quality notification not to be performed in a one-step process. Instead, a transaction starts that commits only when the set of sources that has to be notified is exhausted.

VI - Experiments

In this Section, we first show the experimental methodology, then we show quality improvement experiments and performance experiments.

Experimental Methodology

We perform a set of experiments in order to test the quality improvement functionality of the Data Quality Broker and its performance features. We used two real data sets, each owned by an Italian public administration agency, namely:

- the first data set is owned by the Italian Social Security Agency, referred to as INPS (in Italian, Istituto Nazionale Previdenza Sociale). The size of the database is approximately 1.5 millions of records;
- the second data set is owned by the Chambers of Commerce, referred to as CoC (in Italian, Camere di Commercio). The size of the database is approximately 8 millions of records.

Some data are agency-specific information about businesses (e.g., employees social insurance taxes, tax reports, balance sheets), whereas others are common to both agencies. Common items include one or more identifiers, headquarter and branches addresses, legal form, main economic activity, number of employees and contractors, information about the owners or partners.

As far as quality improvement experiments, we have associated quality values to the INPS and CoC databases. Specifically, we have associated completeness and currency quality values to each field value. Completeness refers to the presence of a value for a mandatory field. As far as currency values, timestamps were already associated to data values in the two databases; such timestamps refer to the last date when data were reported as current. We have calculated the degree of overlapping of the two databases that is equal to about 970000 records.

As far as performance experiments, a P2P environment has been simulated. Each data source has been wrapped by a web service; such web services have been deployed on different computers connected by a LAN at 100 Mbps and interacting each other using the SOAP protocol.

Quality Improvement Experiments

The experimental setting consists of the two described real data bases plus a third source that has the purpose of querying the first two sources and cooperates with them. We have

considered how this CIS behaves with regards to the quality of its data, in two specific cases. In the first case, a “standard” system is analyzed; this system does not perform any quality based check or improvement action. In the second case, the CIS uses the Data Quality Broker functionality of query answering and quality improving.

Values for the frequency of queries and updates on the data bases and average query result size are derived from real use cases. We have estimated the frequency of changes in tuples stored in the two databases to be around 5000 tuples per week. Average query frequency and query result size are, respectively, of 3000 queries per week and 5000 tuples per query. In a real setting, updates are distributed over a week. Anyway, to simplify our experimental setting, we have chosen to limit updates to the beginning of each week.

We consider how the quality of the entire CIS changes throughout a period of five weeks. Note that such variations are due to both updates on the databases and exchanges of data between them. In the standard system, these exchanges are only due to queries. With the Data Quality Broker, each time a query is performed, an improvement feedback may be propagated. For both the Data Quality Broker and the standard system, we calculate the overall *Quality* of the system, as the percentage of the high quality tuples in the system. We adopt simplified quality metrics by considering that a tuple has high quality if it is complete and current on all its fields. Conversely, a tuple has low quality if it is not complete and/or current on some fields.

To clarify how the two systems reacts to updates, we have considered an update set composed by both high quality and bad quality tuples equally distributed.

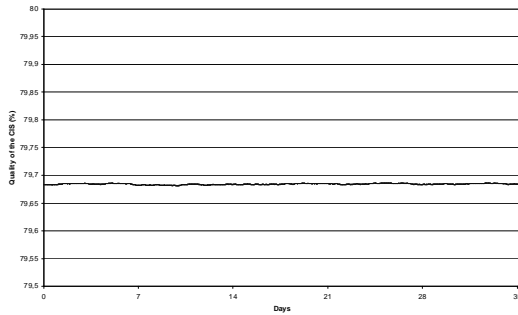
In Figure 9, the behaviors of the Data Quality Broker and the standard system with respect to quality improvement are shown. In the standard system (Figure 9.a), the overall quality is roughly constant, due to the same number of high quality and low quality tuples spread in the system. Instead, with the Data Quality Broker (Figure 9.b), the improvement of quality in each period is enhanced by data quality feedbacks performed by the system and low quality data are prevented to spread. This causes a growing trend of the Data Quality Broker curve, in spite of low quality inserted tuples. The actual improvement is about 0.12%; given that the size of the two databases is about 9.500.000 tuples, the improvement consists of about 11.500 tuples.

