MeSQuaL: the Spirit for Data Quality Auditing

Laure Berti-Équille*
Université de Rennes 1
berti@irisa.fr

Elie Naulleau†
Semiosys
semiosys@semiophore.net

ABSTRACT

We present MeSQuaL[1], a system extending SQL queries for jointly querying relational data and auditing its quality based on user-define functions and constraints. The MeSQuaL framework consists of (a) SQuaL, a SQL-like language for specificatio of data quality requirements; (b) an extensible library of functions for profiling measuring and checking data quality, (c) a web interface and an API for submitting SQuaL queries and interact during the auditing process to relax or reinforce the constraints on data quality; and (d) an interface that assists users in writing SQuaL queries. We discuss the challenges involved in the design and implementation of a declarative and easy-to-use framework for auditing the quality of relational data with constraining on-demand computed data quality indicators. We demonstrate the specificatio steps of data quality auditing scenarios and show how the MeSQuaL system can be effectively and efficiently used to audit and profil the quality of large real-world databases.

1. INTRODUCTION

Reporting the quality of data and maintaining a high level of data quality in a database are highly challenging tasks that require the detection and elimination of a variety of data quality problems, such as duplicate records, inconsistent, obsolete, and incomplete data. This requires the use of a wide range of methods, advanced techniques, and tools involving statistics [2, 9], constraint mining and checking [7], string matching and record linkage methods [5] or probabilistic data management [1]. A plethora of dimensions exist to characterize the quality of data (e.g., accuracy, freshness, consistency, completeness, etc.) and various techniques can be used to evaluate each dimension for ultimately auditing the quality of a database. Nevertheless the auditing specification ultimately depend on the users' requirements, their domain knowledge, and the application domain specificities In this context, what users needs for auditing and managing data quality is an extensible framework offering a customizable, fl xible, and declarative way to evaluate the quality of their data.

Several extensions of SQL query language have been proposed to include constraints on a limited number of fi ed data quality dimensions (typically, one to three) they handle separately using *built-in* functions or predicates [8, 4, 10]. Their main drawback is that they are neither fl xible nor extensible and they don't offer the users the possibility to add and tune new functions and constraints on data quality and interact in the auditing process.

The goal of MeSQuaL is to provide the users with a facility for specifying user-define measures of data quality using SQuaL, the SQL-like query language extension for querying data with guarantees on "checked on-demand" quality and correctness of the results.

In this demonstration, we showcase the principal features of MeSQuaL.

- We show that by integrating user-define constraints and a rich collection of functions to detect data anomalies and evaluate the quality of data, our framework supports interactive data quality auditing. A common way to use our framework is to declaratively specify the data quality dimensions of interest, bind and invoke the functions to measure them and compute relevant indicators, and query the data with constraints of these indicators. The results can help the user in refinin the auditing process or definin the methods to use or comparing the different methods and selecting the most appropriate ones.

- We show the effectiveness of our framework to audit the quality of large databases on patents from EPO[2] and WIPO[3] and on biomedical data extracted from on-line databanks using BioGuideSRS[4] [3].

2. THE MESQUAL SYSTEM

2.1 Illustrative Example

Consider, for example, a scenario in which a company wants to audit and check the quality of its database on patents and inventors populated with data weekly imported from EPO and WIPO, as shown in Figure [1]. At the firs glance on this database, you may observe various data quality problems, such as duplicate records, missing, and inconsistent values. A typical scenario for auditing

*Funded by Marie Curie International Fellowship (FP6-MOIF-CT-2006-041000)
†Partially funded by ANR ARA QUADRIS
[1]My extended SQL for data QuaLity auditing

VLDB '10, September 13-17, 2010, Singapore

[2]European Patent Office (EPO) `http://www.epo.org/patents/patent-information/raw-data/`
[3]World Intellectual Property Organization: `http://www.wipo.int/pctdb/en/`
[4]BioGuide Project : `http://www.bioguide-project.net/`

