DQ²S - A Framework for Data Quality-Aware Information Management

Sandra de F. Mendes Sampaio[a], Chao Dong[b] and Pedro Sampaio[c1]

[a] School of Computer Science, The University of Manchester, Oxford Road, Manchester, UK, M13 9PL,
s.sampaio@manchester.ac.uk
[b] Innovation Technology Department – China Youth Development Foundation - 51 Wangjing West Road, Chaoyang District, Beijing, 100102, P.R.China
dongchao@cydf.org.cn
[c] Manchester Business School, The University of Manchester, Booth Street West, Manchester M15 6PB
p.sampaio@manchester.ac.uk

Abstract

This paper describes the design and implementation of the Data Quality Query System (DQ^{2}S), a query processing framework and tool incorporating data quality profiling functionality in the processing of queries involving quality-aware query language extensions. DQ^{2}S supports the combination of performance and quality-oriented query optimizations, and a query processing platform that enables advanced data profiling queries to be formulated based on well established query language constructs, often used to interact with relational database management systems. DQ^{2}S encompasses a declarative query language and a data model that provides users with the capability to express constraints on the quality of query results as well as query quality-related information; a set of algebraic operators for manipulating data quality-related information, and optimization heuristics. The proposed query language and algebra represent seamless extensions to SQL and relational database engines, respectively. The constructs of the proposed data model are implemented at the user's view level and are internally mapped into relational model constructs. The quality-aware extensions and features are extremely useful when users need to assess the quality of relational data sets and define quality constraints for acceptable data prior to using candidate data sources in decision support systems and conducting big data analytical tasks.

Keywords: Information Management, Data Quality, Query Language Extensions, Data Profiling, Decision Support Systems, Big Data.

1. Introduction

According to IDC, an explosive growth in the demand for "Big Data" analytics is predicted as the digital universe continues to expand (Gantz & Reinsel, 2012). To capture the opportunities arising from big data, data quality issues need to be addressed to enable the execution of the analytical tasks and tools to support users in unlocking the valuable knowledge patterns available from querying large data sets. Therefore, research in data quality including traditional techniques for data cleansing, assessment and measurement of data quality need to be adapted to this new trend.

The challenges that big data imposes over data quality research and practice include not only the fact that, if data is truly "big", then traditional main memory methods become unsuitable for use, but also the fact that the data being used is not well known to its users.

[1] Corresponding Author

The latter can be addressed by profiling the data to reveal errors and guide data cleansing or data repairing processes. The result of data profiling is the generation of metadata that can also be used to measure the quality of a data set against previously established constraints and data quality benchmarks. While basic data profiling can be performed by simply eye-balling database tables, more advanced data profiling can be performed by key-word-searching in data sets or using dedicated data profiling tools (Naumann, 2014).

To facilitate the process of profiling and analysing large data sets, this paper proposes a comprehensive framework for combining data management with data profiling for data cleansing, by allowing users to seamlessly model and store quality-related data properties in the database and associate them with their respective data; this information is used to profile the data in preparation for data cleansing, considering multiple data quality dimensions. The framework provides the following advantages:

- Allows users to query not only data, but also its quality-related information and generate a profile. This is achieved by the use of the DQ^2L query language, a seamless extension to SQL, and an interface to the database which hides from users the complexities related to the presence of the stored quality-related information and data profiling algorithms. Both language and interface provide an intuitive means of associating data with quality-related data via an extended relational data model whose constructs are internally mapped into relational model constructs.
- Allows users to request the quality of data to be measured according to an extensible set of data profiling algorithms applied over the available quality-related information. These algorithms represent an extension to the database engine, but are dedicated to the calculation of data quality scores.
- Allows users to apply filters when querying the data, based on both the stored quality-related information and on data quality scores calculated during query processing.
- Provides a complementary functionality for information quality management that can be bundled as a seamless architectural extension to mainstream database management systems.

Regarding the information quality management perspective adopted by the framework discussed in this paper, latest research by Illari (Illari, 2014), whilst acknowledging that information quality (IQ) is generally defined as information that is 'fit for purpose', also indicates that IQ researchers agree that several dimensions of IQ, and even some aspects of IQ itself, are purpose-independent. Therefore, in our approach, we pursue to represent a set of IQ dimensions that are widely used across several application domains and which can be effectively and efficiently incorporated into an automated information management framework/system. Both perspectives are complementary and are often observed in practice where domain independent and domain specific tools and techniques are applied towards tackling complex data quality problems.

The key contribution arising from our approach is the capability to combine data management and advanced data quality profiling allowing users to profile their data by having data quality scores calculated during query processing. Users have control not only over the data profiling metadata that is generated, but also over how the metadata is generated. The framework also allows individual users to set their own quality constraints while querying the data, without imposing the same constraints to all users. These features are extremely useful when users need to assess the quality of relational data sets and define quality filters for acceptable data and a methodology for quality management that allows information quality to be queried and measured for different purposes prior to conducting big data analytical tasks (Floridi, 2014).

The paper also contributes to the literature by providing a comprehensive description of the key design decisions underpinning the framework, including data quality dimensions, high level query language extensions, query processing architecture, algebraic operators, mapping from query language constructs to algebraic query plans, assessment of query formulation simplicity of using the proposed extension compared to developing the queries in plain SQL. Other contributions of this paper include the following:

- Insights on the development of heuristics for optimizing query execution plans in the presence of algorithms for measuring data quality.
- A review of the main data quality dimensions and their respective measurement techniques.
- A discussion of three information quality management implementation scenarios using DQ^{2}S in the context of an e-Business application.
- A preliminary empirical evaluation of the framework, illustrating the elapsed time of query plans taking into account query optimization actions.

The paper is organized as follows: Section 2 describes background on a number of data quality dimensions and their respective measurement techniques. The measurement formulas presented in this section have an objective nature and can be applied in the context of a variety of application domains. They are used as basis for the DQ^{2}S data quality profiling algorithms. Section 3 describes the design of the proposed query language, DQ^{2}L, which is accompanied by a number of query examples and illustrations of the main features of the data quality model. Section 4 describes the architecture of DQ^{2}S and provides details on its query processing, illustrating the internal query representations. Section 5 discusses query optimization and how it can be extended with heuristics based on data profiling algorithms to measure data quality. Section 6 discusses the empirical evaluation of the framework. Section 7 describes related work and Section 8 summarises the paper and discusses future work.

2. Extending the Relational Model to Support Data Quality Measurement

This section describes the challenges of expressing and storing quality-related information and introduces the Data Quality Model (DQM), which enables users to represent and store quality-related properties associated with relational data.

2.1 The Quality Relation

The success of extensions to the relational model and the SQL language contemplating features and functionalities such as objects (Stonebraker & Moore, 1996), rules (Widom & Ceri, 1996), XML (Scardina, Chang, & Wang, 2004) and multimedia capabilities (Greenwald, Stackowiak, & Stern, 2013) among others has shown that piecemeal extensions to well known and widely accepted technologies often have better acceptability than completely new approaches developed from scratch.

Based on this observation, we follow the extension approach and propose a small number of features and functionalities to be seamlessly incorporated into the relational model, aimed at facilitating the storage, retrieval and manipulation of data with its associated quality-related information. The proposed constructs reflect the need for additional structures to support functionality related to the management of quality-related data properties. Therefore, as originally proposed in (Wang, Reddy, & Kon, 1995), special tables and attributes were designed to allow automatic association between data and its quality and are visible at the view level of the classical ANSI-SPARC architecture. However, these structures are mapped into standard relational model structures at the conceptual and physical levels.

The special tables are called Quality Relations (QRs) and are associated with the relational tables via attributes that have the form of a pair of values (`<attribute_value, FK_QR>`), where the first element of the pair represents the value of the attribute, and the second element represents a foreign key to a QR tuple in which quality information related to the attribute value is stored. The foreign key element is called quality key. Note that each element of the pair is atomic and associated with a domain of values. When an attribute of the table is not associated with a QR, i.e., an attribute for which there is no available quality information, the value of `FK_QR` is `NULL`. All attributes in both relational tables and QRs share this format. Attributes in a QR represent properties associated with a quality dimension. Note, however, that at the conceptual and physical levels of a database system, all tables and attributes are implemented with traditional relational constructs. Figure 2.1 shows properties related to the timeliness data quality dimension stored in the QR associated with attribute price of relation Product via quality key price_QID. This relationship enables navigation from a price value to its timeliness information. Also note that each tuple in relation Product has its price instance associated with a tuple in the QR; the QR attributes represent the available timeliness information for a price instance. Similarly, attributes of a QR can be further associated with other QRs.

2.2 Modelling Dimensions of Data Quality using Quality Relations

This section describes how classical data quality dimensions: accuracy, completeness, timeliness and reputation described in the data quality literature (Olson, 2003), (Wang, Ziad, & Lee, 2001) are represented in the relational data model extension implemented in DQ^2S. The modelled dimensions were chosen due to their general applicability across different data quality applications in domains such as e-Business, e-Science and

Geographical Information Systems. Along with the dimensions, a design for the related QRs is provided as an example in the context of an e-Business application.

2.2.1 Accuracy Dimension

While a recent data quality standard has been published towards defining accuracy and other quality dimensions in the domain of geographical information systems (Standard, 2013), previous work on the accuracy dimension of quality (e.g., (Han, Jiang, & Li, 2010) and (Han, Jiang, & Song, 2008)) focuses on the development of approaches for assessing and quantifying accuracy, and earlier initial work is dedicated to defining accuracy in a general context (Redman, 1997). Earlier general definitions are still applicable across a number of applications. An example of a general definition is the one introduced in (Redman, 1997), where accuracy is defined as the proximity of a value *v* to a value *v'* considered as correct in both content and form. The work in (Mecella et al., 2002) further distinguishes between syntactic accuracy and semantic accuracy, the former being defined as the closeness between *v* and *v'* where *v'* is the value considered syntactically correct; and the latter being defined as the closeness between *v* and *v'* where *v'* is the value considered semantically correct, i.e., a value that is consistent with respect to the real world.

While it is reasonably straightforward to check the syntactic accuracy of a data unit, it is often not feasible to check its semantic accuracy. For example, to syntactically validate the name of an English person, it may be enough to check the name against a dictionary of common English names. However, to semantically validate the name of an English person, verification using data sources that contain information about the person may be necessary, since the terms of comparison have to be derived from the real world (Mecella et al., 2002). For example, if *v* = Juhn and *v'* = John than *v* is low in syntactic accuracy because its value is not acceptable according to a dictionary of English names. Regarding semantic accuracy, if *v* = Robert and *v'* = John, then *v* has low semantic accuracy because, even though its syntactical value is acceptable, the person whose name is stored as Robert actually represents a person named John in the real world. Due to the feasibility challenges in processing semantic data accuracy this work focuses on syntactic accuracy. Two methods are typically used to measure syntactic accuracy, described as follows.

2.2.1.1 The Edit Distance Method

(Batini & Scannapieco, 2006) suggest the comparison function *EditDistance* to evaluate the closeness between the actual value *v* and the expected value *v'*, by calculating the number of steps (i.e., insertions, and/or deletions and/or replacements of digits or characters) for converting *v* into *v'*. For example, if the value for attribute name of the schema element Employee is *v* = John Smth and the expected value is *v'* = John Smith, then *EditDistance*(*v*, *v'*) = 1, since only one step is needed to convert *v* into *v'*, namely the insertion of 'i' between 'm' and 't'.

(Cong, Fan, Geerts, Jia, & Ma, 2007) further suggests the use of the ratio between the edit distance of *v* and *v'* and the maximum length between *v* and *v'*, to measure the similarity of *v* and *v'* enforcing the idea that longer strings with one-character difference between *v* and *v'* are closer than shorter strings with the same characteristic. The ratio is a value between 0 and 1, and the higher this ratio, the more expensive it is to convert *v* into *v'*. Thus, the accuracy quality can be calculated using Formula 1, where the `|v|` and `|v'|` represent the number of characters in the value held in `v` and in the value held in `v'`, respectively.

```
Accuracy(v) = 1 – [EditDistance(v,v') / max(|v|,|v'|)]
```
(1)

2.2.1.2 The Boolean Method

Unlike the Edit Distance Method, the Boolean Method measures accuracy with only two values as possible outcome: *yes* or *no*, representing accurate and inaccurate, respectively (Batini & Scannapieco, 2006), which can also be represented as *1* or *0*. If a value is contained in its corresponding reference value domain, then it can be considered as syntactically accurate (i.e., *yes*) otherwise, it will be deemed as inaccurate (i.e., *no*) (Batini & Scannapieco, 2006). Using the previous example, the name 'John Smth' is considered as to be inaccurate. Thus, in the Boolean Method, accuracy can be expressed as shown in Formula 2, where `v'` represents an element in the relevant reference dictionary.

```
Accuracy(v) = 1, if v = v'
                    OR
Accuracy(v) = 0, if v <> v'
```
(2)

It is worth pointing out that different users have different concerns about data accuracy. For instance, a user may be more concerned with whether a data unit is accurate or not, rather than how many steps should be taken to improve the accuracy of the data unit. For such a user, the Boolean method is more suitable. Another user may be interested in not only detecting inaccurate data units, but also improving their accuracy, preferring in this case the Edit Distance Method. The requirement for developing data quality frameworks that support users with different information production and consumption purposes relating to the same data set is further discussed in (Floridi, 2014).

The need for checking the accuracy of data values has raised the question of how these checks can be performed efficiently. Most commercial RDBMSs provide enforcing mechanisms for their business rules for checks to be carried out when data is entered into or updated in the database, for preventing invalid data. These enforcing mechanisms are typically implemented as functions stored in the database or as application programs running on the client side accessing local or remote sources of domain information. Therefore, for the accuracy dimension, there is no need to store quality related information in a QR.

2.2.2 Completeness Dimension

Completeness, together with accuracy and timeliness are often regarded as the three most used quality dimensions in specific application domains such as in Public Health Information Systems (Chen, Hailey, Wang, & Yu, 2014). While context-dependent definitions for the completeness dimension of quality can be found in (Biswas, Naumann, & Qiu, 2006), (Orme, Yao, & Etzkorn, 2007), (Tomic et al., 2015) and (Sebastian-Coleman, 2013), a generic definition of completeness, proposed in (Pipino, Lee, & Wang, 2002), is "the extent to which data are of sufficient breadth, depth, and scope for the task at hand". (Pipino, Lee, & Wang, 2002) also suggest that there are three main different types of completeness that can be measured by calculating the ratio between the number of incomplete items and the overall number of items, and subtracting this result from 1. The three types are described as follows:

- **Schema completeness:** At the most abstract level, completeness is the degree to which the properties of data (e.g. entities and attributes) are not missing from its associated schema.

- **Column completeness:** At the data level, completeness is the measure of the missing values for a specific property or column in a table. In the context of the relational model, it is also known as *attribute completeness*.

- **Population completeness:** This type of completeness represents the degree of missing values relative to a reference population. For example, if a column should contain at least one occurrence for each of the 50 states of the USA, but it contains only 43 states, then we have population incompleteness.

When assessing completeness of relational data, the semantics of NULL values are important and should be taken into consideration. A NULL value typically indicates a missing value; however, it is critical to investigate the reason why a value is missing before assessing completeness. A value may be missing due to any of the following three reasons (Atzeni, Batini, & Antonellis, 1993): (i) The value does not exist; (ii) The value exists, but is not available; and (iii) It is unknown whether the value exists. Cases (i) and (iii) should not be deemed as cases of incompleteness, since, in case (i), the value does not exist in the real world, and, in case (iii), the existence of the value is not known. Case (ii), however, is a case of incompleteness. For example, a NULL value for the mandatory attribute date of birth in relation Employee is a case of incompleteness since every employee must have a date of birth; thus, incompleteness is tenable for mandatory attributes. However, for non-mandatory attributes, such as e-mail address, cases (i) or (iii) may apply.

