Incorporating Domain-Specific Information Quality Constraints into Database Queries

SUZANNE M. EMBURY, PAOLO MISSIER, SANDRA SAMPAIO, R. MARK GREENWOOD
University of Manchester
and
ALUN D. PREECE
Cardiff University

The range of information now available in queryable repositories opens up a host of possibilities for new and valuable forms of data analysis. Database query languages such as SQL and XQuery offer a concise and high-level means by which such analyses can be implemented, facilitating the extraction of relevant data subsets into either generic or bespoke data analysis environments. Unfortunately, the quality of data in these repositories is often highly variable. The data is still useful, but only if the consumer is aware of the data quality problems and can work around them. Standard query languages offer little support for this aspect of data management. In principle, however, it should be possible to embed constraints describing the consumer's data quality requirements into the query directly, so that the query evaluator can take over responsibility for enforcing them during query processing.

Most previous attempts to incorporate information quality constraints into database queries have been based around a small number of highly generic quality measures, which are defined and computed by the information provider. This is a useful approach in some application areas but, in practice, quality criteria are more commonly determined by the user of the information not by the provider. In this paper, we explore an approach to incorporating quality constraints into database queries where the definition of quality is set by the user and not the provider of the information. Our approach is based around the concept of a *quality view*, a configurable quality assessment component into which domain-specific notions of quality can be embedded. We examine how quality views can be incorporated into XQuery, and draw from this the language features that are required in general to embed quality views into any query language. We also propose some syntactic sugar on top of XQuery to simplify the process of querying with quality constraints.

Categories and Subject Descriptors: H2.3 [**Database Management**]: Languages—*Query Languages*

General Terms: Languages

Additional Key Words and Phrases: Information quality, database query languages, XQuery, views

Corresponding author's address: S. Embury, School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK.
The Qurator project was supported by a grant from the EPSRC.

1. INTRODUCTION

The modern data consumer is both imaginative and resourceful. New data sources, and new uses for existing data, appear far more rapidly than bespoke analysis tools can be created. Generic data analysis tools, such as spreadsheets, can be used by skilled data consumers, but in many cases information management staff must extract the relevant subsets of the data, in formats suitable for interpretation by consumers with domain rather than technical expertise. Declarative query languages are a valuable support tool for technical personnel in this role, allowing rapid prototyping of reports for *ad hoc* or novel analyses and easy export of data to generic analysis tools. Queries can also be embedded into programming languages, to offer a concise and high-level substrate for the implementation of bespoke analysis tools, once the need (and justification) for them becomes apparent.

Unfortunately, these same circumstances (new types of data and new applications for existing data) are exactly those in which problems with the quality of data, and therefore the quality of query results, are likely to arise [Blaha 2001]. Missing or inaccurate data, out of date or imprecise information will all propagate through queries to produce results that are challenging to interpret, reducing the value of the new analysis tool. When resources are available and the importance of the analysis justifies the expense, data sets can be cleaned prior to application of queries [Dasu and Johnson 2003]. However, in many cases, especially in the case of *ad hoc* or novel analyses, pre-query cleaning is not cost-effective, and the data consumer is left with the responsibility for filtering out bad results using their knowledge of the domain semantics. A preferable solution for such cases would be for the end user's quality requirements to be incorporated into the query by the query writer, so that filtering or cleaning of results could be enforced automatically by the query processor, on only the data set that is of relevance to the end user's needs.

The ability to express information quality (IQ) constraints within a query language offers other advantages. In practice, assessing the quality of data is a complex task, demanding extensive domain knowledge and experience of working with the type of data being assessed[1]. If useful constraints on IQ can be expressed within standard query languages, then the knowledge of domain experts regarding IQ can be packaged in a form that is straightforward for technical staff supporting less expert users to access and reuse. This packaging of domain expertise becomes even more useful when the consumer of query results is not a human but a piece of software, and is therefore even less able to detect mistakes or omissions than a novice user. Data errors can easily spread into new databases through the use of computational processes to derive new knowledge from old (as commonly occurs in e-Science), causing the phenomenon known as data pollution [Redman 1996]. In such cases, it can be difficult to discover the source of errors once they have spread across several systems, which makes cleaning even more expensive. Effective, automated assessment of data quality before data sets are used by computational components is one means by which the spread of data pollution can be limited.

[1]For examples of the complexity of domain-specific forms of data quality measure, see the work of Burgoon et al. [2005], Korn et al. [2003] and Heim et al. [2004]

1.1 Provider-Centric IQ Assessment

Several researchers have proposed mechanisms by which queries can be expressed over data and IQ metadata (e.g., [Naumann et al. 1999; Scannapieco et al. 2004; Martinez and Hammer 2005]). As we shall discuss (in Section 2), most of this previous work has taken a provider-centric approach to the assessment of IQ. By this, we mean that the task of defining what forms of IQ should be supported by queries and of assessing individual data values against them is the responsibility of the provider(s) of data or the query facility, rather than of the data consumer. In the small number of proposals where this is not the case, data quality measures are precomputed by the execution of custom code or pre-assessed by human intervention, and so are decoupled from the actual action of the query processor.

This provider-centric approach is useful when it is possible for all consumers of a data set (or collection of data sets) to agree on a small number of widely applicable IQ measures. But, in many domains, and for many data sets, the needs of individual data consumers vary widely, depending on the specific application in hand. This is because IQ, like other forms of quality, is a relative not an absolute concept. According to the most commonly quoted quality definition, information is of high quality if it is *fit for purpose* [Batini and Scannapieco 2006] — something that can only be judged by the consumer of the information. Moreover, what is high quality data for one group of users may be considered poor by others. For example, a common scenario found in both e-business and e-science is that data sets are typically considered to be of acceptable quality for the application for which they were originally created, but are found to be of low quality when applied to a new application [Blaha 2001].

1.2 User-Centric IQ Assessment: a Motivating Example

In this paper, we set out to explore the complementary, consumer-centric approach, in which highly domain-specific IQ constraints can be added to database queries by consumers, without imposing any requirements on the owners of the queried sources to provide specific quality metadata or special-purpose quality measurement functions. This approach is motivated by the observation that, in many information-intensive applications, the decision regarding whether to accept or reject a data item is based on a combination of objective measures (quality indicators) and more subjective, consumer-specific criteria. This is increasingly the case in e-Science, for example, where "fitness for purpose" is defined differently by different scientists with different experimental goals in view, even when broadly the same sets of objective quality indicators are used.

The computational unit we use to encapsulate such user-specific definitions of fitness is the *quality view*: a shareable, reconfigurable IQ component that defines one particular way of assessing the quality of some particular kind of data. Quality views are designed to support users during the quality assessment steps of a quality-aware information lifecycle consisting of (i) information acquisition, (ii) quality assessment, (iii) filtering or editing for quality improvement, and (iv) information use.

To get a flavour of the type of assessments that quality views facilitate, consider the common problem of predicting the correctness of customer address data, when

no correctness assurance is provided by the data supplier[2]. In such a case, the quality assessment performed by the data consumer will typically be based on heuristics and indirect evidence. Criteria may include, for example, counting the number of addresses recorded for individuals in the data set, as well as using a trusted reference set to determine the validity of postcodes/zip codes in addresses. Other sources might contain related information, such as a database of records of bill payments, which can be used to cross-check against the address data for consistency and reasonability.

Application of these criteria can be seen as a process consisting of the following main steps:

(1) Issuing a query against the address data to count the number of distinct addresses per individual. These counts are associated with each address as the first quality indicator.
(2) Validating the postcodes in the address data against a reference database of choice. Any invalid or mismatched postcodes are recorded with each address as the second quality indicator.
(3) Issuing of queries to a bill-payment database, where some of the same customers are expected to be found, so that any discrepancies can be recorded against addresses as the third quality indicator.
(4) Combining all three quality indicators into a single quality score, by means of a user-defined quality function, for each address. The quality function encapsulates the "quality knowledge" used to make the assessment, and may be induced as a predictive model using machine learning algorithms or may be the result of user design.
(5) Using the quality score for a given address to decide whether to accept (i.e., trust) it or to reject it.

Note that steps (2) and (3) above may involve the use of similarity measures, and may result in corrections being made to the data where appropriate. The main point, however, is that quality assessment steps are interleaved with data access (i.e., query) steps. Thus, applying this process within a specific user query is a matter of reducing the scope of the quality assessment steps to the data touched by the query, rather than applying to the entire data set. More generally, we would like to be able to allow any query Q to be transformed into a quality-aware version Q' that adds user-specified IQ constraints and that returns only the subset of Q's results that satisfies those constraints.

1.3 Contributions

The main contribution of this paper, presented in Section 4, is an analysis of the features that must be supported by a query language in order to allow the incorporation of domain-specific IQ constraints in the form of quality views. (The quality view model itself is described and motivated in Section 3.) In particular, we show how IQ constraints over XML data can be incorporated in XQuery expressions, as well as describing an execution model for the resulting quality-aware

[2]A more complex, real-life example from the life sciences is presented in section 4.