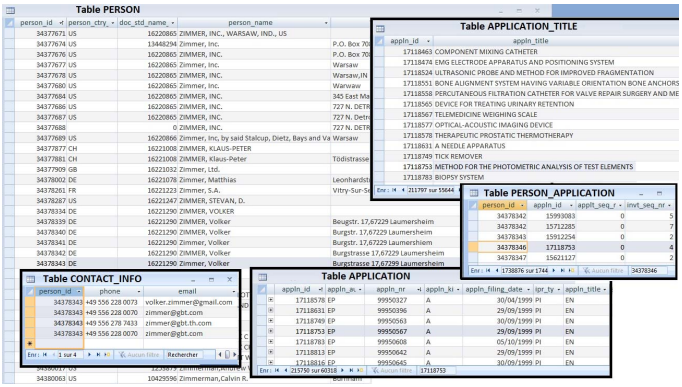

Figure 1: The Patent Database (Sample)

the quality of this database would be to check the presence of duplicates in the tables `PERSON` and `APPLICATION`, the completeness of the records on applications, inventors, in contact information, the correctness of the address, and the freshness of the contact information, phone numbers and emails. Numerous methods and techniques can be invoked to achieve each step of this scenario. In the absence of an extensible and dedicated tool, and without automated support for a variety of advanced data quality checking methods, the users will have to experiment a myriad of external and generic methods for each type of anomaly and combine their results to finall report the quality of the database. To overcome this problem, factorize and re-use the methods in a unifie way, we have developed MeSQuaL an extensible framework for integrating various methods for checking data quality and querying data with constraints on its quality. Our goal is to facilitate usage and help users defin and tune the data quality auditing scenarios that will work best for their domain and application.

2.2 Data Quality Auditing Specification

From the user-centric perspective, a MeSQuaL scenario is composed of the following steps:

Step 1 Declaration (or re-use by invocation) of methods for data quality checking and evaluation. The firs step of a data quality auditing scenario is the definitio (or re-use) of functions that measure objectively the dimensions of interest characterizing the quality of data (for our example, absence of duplicate records of persons, completeness, accuracy and freshness); each dimension may characterize one aspect of the quality of a database object or a particular level of granularity in the database (*i.e.*, the granularity level of the value, tuple, attribute domain, table or database). Measures and indicators are computed based on user-define functions, stored and managed in a metadata repository associated to the considered level of granularity for the current database schema.

Step 2 Declaration of constraints on the measures and indicators. To establish a quality diagnostic, data quality measures have to be compared to user-expected values. To do so, constraints are specifie by the user and checked by the system on the considered dimension in order to determine if the quality is (or not) acceptable.

Step 3 Quality-constrained query declaration. The user can submit a query with constraining the result with the constraints define in Step 2.

```
contract_type_stmt := (CREATE|REPLACE ) CONTRACTTYPE ctype_name
                        dimension[, dimension]*;
dimension :=        dim_name sql_data_type
                      ON ( CELL [attribute_name | table_name]
                             | ROW [table_name]
                             | COLUMN [table_name]
                             | TABLE [schema_name]
                             | DATABASE )
                    BY FUNCTION function_name AS
                      (call_spec_stmt | sql_procedure | web_service_spec);
call_spec_stmt :=   LANGUAGE language_name file_name;
contract_stmt :=    ( CREATE | REPLACE ) CONTRACT contract_name
                    FROM ctype_name[, ctype_name]* constraint[, constraint]*;
constraint :=       [ctype_name.]dim_name operator expression;
operator :=         <> | = | != | > | < | <= | >= | && | || ;
expression :=       simple_expression | complex_expression;
simple_expression := NULL | TRUE | FALSE | sql_data_type_constant;
complex_expression := expression
                      | ( expression ) operator ( expression )
                      | NOT ( expression );
SQuaL_query_stmt := SFW_query_stmt
                    QWITH
                    ( [ctype_name.]contract_name.constraint
                      | [ctype_name.]contract_name )
                      ON ( CELL [attribute_name | table_name]
                             | ROW [table_name]
                             | COLUMN [table_name]
                             | TABLE [schema_name]
                             | DATABASE )]
                    [ ( AND | OR )
                      ( [ctype_name.]contract_name.constraint
                             | [ctype_name.]contract_name )
                      ON ( CELL [attribute_name | table_name]
                             | ROW [table_name]
                             | COLUMN [table_name]
                             | TABLE [schema_name]
                             | DATABASE) ]*;
```

Figure 2: SQuaL Syntax

2.3 Query Language

Each step of a data quality auditing scenario can be expressed in SQuaL, the SQL-extended language we implemented in MeSQuaL on top of three RDBMS (Oracle, MySQL, and PostgreSQL).Data quality measures and indicators can be expressed in the declaration of *quality contract types*. The creation of contract types triggers the execution of user-define functions and the computation of data quality indicators. Constraints on the quality indicators can be expressed in the declaration of *quality contracts* or invoked in SQuaL queries. In more detail, the contract type and contract specifica tions are given using the grammar of Figure 2. As shown in the figure a `contract_type_stmt` statement creates (or replaces) a contract type `CREATE | REPLACE CONTRACTTYPE` and define it with one (or more) quality dimension(s) of interest (*dim_name*) and a measure computed by a user-define function (`BY FUNCTION`) associated to a granularity level in the database, *i.e.* `ON CELL, ROW, COLUMN, TABLE`, or `DATABASE` or to a particular database object (*attribute_name* or *table_name*) with ambiguity resolution.