In addition, (Batini & Scannapieco, 2006) suggest that the following two assumptions should also be taken into account when assessing completeness of relational data: the closed world assumption (CWA) and the open world assumption (OWA). The CWA states that the values present in a relation represent all facts of the real world, i.e., the relation is population-complete; the OWA assumes that values in a relation are not able to represent all of the facts of the real world. By combining the semantics of NULL values with the CWA and OWA assumptions, it is possible to derive two models for the assessment of completeness, described as follows (Batini & Scannapieco, 2006).

- **OWA Assumption without NULL Values:** In this model, a reference data set containing all real world entities is compared against the relation being assessed, and the measure of completeness can be calculated by the ratio between the number of tuples that are present in the relation and the total number of tuples in the reference data set. In other words, the reference relation can be the complete relation R.

- **CWA Assumption with NULL Values:** This model assumes that the relation on focus is population-complete, i.e. there are no entities in the real world that are not present in the relation. However, due to presence of NULL values, other types of completeness are considered, and further classified regarding different data properties, described as follows:

    - **Value Completeness:** takes into consideration the presence or absence of value for an attribute. If an attribute value is not NULL, then it is complete. Otherwise, it is incomplete.

    - **Tuple Completeness:** considers a tuple as complete, if all of its fields are complete (value-complete). It can, therefore, be measured by the ratio between the number of non- NULL values in a tuple and the overall number of values in this tuple.

    - **Relation Completeness:** it considers the presence of non-NULL values in an entire relation. It can, therefore be measured by the ratio between the number of non-NULL values in the relation and the overall number of the values in the relation.

Assessment of schema and population completeness, as well as adoption of the OWA model prescribes the use of a reference data set against which the relation being assessed is to be compared. However, in application domains such as e-Business, such reference data sets are not always available and access to and integration of a potentially large number of external sources may be required. As the access and integration of a number of data sources is out of the scope of this research, we focus on the CWA with NULL Values Model for the assessment of tuple and relation completeness.

A possible implementation of the CWA with NULL values approach involves the automatic and periodic counting of non-NULL values and the saving of that in a histogram, rather than frequent counting and updating whenever information about completeness is needed, which can incur high maintenance costs. Therefore, for the completeness dimension, there is no need to store quality related information in a QR, except for the semantics behind the presence of a NULL value, which can impact on the completeness score.

2.2.3 Timeliness Dimension

The timeliness data quality dimension has grown in importance due to the widespread adoption of real-time information systems and large-scale sensor data management applications (Qin, Han, Mehrotra, & Venkatasubramanian, 2014). Work on the timeliness dimension of data quality has focused on the analysis of processes involved in the implementation of information services to understand their dependences and impact on the timeliness of the information provided (e.g., (Aktaş & Karğin, 2011) and (Yom-Tov & Diaz, 2011)), earlier work has focused on defining timeliness for use in broader contexts. A general definition can be found in (Ballou, Wang, Pazer, & Tayi, 1998) where timeliness is defined as the degree to which data is timely enough for its intended use. For instance, the data of a Sales Company will present low timeliness if the various branches of the company fail to record the latest regional sales information in time for a quarterly meeting. Moreover, in different application domains data sets can vary significantly in their update frequency. For example, a weekly update frequency for the prices displayed in an e-Shop comparison engine will usually satisfy the vast majority of shoppers; whilst for an investor willing to buy stocks, a one-hour update frequency will be unacceptably long, incurring low timeliness quality.

(Pernici & Scannapieco, 2003) and (Bouzeghoub & Peralta, 2004) have divided data into two main categories regarding update frequency: (i) *Static data*, defined as data which will not be updated during its lifecycle, for example data representing mathematical formulas, planet names, continent names, etc.; and (ii) *Dynamic data*, defined as data which has the possibility of being updated during its lifecycle. The update frequency of dynamic data can be user defined or can occur randomly. Considering update time intervals, dynamic data can be further divided into two main categories: (ii.a) *Seldom-update data*, defined as data considered to have a low update frequency (e.g., yearly or monthly), such as customers' home addresses and telephone numbers; and (ii.b) *Frequent-update data*, defined as data considered to have a high update frequency (every minute or second), such as sensor data from weather stations. For static data, a measure of timeliness may not be relevant in the context of many applications, since it is known that static data do not become outdated. In the design phase, when selecting which attributes are relevant for timeliness, only dynamic attributes may be considered.

(Ballou, Wang, Pazer, & Tayi, 1998) provide formulas to measure the timeliness of data, based on the concepts of *currency* and *volatility*, described as follows: Currency is defined as the age of data when it is delivered to the user. It is dependent upon three key factors: (i) the time when the data is delivered to the user as query results (*delivery time*), (ii) the time when the data was entered or modified in the database (*last update time*), and (iii) how old the data was when it was entered into the system (*age*). Based on the measurement formula proposed in (Ballou, Wang, Pazer, & Tayi, 1998), currency can be formally described as shown in Formula 3, where the currency of a unit of data *v* is a function of its delivery and last update times, and its age.

$$\text{Currency}(v) = \text{DeliveryTime}(v) - \text{LastUpdateTime}(v) + \text{Age}(v) \tag{3}$$

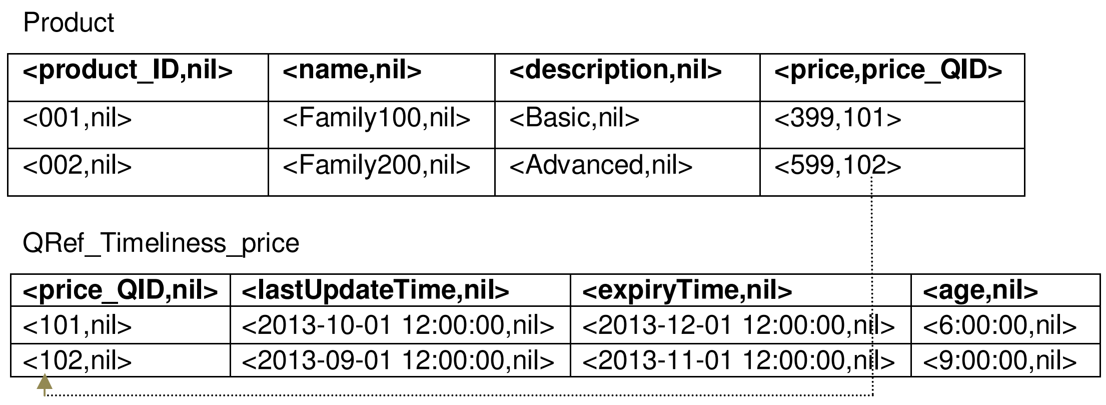
Volatility is defined as the length of time during which the data remains valid. It is also dependent upon three key factors: (i) the time when the data expires or becomes invalid (*expiry time*), (ii) the last update time of data, and (iii) the age of the data. Volatility can be formally defined as shown in Formula 4, where the volatility of a unit of data v is a function of its expiry and last update times, and its age.

$$\mathrm{Volatility}(v) = \mathrm{ExpiryTime}(v) - \mathrm{LastUpdateTime}(v) + \mathrm{Age}(v) \tag{4}$$

Based on the definitions for currency and volatility, timeliness of a data unit v can be formally defined as shown in Formula 5.

$$\mathrm{Timeliness}(v,s) = \{\max[(1 - \mathrm{Currency}(v)/\mathrm{Volatility}(v)),\ 0]\}^{s} \tag{5}$$

Exponent s in Formula 5 is a parameter that allows control of the sensitivity of timeliness to the currency-volatility ratio, and its value should be chosen depending on context. For high volatility the ratio is large, while for low volatility the ratio is small; furthermore, as the ratio increases, timeliness can be slightly (e.g., `s = 0.5`) or significantly (e.g., `s = 2`) affected, or neither (e.g., `s = 1`). To be able to measure the timeliness of a given unit of data using the formulas described above, it is necessary to store values for each of the elements (quality factors) in the formulas, namely *last update time*, *expiry time* and *age*, for that data unit. As quality factor *delivery time* has its value recorded only on the event of the delivery of the data unit to a user, its value will differ from event to event, therefore it should not be stored. The storage of timeliness information in relational databases has been discussed in more detail in (Dong, Sampaio, & Sampaio, 2006), where the concept of QR is introduced as a mechanism for storing quality information. For example, in relation `Product(product_ID, name, category, price)` attribute `price` is a dynamic data unit, as its value can change frequently over time. Attributes `name` and `category` can be deemed as static, since the value for each will not change frequently over time. As illustrated in Figure 2.1, timeliness related information is stored in the QR associated with attribute `price` via quality key `price_QID`, which enables navigation from a `price` value to its timeliness information.



Product

<product_ID,nil>	<name,nil>	<description,nil>	<price,price_QID>
<001,nil>	<Family100,nil>	<Basic,nil>	<399,101>
<002,nil>	<Family200,nil>	<Advanced,nil>	<599,102>

QRef_Timeliness_price

<price_QID,nil>	<lastUpdateTime,nil>	<expiryTime,nil>	<age,nil>
<101,nil>	<2013-10-01 12:00:00,nil>	<2013-12-01 12:00:00,nil>	<6:00:00,nil>
<102,nil>	<2013-09-01 12:00:00,nil>	<2013-11-01 12:00:00,nil>	<9:00:00,nil>

Figure 2.1: A QR for the timeliness of attribute `price` in relation `Product`.

2.2.4 Reputation Dimension

The reputation dimension of data quality was first defined in (Wang & Strong, 1996). This dimension received attention in the context of Healthcare (Deursena, Kostera, & Petkovica, 2008), Science (Gamble & Goble, 2011) and Web applications (Barbagallo, Cappiello, Francalanci, & Matera, 2010), (Peer, Vosgerau, & Acquisti, 2014). A broad definition is one that focuses on the reputation of data sources, indicating whether a data source is of high standing. Usually, long-established data sources have a higher reputation. For instance, stock information from the official NASDAQ website may have a higher reputation than the stock information from a local newspaper website.

There is as yet no widely accepted measurement mechanism for reputation in the literature, due to its highly subjective nature. However, since the notion of reputation is mainly based on users' experience accessing data sources, we have adapted the work discussed in (H. Wang, Yang, Zhao, & Gao, 2006), which proposes measuring the reputation of a data source using users' experience, expressed through an arbitrary number of data source reputation-related attributes, such as accessibility and reliability. As an example, Figure 2.2 shows instances of a relation associated with a data source (represented as the URL from which each instance was obtained), via a QR, and the data source's reputation-related attributes, which enable the calculation of the reputation for that data source.

Weights can be associated with each reputation-related attribute, to represent its relative level of importance among other reputation-related attributes. Each attribute and its weight have to be agreed on by the database administrators, data domain experts and end users upon metadata creation. Associated with each reputation-related attribute in the QR in Figure 2.2 is a score computed upon metadata creation. The computation is based on aggregating individual measures for each pair (data_source, reputation_attribute) submitted by end users of the data sources, since they have experience accessing the fitness for use of each data source.

As any number of end users may submit a score, an aggregate function such as average may be used to calculate an aggregate of the available scores per pair, as shown in Formula 6, where `Score_Attr(s,a)` denotes the overall score for the reputation attribute a, calculated by averaging all available scores for that attribute (`score[a,j]`), submitted by a number (m) of users considering data source s. In other words, the score of a particular attribute is computed as the average of a number of ratings submitted by users of a given data source. A domain [0, 1] can be given to the users of the data source for expressing their ratings or existing ratings have to be mapped in terms of that domain, where 1 represents the highest quality level and 0 represents the lowest.

Finally, the reputation measurement for a data source can be obtained by computing the weighted sum of the aggregated scores for each attribute as shown in Formula 7, where s denotes the data source whose reputation is being calculated, `Weight[a]` denotes the weight assigned to each attribute to reflect its importance among the other reputation attributes, `Score_Attr[s,a]` denotes the overall score for each attribute considering data source *s*, as described in Formula 6, and *n* denotes the number of attributes.

$$\texttt{Score\_Attr}(s,a) = \sum_{j=1}^{m} \texttt{score}[a,j] \;/\; m \tag{6}$$

$$\texttt{Reputation}(s) = \sum_{a=1}^{n} \texttt{Weight}[a] * \texttt{Score\_Attr}[s,a] \tag{7}$$

The example in Figure 2.2 illustrates a QR storing information about the data sources from which values for attributes `part_price` and `quantity_available` in relation `Part_Supply` were obtained. In this example, the used reputation attributes are the level of *accessibility* and *reliability* of the data sources according to the users of these sources. The stored scores represent an aggregation of all scores for each attribute submitted by users, as suggested in Formula 6. Using the scores, the reputation quality for the sources described in the QR can be calculated. Note that more than one instance can be associated with the same data source. The quality of the data source from which an instance was obtained indirectly reflects the quality of that instance, since a user may choose to discard an attribute value if it was retrieved from an unreliable data source, and users can discard a data source and choose to find another one, depending on its reputation.

2.3 Impact of Extensions on Integrity Constraints

In this section, the following rules are proposed to support the relational integrity constraints in the presence of the extensions described in the previous sections.

- Presence of NULL Values in QRs: NULL values for the quality factors mean that quality information is currently unavailable or it is unknown, rather than low quality. For instance, a NULL value for a reputation attribute indicates absence of input from users rather than a low quality score.

Part_Supply

<part_ID,nil>	<supplier_ID,nil>	<part_price,source_QID>	<quantity_available,source_QID>
<201,nil>	<301,nil>	<189.99,01>	<505,01>
<202,nil>	<301,nil>	<299.99,02>	<467,02>

QRef_Reputation_Part_Supply

<source_QID,nil>	<description,nil>	<accessibility,nil>	<reliability,nil>
<01,nil>	<www.alphacomputers.com,nil>	<0.5,nil>	<0.5,nil>
<02,nil>	<www.betacomputers.com,nil>	<0.8,nil>	<0.2,nil>

Figure 2.2: A QR for the reputation of the data sources associated with attributes `part_price` and `quantity_available` in relation `Supplier`.

- Atomicity of Attributes: Attributes of the form <attribute_value, FK_QR> are treated as an atomic unit, i.e., operations on such attribute cascades to its associated QR and quality factors. Consequently, tuple insertion, update and deletion are defined as follows:

  Tuple Insertion: Inserting tuples with non-NULL quality keys into a relation requires the associated QR also to have an inserted tuple with quality information, or requires the association of the inserted tuple with existing quality information in the QR.

  Tuple Deletion: Deleting tuples with non-NULL quality keys from a relation requires the associated quality information to be deleted from the QR, unless the information is associated with other tuples present in the database.

  Tuple Update: Updating tuples in a relation incurs the described tuple insertion and deletion actions.

3. The Data Quality Query Language (DQ^{2}L)

DQ^{2}L was designed to express data quality requests in relational databases as an extension to SQL, aiming at profiling the data with regard to a number of data quality dimensions. The extensions include an additional clause and a number of quality profiling functions to enable users to specify quality requests using declarative query expressions, and to measure the quality of intermediate and final query results, by calculating quality scores for individual data units, and to filter out low quality instances.

DQ^{2}L extends SQL with a single additional clause, called the `WITH-QUALITY-AS` clause, which is similar to the one proposed in (R. Y. Wang et al., 1995), (Dong, Sampaio, & Sampaio, 2006) in syntax and semantics. Therefore it supports the expression of constraints in the quality of the queried data and filtering capabilities. In addition, our framework provides facilities for the definition and implementation of quality profiling functions associated with the dimensions of quality against which data is measured, and which are a part of the database engine. For example, `Accuracy`, `Completeness`, `Timeliness` and `Reputation` are the currently implemented functions, discussed in Section 3.1.

