That's All Folks! LLUNATIC Goes Open Source

Floris Geerts[1] Giansalvatore Mecca[2] Paolo Papotti[3] Donatello Santoro[2,4]

[1] University of Antwerp – Antwerp, Belgium [2] Università della Basilicata – Potenza, Italy
[3] Qatar Computing Research Institute (QCRI) – Doha, Qatar [4] Università Roma Tre – Roma, Italy

ABSTRACT

It is widely recognized that whenever different data sources need to be integrated into a single target database errors and inconsistencies may arise, so that there is a strong need to apply data-cleaning techniques to repair the data. Despite this need, database research has so far investigated mappings and data repairing essentially in isolation. Unfortunately, schema-mappings and data quality rules interact with each other, so that applying existing algorithms in a pipelined way – i.e., first exchange then data, then repair the result – does not lead to solutions even in simple settings. We present the LLUNATIC mapping and cleaning system, the first comprehensive proposal to handle schema mappings and data repairing in a uniform way. LLUNATIC is based on the intuition that transforming and cleaning data are different facets of the same problem, unified by their declarative nature. This holistic approach allows us to incorporate unique features into the system, such as configurable user interaction and a tunable trade-off between efficiency and quality of the solutions.

1. INTRODUCTION

Data transformation and data cleaning represent two important technologies in data management. Data transformation, or data exchange [2], is the process of translating data coming from one or more relational sources into a single target database. Data repairing [3] uses declarative constraints, like functional and inclusion dependencies, to detect and repair inconsistencies in the data.

It is natural to think of data exchange and data repairing as two strongly interrelated activities. In fact, the source databases are often structured according to different conventions and rules, and may be dirty. As a consequence, inconsistencies and errors often arise when the sources are brought together into a target schema that comes with its own integrity constraints.

On the contrary, the database literature has so far studied these two problems in isolation, with the consequence that there is currently neither a clear semantics, nor adequate techniques to handle data translation and data repairing in an integrated fashion. One might expect that pipelining data exchange algorithms [2] and data repairing algorithms like those in [3] is sufficient. Unfortunately, this is not the case. In fact, we have shown [5] that schema mappings and data quality constraints interact in such a way that simply pipelining the two semantics often does not return solutions.

LLUNATIC [4, 5] is the first tool that can solve mapping and cleaning scenarios in a unified fashion. It provides a novel semantics and new algorithms to generate solutions in the presence of dirty data. It offers a number of unique features, as follows:

(i) Users have a common language to express both schema mappings for data exchange purposes, and data quality rules for data repairing, within an integrated tool capable of executing them.

(ii) Users can declaratively express preference rules to select the best strategy to repair data in case of conflicts; for example, they can easily specify that values with higher confidence or currency should be preferred.

(iii) The semantics accommodates user inputs in the process in a principled way, for example to discard unwanted solutions or to interactively resolve conflicts among values for which there is no clear preference rule.

(iv) Finally, the system allows users to fine-tune the trade-off between the quality of the solutions and the efficiency of computing them. This is crucial due to the exponential nature of data repairing algorithms. In fact, different applications typically require different levels of quality. For example, a medical data-integration scenario requires very high accuracy, while a web aggregator of sports-related data may tolerate a minor quality to reduce users' efforts and execution time.

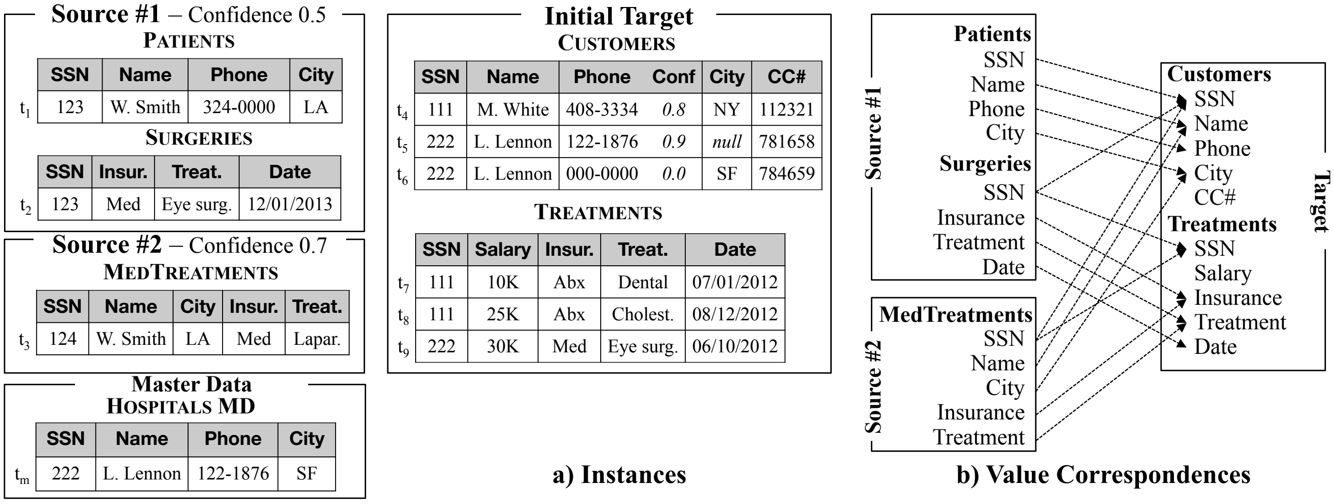
In the following sections we describe the organization of the demonstration. First, we outline the kind of mapping and cleaning scenarios that will be demonstrated. We concentrate on a synthetic example that serves the purpose of showing all of the main features of the system. During the demo, however, we will also discuss scenarios from real-life applications that have been conducted with the system, including medical data, events and sports data from the Web, and bibliographic data about publications. Given the focus of this proposal, we deliberately choose to omit many of the technical details that are in published papers [4, 5]. We rather concentrate on a description of the system from the user perspective, and illustrate what an attendee may learn by playing with it.