Performance Experiments

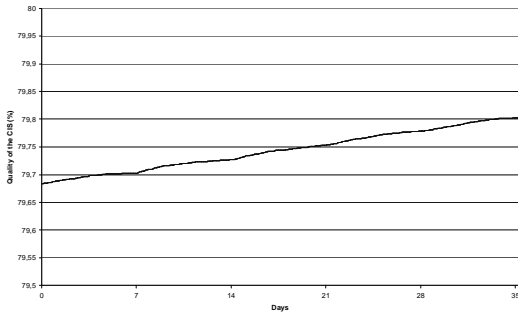
For the performance set of experiments, we have considered the Data Quality Broker and the standard system behavior with fictitious sources, in order to vary some parameters influencing performance experiments.

The first performance experiment shows the time overhead of the Data Quality Broker system with respect to the standard system. In such experiment we draw a *normalized transaction time* defined by the fraction:

$$\frac{\text{DataQualityBrokerElaborationTime} - \text{StandardElaborationTime}}{\text{StandardElaborationTime}}$$



(a) Data quality improvement in the standard system



(b) Data quality improvement with the Data Quality Broker

Figure 9. Data quality improvement in the standard system and with the Data Quality Broker

The elaboration time is the time required by the system for processing a query. The normalized transaction time is drawn when varying the degree of overlapping of data sources. The overlapping degree significantly influences the Data Quality Broker. Indeed, the Data Quality Broker accomplishes its functionalities in contexts where data sources overlap and such an overlapping can be exploited to improve the quality of data. The Figure 10 (top) shows how the normalized transaction time varies in dependence on the percentage of data sources overlapping with two fixed query result sizes, namely $q_1=1000$ tuples, $q_2=5000$ tuples. The number of overlapping sources is fixed to 3. This means that once a query is posed over the system, three sources have data that can be provided as answer to the query, though the system can have a larger number of sources. The Figure 10 shows the actual time overhead of the Data Quality Broker systems with respect to a standard system. The Data Quality Broker system has an acceptable time overhead. The worst depicted case is for the query result size $q_2=5000$ and a percentage of overlapping equal to 40%; in such a case, there is a 50% time overhead with respect to the standard system.

The second performance experiment shows the normalized transaction time with query size varying (see Figure 10 bottom). For a fixed degree of overlapping equals to 15%, we draw the normalized transaction time for three different numbers of overlapping organizations, namely $n_1=3$, $n_2=4$ and $n_3=5$. This experiment shows the behavior of the Data Quality Broker when increasing the number of organizations

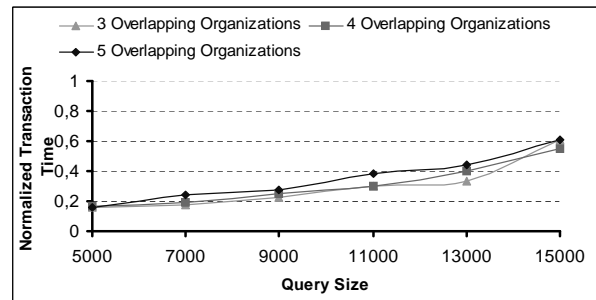
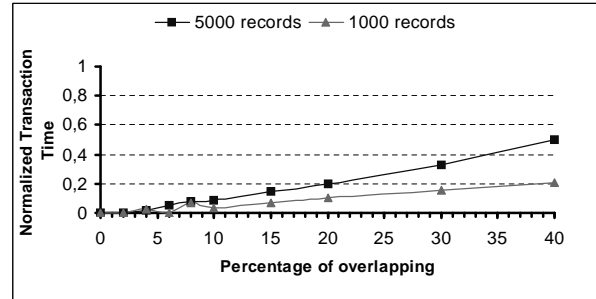


Figure 10. Normalized transaction time wrt percentage of overlapping data sources (top) and normalized transaction time wrt query sizes (bottom)

and the size of queries. Specifically, the normalized transaction time increases slowly with an almost linear trend. The positive result shown in Figure 10 is that when the number of overlapping data sources increases, the trend does not substantially change.

VII - Related Work

The Quality Factory deals with the problem of measuring quality of data. Data quality is typically characterized by a set of dimensions, for which various definitions have been proposed, including (Wang, 1998), (M. Bovee and Srivastava, R.P. and Mak, B.R., 2001) and (Liu & Chi, 2002). In (Scannapieco & Batini, 2004), a set of metrics for characterizing completeness in the relational model are described, while in (Naumann, Freytag, & Leser, 2004) completeness of sources in data integration settings is evaluated. Such definitions do not regard XML data. The problem of considering the quality of an XML document is considered by the proposal of a normal form for XML (Arenas & Libkin, 2004), and by new more expressive data models that better allow XML queries specification and execution (Jagadish, Lakshmanan, Scannapieco, Srivastava, & Wiwatwattana, 2004.). In this paper we have instead described an original methodology for evaluating the quality of XML data sources, laying the foundations for a full characterization of the quality of XML data.