The quality profiling functions can be called from the `WITH-QUALITY-AS` clause as well as the `SELECT` clause. This simplicity combined with the extensibility of the set of algorithms that can be applied to implement each quality dimension distinguishes our framework from previous work. For example, to add a new quality profiling function that measures the reputation of data sources from which instances of the database were obtained (see Sections 2.2.4 and 4.3.4 for more details about the Reputation dimension and operator), considering that an operator for that dimension already exists, the implementation of an algorithm to represent the new Reputation operator is necessary. It is required a single operator to perform all of the new reputation-related functionality, i.e., the functionality must not be divided between two or more new operators, because if it is, then there will be dependencies between the (sub-) operators that will need to be incorporated into the optimization process, causing extensions to become more complex. The insertion of the new operator into a list of all physical operators is also necessary, so that this new operator can be taken into consideration during query optimization. Because the profiling operators share similarities in their implementation, one operator can be used as a template for the implementation of new operators, not incurring new operator dependencies other than the ones already anticipated and incorporated into the optimizers, making unnecessary the addition of new optimization heuristics. Research on common data quality application domains indicates that the dimensions of quality that mostly satisfy data quality requests have been incorporated into DQ^{2}L (Wand & Wang, 1996). However, if a new dimension of quality is to be added into the system, then an extension to the DQ^{2}L syntax is necessary to add a new function to represent this new dimension, in addition to the physical level implementation of the new operator, and the insertion of this into the list of operators to be taken into consideration during logical and physical optimizations. These design decisions are often necessary in order to isolate the typical database user from the need to specify complex extensions to the functionality used by the query processor via advanced application programming interfaces. There is often a trade-off between flexibility and performance and in our framework we opted for a design that attempts to minimize the cognitive burden on the end user and to maximize performance towards scaling the approach for Big Data Scenarios, thus with some sacrifices to the flexibility provided to the end user when considering the addition of new quality dimensions. The complete DQ^{2}L syntax is specified in Appendix A using an extension of Backus Naur Form.

3.1 Querying with DQ^{2}L

In this section, examples of queries expressed in DQ^{2}L are given to illustrate the usability and expressiveness of the language when specifying requests or constraints on the quality of relational data. The example queries are based on the business processes of an e-commerce company that sells computer hardware. The focus is particularly on the business function of order fulfilment, where a variety of data quality problems that affect processes, such as sales, procurement, shipping, customer services, etc., are detected. In the following sections, a number of scenarios describing the company's main business rules are described along with common database queries associated with each scenario, in which data quality requests and constraints are specified. The database schema against which the queries are submitted is described in Figure 3.1. Note that no QRs appear in the schema, indicating that users are unaware of the presence of quality-related information associated with the data.

3.1.1 Scenario I: Sales

The Sales Department accepts two different types of customer orders: direct and indirect. Direct orders are submitted electronically via a Web site, and are less likely to be incomplete, since to be able to submit a direct order, customers have to fill in all of the form's fields. However, indirect orders are received from third party agencies varying in format, and so, are more likely to present completeness problems due to data being lost or transposed during translation.

Completeness checks can be regularly performed on all orders using a simple DQ^{2}L query, to identify incompleteness. Query 1, in Figure 3.2(a), shows how the checks can be expressed in DQ^{2}L. It retrieves order numbers and customer IDs for all orders whose completeness quality is less than 1 (i.e., less than 100%). The query is particularly useful when a broad profile of a database is required for auditing purposes, where the overall completeness of each relation is requested and individual tuples that need to be completed have to be identified. Function completeness is applied over every tuple in table Order, returning true if the tuple completeness (see definition in Section 2.2.2) is less than 1. Notice that the '`WITH QUALITY AS`' clause is used to specify quality-related filters. With standard SQL constructs, this query could not be easily expressed, since all attributes in each tuple would have to be tested for value completeness. Figure 3.2(b) illustrates this query in SQL. As all attributes are tested in the WHERE clause, for relations with a large number of attributes in their schemas, it becomes harder to write this query as all attributes need to be specified, imposing on the user an additional cognitive burden required to formulate the query. However, simpler completeness checks are possible using only SQL, for example, to retrieve all tuples in relation Order and leave for the database user the task of checking which tuples contain null values, or to retrieve all tuples in relation Order that have a null value for a couple of attributes only. Queries to check population completeness can also be expressed using SQL by the specification of predicates in the WHERE clause that require all tuples to have value "x" for attribute *a*, for example.

Subsequently, incomplete orders are put in a pending status until they are revised and corrected. For that, requests for further information are sent to the relevant customers or departments to confirm the order. Normally, a period of 14 days is given for the status of an order to change from pending to progressing. But half way into this period, the status of the order is checked and, if it is still pending, a further message is sent as a reminder about the order status. Query 2 in Figure 3.3(a) supports the process of checking the status of orders that have been pending for at least 7 days. This is done by finding all the orders that are currently pending and, among these, further selecting the orders whose status has remained unchanged for 7 days.

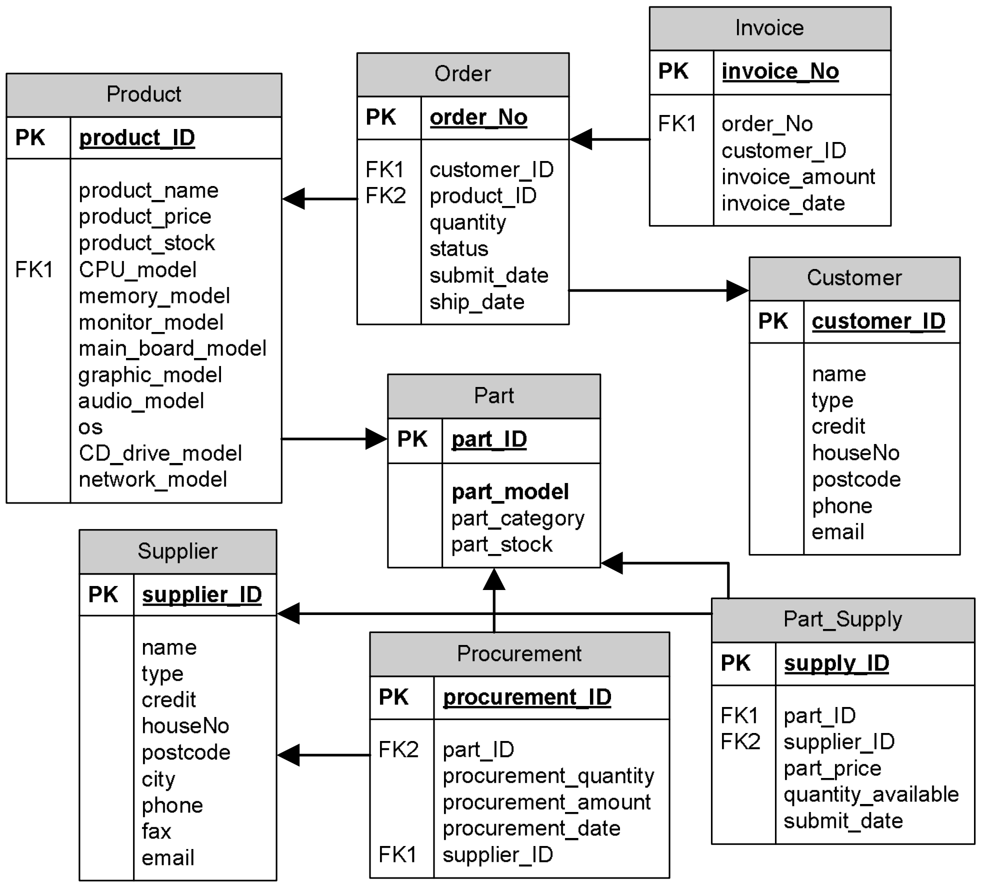


Figure 3.1: Database Schema used in all example queries.

Query 1: "Select all incomplete orders with their order numbers and customer IDs."
In DQ^{2}L:

```
SELECT order_No, customer_ID
FROM Order
WITH QUALITY AS COMPLETENESS(Order) < 1
```

Figure 3.2(a)**:** Query example in DQ^{2}L where the completeness function is used.

Note that Query 2 could not be easily expressed in plain SQL using the database schema shown in Figure 3.1, and to enable the user to express an equivalent plain SQL expression to address the information request, the hidden QRs would have to be made visible. Figure 3.3(b) illustrates Query 2 expressed in SQL, assuming that all QRs are visible to the user. The additional join(s), the decision about which Timeliness formula to use and the complexity associated with expressing the formula using SQL without any errors, represent the additional work that is avoided when DQ^{2}L is used. Note that, by using DQ^{2}L, the user can simply set a timeliness threshold for all pending orders and keep only the pending orders whose timeliness is less than or equal to 0.5.

Query 1: "Select all incomplete orders with their order numbers and customer IDs."
In SQL:

```
SELECT order_No, customer_ID
FROM Order`
WHERE quantity IS NULL OR
        submit_date IS NULL OR
        ship_date IS NULL OR
        status IS NULL;
```

Figure 3.2(b)**:** Query 1 expressed in SQL.

Figure 3.4 shows the QR associated with the `status` attribute of Table `Order`. Note that the timeliness model described in Section 2.2.3 is used to calculate the timeliness of orders. In this example, the 14-day deadline represents the `expiryTime` for the pending status of an order; the 7-day deadline for checking the status of an order represents the `deliveryTime` for the status of the order, since it is the time up to which the change in the order status is assumed to have happened without delay; the time when the status of the order was last updated represents its `lastUpdateTime`; as the semantics associated with the `age` of an order is expressed inside the context of the other quality-related properties, it is set to zero for all tuples. Calculations of `currency` and `volatility` are performed by subtracting attribute values from the QR `QRef_Timeliness_status`, using hours as unit. For example, for the tuple whose `order_No` is 302, `currency` is 168 hours (7 days) and `volatility` is 336 hours (14 days), giving a total `timeliness` of 0.5.

Query 2: "Select the orders that are pending and have been waiting to be validated for more than 50% of the total waiting time."
In DQ^{2}L:

```
SELECT order_No, TIMELINESS(status)
FROM Order
WHERE status = 'Pending'
WITH QUALITY AS TIMELINESS(status) <= 0.5
```

Figure 3.3(a): Query example in DQ^{2}L where the timeliness function is used in both the 'select' clause and the 'with quality as' clause.

3.1.2 Scenario II: Procurement Management

If orders in progress require the purchase of a large number of units of a specific PC part, then it is the responsibility of the procurement department to negotiate the supply of the parts from the available suppliers. In this context, it is important to obtain reliable information from the suppliers about prices and resources in stock to avoid failure in order fulfilment and loss of customers. Query 3 shown in Figure 3.5(a) reflects this scenario, representing a request on the reputation of the source of information about price of a specific part based on the experience of previous users of that source. The same query in SQL is shown in Figure 3.5(b). As for Query 2, the extra work that DQ^{2}L isolates the user from is proportional to the number of joins between relations and QRs, and, in this example, the number of reputation-related attributes the user wants to use with the associated weights. When DQ^{2}L is used, the system automatically decides which attributes and weights are to be used, and specifies which joins are to be performed.

Query 2: "Select the orders that are pending and have been waiting to be validated for more than 50% of the total waiting time."
In SQL:

```
SELECT order_No, GREATEST(1 -
(TIMEDIFF(deliveryTime, lastUpdateTime) /
TIMEDIFF(expiryTime, lastUpdateTime)), 0) as timelinessStatus
FROM  Order, QRef_Timeliness_status
WHERE  status = 'pending' AND GREATEST(1 -
(TIMEDIFF(deliveryTime,lastUpdateTime) /
TIMEDIFF(expiryTime,lastUpdateTime)), 0) <= '0.5' AND
Order.status_QID = QRef_Timeliness_status.status_QID;
```

Figure 3.3(b): Query 2 expressed in SQL.

Order

<order_No,nil>	**…**	**<status,status_QID>**	**…**
<301,nil>	…	<progressing,***01***>	…
<302,nil>	…	<pending,***02***>	…
<303,nil>	…	<pending,***03***>	…

QRef_Timeliness_status

<status_QID,nil>	**<lastUpdateTime,nil>**	**<expiryTime,nil>**	**<deliveryTime,nil>**	**<age,nil>**
<01,nil>	< 2013-09-07 13:00:00,nil>	< 2013-09-15 13:00:00,nil>	< 2013-09-08 13:00:00,nil>	<0,nil>
<02,nil>	< 2013-09-09 13:00:00,nil>	< 2013-09-23 13:00:00,nil>	< 2013-09-16 13:00:00,nil>	<0,nil>
<03,nil>	< 2013-09-02 13:00:00,nil>	< 2013-09-16 13:00:00,nil>	< 2013-09-09 13:00:00,nil>	<0,nil>

Figure 3.4: Extended relational schema for table Order, including the Quality Relation associated with attribute status.

Query 3: "Select the price and reputation of the source from which the price for part 201 was obtained, if the reputation score for the source is greater than 0.8."
In DQ^{2}L:

```
SELECT part_price, REPUTATION(part_price)
FROM Part_Supply
WHERE part_ID = '201' AND REPUTATION(part_price) > 0.8
```

Figure 3.5(a): Query example in DQ^{2}L where the reputation function is used in the 'select' clause.

Query 3: "Select the price and reputation of the source from which the price for part 201 was obtained, if the reputation score for the source is greater than 0.8."
In SQL:

```
SELECT part_price, ((0.5 * accessibility) + (0.5 * reliability)) as
reputation_part_price
FROM Part_Supply, QRef_Reputation_Part_Supply
WHERE Part_Supply.source_QID =
QRef_Reputation_Part_Supply.source_QID AND part_id = '201' AND
((0.5 * accessibility) + (0.5 * reliability)) > 0.8;
```

Figure 3.5(b): Query 3 expressed in SQL.

Figure 2.2 illustrates the association between Table `Supply_Part` and a Reputation QR in which scores on the `accessibility` and `reliability` of data sources from which information about available quantities and prices of computer parts are provided. Note that, in this example, the data sources are web-based and may or may not be completely reliable or accessible at all times. Other reputation-related attributes can be associated with these sources, including trustworthiness or update frequency.

3.1.3 Scenario III: Shipping Management

When delivering products to customers, correct information about delivery periods and addresses is important, as delays impact on customer satisfaction. Data obtained from third party agencies may differ in format and may not be syntactically accurate when stored in the database. For instance, using different data formats for customer address, e.g. postcodes, may result in incorrect or inexistent addresses, thus, address checks are necessary before shipment.

Illustrating the context of this scenario is Query 4 in Figure 3.6, expressing a check in the postcode of a customer to whom goods are to be delivered. The execution of this query involves the validation of a customer's postcode by looking it up in an accurate data source, such as an address book. In other words, the accuracy function call is translated in terms of an SQL query to retrieve data stored in another table followed by an accuracy evaluation using one of the methods described in Section 2.2.1.

Note that the accuracy evaluation method cannot be directly called from SQL, and has to be implemented as a separate function. Therefore, this query could not be easily expressed in SQL, and a more procedural implementation style is likely to be necessary. In Section 5, a discussion on the choice of evaluation method according to a number of criteria is provided.

Query 4: "Select the customer's name and the accuracy of his/her postcode for the customer with id equal to 10."
In DQ^{2}L:

```
        SELECT name, ACCURACY(postcode)
        FROM Customer
        WHERE customer_ID = 10
```

Figure 3.6: Query example in DQ^{2}L where the accuracy function is used in the 'select' clause.

The expression of queries in which multiple dimensions of quality are involved is also possible. As an example, Query 5 in Figure 3.7 requests information about the timeliness of an attribute and the completeness of a relation. The combination of multiple dimensions of quality in a single query, will make it harder to write the equivalent query in plain SQL, highlighting the advantages of our approach.

4. DQ^{2}S Architectural Overview

This section describes the main architectural components of DQ^{2}S, its query processing approach, including extensions to the relational database engine with data profiling operators and the query execution plan generation.

4.1 Design Approach and Main Components

When designing DQ^{2}S, the layered modular architecture of RDMSs was taken into consideration for the fulfilment of the following requirements: (i) no changes to the underlying host RDBMS; (ii) no interference between the DQ^{2}S components and the host's components; and (iii) easy porting of DQ^{2}S to other RDBMSs. These requirements were fulfilled by allowing the DQ^{2}S functionality to be a non-invasive complementary query processing engine (dual-path architecture), allowing users to submit SQL and DQ^{2}L queries alike via the same interface. The DQ^{2}L engine is used as an alternative to the host's engine, for the cases when DQ^{2}L query expressions are submitted.