XQueries. Quality views were originally designed for access from software, however, and the minimal essential set of languages features do little in themselves to shield the query writer from the low-level details of the QV API. We therefore further propose some syntactic sugar (tailored for XQuery, but easily adaptable to other contexts) to make queries using quality views shorter and more readable (Section 5). We illustrate the usefulness of the overall approach by giving some example quality-constrained queries from the application domain of proteomics (Section 5.3) and conclude with a discussion of directions for future work (Section 6).

Before delving into these details, we present a survey of existing approaches to quality assessment in a query context.

2. RELATED WORK

2.1 Assessing Information Quality

Poor information quality manifests itself in a variety of different costs for organisations that rely on the information for both operational and strategic decision making [Batini and Scannapieco 2006]. Errors in data can directly reduce the amount of productive work that an organisation can undertake, due to the need to spend time recovering from errors, but they can also have more far-reaching, less easily quantifiable costs. Customer satisfaction and loyalty can be damaged by errors, reducing the chance for future business, as can employee morale. Similarly, organisations can be prevented from changing their business rules and policies, if software systems cannot easily be adapted, due to unexpectedly poor information quality. Taken together, Redman conservatively estimated these costs to cover 10% of revenue, but suggested that the actual figure could be closer to 20% for some organisations [Redman 1998].

Considerable progress has been made in the development of tools and techniques to address specific information quality problems. For example, mature data deduplication techniques exist that can help to ensure that data sets map accurately to the real world population they model [Elmagarmid et al. 2007]. Related work, on record linkage, provides mechanisms for matching records from different data environments that refer to the same real world entity, despite differences in the representation and format of those entities [Winkler 2006]. More generally, data cleaning toolkits such as Potter's Wheel [Raman and Hellerstein 2001] and XClean [Weis and Manolescu 2007], have been proposed which allow IQ practitioners to implement filtering and transforming rules over data, that can detect and remove anomalies. However, the principal barrier to more generic solutions to information quality is the difficulty of defining what is meant by high or poor quality in real domains, in a sufficiently precise form that it can be assessed in an efficient manner.

No single agreed definition of information quality exists. Instead, it has been shown that information quality is an inherently multi-dimensional concept [Batini and Scannapieco 2006; Civan and Pratt 2006], with different (combinations of) dimensions being of relevance for different applications. In one influential study, Wang and Strong surveyed information professionals and uncovered a vast array of different perspectives on information quality, which they boiled down to 15 key dimensions, including accuracy, completeness, believability and interpretability [Wang and Strong 1996]. A number of methodologies have been developed to help infor-

mation quality practitioners to discover which forms of information quality are of relevance to their stakeholders, and to help convert them into specific quality scores. Key examples are: AIMQ [Lee et al. 2002], TDQM [Wang 1998] and CDQM [Batini and Scannapieco 2006].

2.2 Information Quality in Database Queries

It is usually the case that several of the standard IQ dimensions will apply to any one data source, depending on the specific needs of the various data consumers. For example, while one data consumer may be concerned primarily with the completeness of the data, another may prefer to see only the most up-to-date information, even if it is incomplete. Moreover, these dimensions are not in themselves definitions of quality, but are instead *families* of quality measures [Peralta 2006]. For example, even if we consider the single dimension of completeness in a relatively narrow application domain such as genomics, the completeness of a data set for one scientist might depend on whether all known genes for a particular species are included, whereas for another the key completeness criterion is whether all known strains of the species in question are included. Each dimension, therefore, is in fact a grouping of a potentially infinite number of quality measures.

When we talk about including information quality within database queries, it is this diversity of information quality score that we must access from the query language in question. In some settings, quality scores can be acquired for the data set in advance, and stored for querying through normal data or metadata access mechanisms. For example, in some cases, users can be expected to supply quality scores for the data set in advance, using either questionnaire-based approaches for assessing stakeholder perceptions of IQ at a coarse-grain, such as that advocated by AIMQ [Lee et al. 2002], or specialist tools for capturing fine-grained user assessments of quality, such as that proposed by Führing *at al.* [Führing and Naumann 2007]). For a generic query mechanism, however, we need a way to compute up-to-date quality measures, either to fully populate a quality metadata store for a data set or to compute scores on-demand, for just those parts of the data set accessed by the query. It is unlikely, in most settings, that human users would be able to undertake the IQ annotation burden this implies, and therefore automated (or at worst semi-automated) means of computing quality scores are required, in a form that can be easily specified and manipulated in a standard query language.

In most of the existing IQ-oriented query systems, this has been achieved by selecting a small number of IQ dimensions that are of relevance to their domain, and defining a single form of quality measure for each dimension. By fixing the number and type of IQ criteria that are considered, it becomes possible to place the responsibility for measuring and recording IQ levels with some part of the infrastructure responsible for delivering query results. Hence, these approaches can be considered *provider-centric*. Two options have so far been explored:

—approaches in which participating information sources agree to publish a common set of quality measures for the data sets that they export (structured according to some common quality data model) [Martinez and Hammer 2005; Mihaila et al. 2000], and

—approaches in which query processors (typically, distributed query processors)

are extended with the ability to compute a set of quality measures on demand during query evaluation [Berti-Equille 2004; Naumann et al. 1999; Scannapieco et al. 2004; Simmhan et al. 2006].

The first option is the simplest conceptually, although it has some disadvantages in practice. Under this approach, either the metadata model or the schema of the component data sources must be expanded with additional elements for holding IQ scores for the selected dimensions. The individual sources are responsible for computing these measures, and exporting them along with any query results. To the query processor, this exported quality data looks just like any other kind of data or metadata and so it can be included within queries without the need for additional language support. For example, Martinez and Hammer proposed a mechanism by which biological data, represented in semi-structured form, could be extended with additional tree structures representing the quality of each relevant node and attribute [Martinez and Hammer 2005]. They selected four IQ dimensions and defined how numerical scores could be computed for each based on information known to be present in the data itself. Since, in this approach, quality scores are returned as normal data, it is straightforward to query over them with a standard (unextended) query language. By contrast, Mihaila *et al.* chose to extend the metadata that must be published by participating sources, with the aim of enabling quality-oriented selection of sources in web information systems [Mihaila et al. 2000]. They chose four IQ criteria, based on four quality dimensions appropriate for geo-spatial information systems, and describe a metadata model for their export by sources, which can be queried using a slight variant of SQL[3].

These proposals require a significant commitment from the owners of participating sources, who must agree to provide the resources (CPU time, disk storage, human expertise) needed to compute and maintain the quality measures. This is a threat to source autonomy, and may not be practical in domains where the major data sources are already well established and contain much historical data that would be extremely costly to assess for quality *en masse*. The second provider-centric option, in which the query processor takes responsibility for measuring IQ, does not place such stringent constraints on individual sources (although some shared requirements must still be imposed). One such approach is to extend the query language with special keywords allowing users to specify constraints in terms of a single in-built notion of quality. This is only appropriate for forms of IQ that are sufficiently generic to be applicable across a wide set of domains. Such forms are rare, with the most successful probably being the notion of *data freshness* [Bouzeghoub and Peralta 2004]. Even in this case, however, there are multiple ways in which freshness can be defined and computed [Sampaio et al. 2005], and if the language designers have chosen to implement a form of freshness which is not applicable to a specific user, then no support at all can be provided.

Clearly, it is too costly to add individual language support for a wide range of diverse query criteria. An alternative that is possible in distributed information systems (DIS) is to extend the global metadata model with elements for each IQ

[3]Since the metadata tables look like ordinary tables in this approach, the only additions required are some fuzzy comparison operators for ease of expression of loose quality-related constraints.

criteria required, and to arrange for the query processor (or some other DIS component) to populate the model on demand. Representative examples of proposals based on this option are the DaQuinCIS system [Scannapieco et al. 2004; Batini and Scannapieco 2006] and the query integration work of Naumann *et al.* [Naumann et al. 1999; Naumann 2002].

DaQuinCIS is a platform for cooperative information systems that provides explicit support for the use of IQ in improving integrated query results and in improving the quality of data held in the underlying sources. Again, four IQ dimensions are selected, and provision for including scores or measures for each of them is incorporated into a special data quality model (called D^2Q). This model mirrors the schema of the (semi-structured) data, and allows one IQ measure per dimension to be associated with each data instance and each property of each instance. The data quality model is populated by special *Data Factory* components, which must be provided for each source that joins the system. New data quality scores can be incorporated into the metadata model if all DataFactory components can be extended to provide them. Although originally intended for use by the DaQuinCIS query processor in ranking and selecting results based on quality, the DaQuinCIS team have also shown how the populated data quality models can be queried directly through XQuery, by creating a user-defined function for each quality dimension supported by the model [Milano et al. 2004].

Naumann *et al.* examined the potential uses of quality scores to better perform the process of distributed query evaluation [Naumann et al. 1999; Naumann 2002]. They focussed on the use of vectors of quality scores (corresponding to ten fixed quality measures/dimensions) to prune the search space when generating query plans, both in terms of filtering out poor quality sources from consideration and of ranking the generated plans with the goal of locating the highest quality data soonest. Unlike in DaQuinCIS, three different sources of quality measures are considered. Some are calculated by the query processor, based on information provided by the DIS (such as availability of individual sources), others are computed based on the intermediate result sets returned from sources and the remaining measures are provided by users as individual profiles. This last source of quality measures provides the information consumer with some measure of control over the results returned. However, the profiles can only supply quality criteria that describe whole sources, and users are expected to create one profile for use in answering all queries. Moreover, the set of quality criteria that can be set by the user are still fixed by the original design; users cannot change these definitions or add their own.