The system is available under an open-source license at the following URL: `http://db.unibas.it/projects/llunatic/`.

2. MAPPING AND CLEANING

Consider the data scenario shown in Figure 1. Here we have several different hospital-related data sources that must be correlated to one another. The first repository has information about Patients and Surgeries. The second one about MedTreatments. Our goal is

Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08.



Source #1 – Confidence 0.5

PATIENTS

	SSN	Name	Phone	City
t_1	123	W. Smith	324-0000	LA

SURGERIES

	SSN	Insur.	Treat.	Date
t_2	123	Med	Eye surg.	12/01/2013

Source #2 – Confidence 0.7

MEDTREATMENTS

	SSN	Name	City	Insur.	Treat.
t_3	124	W. Smith	LA	Med	Lapar.

Master Data

HOSPITALS MD

	SSN	Name	Phone	City
t_m	222	L. Lennon	122-1876	SF

Initial Target

CUSTOMERS

	SSN	Name	Phone	Conf	City	CC#
t_4	111	M. White	408-3334	*0.8*	NY	112321
t_5	222	L. Lennon	122-1876	*0.9*	*null*	781658
t_6	222	L. Lennon	000-0000	*0.0*	SF	784659

TREATMENTS

	SSN	Salary	Insur.	Treat.	Date
t_7	111	10K	Abx	Dental	07/01/2012
t_8	111	25K	Abx	Cholest.	08/12/2012
t_9	222	30K	Med	Eye surg.	06/10/2012

a) Instances

b) Value Correspondences

Figure 1: A Hospital Mapping and Cleaning Scenario.

to move data from the source database into a target database that organizes data in terms of Customers with their addresses and credit-card numbers, and medical Treatments paid by insurance plans.

Schema mappings To move data from the sources to the target, users may specify declarative *schema mappings*. As it is common, the mappings for our example are specified under the form of value correspondences in Figure 1.b. Intuitively, the lines from attributes of Source #2 to attributes of the target state that, for each tuple in the MedTreatments source table, there must be corresponding tuples in the Customers and Treatments target tables. Correspondences are translated into a set of mappings under the form of *tuple generating dependencies (tgds)* [10, 8] (which we omit here for the sake of space), and executed as SQL scripts.

Target Constraints Notice that, besides deciding how to populate the target to satisfy the mappings above, users must also deal with the problem of generating instances that comply with target constraints, as follows.

(i) *Functional and Inclusion Dependencies*: Traditionally, database architects have specified constraints of two forms: inclusion constraints and functional dependencies. In our example, we have a foreign-key constraint stating that the SSN attribute in the Treatments table references the SSN of a customer in Customers. The target database also comes with a number of functional dependencies: $d_1 = (\mathsf{SSN}, \mathsf{Name} \rightarrow \mathsf{Phone})$, $d_2 = (\mathsf{SSN}, \mathsf{Name} \rightarrow \mathsf{CC\#})$ and $d_3 = (\mathsf{Name}, \mathsf{City} \rightarrow \mathsf{SSN})$ on table Customers. Here, d_1 requires that a customer's social-security number (SSN) and name uniquely determine his or her phone number (Phone). Similarly for d_2 and d_3. Notice that we do not assume that the target database is empty. In fact, in Figure 1.a we have reported an instance of the target. There, the pair of tuples $\{t_5, t_6\}$ violates both d_1 and d_2; the database is thus dirty.