The approach of having two engines to execute query expressions avoids any need for changes to the host RDBMS. As described in Section 4.2, during query processing a DQ^{2}L expression is translated into SQL form and processed by the DQ^{2}S optimizer. Requirement (iii) was fulfilled by developing the DQ^{2}S layer as a seamless extension to the host's architecture and supporting the communication between DQ^{2}S and the host via a JDBC interface providing unified access to a wide range of database back-ends (Taylor, 2003).

Query 5: "Select the supplier's ID, price and available quantity of all parts whose model include 'WM', (for example, 'WM19'), considering the Part-Supply tuples whose completeness score is greater than 0.8 and whose timeliness score for price is greater than 0.5."
In DQ^{2}L:

```
        SELECT PS.supplier_ID, PS.part_price, PS.quantity_available
        FROM Part_Supply PS, Part P
        WHERE P.part_model LIKE '%WM%' AND
                PS.part_ID = P.part_ID
        WITH QUALITY AS
                COMPLETENESS(Part_Supply) > 0.8 AND
                TIMELINESS(PS.part_price) > 0.5
```

Figure 3.7: Query example in DQ^{2}L where functions `Timeliness` and `Completeness` are both used in the 'with-quality-as' clause.

The main architectural components are illustrated in Figure 4.1 where the DQ^{2}S components are depicted in grey colour and described as follows:

- **Application Interface:** This is the interactive interface to formulate and submit queries as well as browse query results. Two query languages are supported: SQL and DQ^{2}L. Thus, there are two alternative paths to the underlying database through this unified interface, either of which is taken depending on the expression input by the user.

- **Pre-parser:** The role of the pre-parser is to distinguish between DQ^{2}L query expressions and SQL ones; an SQL query is sent to the host RDBMS for processing, while a DQ^{2}L one is sent to the DQ^{2}L processing component.

- **DQ^{2}L Component:** In this component, queries from users are further parsed, translated into a plan, optimized and executed; query results are evaluated, marked, filtered, and possibly ranked. In particular, a DQ^{2}L query is translated into a logical algebraic query plan with data quality assessment operations. Following that, the query optimization process takes place to construct an execution strategy from the logical plan. A detailed description of the query processing framework is presented in Section 4.2. Note the dashed arrow in Figure 4.1 representing the communication channel linking the DQ^{2}L component to the host RDBMS.

- **RDBMS:** The host RDBMS stores data and quality-related data as relations to answer both SQL and DQ^{2}L queries.

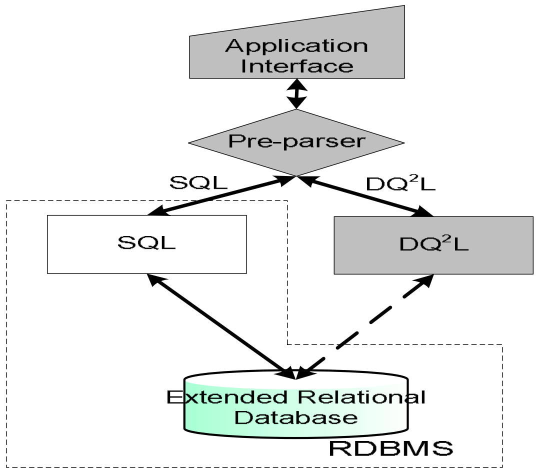


Figure 4.1: Main Architectural Components.

4.2 Query Processing

Figure 4.2 illustrates the functional components involved in the processing of DQ^2L queries. Once a DQ^2L expression takes the DQ^2L path, it is submitted for further parsing, during which it is transformed into a parse tree. The parse tree is translated into an algebraic expression by the *Logical Query Plan Generator* (LQPG), generating a *Logical Query Plan* (LQP) (please, refer to Section 4.4 for a more detailed description of this translation step). At this stage, the LQP contains solely relational algebra operators, namely `select`, `project`, `join`, etc. Next, the LQP is submitted to optimization, which is traditionally composed of logical optimization and physical optimization.

The logical optimization of DQ^2L queries is performed by the *Query Rewriter* (QRw), and is divided into two sub-phases, the first comprising traditional logical optimization and a second comprising an extended logical-level optimization focused on the insertion of data profiling operators into the LQP. During the first sub-phase, the LQP has operators reordered, removed and/or inserted, according to traditional heuristics developed for relational databases. Examples of such heuristics include: placing selection and projection operations so that these are executed the earliest as possible in the plan; placing the most selective joins to be executed before the least selective joins, etc. During the second sub-phase, data profiling operators are inserted into the LQP (e.g., `timeliness`, `accuracy`, etc.) causing operators to be further reordered or inserted into the LQP. The result is an extended LQP with explicit quality assessment functionality (LQP_{ext}). An LQP_{ext} for each example query in Section 3.1 is described in Section 4.3. This sub-phase does not have a counterpart in traditional relational query optimization.

Next, the generated LQP_{ext} is submitted to physical optimization, carried out by the *Physical Query Plan Generator* (PQPG), to generate a *Physical Query Plan* (PQP). During this phase, the operators in the LQP_{ext} are replaced with operators of the extended physical algebra, i.e. the set of algorithms that implement each of the operators in the logical algebra. A description of the process of selecting appropriate algorithms for each operator in the LQP_{ext} is discussed in Section 5.

Finally, the PQP is submitted to execution, carried out by the *Query Executor* (QE), which is also responsible for delivering query results to users. The QE, in turn, is composed of the *Data Evaluator* (DE) and the *Data Manager* (DM). While the DE comprises the database engine (described in Section 4.3), the DM is responsible for communicating with the host RDBMS. The unit of communication between the DM and the host RDBMS is a general and system-independent tuple-like data type, into which data coming from the host are mapped. Thus, the reuse of the DQ^2L query processor across relational products requires this mapping to be implemented making the DM the only component that needs to be rewritten when bundling DQ^2L with a host RDBMS.

4.3 Examples of Extended Logical Query Plans (LQPext)

For the application described in Section 3.1, four logical operators have been added to the database engine, namely `completeness, timeliness, reputation` and `accuracy`, to address the most common data quality problems (Paulson, 2000), (Segev, 2001), (Scannapieco, Mirabella, Mecella, & Batini, 2002). These four data profiling operators represent the four data quality dimensions described in Section 2.2 and the examples relate to the queries shown in Figures 3.2(a), 3.2(b), 3.3(a), 3.3(b), 3.5(a), 3.5(b) and 3.7. The following sections describe each of the four operators.

4.3.1 The Accuracy Operator

The accuracy operator, represented as `accuracy(R.a)`, calculates the accuracy score for each instance of attribute `a` in relation `R` and outputs a new relation R_{res}, whose schema is identical of that of `R`, except that R_{res} has an additional attribute associated with the calculated accuracy score. For example, consider Query 4 described in Figure 3.6. The LQP_{ext} for this query is shown in Figure 4.3, and an illustration of the tuples output by each operator in the LQP_{ext} is illustrated in Figure 4.4. Note from Figure 4.3 that while the select operator retrieves tuples from the `Customer` table and filters out the ones whose value for `customer_ID` is different to 10, the accuracy operator adds attribute `accuracy_postcode` to its input tuple.

4.3.2 The Completeness Operator

The completeness operator, represented as `completeness(List[R.`a_i`])`, calculates the completeness score for a list of attributes a_i in relation `R`. If all attributes in `R` are considered, then the operator can be represented as `completeness(R)`. When a sub-set of the attributes in `R` is considered, tuple completeness is calculated considering only the listed attributes. Once the completeness score for each tuple is calculated, the completeness operator adds a new attribute to each tuple containing the calculated completeness score. For example, consider Query 1 described in Figure 3.2(a). The LQP_{ext} for this query is shown in Figure 4.5. Note that two select operators are present in the plan: the first, at the leaf of the tree, is used to retrieve all tuples from relation `Order`, and the second is used to filter out `Order` tuples for which the completeness score is less than 1. The completeness score is calculated by the completeness operator and attached as a new attribute to each input tuple (`completeness_Order`).

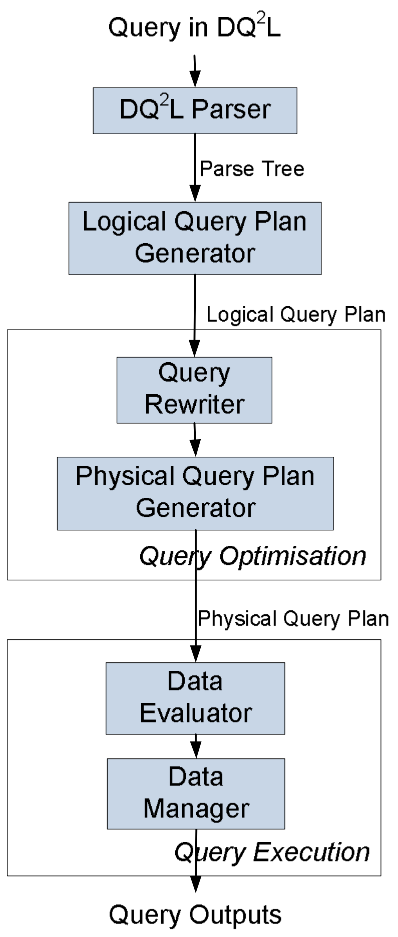
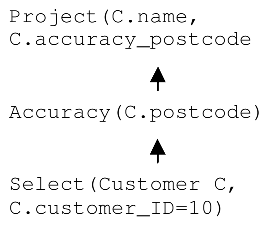


Figure 4.2: Query Processor Components of DQ^{2}S.



Project(C.name, C.accuracy_postcode
Accuracy(C.postcode)
Select(Customer C, C.customer_ID=10)

Figure 4.3: LQP$_{ext}$ for Query 4.

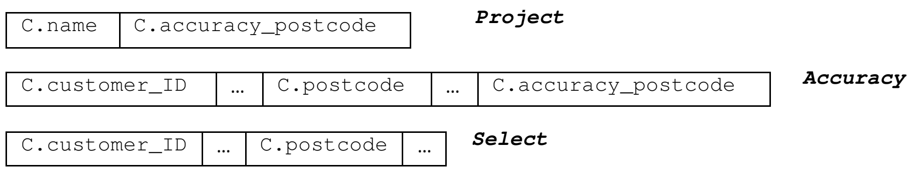


Figure 4.4: Tuple types output by each operator in the LQP$_{ext}$ for Query 4.

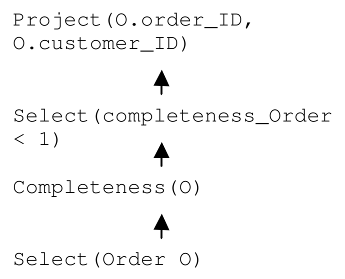

Project(O.order_ID, O.customer_ID)
Select(completeness_Order < 1)
Completeness(O)
Select(Order O)

Figure 4.5: LQP_{ext} for Query 1.

4.3.3 The Timeliness Operator

The timeliness operator, represented as `timeliness(R.a)`, receives an attribute `a` of relation `R` and calculates the timeliness score for each instance of `R.a`, attaching the scores onto `R` as the value of a new attribute. The timeliness score calculation formula was originally proposed in (Ballou, Wang, Pazer, & Tayi, 1998), and was adapted in this paper to be used in the context of the application described in Section 3.1. Figure 4.6 shows an LQP_{ext} for Query 2, illustrated in Figure 3.3(a). The timeliness operator is located between two select operators. The one at the leaf of the tree is used to retrieve all tuples from QR `Q_Ref`, and the second is used to filter out `Q_Ref` tuples for which the timeliness score is less than 0.5 (please, refer to Figure 3.4 for a description of the schema for relation Order and its QR). The timeliness score is calculated by the timeliness operator and attached as a new attribute to each input tuple. Note that the input tuples to the timeliness operator are `Q_Ref` tuples, despite the fact that the timeliness being calculated is that for attribute `Order.status`. The reason for placing timeliness before the join operator on the `Q_Ref` branch of the tree is performance improvement obtained from the decreased cost of the execution of the join due to the high selectivity of the select filtering on the timeliness scores, as discussed in Section 4.4.

4.3.4 The Reputation Operator

The reputation operator, represented as `reputation(R.a, List<attr,w>)`, receives as input an attribute `a` of relation `R` and a list of pairs composed of a reputation indicator (i.e., a reputation attribute) and its assigned weight, specified in a DQ^2L query expression (e.g., `<accessibility, 0.5>`). It uses this input to calculate the reputation score for each instance of `a`, and attaches the scores to `R` as the value of a new attribute. Figure 4.7 shows an LQP_{ext} for Query 3, illustrated in Figure 3.5(a).

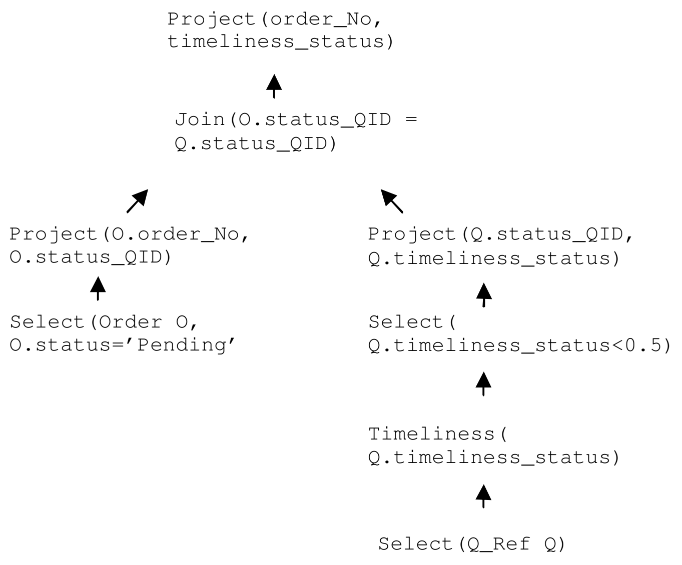

Project(order_No, timeliness_status)
Join(O.status_QID = Q.status_QID)
Project(O.order_No, O.status_QID)
Project(Q.status_QID, Q.timeliness_status)
Select(Order O, O.status='Pending'
Select(Q.timeliness_status<0.5)
Timeliness(Q.timeliness_status)
Select(Q_Ref Q)

Figure 4.6: $\mathrm{LQP}_{\mathrm{ext}}$ for Query 2.

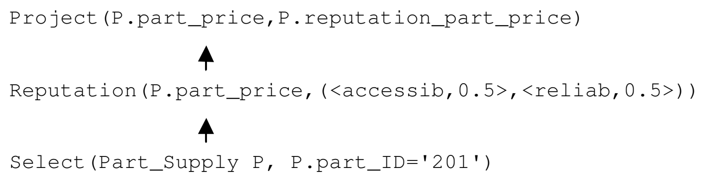

Project(P.part_price,P.reputation_part_price)
Reputation(P.part_price,(<accessib,0.5>,<reliab,0.5>))
Select(Part_Supply P, P.part_ID='201')

Figure 4.7: $\mathrm{LQP}_{\mathrm{ext}}$ for Query 3.

4.4 Mapping DQ^{2}L Query Expressions into Algebraic Query Plans

As described in Section 4.2, a parsed DQ^{2}L query tree is submitted to a number of transformations for the generation of a PQP. The following describes the four-step process of transforming a parsed DQ^{2}L query tree into a PQP. Figures 4.8 and 4.9 illustrate the application of steps 1 and 2, respectively, over Query Q2 (Figure 3.3(a)). A description of the schema for relation `Order` and its QR (`QRef_Timeliness_status`) are shown in Figure 3.4.