Berti-Equille took this notion of user configurability a step further with the design of the XQual language [Berti-Equille 2004; Berti-Équille 2007]. In XQual, the user can define IQ *contracts*, which specify constraints over a set of quality dimensions and which can be used to wrap SQL queries with quality requirements. The contract specification for each dimension of interest includes the name of a user-defined function that has the task of generating the quality scores for the data set to which the contract applies. These functions are invoked when the contract is created, and must precompute the quality measure for the complete data set (at the granularity preferred by the consumer) and then record them in a shared metadata store. The measures can then be accessed by the XQual processor, which evaluates queries

only over that subset of the data which satisfies the quality constraints given in the contracts applied to the query.

The IQ contracts of XQual are significant because they are an early attempt to allow query writers to associate (partially) declaratively specifications of IQ preferences with their queries. This idea was recently developed further by Simmhan et al. [2006]. Recognising that different consumers will have very different IQ needs, and that it is unrealistic to expect information providers to cater for them all, these authors proposed a rule language that describes how to assign scores to data items based on information available from the provider. An example rule in this language is:

```
switch (transferTimeSecs) {
  case < 10 : return qualityScore 7;
  case > 60 : return qualityScore -7;
  default   : return qualityScore 0;
}
```

(taken from [Simmhan et al. 2006], p. 2). Here, `transferTimeSecs` is the data item being assessed for quality, and `qualityScore` is the result of the quality assessment process. This work, however, is at an early stage and it is not yet clear whether this approach can provide the expressiveness needed for the complexity of quality measures encountered in real applications or how the scores would be accessed from within a query language.

2.3 Discussion

As we have seen, the current proposals for query mechanisms incorporating IQ constraints have focussed on techniques that are limited to either a fixed (built-in) set of quality criteria (typically also limited to one criterion per quality dimension), user specified scores for fixed quality criteria or else selection and computation over the available metadata to define and name new criteria. While these approaches all have value in contexts where the selected dimensions and criteria are useful to a wide range of users, their applicability is limited where highly domain-specific forms of IQ are required. We can summarise the limitations of the existing proposals for such application areas as follows:

—In most of the surveyed approaches, information consumers are limited to the forms of IQ that the designers of the data source or DIS thought appropriate. There is little scope for users to express their own requirements as to how the quality of information should be judged.

—Although some of the systems surveyed do include extension points for the introduction of custom code for user-defined quality measures, the interface that the code must adhere to is undefined and the mechanisms by which the code operates (and is introduced to the system) are unclear. Moreover, the interation between the custom code and the query evaluation is decoupled, meaning that the ways in which quality can be introduced into queries is limited (e.g., as an additional constraint wrapped around an otherwise conventional query).

—Where some more principled form of user-configurability is supported, IQ assessment must still be performed using the data/metadata exported by the queried

sources. In contrast, many real world approaches to IQ assessment require access to additional sources of information. For example, completeness of a data set is often defined with respect to some named reference set, which may not be stored within the sources being queried.

—The proposals for declarative IQ assessment languages surveyed are not yet mature, and are limited to simple computations or conditions over attributes of individual instances. By contrast, real world IQ assessment procedures are often highly complex decision procedures, some of which require the ability to perform aggregate computations over the entire data set. (A familiar example of the latter is in data deduplication, where the confidence in, say, an address record depends on the number and strength of matches found across the entire address list [Winkler 2004].)

—Provider autonomy can be compromised by the requirement to export particular forms of quality metadata, or particular data items from which to compute quality metadata. If consumer requirements change and new metadata must be exported, it may not be practical to modify large volumes of existing data to include the new measures.

In our previous work on the Qurator project, we have proposed a model-based approach to the specification of software components that can assess quality in a domain-specific manner [Missier et al. 2006]. These components, called *quality views* (QVs), are designed to be reusable in a range of information processing environments, and therefore should be amenable to incorporation within a standard query language. In the remainder of this paper, we examine whether and how quality views can be used to embed consumer-oriented notions of IQ assessment into the XQuery language.

3. QUALITY VIEWS: SHARABLE IQ COMPONENTS

3.1 A Model for Quality Assessment Components

Although it is common in the IQ literature to talk of "measuring", "evaluating" or "assessing" the quality of information, in practice it is almost never the case that we can actually measure the fit of the data to the state of the real world. The best we can hope for is to compute a close *estimate* of its quality. Consider, for example, the common case of customer address data, which must be assessed for accuracy/correctness. However rich and detailed the data model, there is no attribute stored within standard address data that can tell us, by itself, whether the person named is indeed currently resident at the given address. Instead, we hypothesise the features that high quality data has, and check our data set for consistency with our hypotheses. In the case of address data, we make a complex sequence of reasonability checks using the attributes in the data set and other relevant sources of information. For example, we might count the number of addresses recorded for each individual in the data set, as well as using a trusted reference data set to determine the validity of postcodes/zip codes appearing in the addresses. Other sources containing related information, such as a database of records of bill payments, might be used to cross-check against the address data. Having gathered all these individual pieces of evidence for a specific address, we must combine them in

some way to make a defensible guess about its quality, rather than arriving at a definite, absolute quality measure.

As this simple example shows, estimating IQ in practice is a more complex task than the calculation of a single score or measure. First, we must gather the individual pieces of evidence that will be used to estimate IQ, and then we must apply one or more decision procedures to the evidence to assign a specific quality class or score to each item in the data set under study. Both the evidence gathering tasks and the decision procedures themselves will vary dramatically in their scale and complexity, depending on the needs of the application domain. In the case of evidence gathering, we might simply need to extract some values from the data set itself, and perform a straightforward calculation over them. In other cases, however, we might need to join the data set with another database in order to extract related information (including applying any necessary transformations to resolve heterogeneities), or to invoke a sequence of external services in order to compute the required information. Similarly, decision procedures can range from the standard aggregation operations noted by Pipino et al. [2002] (e.g., maximum, average) to fully-fledged decision models based on decision trees, clustering algorithms or sets of learnt association rules (e.g., the rules for assessing quality of transcriptomics data proposed by Burgoon et al. [2005]).

In our previous work, we have proposed a model for the rapid specification and deployment of software components that can encapsulate this kind of IQ estimation process. Given the complexity of the task as just described, designing a purely declarative language for specifying complete IQ measures is a more challenging prospect than might at first be apparent. However, this does not mean that it is necessary to fall back on the assumption that all IQ assessment components must be custom-coded from scratch. In our work, we have attempted to find a compromise position that allows some elements of the IQ assessment process to be specified declaratively, at a high level, while leaving the more challenging, procedural aspects to be specified in the form of Web services with standard roles and agreed interfaces, allowing them to be reused and reconfigured with ease.

The principal component in our model is the *quality view* (QV) [Missier et al. 2006]. Viewed from the outside, a QV is a software component that takes in a data set, assesses the quality of each item in the data set according to the specific notion of quality embodied by the QV, and then uses the results of this process to transform the input data set in some way. It can thus be seen as playing the role of the middle (transformational) stage in the well-known ETL paradigm. A range of commonly encountered forms of quality manipulation can be provided through this model. For example, one familiar style of transformation is the quality-based filter, in which poor quality data items in the input data set are blocked, while the better quality items are allowed to pass through to the output. Another example transformation might be to rank the elements of the data set according to their quality score, or else to augment the data set with a new attribute describing its quality. More complex transformations might attempt to clean the low quality data items before reincorporating them into the output data set.

Internally, a QV is an instantiation of the generic pattern for quality manipulation that we have observed in our studies of IQ assessment in various e-Science

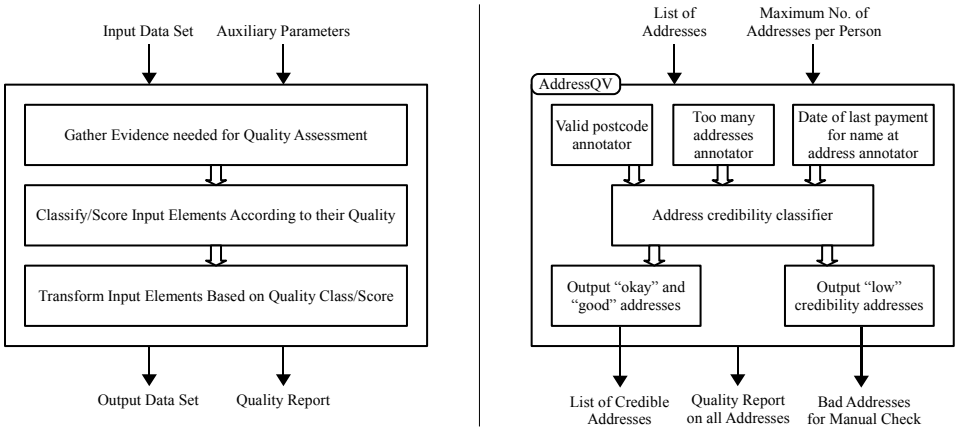

Fig. 1. a) Quality View Pattern b) Example Quality View

domains [Preece et al. 2006; Missier et al. 2007] and that we described earlier in this section. The pattern is shown in abstract in part a) of Figure 1, with a specific instantiation of the pattern, expressing a QV taken from the simple address data example used earlier, given in part b). As illustrated, a QV is a layered configuration of components. In the top most layer, components (called *quality evidence functions* or *annotators*) are applied to each item in the input data set in order to gather together the various pieces of evidence required for the quality assessment procedure described by the QV. In the example in Figure 1b) three different evidence functions have been included in the QV. This evidence is then passed on to the middle layer of components. These components are called *quality assertions* (QAs) and they have the job of applying a decision procedure to the elements in the input data set (and the evidence gathered) in order to assign a quality class or score to each one. The example QV in Figure 1 has a single QA, but in general, a QV may have zero or more of these middle layer components. Each QA takes as input the data set being classified, and selected values from the QV's evidence base. For each item in the data set, the QA will output a value from its own quality classification scheme. A quality classification scheme is a partially ordered set of labels (for example, $\{ \mathit{high} > \mathit{medium} > \mathit{low} \}$), with one label for each level of quality that is relevant to the kind of quality assessed by the QA.