Our framework provides several *built-in* functions (currently in Java, SAS, and C) or the possibility to call stored procedures (*sql procedure*) or web services *web service spec* for checking data quality. More specificall , we provide functions to check : *i)* whether records are duplicates using a variety of string matching methods (based on SimMetrics library[5], *ii)* whether values are outlying using various statistical methods (coded in SAS), *iii)* inconsistent based on constraints given as inputs by the user, and *iv)* out-of-date or incomplete using more basic utility functions providing timestamp comparison or countings.

[5]SimMetrics: `http://sourceforge.net/projects/simmetrics/`

```
CREATE CONTRACTTYPE Freshness (
  T_UpdateFrequency FLOAT ON TABLE BY FUNCTION java_T_freshness AS LANGUAGE JAVA 'java_T_freshness.java',
  DB_UpdateFrequency FLOAT ON DATABASE BY FUNCTION java_DB_freshness AS LANGUAGE JAVA 'java_DB_freshness.java',
  CL_Age INTEGER ON CELL BY FUNCTION java_CL_age_from_creation AS LANGUAGE JAVA 'java_CL_age.java' );
\\
CREATE CONTRACTTYPE Consistency (
 CL_AddressStreet_Check BINARY ON CELL PERSON.person_address BY FUNCTION c_R_address_check AS LANGUAGE C 'address_check.c',
 CL_ZipCity_Check BINARY ON CELL PERSON.person_address BY FUNCTION c_R_zip_check AS LANGUAGE C 'zip_check.c',
 CL_Phone_Check BINARY ON CELL CONTACT_INFO.phone BY FUNCTION c_CL_phone_check AS LANGUAGE C 'phone_check.c',
 CL_Email_Check BINARY ON CELL CONTACT_INFO.email BY FUNCTION c_CL_email_check AS LANGUAGE C 'email_check.c');
\\
CREATE CONTRACTTYPE Outlyingness (
 CL_outlier_Tukey BINARY ON CELL BY FUNCTION sas_CL_outlier1 AS LANGUAGE SAS 'TukeyOutliers.sas',
 CL_outlier_MV FLOAT ON CELL BY FUNCTION sas_CL_outlier2 AS LANGUAGE SAS 'MultivariateOutliers.sas',
 CL_outlier_Cluster FLOAT ON CELL BY FUNCTION sas_CL_outlier3 AS LANGUAGE SAS 'ClustersOutliers.sas',
 CL_outlier_Ranges FLOAT ON CELL BY FUNCTION sas_CL_func_outlier4 AS LANGUAGE SAS 'RangesOutliers.sas',
 CL_outlier_Reg FLOAT ON CELL BY FUNCTION sas_CL_outlier5 AS LANGUAGE SAS 'RegressionOutliers.sas');
\\
CREATE CONTRACTTYPE Completeness (
 CO_nullValuesPercentage FLOAT ON COLUMN BY FUNCTION  plsql_CO_nullValues AS plsql_CO_nullValuesPercentage,
 T_nullValuesPercentage FLOAT ON TABLE BY FUNCTION  plsql_T_nullValues AS plsql_T_nullValuesPercentage,
 R_nullValuesPercentage FLOAT ON ROW BY FUNCTION  plsql_R_nullValues AS plsql_R_nullValuesPercentage,
 DB_nullValuesPercentage FLOAT ON DATABASE BY FUNCTION  plsql_DB_nullValues AS plsql_DB_nullValuesPercentage);
\\
CREATE CONTRACTTYPE Uniqueness (
 R_DupStatus1 BINARY ON ROW PERSON BY FUNCTION java_R_Dup1 AS LANGUAGE JAVA 'levenshtein_string_matching.java',
 R_DupStatus2 BINARY ON ROW PERSON BY FUNCTION java_R_Dup2 AS LANGUAGE JAVA 'edit_string_matching.java',
 R_DupStatus3 BINARY ON ROW PERSON BY FUNCTION java_R_Dup3 AS LANGUAGE JAVA 'jarowinkler_string_matching.java');
```