Note from Figure 4.8 that the FROM and WHERE clauses of the SQL expression appear expanded when compared to their counterparts in the DQ^{2}L expression in Figure 3.3(a), to include the following details: table `QRef_Timeliness_status` (or `QRef`, for short) and predicates '`timeliness(status)<=0.5`' and '`O.status_QID = Q.status_QID`'. The first predicate includes a call to a quality function involving the `Order.status` attribute, which accesses data stored in `QRef`. This function call is translated in terms of an operator from the extended algebra during STEP 3. However, at STEP 2, it is treated as an unchecked attribute. Also note that attribute '`Order.status_QID`' represents the foreign key attribute that associates tuples from table `Order` with tuples from `QRef`.

STEP 1. Mapping of a parsed DQ^{2}L query tree into a SQL query tree:
During this step, the FROM clause of a DQ^{2}L query is expanded to include all the QRs not explicitly mentioned in the query, but which are relevant to its execution; the WHERE clause is also expanded to include any predicates involving the QRs, specified in the 'WITH QUALITY AS' clause of the original DQ^{2}L expression; and the 'WITH QUALITY AS' clause is removed. The result is a SQL query tree that is parsed and type-checked against schema information. However, calls to data profiling functions, such as `timeliness()`, are not ignored and are, instead, treated as unchecked attributes.

STEP 2. Mapping of a SQL query tree into a logical (relational) algebra plan:
This step encompasses the traditional procedures for generating a (logical) relational algebra expression from a parsed query tree. At the end of this process, a LQP is generated, which contains operators of the relational algebra, but does not include any data profiling operators, since calls to data profiling functions continue to be ignored.

STEP 3. Mapping of a logical (relational) algebra plan into a logical DQ algebra plan:
In this step, data profiling function calls are replaced with appropriate attribute names and related (logical) data profiling operators are added to the query plan. This step is followed by logical optimization for DQ^{2}L queries, performed by the QRw component. The result of this step is an LQP$_{ex}$, as discussed in Section 4.2.

STEP 4. Mapping of a logical DQ algebra plan into a physical DQ algebra plan:
During this step, an LQP$_{ex}$ is mapped into a PQP, considering traditional physical optimisation rules and heuristics, as well as a few additional optimisation rules, discussed in Section 5.

"Select the orders that are pending and have been waiting to be validated for more than 50% of the total waiting time."
In SQL syntax:

```
SELECT O.order_No, TIMELINESS(O.status)
FROM Order O, QRef_Timeliness_status Q
WHERE O.status = 'Pending' and TIMELINESS(O.status) <= 0.5
and O.status_QID = Q.status_QID;
```

Figure 4.8: Equivalent SQL expression for Query 2.

During STEP 3, the LQP generated in STEP 2 is mapped into an LQP$_{ext}$, by replacing data profiling function calls with operators of the extended relational algebra. An LQP$_{ext}$ for Query 2 is illustrated in Figure 4.6. Note the following details: In the LQP expression (Figure 4.9), the predicate involving the `Order.status` attribute is a part of the select operator that retrieves tuples from the `Order` relation; in the corresponding LQP$_{ext}$

(Figure 4.6), this predicate is translated into the combination of a timeliness and a select operators which are executed before the join operator. This change is the result of optimization performed by the QRw component whereby the timeliness score for each tuple in relation `QRef` is calculated and the tuples are filtered based on this score before to the join between `Order` tuples and `QRef_Timeliness_status` tuples takes place, to minimise the cost associated with the join operator. Details of STEP 4 are discussed in Section 5.

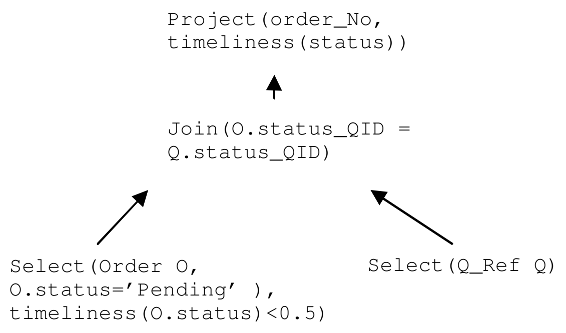

Project(order_No, timeliness(status))
Join(O.status_QID = Q.status_QID)
Select(Order O, O.status='Pending'), timeliness(O.status)<0.5)
Select(Q_Ref Q)

Figure 4.9: LQP for query expression in Figure 4.8.

5. Query Optimization in the Presence of Data Profiling Operators

This section shows how traditional heuristics developed for optimizing query execution plans in relational databases can be applied in the presence of data profiling algorithms. The heuristics focus on the reordering of operators in a query plan, performed during logical optimization, and are described in Section 5.1. Section 5.2 discusses the selection of appropriate algorithms for data profiling with a focus on specific quality dimensions and the task at hand. A discussion about the trade-offs between selecting suitable algorithms and performance issues is also provided.

5.1 Applying Traditional Heuristics over Extended Query Plans

Although heuristics are typically combined with cost models for efficient cost based optimization, cost models are not the main focus in this paper and shall be addressed in future work. In this section, we revisit the most common heuristics used in the pruning of the search space of possible execution plans for a relational query in the presence of the data profiling operators discussed in Section 4.

As suggested in Section 4, there is no dependence relationship between the data profiling operators in the sense that the attribute generated by an operator is not used in the execution of another one of those operators. However, dependencies between data profiling operators and traditional operators of the relational algebra may be identified. Figure 4.6 illustrates an example, where attribute `timeliness_status`, added to each input tuple by the timeliness operator, is involved in a selection predicate applied to the tuples prior to the join between tables `Q_Ref` and `Order`, indicating a dependency between timeliness and this particular instance of the select, which forces timeliness to be executed before this select.

Below, the relational heuristic rules of execution of selections and projections as early as possible in query execution for decreasing the size of intermediate results (Yu & Meng, 1998), (Garcia-Molina, Ullman, & Widom, 2013), (Connolly & Begg, 2014) are revisited to deal with LQP_{ext}.

- **Heuristic Rule 1:** If the attribute generated by a data profiling operator is not relevant to any select operator, execute the data profiling operator as late as possible.

  Because data profiling operators increase the size of the input tuples, pulling them up in the LQP_{ext} tree can minimize resource consumption and, hopefully, improve performance by keeping the size of intermediate results small for most of the query execution time.

- **Heuristic Rule 2:** If the attributes generated by data profiling operators are relevant to any selection predicate, execute the operators as early as possible.

  This rule is based on the traditional heuristic of placing selection operations as early as possible in the query plan to decrease the size of intermediate results. The earlier a data profiling operator is performed, the earlier the select operator that uses its data profiling attribute can be executed. Data profiling operators add one attribute to the input tuples, however a decrease in the number of input tuples represents a more significant performance improvement in most cases, especially when the number of input tuples is large and the selectivity of the predicate is high.

- **Heuristic Rule 3:** Projections that discard attributes relevant to the execution of data profiling operators should be pushed down the query plan for only as far as these attributes are not discarded before they are used.

  This rule extends the traditional heuristic of pushing projections down the query plan to improve system performance in case of memory constraints (Yu & Meng, 1998), (Garcia-Molina, Ullman, & Widom, 2013), since projections discard attributes that are irrelevant to the query execution.

- **Heuristic Rule 4**: Assuming a pipelined approach to implementation, for queries in which quality profiles involving multiple dimensions are requested, the use of query plans shaped as left-deep or right-deep trees can gain in performance since these queries incur multiple joins.

  This rule extends the traditional heuristic of exploiting pipelining to concurrently execute the operators of a left-deep or right-deep tree to improve system performance (Garcia-Molina, Ullman, & Widom, 2013).

To illustrate the application of the heuristic rules, consider the LQP$_{ext}$ in Figure 4.6. The application of Heuristic Rules 1 and 2 has ensured that the selection involving the timeliness attribute is executed before the join, by placing both the selection and the timeliness operators down on the left branch of the plan. Note that, although the input to the timeliness operator is attribute `status` of relation `Order`, its placement onto the left branch of the plan was necessary to enable the calculation of the timeliness score for attribute `status` as well as the filtering of tuples based on this score before the join between relation `Order` and its QR. The application of DQ Heuristic Rule 3 represents a performance improvement when dealing with the problem of memory constraints. In such case, projections are executed before the join to decrease the size of tuples in the LQP$_{ext}$ in Figure 4.6.

In the presence of predicates involving multiple data profiling attributes, the most selective predicate should be executed first, as suggested in the example illustrated in Figure 3.7, which shows a more complex query with two data profiling operators, two selections over data profiling attributes and two join operators. The corresponding LQP$_{ext}$ in Figure 5.1 shows the joins ordered according to predicate selectivity and all select operators being executed before the joins, to decrease runtime costs. Since pipelining is not supported in the current prototype, the application of Heuristic Rule 4 is not yet possible.

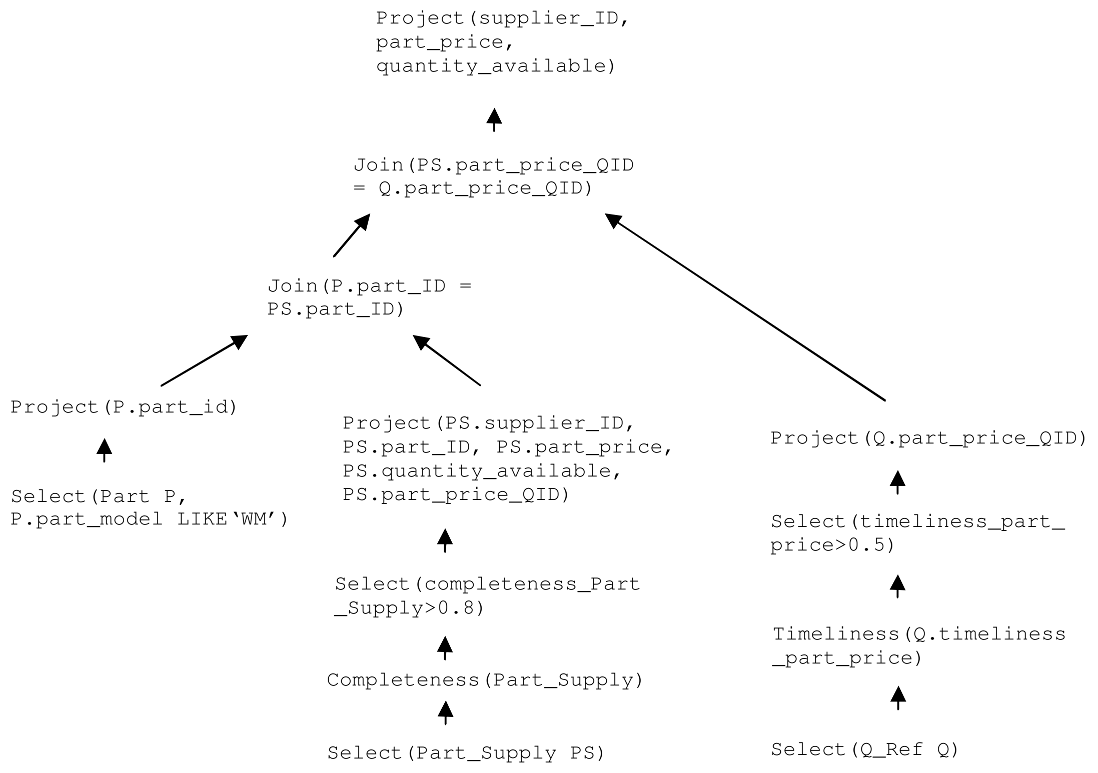

Project(supplier_ID, part_price, quantity_available)
Join(PS.part_price_QID = Q.part_price_QID)
Join(P.part_ID = PS.part_ID)
Project(P.part_id)
Select(Part P, P.part_model LIKE'WM')
Project(PS.supplier_ID, PS.part_ID, PS.part_price, PS.quantity_available, PS.part_price_QID)
Select(completeness_Part_Supply>0.8)
Completeness(Part_Supply)
Select(Part_Supply PS)
Project(Q.part_price_QID)
Select(timeliness_part_price>0.5)
Timeliness(Q.timeliness_part_price)
Select(Q_Ref Q)

Figure 5.1: LQP$_{ext}$ for Query 5 in Figure 3.7.

5.3 Physical Optimization

Adding data profiling capabilities to query processing incurs, in most cases, the execution overhead of an additional join operation between a relation and its QR (a detailed performance evaluation will be provided in future work). Moreover, the selection of the best algorithm for executing a data profiling operation should not be based on execution cost, but on the task at hand. For example, if two different methods for measuring accuracy are available, e.g., EditDistance and Boolean, then, even though the Boolean algorithm may be the cheapest in terms of running costs, the algorithm implementing the most suitable method for measuring the accuracy of a particular data unit for a particular user should be selected.

While a user should be allowed to choose the most suitable methods to measure the quality of a data item for the task at hand, some users may prefer for this choice to be made automatically. For this purpose, a mechanism for automatically identifying users' quality requirements was devised which uses ideas from view management in database systems. Based on information associating data units with quality measurement methods and user groups, a query optimizer is able to select the method that represents the preferred choice by the users of a certain group in most cases. For example, in an organization, different departments have different quality requirements, data access patterns and views of how quality should be measured, so each department can represent a user group. By collecting statistics on data access and selected method to measure quality to build user group profiles, and by enabling these profiles to be accessed and interpreted by the query optimizer, a quality-targeted optimization is achieved. These profiles are called *view_packages*.

Consider the view_packages described in Tables B.1 and B.2 in appendix B, associated with users from the Sales and the Shipping departments of the company described in our examples, respectively. Each view_package has an *`id`*, a *`name`*, and a list of the most *`frequently accessed attributes`* by its group users. The association of an attribute with a data quality measurement method represents the most appropriate choice at physical-level optimization regarding a dimension of quality. However, an attribute may be associated with a number of evaluation methods, if each such method is related to a different dimension of quality. Data profiling operators applied over attributes for which no association with evaluation methods is available should be selected based on response time, if multiple alternatives are present.

The set of all view_packages is extensible and is stored in the Data Dictionary, being accessed by the PQPG component. In addition, individual users are able to write their own view_package, which has priority over any the group package, and users can select their preferred quality measurement methods at query submission, via the user interface.

6. Preliminary Empirical Evaluation

This section describes a preliminary evaluation of the approach, with a focus on the optimization heuristics suggested in Section 5.

6.1 Test Queries and Environment

Queries 2 and 5, described in Figures 3.3(a) and 3.7, respectively, were used in this preliminary evaluation, since they share properties with all other queries presented in Section 3. Query 2 is simple, containing one profiling operator and one join. Query 5 is more complex, containing two profiling operators and two joins. For both queries, the scores generated by the profiling operators were used to filter the query results. Two query plans were tested for each query, the first being an optimized plan, resulting from the application of the heuristics described in Section 5, and the second, non-optimized, having the profiling operators added to the top of the query plan, as the query processing steps described in Section 4.4 suggest. Figures 4.6, 6.1, 5.1 and 6.2 show the query plans.

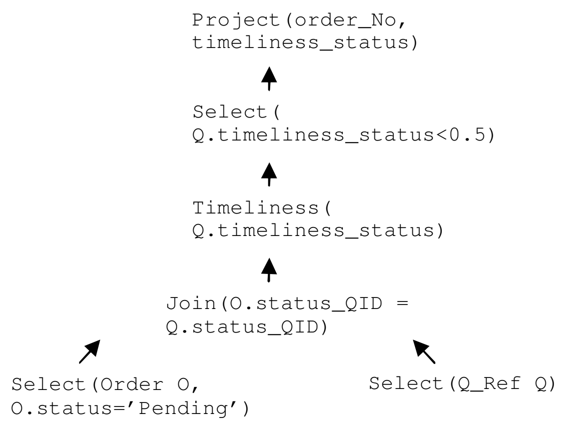

Project(order_No, timeliness_status)
Select(Q.timeliness_status<0.5)
Timeliness(Q.timeliness_status)
Join(O.status_QID = Q.status_QID)
Select(Order O, O.status='Pending')
Select(Q_Ref Q)

Figure 6.1: Non-optimized LQP_{ext} for Query 2.