The components making up the bottom layer of the QV pattern implement the quality-oriented transformation of the data set, based on the evidence and the quality classifications produced by the quality evidence functions (QEFs) and the QAs. The transformations are specified as condition-action rules (CAs), with the conditions being stated declaratively within the QV specification and the actions being either a call to a transforming Web service or the application of an XSLT expression. It is possible to chain CAs to produce a single output stream, or to apply them in parallel to produce multiple output streams, as illustrated in the example QV. This latter option is particularly useful for routing data of different quality levels to different processing components after exit from the QV. For example, high quality data could be passed through to a browser for inspection by the user, while

Identifier	*Address Credibility Classifier*
627634	low
723943	okay
829222	high
⋮	⋮

Fig. 2. Example of the Quality Classification Data Structure

low quality data is diverted to an error queue for off-line examination and possible manual correction.

All QVs must adhere to a standard interface, so that they are easily reusable and exchangeable in practice. The input data sets must be a collection of XML elements, the first attribute of which is an identifier of some kind for the element. A list of optional parameters may also be supplied, which will then be available for use by any of the internal components. Our example *AddressQV* takes a single parameter, which is consumed only by the *Too many addresses annotator* component and which indicates the maximum number of addresses for an individual that the QV is prepared to tolerate before considering the data as suspicious. We might also have given an additional parameter: a threshold to be used by the CAs in deciding whether to pass the addresses through to the output or to queue them for manual attention. In this case, the QV designer elected to hard-code this threshold.

The format of the output of a QV depends upon the CAs that make up its bottom layer, and their configuration. Since a CA may apply an arbitrary transformation to the data set, there are no constraints on the format of data sets output by QVs; they can produce whatever form of data the consumer of the QV requires. However, to better support reuse, all QVs also output a full quality report on the data set. This is in XML form, and consists of a table, giving the output of each QA for each item of the input data set. An example of the report that might be produced by the example *AddressQV* is given in Figure 2. In this case, since the QV contains only one QA, the quality report contains only one column of quality classes. In general, the report will contain one column for each QA in the QV. By exporting this information as standard, QVs can provide straightforward programmatic access to quality decisions without having to limit the range of transformations that can be made by individual QVs.

3.2 The Information Quality Ontology

As well as defining standard interfaces for QV components as a whole, we have also defined them for the internal QEF and QA components of quality views, in order to increase the ease with which they can be reused and reconfigured to create new QVs. However, while XML schemas can describe the structure of input and output parameters, they have been found to be insufficient by themselves to support component reuse; further annotation expressing the semantics of components and their parameters are required in order to ease their discovery and provide tool support for their composition [Medjahed et al. 2003]. Therefore, each QV, QA and QEF Web service is annotated in this way relative to a specially designed Information Quality ontology [Preece et al. 2006], a fragment of which is shown in

Figure 3. Here, concepts are shown using ellipses, while relationships between the concepts are shown using arrows.

This ontology is split into two levels. The upper-level ontology contains generic IQ concepts and concepts for the important QV components. A sample of these are shown on the left in the figure (the ellipses with solid outlines). Following the standard semantic Web service approach, components are annotated with terms describing their general function (i.e., task annotation), and their input and output parameters are annotated with terms describing the semantic real-world objects. For example, QA components are labelled as having a task of *Quality Assertion*, and as taking input with the semantics of *Quality Evidence*.

The lower-level ontology contains domain-specific specialisations of these generic terms, describing the semantics of the inputs and outputs of the actual QA and QEF components that have been created and registered with Qurator. For example, the right-hand side of Figure 3 (the ellipses with grey outlines) shows part of the semantic model for the *AddressQV*. The concepts shown model the semantics of the three annotation functions, including a declaration of the type of input data they expect (*Person Address* data), and the kinds of evidence that they produce (e.g., *Valid Postcode*). Similarly, a concept exists for the *Address Credibility Classifier*, showing the types of evidence that it requires as input. Its output (instances of its classification model) are not shown in the figure, for reasons of space. The unlabelled arrows represent inherited versions of the relationships shown between the upper-level classes. A full description of the semantics of quality views and their components, along with more details of the contents of the Qurator ontology, can be found elsewhere [Missier 2008].

3.3 Deploying and Invoking Quality Views

One of our aims in designing QVs was to create components that could be easily incorporated into the information consumer's preferred information environments, rather than requiring data to be exported into and out of a dedicated quality management system. The high-level specification of quality views is designed to be independent of any specific technical context. Instead, compilers can be created

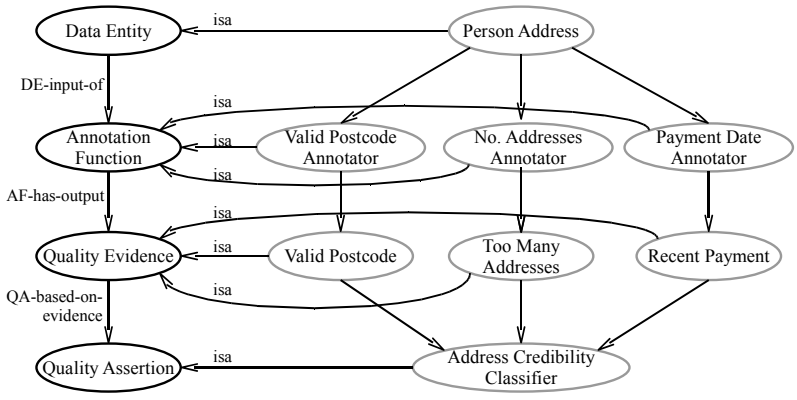

Fig. 3. A Fragment of the Information Quality Ontology

that can convert these specifications into executable components suitable for use within particular information environments. For example, since scientific workflows are currently a popular implementation technology for *in silico* experiments, we created a compiler that converts QV specifications into quality assessment and transformation workflows that can be embedded within larger scientific workflows at the point where quality management is required [Missier et al. 2006].

For the purposes of embedding QVs into query languages, however, their most useful incarnation is as standalone Web services. We have provided a single point of invocation for all QV implementations known to our semantic registry [Preece et al. 2006]. This Web service requires the following inputs:

—The semantic type of the QV to be invoked (specified as the URL of a concept in the IQ ontology).

—The data set to which the QV component is to be applied.

—Any additional parameters needed by the QV. These are either constants, that are used by the QV to configure its internal functions, or references to external data sources that the QV needs to access in order to compute the quality values.

By this means, QVs can be invoked from within any information environment that supports access to Web services.

4. EMBEDDING QUALITY CONSTRAINTS INTO XQUERY

Since the specification of the XQuery language[4] includes constructs for the declaration of externally defined functions, it is straightforward to envisage how the QV invocation Web service just described can be accessed from within XQuery expressions. This process is further assisted by the fact that the inputs expected by QVs and the quality reports they produce are already in XML format, so that the language constructs needed to manipulate them are already present in XQuery. The main question, therefore, is how easy is it to write XQuery expressions that can adhere to the semantic constraints imposed by the QV specification.

In order to illustrate the issues that arise, we will consider the use of QVs within an example query taken from our work with proteomics scientists. By analogy with the genome, the *proteome* is the set of proteins that an organism is capable of producing throughout its lifetime. *Proteomics*, therefore, is the study of the conditions which trigger the production of these proteins and their individual roles in sustaining or damaging the organism which hosts them. A typical proteomics experiment involves taking one or more samples from organisms of interest, produced under contrasting circumstances (for example, from a diseased individual and from a healthy one) and then attempting to discover the differences between the sets of proteins present in each sample.

These *protein identification* experiments typically consist of a "wet lab" phase followed by a "dry lab" (i.e., computational analysis) phase. In this latter stage, the results produced by the wet lab phase are analysed in order to extract a list of protein hits—that is, a list of the proteins thought to be present in the sample. The results of such experiments are then stored in one of the public repositories

[4] `www.w3.org/TR/xquery`

for protein identification data (such as PRIDE[5] and PedroDB [Garwood et al. 2004]) where they can be freely accessed. This allows scientists to query over large collections of experiments to discover whether some protein of interest has been seen by other scientists, and perhaps to formulate some hypotheses regarding the role of the protein based on the complete set of circumstances under which it has so far been observed.