Quality-aware querying, performed by the Data Quality Broker, is a problem explicitly addressed in a few works. In (Naumann, Leser, & Freytag, 1999), an algorithm for querying for best quality data in a LAV integration system is proposed. We share with such a work the idea of querying for

best quality data; however, the main difference is the semantics of our system: our aim is not only querying, but also improving quality of data. To such a scope, the query processing step has a specific semantics that allows to perform quality improvement on query results.

The MIT Context Interchange project (COIN) (Bressan et al., 1997) is based on the idea of modeling a “context” for integrating heterogeneous sources. Such a context consists of metadata that allows for solving problems, such as instance level conflicts that may occur in the data integration phase. The Data Quality Broker differs mainly for considering a much more general and explicit way of representing quality of data. Instead, the COIN approach focuses only on one aspect of data quality, namely *data interpretability*.

In (Mihaila, Raschid, & Vidal, 2000), the basic idea is querying web data sources by selecting them on the basis of quality values on provided data. Specifically, the authors suggest to publish metadata characterizing the quality of data at the sources. Such metadata are used for ranking sources, and a language to select sources is also proposed. In the Data Quality Broker system, we associate quality to data (at different granularity levels) rather than to a source as a whole. This makes things more difficult, but allows to pose more specific queries.

As an e-Government initiative, the Italian Public Administration in 1999 started a project, called “Services to Businesses”, which involved extensive data reconciliation and cleaning (Bertoletti, Missier, Scannapieco, Aimetti, & Batini, 2005). The approach followed in this project consisted of three different steps: (i) linking *once* the databases of three major Italian public administrations, by performing a record matching process; (ii) correcting matching pairs and (iii) maintaining such status of aligned records in the three databases by centralizing record updates and insertions only on one of the three databases. This required a substantial re-engineering of administrative processes, with high costs and many internal changes for each single administration. Differently from the approach adopted in the “Services to Businesses” project, the Data Quality Broker is implemented in a completely distributed way through a P2P architecture, thus avoiding bottlenecks on a single cooperating organization. Even more important, no kind of re-engineering actions need to be engaged when choosing to use the Data Quality Broker, as query answering and quality improvement can be performed with a very low impact in terms of changes on cooperating organizations.

VIII - Concluding Remarks

This paper provides two major contributions. First, it describes the issues related to the implementation of a Quality Factory in cooperative information systems, where a Quality Factory has the purpose of evaluating the quality provided by each cooperating organization. A general methodology for designing a Quality Factory is proposed and the design of a specific Quality Factory for XML data is described. Second, it provides the implementation details of the Data Quality Broker module, responsible for data and quality exchanges

in CISs.

The Data Quality Broker has been implemented in this paper as a peer-to-peer system. Specifically, we have described the detailed design and implementation of two modules composing the Data Quality Broker, namely the Query Processor and the Transport Engine. We have also described some experiments that validate our approach with respect to quality improvement effectiveness. Such experiments show that the Data Quality Broker succeeds in controlling and improving quality of data in a CIS. Moreover, when compared to a standard system, i.e. a system with no quality management features, the Data Quality Broker exhibits a limited performance degradation. Such a performance degradation is not a serious problem in specific scenarios, such as e-Government, in which the quality of data is the main enabling issue for service provisioning. Indeed, we remark that such scenarios are the reference ones for our system. Future works will address two main lines: (i) the Quality Factory will be extended with further metrics for quality measuring of XML data: besides completeness, we aim to investigate also metrics for accuracy and consistency; (ii) the Data Quality Broker will be more extensively validated, in particular by pushing the adoption of the proposed P2P system in some Italian e-Government pilot initiatives.