Three MySQL relational databases with similar schemas were used in the experiments. Refer to the schema described in Figure 3.1 and the QRs described in Figures 2.1, 2.2, and 3.4 for the three databases. The first database contained 1,000 tuples in each relation (DB1), while the second contained 10,000 tuples (DB2) and the third 100,000 tuples in each relation (DB3). When populating the databases, we made sure that value distributions remained with the same proportions to keep the selectivity of the predicates constant as the size of the database increased.

Each query plan was run three times over each database and the average elapsed time was obtained for each pair ($queryPlan_i$, $DBsize_j$). The experiments were performed on an Intel (R) Core (TM) i5-4200U CPU @ 1.60 GHz 2.30 GHz machine, with 8.00 GB of RAM, running a 64-bit Windows 7 OS. In between runs both OS and MySQL database management system caches were flushed.

6.2 Results

Figure 6.3 shows the experimental results. In the figure's legend, four query plans are specified, two for Query 2 and two for Query 5. The notation (NO) and (O) distinguishes between the non-optimized and the optimized query plans, respectively. Axis x shows the variation in database size and axis y shows the average elapsed time associated with each pair (queryPlan$_i$, DBsize$_j$).

Note that, for both queries, the optimized query plans (O) presented a shorter elapsed time than their non-optimized counterparts (NO). Table C.1 in appendix C shows the average elapsed times for Query 2. For DB1, the optimized plan was 1.38 times faster than the non-optimized plan. For DB2, the optimized plan was 2.64 times faster than the non-optimized plan. And for the DB3, the optimized plan was 3.42 times faster than the non-optimized plan. As the size of the data increased from DB1 to DB2, the application of the heuristics showed a significant advantage, being that of the optimized plan over the non-optimized one 28% for DB1, and 62% DB2. However, this advantaged was less significant when the data was increased from DB2 to DB3 (only 71%), due to the intense paging carried out during query execution, as there was not enough memory for the input relations. The paging could have been minimized by a change in implementation approach from fully materialization of relations in memory to pipelining of tuples from the leaves of the query plan to its root.

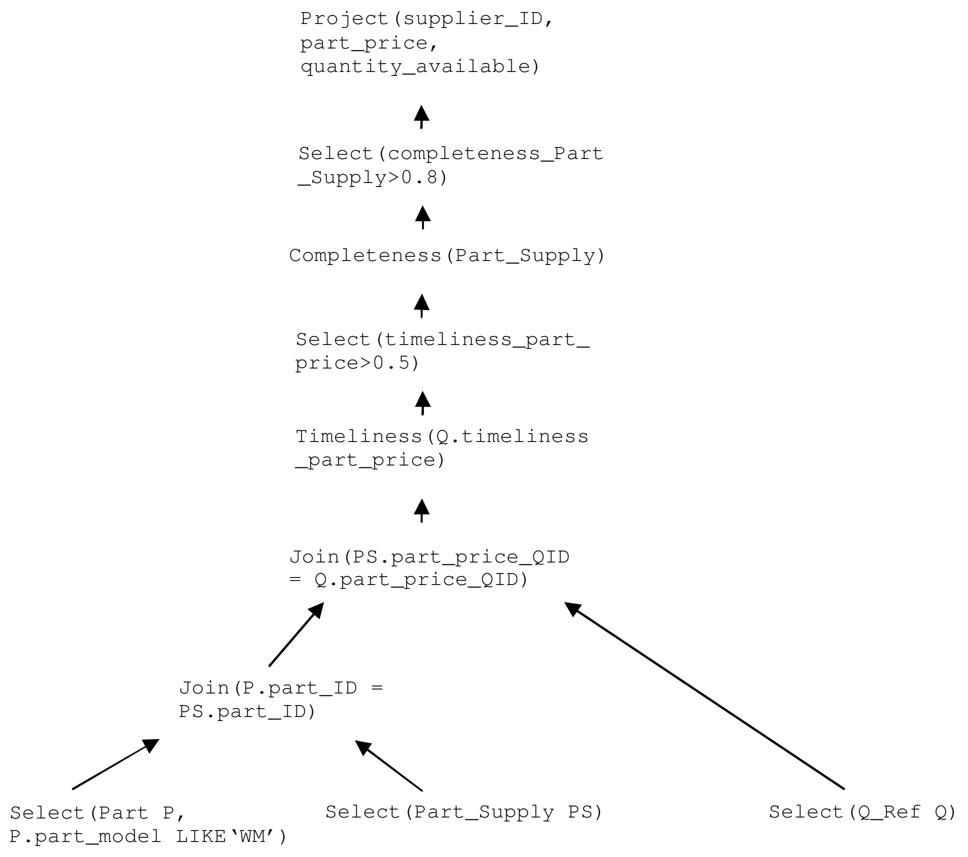

Project(supplier_ID, part_price, quantity_available)
Select(completeness_Part_Supply>0.8)
Completeness(Part_Supply)
Select(timeliness_part_price>0.5)
Timeliness(Q.timeliness_part_price)
Join(PS.part_price_QID = Q.part_price_QID)
Join(P.part_ID = PS.part_ID)
Select(Part P, P.part_model LIKE'WM')
Select(Part_Supply PS)
Select(Q_Ref Q)

Figure 6.2: Non-optimized LQP$_{ext}$ for Query 5.

Table C.1 in appendix C shows the average elapsed times for Query 5. For DB1, the optimized plan was 1.09 times faster than the non-optimized one. For DB2, the optimized plan was 1.58 times faster than the non-optimized plan. And for the DB3, the optimized plan was 1.49 times faster than the non-optimized plan. Similar to Query 2, the application of heuristics for Query 5 showed to be more advantageous when data increased from DB1 to DB2 (from 9% to 37% improvement) and less significant with the increase from DB2 to DB3 (from 37% to 33% improvement), also due to the intense paging carried out during query execution.

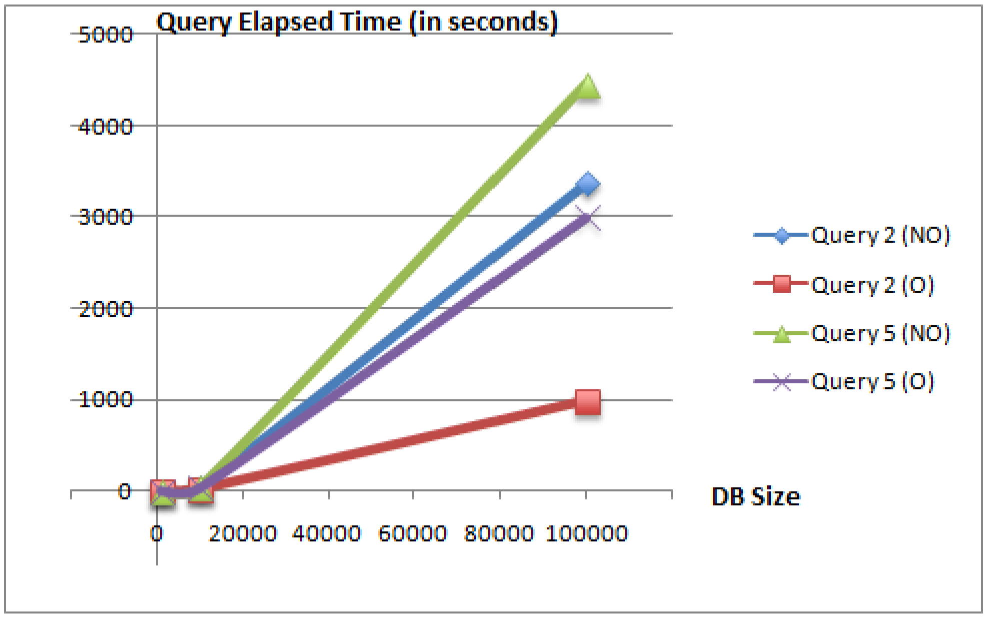


Figure 6.3: Experimental results.

The three heuristics described in Section 5 optimized the execution of Query 2 and Query 5 by decreasing the sizes of intermediate results. Considering the optimized query plan of Query 2 in Figure 4.6, the project operator on the left branch of the tree decreases the size of each tuple from relation `Order` by 6 attributes. Taking all tuples flowing from that branch, for DB1, this represents a decrease of (500 x 6) attributes, taking into account that the selectivity of predicate `O.status='Pending'` is 50%; for DB2 it represents a decrease of (5,000 x 6) attributes; and for DB3, it represents a decrease of (50,000 x 6) attributes. The project operator on the right branch of the tree, in turn, decreases the size of each tuple by 5 attributes. This decrease in tuple size not only saves memory space, but also makes the sequential search for an attribute in a tuple to become faster, causing the execution of several operators to become more efficient. The select operator that depends on the timeliness calculation on the right branch of tree discards 50% of the tuples flowing from that branch, making the join faster as its right input becomes significantly smaller.

Considering the optimized query plan of Query 5 in Figure 5.1, the project operator on the left branch of the tree decreases the size of each tuple from relation `Part` by 2 attributes. Taking into account that the selectivity of the predicate is 50%, that represents a decrease of (500 x 2), (5,000 x 2) and (50,000 x 2) for DB1, DB2 and DB3, respectively. The project operators on the right branch and the extreme right branch of the tree decrease the size of each of their input tuples by 2 and by 5 attributes, respectively. The select operator that depends on the completeness calculation has a selectivity of 0%, given that there were no incomplete tuples in relation `Part_Supply` in any of the databases, and so, the placement of this select in the query plan does not incur any benefit. Therefore, the join between relations `Part` and `Part_Supply` has the same cost as its correspondent in the non-optimized plan. This may be one of the reasons why the optimized plan for Query 2 represented a greater improvement over its non-optimized equivalent than it seems to be the case for Query 5. On the other hand, the select operator that depends on the timeliness calculation has a selectivity of 50%, making the join between relations `Part_Supply` and the timeliness `Q_Ref` faster.

This preliminary evaluation has shown that the combination of data profiling operators with relational algebra operators in the query engine, in particular, the application of select operations over data quality profiling scores, has provided an opportunity for further decreasing the sizes of intermediate results during query processing, improving overall elapsed time of query execution. We believe this to be a promising way of discarding lower quality data from query results using more complex criteria other than simple attribute value-based filtering of very large data sets, as the query is executed. Our future work includes increasing sizes of relations horizontally as well as vertically to further test the benefits of our approach to query processing, implement our operators using multi-pass algorithms and implement our framework using the MapReduce programming paradigm.

7. Related Work

Techniques and tools have been proposed to facilitate the task of data profiling, enabling users to (semi-) automatically profile their data, preparing it for optimization, as described in the work by Poosala *et al.* (Poosala, Haas, Ioannidis, & Shekita, 1996), knowledge discovery, as described in the work by Yao and Hamilton (Yao & Hamilton, 2008), or data repairing and cleansing, as described in the works by Fan *et al.* (Fan, Geerts, Jia, & Kementsietsidis, 2008), Bravo *et al.* (Bravo, Fan, & Ma, 2007) and Huhtala *et al.* (Huhtala, Kärkkäinen, Porkka, & Toivonen, 1999); the four previous references are mostly focused on detection of inconsistencies in relational data sets, identified as violations of dependencies between attributes.

More recent work in detection of data quality problems focused on violation of dependencies in relational data, e.g., functional, conditional functional and inclusion dependencies, includes the work by Beskales *et al.* (Beskales, Ilyas, Golab, & Galiullin, 2014), Dallachiesa *et al.* (Dallachiesa et al., 2013) and Geerts *et al.* (Geerts, Mecca, Papotti, & Santoro, 2013). Acknowledging the sole use of such dependencies may overlook other, more subtle, data quality problems, other work such as Fan *et al.* (Fan, Li, Ma, Tang, & Yu, 2012) and Yakout *et al.* (Yakout, Berti-Equille, & Elmagarmid, 2013) have relied on mechanisms to extend the range of data quality problems in consideration, including the use of master data and statistical machine learning. In the same spirit, our work seeks to offer an additional mechanism for revealing data quality problems to the database user by facilitating the querying of stored data quality information with which relational data is annotated, and by easing the measurement of data quality levels and data filtering, considering objective definitions of common data quality dimensions. To obtain data quality measurements in DQ^{2}S, quality-related metadata is associated with the stored data. From assessing the data quality literature, previous work that has also used metadata to describe quality-related information includes the following:

- Mecella *et al.* propose a framework for managing data quality in a distributed and cooperative information system, which includes an XML data model that enables each site in the system to export its data and quality data according to an agreed model. The framework also includes a centralised broker responsible for answering requests from all sites and serving a requesting site with the best quality units of data according to the exported information (Mecella et al., 2003);

- Shankaranarayanan and Cai propose a data quality management tool to be used in integration with Information Systems that support decision making tasks. The tool allows users to construct maps describing stages in the lifecycle of the target data, which include information about how the data has been composed and processed at each stage, taking completeness into account (Shankaranarayanan & Cai, 2006);

- Furber and Hepp (Furber & Hepp, 2011) propose a conceptual, ontology-based model for representing quality-related knowledge and requirements for Web data. In this model, data quality requirements are expressed as executable rules, enabling the cleansing of data sources via Semantic Web formalisms;

- Klein *et al.* (Klein, Do, Hackenbroich, Karnstedt, & Lehner, 2007) propose a model to propagate streams of data along with its corresponding quality information in sensor-based data servers. In addition, meta-model constructs are proposed, extending the relational model, in order to allow the storage of the streaming data in a relational database. It is not clear how query processing is affect by the proposed extensions;

- The work that is most related to ours is by Mutsuzaki *et al.* (Mutsuzaki et al., 2007) which propose extensions to SQL and the relational data model to enable users to access information about lineage of data as well as uncertainty in the context of the Trio database management system. Queries in the Trio Query Language (TrioQL) are translated into SQL statements, and are executed over standard relational tables in which lineage-related data is also stored (Mutsuzaki et al., 2007). In this approach, functional extensions to the relational model are added as stored procedures, and so, it cannot be combined and optimized together with other data manipulation functions.

Another work proposing query language extensions to facilitate data quality assessment is (Embury, Missier, Sampaio, Greenwood, & Preece, 2009), proposed by Embury *et al.*, where the focus is in the specification of domain-specific quality constraints to be enforced during query processing, considering XQuery as the target query language. In this work, the quality assessment can also be shared amongst users in the form of Web services. However, in this approach, each user must be able to implement its own services and/or select the most suitable service for the task at hand and explicitly call it using XQuery. The approach relies on a workflow-based execution of web services (quality views) without the use of mainstream query optimization strategies.

Regarding data quality management frameworks and systems, the DQAQS Framework by Yeganeh *et al.* (Yeganeh, Sadiq, & Sharaf, 2014) provides a comprehensive solution to the problem of quality-aware information management enabling users to express quality related queries on top of multiple relational data sources integrated via a wrapper-mediator architecture. The data quality dimensions are user defined and extensible, and profiling is based on both attribute and conjunctive conditions on the universe of possible queries issued over the global schema. The query processing approach involves the mediator translating the query into a set of query plans against different data sources and the utility of a query plan is estimated based on data quality of query results combined with other optimization factors such as execution time. Statistical formulas are used to estimate the data quality of a query plan based on the estimated data quality result against individual data sources that take part in the plan. Plans are ranked based on the overall utility function. The DQAQS mechanism to generate and update the profile is costly, both from disk space used and CPU load perspective. This presents a particular challenge to maintain and update the profiles especially when taking into account big data scenarios. Our approach to data profiling could be seen as complementary to theirs in that they could use DQ^{2}L to issue quality profiling queries to perform complex calculations and filter data of undesirable quality, particularly in cases where specific data quality metadata is not necessary. The mediator would be responsible for translating the equivalent SQL query into a set of plans and gathering results.

The SLIMPad system and application architecture by Delcambre *et al.* (Delcambre et al., 2001) is aimed at extending information sources with data management functionality for superimposed annotations representing underlining, bookmarks and cross-references. The data model for the annotations and data regarding annotations are managed externally from the host data sources. The digital “bundles” linked to the data sources can represent observations about data quality, however, the computations regarding quality measurements need to be manually calculated by the end user developing the annotations. This approach is not directly aimed at data quality management, however, due to the flexibility of the proposed annotations, the system can provide some level of support to tagging quality information to data sources and instances.