A typical query over proteomics data, therefore, requests details of protein identifications from one or more experiments that meet some specific criteria of interest. For example, the following query (expressed against the PedroDB XML schema [Taylor *et al.* 2003], using XQuery) requests details of all proteins that have potentially been observed in a particular experiment that are not *actins*[6].

```
<html>
<ul> {
  let $inputDoc := doc("proteomicsDB/PSF1-ACE2-CG.xml")
  for $d in $inputDoc//DBSearch, $h in $d/ProteinHit
  where not (fn:contains($h/Protein/description, " actin "))
  return
    <li> accession: {data($h/Protein/accessionNumber)},
         gene: {data($h/Protein/geneName)},
         name: {data($d/username)},
         date: {data($d/idDate)}
    </li>
} </ul>
</html>
```

An actin is a type of protein that tends to be highly conserved between species, a property that makes them uninteresting to the scientist who commissioned this example query. For each non-actin protein potentially observed in this experiment, the query returns the accession number (i.e., unique identifier) of the protein, the name of the gene to which it corresponds, the name of the scientist who carried out the experiment and the date on which the observation was recorded.

This query, when executed against the `PSF1-ACE2-CG.xml` experiment result set, returns around 80 protein hits — rather more than the handful of strong candidates for follow-up research the scientist was hoping for. One difficulty in making inferences over the results of protein identification experiments is that they often contain a large proportion of false positive and false negative results [Carlige et al. 2004; Topaloglou 2006]. This is due in part to the inherent uncertainty of the wet lab processes involved and in part to the paucity of information that is available about each identification to the dry lab analysis tools.

To make this query more useful to the scientist, we would like to add some quality control constraints, to ensure that only high quality results are returned. Several measures of quality for proteomics data are beginning to emerge in the literature. For example, proteomics scientists at the University of Aberdeen have developed a computable quality measure that is able to assess protein identification

[5] `www.ebi.ac.uk/pride`

[6] We are grateful to Dr Simon Hubbard for suggesting this example query.

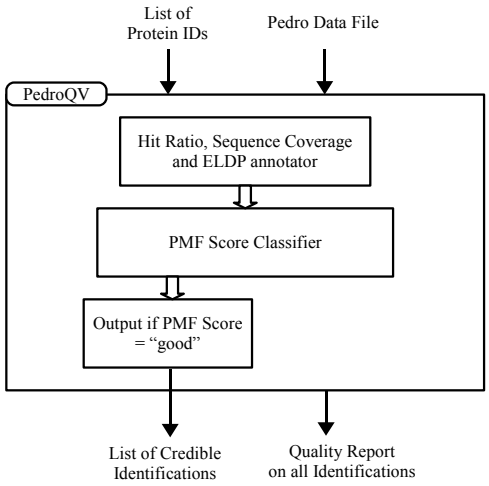

Fig. 4. A Quality View Based on the PMF Score Quality Measure

results[7] and identify some kinds of false positive match [Stead et al. 2006]. We have embedded this score (called the PMF score) within a QV, to provide a Web service for classifying proteomics data according to this measure of quality. The components of the QV are illustrated in Figure 4. Here, the top level component computes the additional evidence needed to assess IQ based on the data in the input document (a process requiring access to an external public Web service). The middle layer component calculates the PMF score using this evidence and discretises it into quality classes such as "good" and "poor". The final layer of the QV contains a single filtering condition-action rule that removes poor quality identifications from the input data set. This was the behaviour required by the original consumers of this QV, but for our purposes we want to apply our own quality constraints from within the query (for example, so that we can experiment with different strengths of constraint until our scientist is happy), and therefore we will make use of the quality report output, rather than the output of the condition-action rules.

The extended version of our query, accessing the *PedroQV* Web service, is shown in Figure 5. The components that have been added or changed in order to add the quality constraints are shown in bold font. We will now walk through each of these added query elements.

The `namespace` declaration that begins the extended query is responsible for binding a local XQuery function with the operations supported by the QV Web service. This syntax is not part of the XQuery standard, but is instead based on the proprietary mechanism for importing external functions provided by the Saxon-B XQuery engine[8]. Other XQuery evalulation engines will either implement the standard syntax or have their own external function mechanism.

[7]Strictly speaking, this quality score is only suitable for identifications produced using protein mass fingerprinting (PMF) technology.

[8]`saxon.sourceforge.net`

```
declare namespace qvi = "java:org.qurator.util.QVInvoker";

declare variable $prefix :=
               "urn:net.sourceforge.taverna.scuflworkers.uniprot:"

declare function local:prepareDataInput($inputDoc) {
   for $x in data($inputDoc//ProteinHit/Protein/accessionNumber)
   return fn:concat($prefix, fn:tokenize($x,' ')[1])
};

<html>
<ul> {

   let $inputDoc := doc("proteomicsDB/PSF1-ACE2-CG.xml")
   let $qvResults := qvi:QVInvoke(
               qvi:new(),
               "http://www.qurator.org#PMFScoreQV",
               local:prepareDataInput($inputDoc),
               ($inputDoc)
   )
   for $d in $inputDoc//DBSearch, $h in $d/ProteinHit
   for $qc in $qvResults/enrichedData/EDItem
   where (not fn:contains($h/Protein/description, " actin "))
   and
      fn:contains(fn:concat($prefix, $d/accessionNumber),
               $qc/@dataRef)
   and
      $qc/assertionValue[@AssertionTagName = "PMFScore",
               @AssertionTagValue != "poor"]
   return
      <li> accession:  {data($h/Protein/accessionNumber)},
         gene:  {data($h/Protein/geneName)},
         name:  {data($d/username)},
         date:  {data($d/idDate)}
      </li>

} </ul>
</html>
```

Fig. 5. Extended Query with Embedded QV Quality Constraints

The `$prefix` variable and `prepareDataInput` local function are both concerned with the preparation of the input data set for use by the QV. In this case, the data to be quality assessed is the entire input experiment set. Although this data is already in XML format, it does not quite meet the input requirements for the QV component, which expects a list of identifiers of the data items to be classified. This function therefore prepares this list of identifiers. The query writer must state the path expression that can be used to retrieve the identifying attributes of each item of the input data set, i.e., of each protein identification. The preparation function then extracts these identifiers, and converts them into the Life Science Identifiers[9] (LSIDs) expected by the QV. In this example, protein identifications are identified

[9] `lsid.sourceforge.net`

by protein accession numbers and all that is needed in this particular case to convert them to LSIDs is to prefix a unique reference string onto this accession key. Clearly, other forms of identifier may require more complex preparation steps.

The next new component of the query is the second `let` expression, within the main query body. This expression uses the newly defined namespace to invoke the required quality view by means of the `QVInvoke` Web service operation described in section 3.3. The arguments given to this operation are:

—`qvi:new()` This first argument is required by the Saxon implementation of external functions.

—`"http://www.qurator.org#PMFScoreQV"` This second argument is a URI identifying the particularly quality view to be invoked. The `QVInvoke` Web service uses this string to search in the Qurator registry for this QV, and invokes it.

—`local:prepareDataInput($inputDoc)` The third argument is an XQuery expression specifying the data set which is to be classified by the named QV (i.e., the prepared version of the experiment data).

—`($inputDoc)` Finally, the fourth argument is a list of any additional parameters needed by the QV. In this case, the only parameter is the input document itself, from which the QV will extract additional elements to be used in the computation.

The quality report exported by this QV (in XML form) is returned from this call, and assigned to the variable `$qvResults`, for use within the remainder of the query.

Next, we consider the new `for` expression that has been nested within the `for` construct from the original query. For each item in the set of `DBSearch` elements (i.e., individual protein identifications) within the input document, we now also extract the set of quality classifications for the item from the QV results. In fact, we are only concerned with the quality classifications that have the same accession number as that of the current protein identification. This constraint is stated by means of the additional conjunct in the `where` clause:

```
fn:contains(fn:concat($prefix, $d/accessionNumber),
            $qc/@dataRef)
```

Effectively, this constraint specifies a join of the protein identification data to the quality report, so that we can make use of the quality classification for each identification in the rest of the query. The process is complicated by the lack of a common join key; hence the need to construct an LSID based on the key of the current identification for comparison with the keys used in the quality report.

The `where` clause of the extended query also contains the constraint from the original query that the identification should not be for an Actin protein, and a second new conjunct:

```
$qc/assertionValue[@AssertionTagName = "PMFScore",
                   @AssertionTagValue != "poor"]
```

that states the constraints on the quality of the protein identifications that we want the query to return. The element tags used here are taken from the schema used for quality reports returned from QVs. The expression as a whole tests that

the quality class for the current identification is better than "poor" (from a set of "poor", "average" and "good").

Since, in this case, we only wish to remove poor quality identifications from the query result, the remainder of the query is unchanged from the original. An alternative would have been to extend the list of attributes returned from the query to include the quality classification for the identifications, to rank the results according to their quality, or to compute data cleaning transformations on selected items before returning them to the user.