(ii) *Conditional Dependencies*: Besides standard functional and inclusion dependencies, the recent literature has shown that more advanced forms of constraints are often necessary [3]. Therefore, an expressive data-cleaning tool needs to support a larger class of data-quality rules. Here we mention *conditional functional dependencies* and *conditional inclusion dependencies* [3], among others. We designed our example in such a way to incorporate some of these as well. We assume two conditional functional dependencies (CFDs): (i) a CFD $d_4 = (\mathsf{Insur}[\mathsf{Abx}] \rightarrow \mathsf{Treat}[\mathsf{Dental}])$ on table Treatments, expressing that insurance company 'Abx' only offers dental treatments ('Dental'). Tuple t_8 violates d_4, adding more dirtiness to the target database. (ii) In addition, we also have an *inter-table* CFD d_5 between Treatments and Customers, stating that the insurance company 'Abx' only accepts customers who reside in San Francisco ('SF'). Tuple pairs $\{t_4, t_7\}$ and tuples $\{t_4, t_8\}$ violate this constraint.

(iii) *Master Data and Editing Rules*: Finally, as it is common in corporate information systems [7], an additional *master-data table* is available; this table contains highly-curated records whose values have high accuracy and are assumed to be clean. We also assume an additional *editing rule*, d_6, that states that whenever a tuple t in Customers agrees on the SSN and Phone attributes with some master-data tuple t_m in Hospitals MD, then the tuple t must take its Name and City attribute values from t_m, i.e., $t[\mathsf{Name}, \mathsf{City}] = t_m[\mathsf{Name}, \mathsf{City}]$. Tuple t_5 does not adhere to this rule as it has a missing city value (NULL) instead of 'SF' as provided by the master-data tuple t_m.

Mapping and Cleaning In summary, our example is such that: (a) it requires to map different source databases into a given target database; (b) it assumes that the target database may be non-empty, and that both the sources and the target instances may be dirty and generate inconsistencies when the mappings are executed; (c) it comes with a rich variety of data-quality constraints over the target.

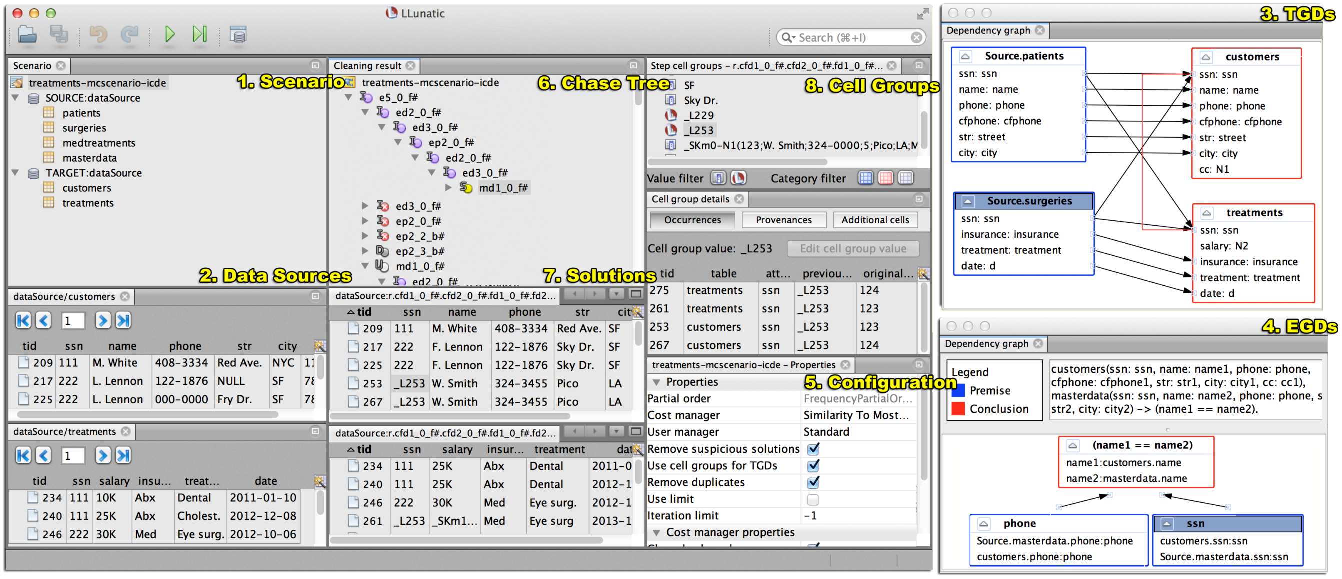
Given the source instances and the target, our goal is to generate a target instance that satisfies the mappings and that it is clean wrt target constraints. We call this a *mapping & cleaning scenario*.

LLUNATIC is the first system to provide a unified semantics and a set of executable algorithms to solve mapping and cleaning scenarios. The semantics is a conservative extension of the one of data exchange and incorporates many of the features found in data-repairing algorithms (see [6] for a comparison to other semantics).