The pSQL system by Bhagwat *et al.* (Bhagwat, Chiticariu, Tan, & Vijayvargiya, 2004) proposes an annotation management system for relational databases providing support for “Where” data provenance analysis. Relations are extended with additional columns to provide annotations and schemes are developed for end users to specify how annotations should propagate. Annotations are restricted to attributes of tuples. An extension of SQL is developed to specify the propagation of annotations according to user-defined schemes. The system architecture has a translator module that rewrites pSQL queries into target SQL queries sent for execution at the host DBMS API. Tuples returned from the host DBMS are post-processed to merge annotations and provide annotated outputs for display to end-users. No optimization is performed on generated SQL queries. DQ^{2}S shares architectural features with pSQL, but differs in functionality.

Feature / Approach	Query Language Design	Query Processing and Optimization	Architectural Approach	Framework/ System Focus	Data Types Supported
DQ^{2}S	Development of high-level query language constructs to express quality preferences based on SQL extensions. Profiling is developed on the fly alongside end user queries	Translation algorithm maps queries with extended data quality requests and profiling information into mainstream SQL queries. Development and integration of algebraic query optimization into framework extensions	External quality-aware engine layered on top of relational DBMS with pre-processing and post-processing of host API results	Data quality management for standardized provider defined quality measures	Relational data with data quality metadata
DQAQS/ Squid (Yeganeh, Sadiq, & Sharaf, 2014)	Quality related user preferences are encoded in a model based on partial order prioritizations. Development of SQL extensions to capture user preferences on data quality in the form of partial orders	Mediator translates the query into a set of query plans against different data sources and utility of query plans is estimated based on data quality of query results combined with other optimization factors such as execution time	Wrapper/mediator quality aware data integration. DQ profiling involves significant CPU and storage overheads to generate and update profiles	Data quality management for user defined quality measures	Relational data with data quality metadata
Quality Views/ Qurator (Embury, Missier, Sampaio, Greenwood, & Preece, 2009)	User defined quality preferences incorporated as extensions to XQuery. Development of data quality language features that can be combined with other declarative query languages	Quality views incorporate declarative and procedural constructs for computing transformations of input data sets into quality-annotated output data sets. Workflow-based execution of web services (quality views) without the use of mainstream query optimization strategies	Quality views implemented as reusable web services with standard interfaces and annotations expressing the semantics of components and parameters specified using ontologies	Data quality management for user defined quality measures	Semi-structured and multi-model type support
SLIMPad (Delcambre et al., 2001)	Graphical user interface to visualize and navigate generic annotations. Data manipulation constructs for creating, updating, removing, storing and loading annotations	Based on allowing components to obtain pointers to referenced objects, which can be used for retrieval of annotations. Query processing and optimization of annotations not performed	Superimposed application implemented as plug-in to existing storage managers. Extensible architecture with minimal coupling interface to host	Tool to manage super-imposed information and annotations	Semi-structured and multi-model type support
pSQL (Bhagwat, Chiticariu, Tan, & Vijayvargiya, 2004)	Development of high-level query language constructs to manipulate data provenance information.	Translation algorithm maps queries with extended provenance information into mainstream SQL queries. No optimization is performed on generated SQL queries	Additional column storing annotations assumed to avoid lazy computation of provenance. Layered component with pre-processing and post-processing of host API results	"Where" provenance management and also data sensitivity and access control management	Relational data with data provenance and annotation metadata
Trio-One (Mutsuzaki et al., 2007)	Development of SQL extensions to express uncertainty and lineage requests	Encodes uncertainty and lineage present in the data model as relational tables and uses query rewriting for query processing	Layered on top of a relational base with extensions added as relations and stored procedures	Handling uncertainty and lineage	Relational data with metadata for lineage and uncertainty

Table 7.1: Data Quality Management Frameworks and Systems.

Table 7.1 illustrates differences and similarities between DQ^{2}S and some of the frameworks and systems found in the literature. The contributions discussed in Table 7.1 work primarily by building database quality profiling and management engines layered on top of database management systems.

Other complementary approaches and techniques addressing data quality issues include: inconsistency detection in integration of data from multiple sources from a computational complexity perspective, consistent query answering, duplicate record detection, database repairing, handling missing data and model-based techniques for data quality improvement, discussed as follows.

The problem of data inconsistency and failure to satisfy integrity constraints in relational databases resulting from integration of data drawn from a variety of sources has been studied by Chomicki and Marcinkowski (Chomicki & Marcinkowski, 2005) from a computational complexity perspective. The authors argue that aborting transactions leading to integrity violations is not a viable strategy when dealing with multi-source data integration, proposing integrity-restoration as a separate maintenance process executed to improve data quality. This approach provides an important contribution to the data quality management literature by investigating the computational complexity of repair checking and consistent query answering (CQA) techniques for data retrieved from multiple data sources. The repair checking techniques investigated can be highly valuable towards improving the data quality of data warehouses generated using Extraction, Transform and Load (ETL) processes.

Bertossi (Bertossi, 2006) provides an extensive overview of the key principles and theoretical issues involved in consistent query answering in relational databases, including discussions on the characterization of the CQA problem, computational complexity analysis of CQA, semantics of different database repair techniques applied in CQA, consistency computational complexity analysis of CQA taking into account NULL values, dynamic and incremental computations.

In (Greco, Greco, & Zumpano, 2001), a set of sound and consistent techniques for computing repairs and consistent answers over inconsistent databases are developed using a logic programming framework. The approach to repairing and CQA is based on rewriting integrity constraints into disjunctive rules that can be used to generate repairs for the database and produce consistent answers. The main limitations of the logic programming-based approach arise from the computational complexity associated with computing some of the techniques, which may limit the ability to compute consistent answers in "big data" applications.

Data quality can also be improved via the application of duplicate record detection, where databases are analyzed to identify different records in a database (duplicates) representing the same entity in the real world, leading to accuracy problems. Elmagarmid *et al.* (Elmagarmid, Ipeirotis, & Verykios, 2007) provide a comprehensive survey of the area including discussions about similarity metrics for detecting duplicate records in databases and an analysis of the efficiency and scalability of duplicate record detection algorithms.

Wang and Wang (H. Wang & Wang, 2009) propose an approach to derive knowledge from missing data in survey data sets. The approach uses association rules to express patterns of missing data such as clashing, hiding and disclosing, which can be very useful in performing data profiling of data sources in data quality management.

The Multidimensional Robust Data Quality Analysis (MRDQA) approach by Mezzanzanica *et al.* (Mezzanzanica, Boselli, Cesarini, & Mercorio, 2015) proposes model checking aimed at formalizing and verifying data quality and the effectiveness of business processes used to create and populate data sources and develops techniques and tools to verify the consistency of databases before and after the application of database cleansing functions. The authors argue that applying model-driven verification methods to data cleaning activities help to identify the quality constraints that need to be modelled, with the positive impact in generating higher quality data sets for knowledge discovery tasks. The verification methods proposed include iterative techniques to evaluate the effectiveness of data cleansing functions and the development of visualization techniques to identify problems in data sets.

8. Conclusions and Future Work

This paper presented DQ^{2}S a comprehensive framework and tool for combining traditional data management with data profiling targeted at data cleansing. The framework allows database users to profile their data while querying the database in a declarative way, in preparation for data cleansing, considering multiple dimensions of data quality, such as accuracy, completeness, timeliness, etc. For that, modelling and storage of quality-related data properties can be performed using the same means for modelling and storing relations in a relational database. The quality-related data properties together with the data profiling algorithms represent the criteria under which data is assessed, measured and filtered, in accordance with definitions of data quality dimensions chosen and modelled by the user. An implementation of the framework is also described in which a number of data quality dimensions are modelled and implemented for illustration using e-Business application scenarios. The proposed architecture represents a seamless extension to relational database management systems; and the proposed query language and data model represent user-friendly extensions to SQL and the relational data model.

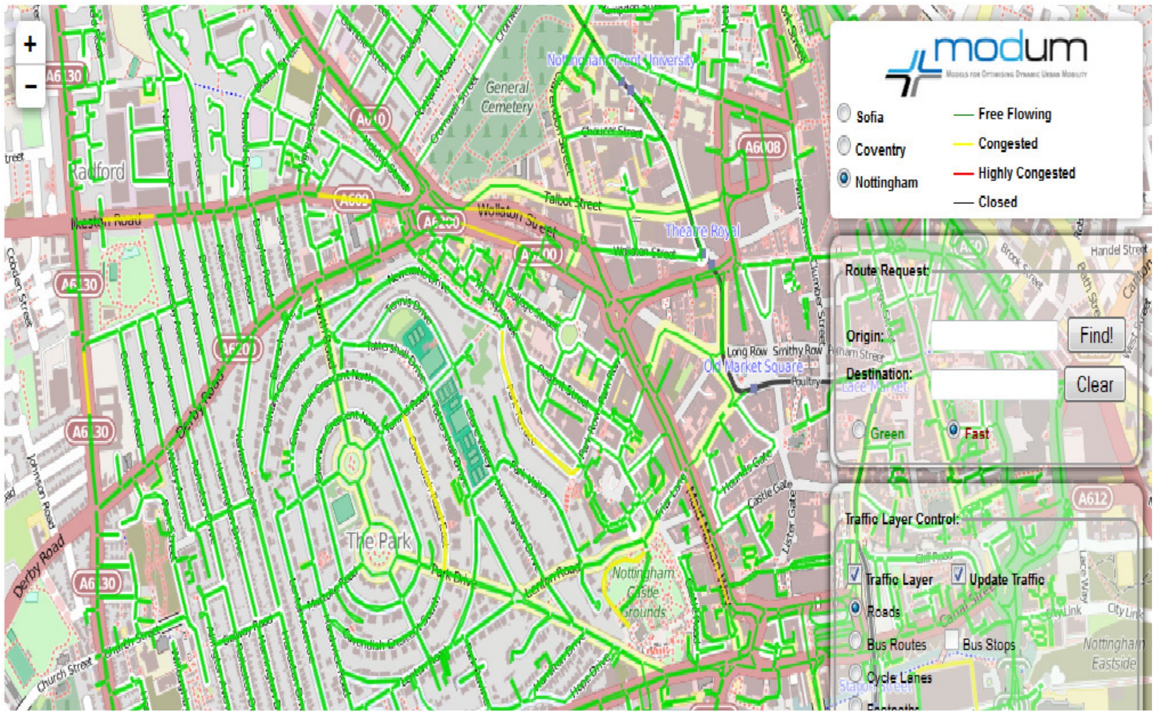
DQ^{2}S enables the ad-hoc exploration and profiling of data and its associated quality as it is queried, using data quality-aware SQL query extensions. The level of sophistication of the profiling algorithms is variable and decided by the end user and/or database application programmer, ranging from simple calculation of scores, to aggregation of multiple scores, including complex calculation strategies and access to multiple sources of information on data-quality properties.

The query language allows comparisons to be made between scores, use of thresholds and filtering of data based on arbitrary predicates. To illustrate the features of our approach to quality profiling, we have selected a number of quality dimensions that are widely used across several application domains and that can be effectively incorporated into an automated information management framework/system. Our choice for a solution with a limited number of profiling operators based on dimensions of quality that have a general purpose is based on the premise that, according to the latest research on purpose and dimensions of Information Quality (IQ) by Illari (Illari, 2014), it is generally agreed that "While the MIT group thinks IQ is best generally defined as information that is 'fit for purpose', both they and many others still think that at least some dimensions of IQ, and even some aspects of IQ itself, are purpose-independent.". Based on the "Law of the Vital Few" (i.e., the Pareto Principle), and experience in practical data quality management projects, only a limited number of quality dimensions and definitions often represent the majority of quality requests from end users. There is also a trade-off between flexibility, performance and usability of data quality techniques and tools. Highly flexible frameworks tend to overburden the end user with the need to learn highly complex application programming interfaces towards expressing quality-aware queries. The balance lies somewhere in a spectrum between highly flexible and extensible solutions and less flexible but efficient and user-friendly frameworks. Typically, what we found in practice is that a combination of complementary tools and techniques will be needed in a data quality management project.

The combined set of features supported by DQ^{2}S allows individual users to set their own quality constraints while querying the data, without imposing the same constraints to all users. These features are extremely useful when users need to assess the quality of relational data sets and define quality filters for acceptable data, as well as a methodology for quality management that allows information quality to be queried and measured for different purposes prior to conducting big data analytical tasks (Floridi, 2014). Future work will include the exploration and quality auditing of large data sets generated by the EC-Funded MODUM project (MODUM: Models for Optimising Dynamic Urban Mobility, 2014) on multi-modal transportation planning, including data generated by urban traffic data sensors from highways in the United Kingdom with data sets in the range of 1-10 Terabytes.

Figure 8.1 shows a screen shot taken from the MODUM system interface, delivering real-time traffic information to a user interested in the traffic conditions in the city of Nottingham on Tuesday the 21st of October 2014 around nine o'clock. In the figure, green lines indicate free flow of vehicles on roads, and yellow lines indicate mildly congested roads. The data feeding the system is based on data collected by sensors distributed across the city. Figure 8.2 shows a fragment of the data in relational format. Traffic dynamic properties such as average speed of vehicles and flow of vehicles across roads are present in the data. Figure 8.3 shows a DQ^{2}L query requesting the measurement of the completeness for attribute `flow`.

We also plan to conduct further experimental evaluation aimed at identifying overheads resulting from the score calculations and access to quality relations for a number of queries of varying complexity. Regarding suitability for use in big data scenarios, additional implementation of the data profiling algorithms as multiple-pass algorithms will be necessary, considering that main-memory solutions are not appropriate for such scenarios. The use of data partitioned parallelism will also be considered as well as cloud-based deployment strategies.



( ) Sofia
( ) Coventry
[✓ Nottingham]

— Free Flowing
— Congested
— Highly Congested
— Closed

Route Request:
Origin: Find!
Destination: Clear
( ) Green [✓ Fast]

Traffic Layer Control:
[✓ Traffic Layer] [✓ Update Traffic]
[✓ Roads]
( ) Bus Routes ☐ Bus Stops
( ) Cycle Lanes

Figure 8.1: MODUM System interface showing traffic information.

Link

<link_ID, nil>	<co2emissions, nil>	<density, nil>	<avgspeed, nil>	<flow, nil>
<-100170072#1, nil>	<5.38e+01, nil>	<4.13e-02, nil>	<1.10e+01, nil>	<4.55e-01, nil>
<-100170078, nil>	<2.45e+01, nil>	<1.47e-02, nil>	<1.28e+01, nil>	<1.89e-01, nil>

Figure 8.2: MODUM System traffic data.

```
SELECT COMPLETENESS(flow)
FROM Link
```

Figure 8.3: DQ2L query requesting the completeness of column `flow`.

References

Aktaş, R., & Karğin, M. (2011). Timeliness of Reporting and the Quality of Financial Information. *International Research Journal of Finance and Economics*(63), 71-77.

Atzeni, P., Batini, C., & Antonellis, V. D. (1993). *Relational Database Theory: A Comprehensive Introduction*: Addison Wesley.

Ballou, R., Wang, R., Pazer, H., & Tayi, G. K. (1998). Modeling Information Manufacturing Systems to Determine Information Product Quality. *Management Science, 44*(4), 462-484.

Barbagallo, D., Cappiello, C., Francalanci, C., & Matera, M. (2010). Enhancing the Selection of Web Sources: A Reputation-Based Approach. *International Conference on Enterprise Information Systems, Lecture Notes in Business Information Processing, 73*, 464-476. doi: 10.1007/978-3-642-19802-1_32

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

Bertossi, L. (2006). Consistent Query Answering in Databases. *ACM SIGMOD Rec., 35*(2), 68 - 76.

Beskales, G., Ilyas, I. F., Golab, L., & Galiullin, A. (2014). Sampling from Repairs of Conditional Functional Dependency Violations. *VLDB Journal, 23*, 103-128.