Although many of the details of this example query are specific to the XQuery language, we can extract from it the general requirements for expressing quality-oriented constraints based on QVs in any database query language. In summary, for this to be possible, the query language must possess the following features:

—The ability to import externally defined functions into specific queries, so that the quality view Web service can be invoked from within query expressions.

—The ability to prepare data sets for classification by imported QVs, by converting their identifiers to the form expected by the QV. In this case, this requires only a string concatenation function and the ability to create temporary data structures based on the inputs to the query, that can be passed as input to an imported function. In other cases, the input data set may already contain the required identifiers, or a more complex transformation process may be required.

—The ability to receive complex data structures as the result of external function applications, and to access the components of those structures as first-class data items within the parts of the query in which they are in scope.

5. QXQUERY: SYNTACTIC SUGAR FOR A QUALITY-ORIENTED XQUERY

It will be apparent to the reader that the process of adding quality-oriented constraints to our original example query has lengthened and complicated it beyond what might be expected of the addition of a simple filter condition. The intent of the new version of the query is now much more difficult to fathom, due to the intrusion of many additional details resulting from the specific form of the quality view Web services and the data structures that they expect to receive and return. These data structures were designed to be easily and efficiently processed by software. As is often the case, the cost for this is that the structures are not in a particularly convenient or intuitive form for human query writers.

One solution to this problem is to extend the XQuery language with special constructs for interacting with quality views in a more natural and less cumbersome way. Since XQuery already possesses all the functionality required to execute QV-based queries, these extensions are merely syntactic sugar and can be implemented by a preprocessor which transforms queries that make use of them into pure XQuery expressions that can be executed using standard query evaluators. We set out to explore the minimal syntactic extensions needed to support the specification of QV-based queries in XQuery, and to hide as many of the QV implementation details as possible from the query writer or reader. We call this extended language *QXQuery* (for Quality-oriented XQuery).

Two basic forms of functionality must be supported by QXQuery over and above

what is supported by XQuery. These are invoking QVs and accessing their results. We consider the syntactic extensions needed for each of these separately.

5.1 Invoking Quality Views

As the example query shown in Figure 5 illustrates, the invocation of QVs from XQuery requires the definition of the QV invocation Web service as an external function, the preparation of the input data set and the invocation of the QV external function with the appropriate parameters using a `let` clause. If we abstract from these three tasks the information that must be provided by the query writer, it becomes clear that a single new construct can be used, from which the three groups of code can be generated. This construct is a *quality view clause*, and it has the following syntax:

```
<QVClause> ::= using quality view <qualView>
               on <PathExpr> with key <PathExpr>
               as <VarName>

<qualView> ::= <URI> '(' ( <ExprSingle> ( ',' <ExprSingle> )* )? ')'
```

Here, `<PathExpr>`, `<ExprSingle>` and `<VarName>` are all defined by the standard XQuery syntax, and are hopefully self-explanatory. To illustrate the role of this *QVClause*, the following QXQuery fragment shows how we would use it to specify the application of the `PedroQV` quality view to a set of experimental results held in the variable `$expRes`, with the resulting quality report being assigned to the variable `$qv`.

```
using quality view "uri://PMFScoreQV"() on $expRes with key
/ProteinHit/Protein/accessionNumber as $qv
```

As this fragment shows, the quality view to be invoked is indicated (in the `<qualView>` construct) by giving the URI of the class in the Qurator ontology that corresponds to the required quality view, and expressions corresponding to the parameter values to be passed to it. The URI is used by the QXQuery preprocessor to provide the value for the second argument of the `QVInvoke` function when the QXQuery is translated into pure XQuery, while the additional parameters are bundled together to provide the fourth argument to `QVInvoke`.

These parameters, however, do not include the main input data set for the QV, which is specified separately, as an arbitrary path expression, after the `on` keyword. This syntax has the effect of stressing the special role of this input to the QV (as the data set which is to be classified and transformed by it) and of ensuring that it is never possible to invoke QV in QXQuery without also supplying a data set for it to operate on. It is also necessary to distinguish the input data set so that the preprocessor can create the code that will transform it into the format required by the QV (in this case, a sequence of LSIDs). The processed data set can then be passed as the third parameter to the `QVInvoke` function.

Note that the query writer must also specify the `key` for the input data set. This is an XPath expression that can be used to locate the identifying attributes for each item in the input data set. It is required for the generation of the input preparation function, which takes the following form:

```
declare function local:prepareDataInput_<N>($inputDoc) {
  for $x in data($inputDoc/<KEY_PATH>)
  return fn:concat(<LSID_PREFIX>, fn:tokenize($x,' ')[1])
};
```

The values here in angle brackets and capitals indicate the values that must be supplied by the preprocessor, using the information given in the `QVClause`.

—`<N>`: since a separate data preparation function must be generated for each QV assigned to each data set, a number is appended to the name of each function to ensure that it is unique within the generated query. The preprocessor keeps track of which function name applies to which QV and data set combination.

—`<KEY_PATH>`: this is the key path expression specified in the *QVClause*.

—`<LSID_PREFIX>`: this is the prefix needed to convert the identifying attributes into LSIDs, as expected by the QV. This value is domain dependent and is therefore stored in the quality ontology, alongside the semantic types used to describe the input types expected by QVs.

Once fully instantiated, these function definitions iterate over the input data set, extracting the key values, and converting them into the identifiers expected by the QV.

The final component of the syntax for quality view clauses is introduced by the `as` keyword, and gives the name of the variable that will hold the results of applying the quality view to the transformed data set. This variable is considered to have the same scope as the left hand side variable of a standard `let` clause.

In the context of complete queries, we would like these quality view clauses to be used with the same flexibility that the `let` and `for` clauses can be used in XQuery. To allow this, we must expand the syntax of the XQuery FLWOR expressions[10] to:

```
<FLWORExpr> ::= (<ForClause> | <LetClause> | <QVClause>)+
                <WhereClause>? <OrderByClause>?
                return <ExprSingle>
```

The only change here from the standard XQuery syntax is the addition of the `<QVClause>` non-terminal. This extension of the FLWOR syntax means that a QXQuery can contain zero, one or many quality view clauses, and that they can be interleaved between arbitrary numbers and combinations of `let` and `for` clauses as is necessary to express the query requirements. A full analysis of the effects of this syntactic change on the normalisation, static type checking and the dynamic evaluation rules of XQuery can be found elsewhere [Embury et al. 2007].

5.2 Accessing Quality Classifications

The syntactic extension just described provides us with the ability to invoke QVs on unprepared data sets within queries. However, the reader will recall that we also need the ability to access and manipulate the results of the QV application in the rest of the query. In theory, since the results of QVs are already XML documents,

[10]Pronounced "flower" — these are the basic query forming constructs in XQuery.

and since the quality view clause binds them to a normal XQuery variable, we do not need any additional syntax to support this requirement. However, without some additional syntactic support, the quality-constrained query writer (or reader) is forced to be aware of the internal structuring of the quality classification report output by the QV. Ideally, it should be possible to access the classification results at a higher conceptual level, in terms of the specific measures of quality that are implemented by the QV, without direct reference to the low-level data structures used.

For example, `PedroQV`, the QV used in our example query, contains a quality assertion that classifies protein identifications according to a scoring model defined by Stead, Preece and Brown, called the *PMF Score* [Stead et al. 2006]. Rather than having to access the classification for a given identification `$d` using the cumbersome path expressions we saw in the example query in Section 4, it would be much simpler for the query writer to indicate the quality measure he or she is interested in by name, and to allow the preprocessor to fill in all the necessary tag names and expressions. This is achieved in QXQuery by the automatic generation of an additional pre-defined function for each QV and data set pair:

```
define function hasQuality($item, $qvResult, $qassertion) {
  let $key = fn:concat(<ID_PREFIX>, $item/<KEY_PATH>)
  return $qvResult/EDItem[@DataRef = $key]
                 /AssertionValue[@AssertionTagName = $qassertion]
                 @AssertionTagValue
}
```

This function takes as parameters the item of data whose quality class we wish to retrieve, the QV result document from which we wish to retrieve it and the name of the quality assertion result that we are interested in[11]. The values for `<ID_PREFIX>` and `<KEY_PATH>` are inserted by the QXQuery preprocessor to create a complete XQuery function definition, and are the same as the values discussed earlier, in the generation of the data preparation function.

Using the `hasQuality` function, the quality filter from the example query given in Section 4 can be specified more succinctly as:

```
hasQuality($d, $qvResult, "PMFScoreClassifier") != "poor"
```

The writer of a QXQuery expression can assume the existence of this function without having to declare it. The preprocessor will construct all the necessary local definitions when transforming the query into an executable XQuery expression. In fact, we must generate one `hasQuality` function for each QV that is applied to each distinct data set type, and therefore the preprocessor must take care to give each one a unique name, and to rename applications of the generic `hasQuality` function in the query body so that the correct local definition of `hasQuality` is invoked in each case.

[11]This last parameter is needed because a given QV can be composed of several different quality assertions, and so may produce quality classifications for data items according to several different classification approaches. Thus, when accessing a given QV report, it is necessary to state which form of quality classification is required.