Table 1: Examples of Quality Contract Types

```
CREATE CONTRACT fresh FROM Freshness (
  T_UpdateFrequency < 0.50,
  DB_UpdateFrequency  < 0.50,
  CL_Age < 86400);
\\
CREATE CONTRACT consistent FROM Consistency (
  CL_AddressStreet_Check = TRUE,
  CL_ZipCity_Check = TRUE,
  CL_Phone_Check = TRUE,
  CL_Email_Check = TRUE);
\\
CREATE CONTRACT complete FROM Completeness (
  R_nullValuesPercentage <= .25;
\\
CREATE CONTRACT notduplicate FROM Uniqueness (
  (R_DupStatus1 = FALSE)
  OR (R_DupStatus2 = FALSE)
  OR  (R_DupStatus3 = FALSE));
\\
CREATE CONTRACT misc
  FROM Freshness, Consistency, Completeness, Uniqueness (
  CL_Age < 2500000
  && CL_Phone_Check = TRUE
  && CL_Email_Check = TRUE
  &&  R_nullValuesPercentage = 0
  &&  NOT(R_DupStatus1 = TRUE) AND (R_DupStatus3 = FALSE));
```

Table 2: Examples of Contract Instances

A *quality contract* define in `contract_stmt` of Figure 2 derives from one (or more) pre-existing quality contract type(s) and is a set of one-sided range constraints on the dimensions declared in the contract type(s) (`FROM` *ctype_name*) the contract instance is derived from. Constraints are simple or complex expressions involving the pre-declared dimensions. Table 1 presents several examples of contract type declaration for our auditing scenario on Patent database. In this scenario, five contract types have been created for evaluating the freshness, the consistency, the outlyingness, completeness, and uniqueness of the database. In `Freshness` contract type, three Java functions are invoked to compute how frequently the tables (`T_UpdateFrequency`) and the database (`DB_UpdateFrequency`) have been updated since their creation time and how old (in seconds) the values are (`CL_Age`). The computed indicators are associated to the declared level of granularity: to each table, to the database, and to each value, respectively. The corresponding contract instance named `fresh` derived from `Freshness` is given in Table 2. It sets the constraints on these dimensions to be less than 50% and one day (86400 seconds).

In `Consistency` contract type in Table 1, four C functions are invoked: `CL_AddressStreet_Check` dimension checks whether the addresses in table `PERSON` exist and are in conformance with an external dictionary referencing all national addresses per country and it associates the corresponding result to each value of `PERSON`.*person_address*; `CL_ZipCity_Check` dimension checks whether the zip code exists and is correct using similarly an external dictionary of zip codes per country and it also associates its result to each value of `PERSON`.*person_address*; `CL_Phone_Check` dimension checks whether the phone number is syntactically correct and corresponds to the dialing codes depending on the person's address and on the country and it associates the checking result to each value of `CONTACT_INFO`.*phone*; `CL_Email_Check` checks whether the email is valid (based on "try and error") and it associates the result to each value of `CONTACT_INFO`.*email*. The corresponding contract instance named `consistent` derived from `Consistency` in Table 2 sets the constraints on these four dimensions to be `TRUE`.

In `Outlyingness` contract type, five SAS functions check whether numerical values are outlying using univariate, multivariate, clustering, ranges and regression-based methods. They associate the outlier status to each cell even though the data distribution they use as input may involve the values of one or more attributes.

In `Completeness` contract type, four Oracle PL/SQL procedures `plsql_XX_nullValues` return the percentage of missing values per row, column, table and for the whole database respectively, and they associate their result accordingly. The corresponding contract instance named `complete` derived from `Completeness` in Table 2 sets the constraint on records to have less than 25% of NULL values.

In `Uniqueness` contract type in Table 1, three Java functions check whether the records are duplicates based on Levenshtein, Edit, and Jaro-Winkler similarity distances. The corresponding contract instance named `notduplicate` derived from `Uniqueness` in Table 2 considers the records of table `PERSON` as duplicates when at least one of the string-matching functions returns `FALSE`.

Finally in Table 2, a contract instance named `misc` is derived from the previous contract types, namely `Freshness, Consistency, Completeness, Uniqueness` to check other constraints on the age of the values, the correctness of phone numbers and emails, and the duplicate status of the records in `PERSON` table by re-using the contract type declarations and changing the constraints.