Bhagwat, D., Chiticariu, L., Tan, W., & Vijayvargiya, G. (2004). An Annotation Management System for Relational Databases. *VLDB Endowment, 30*, 900 - 911.

Biswas, J., Naumann, F., & Qiu, Q. (2006). Assessing the Completeness of Sensor Data. *Database Systems for Advanced Applications (DASFAA), Lecture Notes in Computer Science, 3882*, 717-732. doi: 10.1007/11733836_50

Bouzeghoub, M., & Peralta, V. (2004). A Framework for Analysis of Data Freshness *Proceedings of the International Workshop on Information Quality in Information Systems (IQIS)* (pp. 59-67): ACM.

Bravo, L., Fan, W., & Ma, S. (2007). Extending Dependencies with Conditions *Proceedings of the International Conference on Very Large Databases (VLDB)* (pp. 243-254): VLDB Endowment.

Chen, H., Hailey, D., Wang, N., & Yu, P. (2014). A Review of Data Quality Assessment Methods for Public Health Information Systems. *International Journal of Environmental Research and Public Health, 11*(5), 5170-5207. doi: 10.3390/ijerph110505170

Chomicki, J., & Marcinkowski, J. (2005). Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation, 197*(1 - 2), 90 - 121.

Cong, G., Fan, W., Geerts, F., Jia, X., & Ma, S. (2007). Improving Data Quality: Consistency and Accuracy *Proceedings of the International Conference on Very Large Databases (VLDB)* (pp. 315-326): VLDB Endowment.

Connolly, T. M., & Begg, C. E. (2014). *Database Systems – A Practical Approach to Design, Implementation, and Management* (6th ed.): Addison-Wesley.

Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A. K., Ilyas, I. F., Ouzzani, M., & Tang, N. (2013). NADEEF: A Commodity Data Cleaning System *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 541-552): ACM.

Delcambre, L., Maier, D., Bowers, S., Weaver, M., Longxing, D., Gorman, P., . . . Lyman, J. A. (2001). Bundles in Captivity: an Application of Superimposed Information *Proceedings of the International Conference on Data Engineering* (pp. 111 - 120).

Deursena, T., Kostera, P., & Petkovica, M. (2008). Hedaquin, A Reputation-Based Health Data Quality Indicator. *Electronic Notes in Theoretical Computer Science, 197*(2), 159–167.

Dong, C., Sampaio, S. F. M., & Sampaio, P. R. F. (2006). Expressing and Processing Timeliness Quality Aware Queries: The DQ2L Approach. *Quality of Information Systems (ER Workshops), Lecture Notes in Computer Science, 4231*, 382-391.

Elmagarmid, A. K., Ipeirotis, P. G., & Verykios, V. S. (2007). Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering, 19*(1), 1 - 16.

Embury, S. M., Missier, P., Sampaio, S., Greenwood, R. M., & Preece, A. D. (2009). Incorporating Domain-Specific Information Quality Constraints into Database Queries. *Journal of Data and Information Quality, 1*(2).

Fan, W., Geerts, F., Jia, X., & Kementsietsidis, A. (2008). Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS), 33*(2), 1-48.

Fan, W., Li, J., Ma, S., Tang, N., & Yu, W. (2012). Towards Certain Fixes with Editing Rules and Master Data. *VLDB Journal, 21*(2), 213-238.

Floridi, L. (2014). Big Data and Information Quality *The Philosophy of Information Quality* (Vol. 358, pp. 303-315): Springer.

Furber, C., & Hepp, M. (2011). Towards a Vocabulary for Data Quality Management in Semantic Web Architectures *Proceedings of the International Workshop on Linked Web Data Management (LWDM)* (pp. 1-8): ACM.

Gamble, M., & Goble, C. (2011). Quality, Trust, and Utility of Scientific Data on the Web: Towards a Joint Model *Proceedings of the International Web Science Conference (WebSci)* (pp. 1-8): ACM.

Gantz, J., & Reinsel, D. (2012). The Digital Universe in 2020: Big Data, Bigger Digital Shadows and Biggest Growth in the Far East. Retrieved 02/04/2015, from http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2013). *Database Systems: The Complete Book*: Pearson Education.

Geerts, F., Mecca, G., Papotti, P., & Santoro, D. (2013). The LLUNATIC data-cleaning framework. *Proceedings of the VLDB Endowment, 6*(9), 625-636. doi: 10.14778/2536360.2536363

Greco, G., Greco, S., & Zumpano, E. (2001). A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases. *International Conference on Logic Programming, Lecture Notes in Computer Science, 2237*, 348 - 364.

Greenwald, R., Stackowiak, R., & Stern, J. (2013). *Oracle Essentials: Oracle Database 12c*: O'Reilly.

Han, J., Jiang, D., & Li, L. (2010). Automatic accuracy assessment via hashing in multiple-source environment. *Expert Systems with Applications: An International Journal, 37*(3), 2609-2620.

Han, J., Jiang, D., & Song, A. (2008). Quantifying Accuracy Dimension within Available Context *Proceedings of the International Symposium on Information Science and Engineering (ISISE)* (Vol. 1, pp. 214-218): IEEE.

Huhtala, Y., Kärkkäinen, J., Porkka, P., & Toivonen, H. (1999). TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Computer Journal, 42*(2), 100-111.

Illari, P. (2014). IQ:Purpose and Dimensions *The Philosophy of Information Quality* (Vol. 358, pp. 281-301): Springer.

Klein, A., Do, H. H., Hackenbroich, G., Karnstedt, M., & Lehner, W. (2007). Representing Data Quality for Streaming and Static Data *Proceedings of the International Conference on Data Engineering (ICDE) Workshops* (pp. 3-10): IEEE.

Mecella, M., Scannapieco, M., Virgillito, A., Baldoni, R., Catarci, T., & Batini, C. (2002). Managing Data Quality in Cooperative Information Systems. *On the Move to Meaningful Internet Systems 2002: CoopIS/DOA/ODBASE, Lecture Notes in Computer Science, 2519*, 486-502.

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 on Data Semantics I, Lecture Notes in Computer Science, 2800*, 208-232.

Mezzanzanica, M., Boselli, R., Cesarini, M., & Mercorio, F. (2015). A Model-Based Evaluation of Data Quality Activities in KDD. *Information Processing & Management, 51*(2), 144 - 166.

MODUM: Models for Optimising Dynamic Urban Mobility. (2014). Retrieved 17/10/2014, from http://modum-project.eu/

Mutsuzaki, M., Theobald, M., Keijzer, A. d., Widom, J., Agrawal, P., Benjelloun, O., . . . Sugihara, T. (2007). Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS *Proceedings of the Conference on Innovative Data Systems Research* (pp. 269-274).

MySQL.com. (2015). Retrieved 20/05/2015, from http://www.mysql.com.

Naumann, F. (2014). Data Profiling Revisited. *ACM SIGMOD Record, 42*(4), 40-49.

Olson, J. E. (2003). *Data Quality: The Accuracy Dimension*: Morgan Kaufmann.

Orme, A. M., Yao, H., & Etzkorn, L. H. (2007). Indicating Ontology Data Quality, Stability, and Completeness throughout Ontology Evolution. *Journal of Software Maintenance, 19*(1), 49-75.

Paulson, L. D. (2000). Data Quality: A Rising E-Business Concern. *IT Professional, 2*(4), 10-14.

Peer, E., Vosgerau, J., & Acquisti, A. (2014). Reputation as a Sufficient Condition for Data Quality on Amazon Mechanical Turk. *Behavior Research Methods, 46*(4), 1023-1031.

Pernici, B., & Scannapieco, M. (2003). Data Quality in Web Information Systems. *Journal on Data Semantics I, Lecture Notes in Computer Science, 2800*, 48-68.

Pipino, L. L., Lee, Y. W., & Wang, R. Y. (2002). Data Quality Assessment. *Communications of the ACM, 45*(4), 211--218.

Poosala, V., Haas, P. J., Ioannidis, Y. E., & Shekita, E. J. (1996). Improved Histograms for Selectivity Estimation of Range Predicates *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 294-305): ACM.

Qin, Z., Han, Q., Mehrotra, S., & Venkatasubramanian, N. (2014). Quality-Aware Sensor Data Management *The Art of Wireless Sensor Networks, Signals and Communication Technology* (Vol. 1, pp. 429-464): Springer.

Redman, T. C. (1997). *Data Quality for the Information Age*: ACM.

Scannapieco, M., Mirabella, V., Mecella, M., & Batini, C. (2002). Data Quality in e-Business Applications. *Web Services, E-Business, and the Semantic Web (WES), Lecture Notes in Computer Science, 2512*, 121-138.

Scardina, M., Chang, B., & Wang, J. (2004). *Oracle Database 10g XML & SQL: Design, Build, & Manage XML Applications in Java, C, C++, & PL/SQL*: Mcgraw-hill.

Sebastian-Coleman, L. (2013). *Measuring Data Quality for Ongoing Improvement: A Data Quality Assessment Framework*: Morgan Kaufmann.

Segev, A. (2001). Data Quality Challenges in Enabling eBusiness Transformation *Proceedings of the International Conference on Information Quality* (pp. 83-91): MIT.

Shankaranarayanan, G., & Cai, Y. (2006). Supporting data quality management in decision-making. *Decision Support Systems, 42*(1), 302-317.

Standard, I. (2013). ISO 19157:2013, Geographic Information — Data Quality. Retrieved 02/04/2015, from http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=32575

Stonebraker, M., & Moore, D. (1996). *Object-relational DBMSs: The Next Great Wave*: Morgan Kaufmann.

Taylor, A. (2003). *JDBC: Database Programming with J2EE*: Prentice Hall.

Tomic, K., Sandin, F., Wigertz, A., D.Robinson, Lambe, M., & Stattin, P. (2015). Evaluation of Data Quality in the National Prostate Cancer Register of Sweden. *European Journal of Cancer, 51*(1), 101-111.

Wand, Y., & Wang, R. Y. (1996). Anchoring Data Quality Dimensions in Ontological Foundations. *Communications of the ACM, 39*(11), 86-95.

Wang, H., & Wang, S. (2009). Discovering Patterns of Missing Data in Survey Databases: An application of Rough Sets. *Expert Systems with Applications: An International Journal, 36*(3), 6256 - 6260.

Wang, H., Yang, D., Zhao, Y., & Gao, Y. (2006). Multiagent System for Reputation-based Web Services Selection *Proceedings of the International Conference on Quality Software (QSIC)* (pp. 429-434): IEEE Computer Society.

Wang, R. Y., Reddy, M. P., & Kon, H. B. (1995). Toward Quality Data: an Attribute-Based Approach. *Decision Support Systems, 13*(3-4), 349-372.

Wang, R. Y., & Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems, 12*(4), 5-33.

Wang, R. Y., Ziad, M., & Lee, Y. W. (2001). *Data Quality* (Vol. 23): Kluwer.

Widom, J., & Ceri, S. (1996). *Active Database Systems: Triggers and Rules for Advanced Database Processing*: Morgan Kaufmann.

Yakout, M., Berti-Equille, L., & Elmagarmid, A. K. (2013). Don't Be SCAREd: Use SCalable Automatic REpairing with Maximal Likelihood and Bounded Changes *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 553-564): ACM.

Yao, H., & Hamilton, H. J. (2008). Mining Functional Dependencies From Data. *Data Mining and Knowledge Discovery, 16*(2), 197-219.

Yeganeh, N. K., Sadiq, S., & Sharaf, M. A. (2014). A Framework for Data Quality Aware Query Systems. *Information Systems, 46*, 24 - 44.

Yom-Tov, E., & Diaz, F. (2011). Location and Timeliness of Information Sources During News Events *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 1105-1106): ACM.

Yu, C. T., & Meng, W. (1998). *Principles of Database Query Processing for Advanced Applications*: Morgan Kaufmann.

APPENDIX A - DQ^{2}L Syntax in BNF

The syntax specification of DQ^{2}L is expressed using the Backus Naur Form (BNF) language for grammar specification, which was automatically generated by JJDoc provided by JavaCC. BNF enables the specification of the concrete syntax of the language, and checks the syntactic correctness for the strings of characters. The BNF language has been slightly extended by JJDoc in the following aspects:

- The meta-symbols "(" and ")*" are used to enclose a sequence of symbols that can occur any number of times (zero or more).
- The meta-symbols "(" and ")+" are used to enclose a sequence of symbols that can occur one or more times.
- Lexical tokens of the language are placed within double quotes, e.g., "SELECT".

The distinguished symbol of the grammar is Start. In order to resolve ambiguities in parsing, alternative forms of each right-hand side of the rule are in order of decreasing precedence. The reserved words of the language are: SELECT, FROM, WHERE, ORDER BY, AND, WITH QUALITY AS, ACCURACY, TIMELI-NESS, COMPLETENESS, REPUTATION, TOPSIS.

```
             Start ::= SFWQ <EOF>

              SFWQ ::= SelectStatement FromStatement ( WhereStatement )? (
                       QualityStatement )? ( OrderStatement )?

   SelectStatement ::= "SELECT" Item ( "," Item )*

              Item ::= ( Attribute | QualityFunction )

     FromStatement ::= "FROM" Relation ( "," Relation )*

    WhereStatement ::= "WHERE" Condition ( "AND" Condition )*

         Condition ::= Attribute <OPER> ( Constant | Attribute )

  QualityStatement ::= "WITH QUALITY AS" QualityConstraint ( "AND"
                       QualityConstraint )*

 QualityConstraint ::= QualityFunction <OPER> NumLiteral

    OrderStatement ::= "ORDER BY " RankingReference

  RankingReference ::= Attribute
                     | QualityFunction
     QualityFunction ::= AccuracyFunction
                       | CompletenessFunction
                       | TimelinessFunction
                       | ReputationFunction
                       | TOPSISFunction
    AccuracyFunction ::= "ACCURACY" "(" Attribute ")"
CompletenessFunction ::= "COMPLETENESS" "(" Relation ")"
  TimelinessFunction ::= "TIMELINESS" "(" Attribute ")"
  ReputationFunction ::= "REPUTATION" "(" Attribute "," NumLiteral ":"NumLiteral ")"
      TOPSISFunction ::= "TOPSIS" "(" Weights ")"
             Weights ::= QualityFunction ":" NumLiteral ( ";" QualityFunction ":"
                         NumLiteral )*
           Attribute ::= <VARIABLE>
            Relation ::= <VARIABLE>
            Constant ::= NumLiteral
                       | TextLiteral
          NumLiteral ::= <NUM_LITERAL>
         TextLiteral ::= <TEXT_LITERAL>
```

APPENDIX B - View-Packages

View_Package_id	1	
View_Package_name	Sales	
Most accessed attributes	**Access frequency**	**Preferred method**
`Order.product_ID`	97%	
`Order.status`	89%	
`Order.submit_date`	76%	
`Product.product_price`	70%	`<reputation,[<accessibility, 0.5>,<reliability,0.5>]>`
`Customer.name`	61%	`<accuracy,editDistance>`

Table B.1: View_package for Department Sales.

View_Package_id	2	
View_Package_name	Shipping	
Most accessed attributes	**Access frequency**	**Preferred method**
`Order.product_ID`	98%	`<accuracy,boolean>`
`Order.status`	98%	`<accuracy,boolean>`
`Order.submit_date`	93%	
`Product.product_price`	65%	
`Customer.name`	61%	

Table B.2: View_package for Department Shipping.

APPENDIX C - Experimental Results

	Query Elapsed Time (in seconds)			
DB Size	Query2 (NO)	Query2 (O)	Query5 (NO)	Query5 (O)
1,000	*1.32*	*0.95*	*1.16*	*1.06*
10,000	*36.79*	*13.92*	*46.32*	*29.24*
100,000	*3370.9*	*982.86*	*4456.14*	*2987.63*

Table C.1: Average Elapsed Times of optimized (O) and non-optimized (NO) Queries 2 and 5, varying database size.