5.3 Example Queries

Taken together, the QXQuery extensions described above allow us to write our example quality-filtered proteomics query as follows:

```
<html>
<ul> {
  let $inputDoc := doc("proteomicsDB/PSF1-ACE2-CG.xml")
  using quality view "http://www.qurator.org#PMFScoreQV"()
        on $inputDoc//DBSearch
        with key /ProteinHit/Protein/accessionNumber
        as $qvResults
  for $d in $inputDoc//DBSearch, $h in $d/ProteinHit
  where not fn:contains($h/Protein/description, " actin ")
    and hasQuality($d, $qvResults, "PMFScoreClassifier") != "poor"
  return
    <li> accession: {data($h/Protein/accessionNumber)},
         gene: {data($h/Protein/geneName)},
         name: {data($d/username)},
         date: {data($d/idDate)}
    </li>
} </ul>
</html>
```

This query is translated by the preprocessor into the pure XQuery expression given in Figure 5 for execution using a standard XQuery processor, such as SaxonB.

A variant of this query illustrates how we can use the `hasQuality` function to return details of the quality classification of the query results, rather than filtering poor results from the output:

```
<html>
<ul> {
  let $exps := doc("proteomicsDB/ProjectResults.xml")
                                              //MassSpecExperiment
  for $exp in $exps
  using quality view "http://www.qurator.org#PMFScoreQV"()
        on $exp//DBSearch
        with key /ProteinHit/Protein/accessionNumber
        as $qvResults
  for $d in $exp//DBSearch, $h in $d/ProteinHit
  return
    <li> sample:  {data($exp/Analyte/sampleID)},
         gene:    {data($h/Protein/geneName)},
         quality: {data(hasQuality($d, $qvResults,
                                   "PMFScoreClassifier")}
    </li>
} </ul>
</html>
```

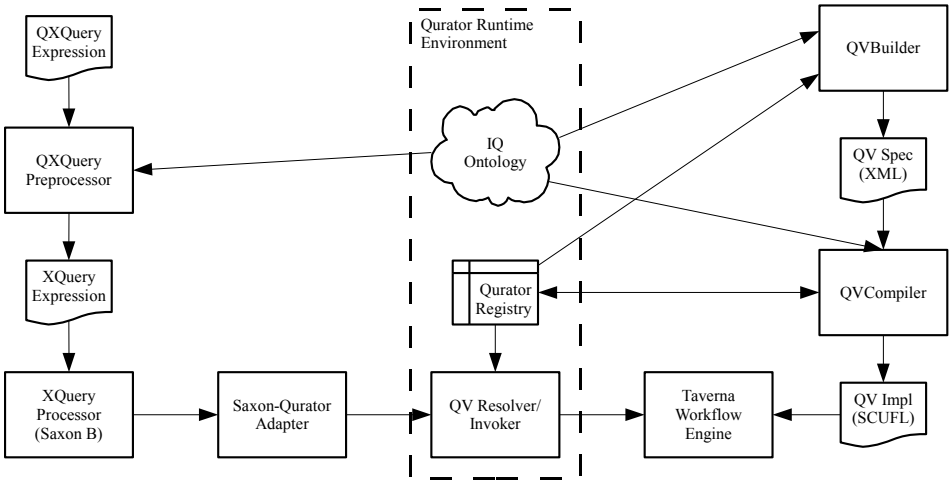

Fig. 6. QXQuery Execution Model

This query fetches all the experimental results from the given PedroDB file, classifies each individual protein identification using the `PedroQV` quality view, and displays for each identification the identifier of the biological sample in which the protein might have been observed, the name of the gene from which this protein would have been transcribed and the quality classification assigned to it. If the output of the quality view is a numerical score, rather than a class label, then it is also possible to rank query results (in decreasing order of quality, for example) using the XQuery `order by` clause.

5.4 QXQuery Execution Model

Our prototype implementation of QXQuery generates XQuery tailored for execution using the Saxon-B XQuery engine, and exploits the existing Qurator infrastructure for quality view creation, compilation and invocation [Missier et al. 2006]. The system architecture, and the specific invocation steps, are illustrated in Figure 6. On the right hand side of the diagram, the components involved in the construction of quality views are shown. At some point prior to query creation, it is assumed that the Qurator registry and ontology have been populated with components and complete quality views. The QVBuilder tool presents the set of available components in a visual manner, and uses the semantics in the ontology to help the user to construct a configuration of QEFs, QAs and CAs that is semantically legal. The output of this process is a platform-independent QV specification, in XML format, which is registered with Qurator and characterised in the ontology.

This specification can then be converted into an executable component suitable for execution within the user's preferred information environment. The figure shows just one example compiler, which converts QV specifications into quality assessment sub-workflows (in SCUFL format) suitable for inclusion within experiments encoded as Taverna scientific workflows [Hull et al. 2006]. Other compilers will generate other forms of executable QV. However, all compilers have the responsibility to

register the implementations they create with Qurator, so that they can be found by other potential users.

Assuming this pre-existing resource of quality views is already in place, the execution model for QXQueries is as follows (see left hand side of Figure 6). The QXQuery expression created by the query writer is first preprocessed as described earlier to create a pure XQuery expression with embedded calls to an externally defined function. This expression is then executed using an XQuery evaluation engine. The calls to the external function map to the Saxon-Qurator adaptor component, which we have constructed, and which is assumed to reside on the same machine as the XQuery engine. This straightforward component converts the external function call into a call to the Qurator QV invocation Web service (which can reside on any accessible machine) for execution of the QV referenced in the query with the provided inputs. The QV invoker looks for an implementation of the required QV in the registry, and (since, in this case, the QV is implemented using SCUFL) interacts with the Taverna engine to request execution of the relevant workflow and to receive its results.

6. CONCLUSIONS

Qurator quality views were designed for use within a service-oriented environment, rather than being invoked directly under the control of human query writers. Despite this, they have proven amenable to incorporation within queries, allowing highly-domain specific quality measures to be accessed by query writers, regardless of the capabilities of the data sources being queried, or the specific types of metadata they export. Some specific features are required of the supporting query language, as we identified in Section 4, but nothing that would not be reasonably expected of a modern query language. Using this approach, each data consumer can choose their own preferred approach to assessing quality, with the responsibility for locating (or computing) the required evidence being hidden within the quality view component itself.

However, the origin of the QV design makes its presence felt in the added complexities it imposes on queries and the demands it makes on human query writers in understanding the details of the QV interfaces. This results in lengthy queries, in which the original intent is obscured by details of quality view use. This problem can be alleviated, however, by the introduction of additional syntactic constructs within the query language, that present the illusion of a more abstract interaction with the quality view, thus simplifying the resulting queries and improving their readability. In the case of XQuery, only one new language construct was required, and even then only as syntactic sugar, since all the capabilities needed to query over quality views are present in the XQuery language itself.

Clearly, the execution process for QXQuery expressions is less efficient than if a single block of custom code had been written to evaluate the quality measure and incorporated directly into the query. However, we have found in practice that the communication and transfer costs of QV execution are insignificant compared with the costs of executing the domain-specific aspects of quality measurement, which would also be present in any custom-coded solution. Nor does the transformation from QXQuery to XQuery add any significant performance costs that would not

also need to be present in a custom solution, since the complexities it introduces concern, for the most part, details of tag names and function calls rather than adding expensive database operations. Typically, one loop and one join condition are added in order to process the quality report alongside the initial data set. If the user needs higher performance than can be achieved with QVs, then presumably he or she will be willing to invest the effort to custom code their quality measures. For other users, who wish to rapidly incorporate wisdom from others regarding quality measurement into their queries but who do not have the resources to design and code such measures for themselves, then QXQuery and quality views may represent an acceptable compromise.

So far, we have shown the range and usefulness of the quality view model in several application domains within the life sciences, but the wider applicability of the approach remains to be demonstrated. One obvious limitation of our present model is our dependence on LSIDs as an identification scheme between data sources. It will be much more difficult to keep the detailed format requirements of quality views hidden from the query writer in other domains, where such standard identifying schemes do not exist. Another potential limitation is our assumption that information consumers will find it worthwhile to invest in the creation of a wide range of quality views and their internal components, and that they can be reliably discovered and reused by others. It has yet to be proven whether the semantic metadata and high-level QV specifications that we currently provide are sufficient to enable the kind of sharing and reconfiguration that our approach aims to facilitate.

A further question that we have yet to explore is that of optimisation. At present, the quality views that we embed into QXQuery expressions are viewed by the query processor as black box components, despite the fact that declarative descriptions of their internal configurations can be accessed through the Qurator runtime. it should therefore be possible to improve the processing of quality-oriented queries by unfolding quality view configurations within the query expression before it is passed to the query evaluator. The use of cached and shared repositories of quality evidence and classifications produced by QVs is another potential optimisation route. Our current QVs have the capability to access cached results when they are available, but this option must be selected statically, at QV creation time. We are currently exploring how the decision to access previously cached evidence and quality scores can be made dynamically within QVs.

REFERENCES

Batini, C. and Scannapieco, M. 2006. *Data Quality: Concepts, Methodologies and Techniques.* Springer, Berlin.