The detail description of the examples in Tables 1 and 2 intends to show that the declaration of contract types and instances is an important part of the data quality auditing specifications MeSQuaL is a system that aims at offering a flexible, extensible and declarative way of incorporating and re-use a set of functions for data quality auditing. As mentioned, we provide a library of functions that can be easily extended. The call and execution of functions is triggered immediately after the validation of the contract type declaration.

To take the full advantage of the contract declaration in the RDBMS, we have extended the query language for invoking the contracts and constraints in the queries. The syntax of a SQuaL query is given in `SQuaL_query_stmt` of Figure 2. Once declared, one (or several) contract(s) and constraints can be invoked and used in the `QWITH` part of the SQuaL query. `'SFW_query_stmt'` represents the classical `SELECT-FROM-WHERE` statement of the query supported by the RDBMS. Existing contract instances and new constraints on existing dimensions from declared contract types can be declared in the `QWITH` statement as shown in the examples of Table 3. SQuaL query #1 retrieves the name, address, phone number, and email of inventors of patents on "photometric analysis" with guarantees on the consistency of the values in `PERSON` table (line 7) – referring to the previous declarations of contract types and instances in Tables 1 and 2, and on the age of the values for address, phone, and email (lines 8-10). The SQuaL query #2 gives the records that don't satisfy the constraints declared in the `misc` contract of Table 2. This kind of query may be very useful to audit the database and discover the data that needs to be cleaned or consolidated. SQuaL query #3 enables the users to compare the functions invoked in this example to check whether records are duplicates based on various similarity distance functions. In this query, only the "border-line" duplicates will be return showing different sensitivity of the detection methods.

2.4 Architecture

From the system-centric perspective, our approach has been

SQuaL Query#1

```
1: SELECT person_name,person_address,phone,email,appln_title
2: FROM PERSON, CONTACT_INFO, APPLICATION_TITLE, PERSON_APPLICATION
3: WHERE PERSON.person_id=CONTACT_INFO.person_id AND
4:       PERSON.person_id=PERSON_APPLICATION.person_id AND
5:       APPLICATION_TITLE.appln_id=PERSON_APPLICATION.appln_id
6:       appln_title like '%PHOTOMETRIC ANALYSIS%'
7: QWITH consistent AND CL_Age ON CELL person_address
8:       AND CL_Age ON CELL phone AND CL_Age ON CELL email;
```

SQuaL Query#2

```
1: SELECT P1.person_name, PA1.appln_title
2: FROM PERSON P1,APPLICATION_TITLE AT1,PERSON_APPLICATION PA1
3: WHERE P.person_id=PA1.person_id AND
4:    AT1.appln_id=PA1.appln_id
5:    AND NOT EXISTS (SELECT P2.person_name, PA2.appln_title
6:        FROM PERSON P2, APPLICATION_TITLE AT2,
7:             PERSON_APPLICATION PA2
8:        WHERE P2.person_id=PA2.person_id AND
9:              AT2.appln_id=PA2.appln_id AND
10:             P2.person_id=P1.person_id
11:       QWITH misc);
```

SQuaL Query#3

```
1: SELECT *
2: FROM PERSON
3: QWITH Uniqueness.R_DupStatus1 !=  Uniqueness.R_DupStatus2
4:    OR Uniqueness.R_DupStatus2 !=  Uniqueness.R_DupStatus3
5:    OR Uniqueness.R_DupStatus1 !=  Uniqueness.R_DupStatus3;
```

Table 3: Examples of SQuaL Queries

translated into three main components that pragmatically integrate data quality awareness in the system: (a) SQuaL, a SQL-like language for specificatio of data quality requirements; (b) an extensible library of functions for profiling measuring and checking data quality organized in four main packages dedicated to *i)* outlier detection, *ii)* duplicate detection, *iii)* inconsistency detection, and *iv)* basic utility functions for counting and comparing, (c) a web interface and an API for submitting SQuaL queries and interact during the execution to relax or reinforce the constraints on data quality; and (d) an interface that assists users in writing SQuaL queries. This approach has several advantages including the ability to easily implement this framework on existing relational databases (currently Oracle, MySQL, PostgreSQL) with minimum effort and without any need for externally written program code and take advantage of the underlying DBMS optimizations in the query engine while evaluating the contracts.