References

- Arenas, M., & Libkin, L. (2004). A normal form for XML documents. *ACM Trans. Database Syst.*, 29.
- Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., & Patel-Schneider, P.F. (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press
- Bertoletti, M., Missier, P., Scannapieco, M., Aimetti, P., & Batini, C. (2005). Improving Government-to-Business Relationships through Data Reconciliation and Process Re-engineering. In R. Wang (Ed.), *Advances in management information systems-information quality monograph (amis-iq) monograph*. Sharpe, M.E.
- Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., & Simèon, J. (2003, November). *XQuery 1.0: An XML Query Language*. W3C Working Draft. Available from <http://www.w3.org/TR/xquery>.
- Bouzeghoub, M., & Lenzerini, M. (2001). Special Issue on Data Extraction, Cleaning, and Reconciliation. *Information Systems*, 26(8).
- Bressan, S., Goh, C., Fynn, K., Jakobisiak, M., Hussein, K., Kon, K., et al. (1997). The COntext INterchange Mediator Prototype. In *Proceedings acm sigmod international conference on management of data (sigmod 1997)*.
- Buneman, P., Davidson, S., Fan, W., Hara, C., & Tan, W. (2001). Keys for XML. In *Proceedings of www 2001*.
- Conrad, R., Scheffner, D., & Freytag, J. (2000). Xml conceptual modeling using uml. In *19th international conference on conceptual modeling*.
- Fernandez, M., Malhotra, A., Marsh, J., Nagy, M., & Walshand, N. (2002, November). *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft. Available from <http://www.w3.org/TR/query-datamodel>.

- Hull, R., & King, R. (1987). Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3), 201-260.
- IPSI-XQ. (n.d.). Available from http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html.
- Jagadish, H., Lakshmanan, L., Scannapieco, M., Srivastava, D., & Wiwatwattana, N. (2004). Colorful XML: One Hierarchy Isn't Enough. In *Proceedings of the 2004 acm sigmod conference (sigmod 2004)*.
- JSR-101-Expert-Group. (2003, October). *Java(tm) api for xml-based remote procedure call (jax-rpc) specification version 1.1*. Sun Microsystems, Inc.
- Lenzerini, M. (2002). Data Integration: A Theoretical Perspective. In *Proceedings of the 21st acm symposium on principles of database systems (pods 2002)*.
- Liu, L., & Chi, L. (2002). Evolutionary Data Quality. In *7th international conference on information quality*.
- M. Bovee and Srivastava, R.P. and Mak, B.R. (2001). A Conceptual Framework and Belief-Function Approach to Assessing Overall Information Quality. In *Proceedings of the 6th international conference on information quality*.
- Mecella, M., Scannapieco, M., Virgillito, A., Baldoni, R., Catarci, T., & Batini, C. (2003). The DaQuinCIS Broker: Querying Data and their Quality in Cooperative Information Systems. *Journal of Data Semantics*, 1(1). Shorter version also appeared in CoopIS 2002.
- Mihaila, G., Raschid, L., & Vidal, M. (2000). Using Quality of Data Metadata for Source Selection and Ranking. In *Proceedings of the 3rd international workshop on the web and databases (webdb'00)*.
- Milano, D., Scannapieco, M., & Catarci, T. (2004). Quality-driven Query Processing of XQuery Queries. In *Proc. of the international workshop on data and information quality (diq 2004)*.
- Milano, D., Scannapieco, M., & Catarci, T. (2005). Using Ontologies for XML Data Cleaning. In *Second INTEROP dissemination workshop*.
- Mylopoulos, J., & Papazoglou, M. (1997). Cooperative Information Systems (Special Issue). *IEEE Expert Intelligent Systems & Their Applications*, 12(5).
- Naumann, F., Freytag, J., & Leser, U. (2004). Completeness of integrated information sources. *Information Systems*, 29(7).
- Naumann, F., Leser, U., & Freytag, J. (1999). Quality-driven Integration of Heterogenous Information Systems. In *Proceedings of 25th international conference on very large data bases (vldb'99)*.
- Scannapieco, M., & Batini, C. (2004). Completeness in the Relational Model: A Comprehensive Framework. In *9th international conference on information quality*.
- Scannapieco, M., Virgillito, A., Marchetti, M., Mecella, M., & Baldoni, R. (2004). The DaQuinCIS architecture: a Platform for Exchanging and Improving Data Quality in Cooperative Information Systems. *Information Systems*, 29(7).
- Ullman, J. (1997). Information Integration Using Logical Views. In *Proceedings of the 6th international conference on database theory (icdt '97)*.
- Wang, R. (1998). A Product Perspective on Total Data Quality Management. *Communications of the ACM*, 41(2).
- Wang, R., & Strong, D. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4).
- Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3).