Berti-Equille, L. 2004. Quality-Adaptive Query Processing over Distributed Sources. In *Proceedings of the 9th International Conference on Information Quality (IQ'04).* Boston MA, USA, 285–296.

Berti-Équille, L. 2007. Quality Awareness for Managing and Mining Data. Habilitation, L'Université de Rennes. June.

Blaha, M. 2001. A Retrospective on Industrial Database Reverse Engineering Projects. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01).* IEEE Computer Society Press, Stuttgart, Germany, 136–146.

Bouzeghoub, M. and Peralta, V. 2004. A Framework for the Analysis of Data Freshness.

In *Proceedings of International Workshop on Information Quality in Information Systems (IQIS'04)*, F. Naumann and M. Scannapieco, Eds. ACM Press, France.

Burgoon, L., Eckel-Passow, J., Gennings, C., Boverhof, D., Burt, J., Fong, C., and Zacharewski, T. 2005. Protocols for the Assurance of Microarray Data Quality and Process Control. *Nucleic Acids Research 33,* 19, e172.

Carlige, B. et al. 2004. Potential for False Positive Identifications from Large Databases through Tandem Mass Spectrometry. *Journal of Proteomics Research 3*, 1082–1085.

Civan, A. and Pratt, W. 2006. Supporting Consumers by Characterising the Quality of Onlinee Health Information: a Multidimensional Framework. In *Proceedings of 39th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 88a.

Dasu, T. and Johnson, T. 2003. *Exploratory Data Mining and Data Cleaning*. John Wiley, New York, USA.

Elmagarmid, A., Ipeirotis, P., and Verykios, V. 2007. Duplicate Record Detection: a Survey. *IEEE Transactions on Knowledge and Data Engineering 19,* 1 (Jan.), 1–16.

Embury, S., Sampaio, S., Missier, P., and Greenwood, R. 2007. The Syntax and Semantics of QXQuery. Tech. rep., School of Computer Science, University of Manchester. Available from `www.qurator.org`.

Führing, P. and Naumann, F. 2007. Emergent Data Quality Annotation and Visualization. In *Proceedings of 2007 International Conference on Information Quality (IQ'07)*. Cambridge, MA, USA.

Garwood, K. et al. 2004. PEDRo: a Database for Storing, Searching and Disseminating Experimental Proteomics Data. *BMC Genomics 5,* 1 (Sept.).

Heim, S., Hahn, K., Sämann, P., Fahrmeir, L., and Auer, D. 2004. Assessing DTI data quality using bootstrap analysis. *Magnetic Resonance in Medicine 52,* 3, 582–589.

Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M., Li, P., and Oinn, T. 2006. Taverna: a Tool for Building and Running Workflows of Services. *Nucleic Acids Research 34,* Web Server issue, W729–W732.

Korn, F., Muthukrishnan, S., and Zhu, Y. 2003. Checks and Balances: Monitoring Data Quality Problems in Network Traffic Databases. In *Proceedings of 29th International Conference on Very Large Databases (VLDB'03)*. ACM Press, Berlin, Germany, 536–547.

Lee, Y., Strong, D., Kahn, B., and Wang, R. 2002. AIMQ: a Methodology for Information Quality Assessment. *Information and Management 40,* 2 (Dec.), 133–146.

Martinez, A. and Hammer, J. 2005. Making Quality Count in Biological Data Sources. In *Proceedings of the International Workshop on Information Quality in Information Systems (IQIS'05)*. ACM Press, Baltimore, USA, 16–27.

Medjahed, B., Bouguettaya, A., and Elmagarmid, A. 2003. Composing Web Services on the Semantic Web. *VLDB Journal 12,* 4, 333–351.

Mihaila, G., Raschid, L., and Vidal, M.-E. 2000. Using Quality of Data Metadata for Source Selection and Ranking. In *Proceedings of the International Workshop on Web Databases (WebDB'00)*. Dallas, USA, 93–98.

Milano, D., Scannapieco, M., and Catarci, T. 2004. Quality-Driven Query Processing of XQuery Queries. In *Proceedings of CAiSE 2004 Workshop on Data and Information Quality*, J. Grundspenkis and M. Kirikova, Eds. Vol. 2. Riga, Latvia, 78–89.

Missier, P. 2008. Modelling and Computing the Quality of Information in e-Science. Ph.D. thesis, School of Computer Science, University of Manchester.

Missier, P., Embury, S., Greenwood, R., Preece, A., and Jin, B. 2006. Quality Views: Capturing and Exploiting the User Perspective on Data Quality. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, U. D. *et al.*, Ed. ACM Press, Seoul, Korea, 977–988.

Missier, P., Embury, S., Hedeler, C., Greenwood, R., Pennock, J., and Brass, A. 2007. Accelerating Disease Gene Identification Through Integrated SNP Data Analysis. In *Proceedings of 4th International Workshop on Data Integration in the Life Sciences (DILS'07)*, S. Cohen Boulakia and V. Tannen, Eds. Springer, Philadelphia, PA, USA, 215–230.

Naumann, F. 2002. *Quality-Driven Query Answering for Integrated Information Systems*. LNCS 2261. Springer, Berlin, Germany.

Naumann, F., Leser, U., and Freytag, J. 1999. Quality-Driven Integration of Heterogeneous Information Systems. In *Proceedings of 25th International Conference on Very Large Databases (VLDB'99)*. Morgan Kaufmann, Edinburgh, Scotland, 447–458.

Peralta, V. 2006. Data freshness and data accuracy: A state of the art. Tech. rep., Universidad de la Republica, Uruguay.

Pipino, L., Lee, Y., and Wang, R. 2002. Data Quality Assessment. *CACM 45,* 4 (Apr.), 211–218.

Preece, A., Jin, B., Missier, P., Embury, S., Stead, D., and Brown, A. 2006. Towards the Management of Information Quality in Proteomics. In *Proceedings of 19th IEEE International Symposium on Computer-Based Medical Systems (CBMS'06)*. IEEE Computer Society Press, Salt Lake City, US, 936–940.

Preece, A., Jin, B., Pignotti, E., Missier, P., and Embury, S. 2006. Managing Information Quality in e-Science Using Semantic Web Technology. In *Proceedings of 3rd European Semantic Web Conference (ESWC06)*. LNCS 4011. Springer, 472–486.

Raman, V. and Hellerstein, J. 2001. Potter's Wheel: an Interactive Data Cleaning System. In *Proceedings of 27th International Conference on Very Large Databases (VLDB'01)*, P. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. Snodgrass, Eds. Morgan Kaufmann, Roma, Italy, 381–390.

Redman, T. 1996. *Data Quality for the Information Age*. Artech House, Boston, USA.

Redman, T. 1998. The Impact of Poor Data Quality on the Typical Enterprise. *Communications of the ACM 41,* 2, 79–82.

Sampaio, S., Dong, C., and Sampaio, P. 2005. Incorporating the Timeliness Quality Dimension in Internet Query Systems. In *Proceedings of the WISE 2005 Workshops*, M. Dean et al., Eds. LNCS Vol. 3807. Springer Verlag, 53–62.

Scannapieco, M., Vigillito, A., Marchetti, C., Mecella, M., and Baldoni, R. 2004. The DaQuinCIS Architecture: a Platform for Exchanging and Improving Data Quality in Cooperative Information Systems. *Information Systems 29,* 7, 551–582.

Simmhan, Y., Plale, B., and Gannon, D. 2006. Towards a Quality Model for Effective Data Selection in Collaboratories. In *Proceedings of 22nd International Conference on Data Engineering Workshops (ICDEW'06)*. IEEE Computer Society Press.

Stead, D., Preece, A., and Brown, A. 2006. Universal Metrics for Quality Assessment of Protein Identifications by Mass Spectrometry. *Molecular and Cell Proteomics 5,* 7, 1205–1211.

Taylor *et al.*, C. 2003. A Systematic Approach to Modelling, Capturing and Disseminating Proteomics Experimental Data. *Nature Biotechnology 21*, 247–254.

Topaloglou, T. 2006. Informatics Solutions for High-Throughput Proteomics. *Drug Discovery Today 11,* 11/12 (June), 509–516.

Wang, R. and Strong, D. 1996. Beyond Accuracy: what Data Quality Means to Data Consumers. *Journal of Management Information Systems 12,* 4, 5–34.

Wang, R. Y. 1998. A Product Perspective on Total Data Quality Management. *Communications of the ACM 41,* 2, 58–65.

Weis, M. and Manolescu, I. 2007. Declarative XML Data Cleaning with XClean. In *Proceedings of 19th International Conference on Advanced Information Systems Engineering (CAiSE'07)*, J. Krogstie, A. Opdahl, and G. Sindre, Eds. Lecture Notes in Computer Science, vol. 4495. Springer, 96–110.

Winkler, W. 2004. Methods for Evaluating and Creating Data Quality. *Information Systems 29*, 531–550.

Winkler, W. 2006. Overview of Record Linkage and Current Research Directions. Statistical research report series rr2006/02, US Bureau of the Census, Washington D.C., USA.

Received September 2007; revised Month Year; accepted Month Year