3. DEMONSTRATION DESCRIPTION

Demonstration Scenarios. In this demonstration, we will go through the steps of data quality auditing in several interesting real world application scenarios. The main goal is to show the applicability of our framework in a variety of needs and prove the need for the functionality of all the components in our framework. At the conference, we will demonstrate MeSQuaL using three demonstrations scenarios for at least two real-world databases. The firs use case, named **Patent** includes the examples used throughout this paper as well as more real-life and large-scale data auditing scenarios based on data from the on-line databases of European and International patents offices (EPO and WIPO). The second use case, named **BioMed** is based on a large biomedical database integrating data from various on-line databanks on human diseases using BioGuideSRS [3]. We show the application of the MeSQuaL system in the following scenarios: **(1) Querying.** We will demonstrate the effectiveness of the SQuaL queries for various settings, contract types and instances declaration on the two databases; **(2) Auditing.** We will present sequences of SQuaL queries to achieve specifi auditing scenarios on the target databases; **(3) Comparison.** Using SQuaL queries, we will show how the users can easily compare the results of various functions for detecting the same data quality problems.

Implementation. We implemented MeSQuaL's query interpreter

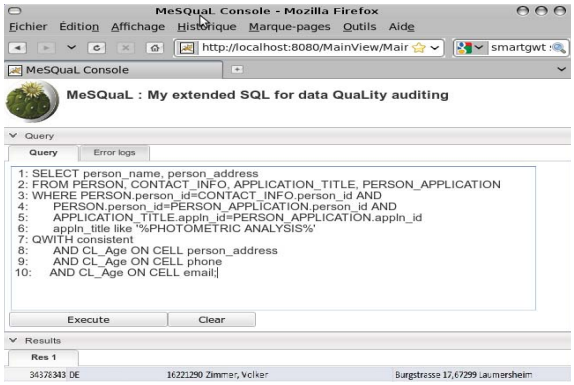

Figure 3: MeSQuaL screenshot

and translator in Java. The functions for evaluating data quality are implemented in Java, SAS, and C and Web services can also be invoked irrespective of the language used to compute data quality indicators. We provide an easy-to-use web interface and API for submitting of MeSQuaL queries. The interface will be available at `http://www.irisa.fr/mesqual`. Our current implementation supports extended SQL queries for Oracle, PostgreSQL, and MySQL databases. We also provide an interface that assists users in declaring contract types and instances and writing SQuaL queries. Figure [3] shows a screenshot of MeSQuaL in action with a snapshot of a simple SQuaL query on the web interface. Note that the screenshot exactly represents the sample scenario discussed in this paper. We will show how a simple SQL-like language extension, SQuaL, can be used for specificatio of data quality auditing in a variety of user and domain needs. We will show attendees how MeSQuaL helps auditing the quality of data with extended-SQL queries, and how a user can interactively experiment with, customize and refin different quality checking functions to better understand what are the most effective methods for her domain or database. Finally, we will discuss the details MeSQuaL's query processing, limitations, and future directions. One interesting perspective of evolution for MeSQuaL's architecture is to add methods for discovering conditional functional dependencies [6] and computing pattern tableaux [11].

4. REFERENCES

[1] P. Andritsos, A. Fuxman, and R. J. Miller. Clean Answers over Dirty Databases: A Probabilistic Approach. In *Proc. of ICDE 2006*, 2006.

[2] L. Berti-Équille and T. Dasu. New directions in data quality mining. Tutorial KDD 2009.

[3] S. C. Boulakia, O. Biton, S. B. Davidson, and C. Froidevaux. BioGuideSRS: querying multiple sources with a user-centric perspective. *Bioinformatics*, 23(10):1301–1303, 2007.

[4] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, S. Seltzsam, and K. Stocker. ObjectGlobe: Open Distributed Query Processing Services on the Internet. *IEEE Data Eng. Bull.*, 24(1):64–70, 2001.

[5] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection a survey. *IEEE TKDE*, 19(1):1–16, 2007.

[6] W. Fan, F. Geerts, and X. Jia. Semandaq: a data quality system based on conditional functional dependencies. *PVLDB*, 1(2), 2008.

[7] W. Fan, F. Geerts, and X. Jia. Conditional dependencies: A principled approach to improving data quality. In *Proc. of BNCOD*, 2009.

[8] H. Guo, P.-Å. Larson, and R. Ramakrishnan. Caching with Good Enough Currency, Consistency, and Completeness. In *Proc. of VLDB 2005*, 2005.

[9] J. Hellerstein. Quantitative data cleaning for large databases. Technical report, UC Berkeley, Feb. 2008.

[10] M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. D. Sarma, R. Murthy, and T. Sugihara. Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS (Demo). In *CIDR 2007*.

[11] D. Srivastava. Data auditor: Analyzing data quality using pattern tableaux. In *Proc. of ER 2009*.