3. QUICK OVERVIEW OF THE SYSTEM

The GUI of the LLUNATIC system is reported in Figure 2. Any experience with the system starts by specifying a scenario (*item 1* in Figure 2), i.e., a set of source databases and a target database, that does not need to be empty. Among the source databases, users may indicate some that are considered as *authoritative* – like master data. Both the source and the target database can be browsed to inspect the data (*item 2*).

The next step is concerned with specifying the mappings, i.e., the tgds, and the data quality constraints over the target. The system provides a graphical user interface for this task (*items 3 and 4*).



dataSource/customers

tid	ssn	name	phone	str	city	
209	111	M. White	408-3334	Red Ave.	NYC	11
217	222	L. Lennon	122-1876	NULL	SF	78
225	222	L. Lennon	000-0000	Fry Dr.	SF	78

dataSource/treatments

tid	ssn	salary	insu...	treat...	date
234	111	10K	Abx	Dental	2011-01-10
240	111	25K	Abx	Cholest.	2012-12-08
246	222	30K	Med	Eye surg.	2012-10-06

dataSource:r.cfd1_0_f#.cfd2_0_f#.fd1_0_f#.fd2...

tid	ssn	name	phone	str	cit
209	111	M. White	408-3334	Red Ave.	SF
217	222	F. Lennon	122-1876	Sky Dr.	SF
225	222	F. Lennon	122-1876	Sky Dr.	SF
253	_L253	W. Smith	324-3455	Pico	LA
267	_L253	W. Smith	324-3455	Pico	LA

dataSource:r.cfd1_0_f#.cfd2_0_f#.fd1_0_f#.fd2...

tid	ssn	salary	insur...	treatment	dat
234	111	25K	Abx	Dental	2011-0
240	111	25K	Abx	Dental	2012-1
246	222	30K	Med	Eye surg.	2012-1
261	_L253	_SKm1...	Med	Eye surg	2013-1

Cell group value: _L253

tid	table	att...	previou...	original...
275	treatments	ssn	_L253	124
261	treatments	ssn	_L253	123
253	customers	ssn	_L253	123
267	customers	ssn	_L253	124

treatments-mcscenario-icde – Properties

Properties	
Partial order	FrequencyPartialOr...
Cost manager	Similarity To Most...
User manager	Standard
Remove suspicious solutions	☑
Use cell groups for TGDs	☑
Remove duplicates	☑
Use limit	□
Iteration limit	-1
Cost manager properties	

customers(ssn: ssn, name: name1, phone: phone, cfphone: cfphone1, str: str1, city: city1, cc: cc1), masterdata(ssn: ssn, name: name2, phone: phone, s str2, city: city2) -> (name1 == name2).

Figure 2: LLUNATIC in action

In addition, users may specify configuration options (*item 5*), as explained in the following section.

Given a set of source instances, a target instance, a set of tgds and a set of data quality rules, LLUNATIC computes a set of *minimal solutions*, i.e., target instances that satisfy the constraints (tgds and egds), and "minimally change" the original target instance. To do this, it runs a parallel-chase procedure, that generates a chase tree (*item 6* in Figure 2). Leaves in the chase tree are solutions that can be inspected by users (*item 7*) to analyze the modifications to the original database.

Traditionally [3], solutions to data-repairing problems have been considered simply as sets of updates to the cells of the original database. Here, by *cell* we denote a tuple-attribute pair $t.A$, i.e., the value of attribute A in tuple t. In contrast, LLUNATIC models updates to the target database in terms of a novel data structure called *cell groups*. Cell groups are essentially "repair instructions with lineage", since they carry full provenance information for each change to the database, i.e.: (i) which conflicting cells in the target were modified, and which were their values; (ii) which value was chosen to repair the conflict; (iii) whether this value comes from one or more of the cells in the source databases, and if these cells are authoritative. Cell groups are the core element of the semantics, and represent the cornerstone of user interactions in LLUNATIC, as it will be discussed in the following section. In fact, proper care has been devoted to provide users with a flexible interface to inspect solutions and their cell groups (*item 8*).

4. EXPERIENCES WITH THE SYSTEM

The demonstration will be centered around a number of experiences that should illustrate what are the main challenges in mapping and cleaning scenarios and how the system solves them. Attendees will be able to interact directly with the system, in such a way that the process will resemble a hands-on tutorial. In the following paragraphs we illustrate these experiences.

a. Need and Benefits of an Integrated Semantics To start, we shall demonstrate why a new, integrated semantics for mapping and cleaning is necessary, and we cannot simply reuse existing algorithms for schema-mappings and for data-repairing. We have formally proven that such an approach does not work in general [5, 6]. To show this in a practical way, LLUNATIC can be configured to run with several semantics. One of them is the result of pipelining a standard chase algorithm for tgds [9, 8], with some of the popular algorithms to repair functional dependencies [1]. By running the system with this semantics on the scenario in Figure 1, and others, we will show that even in simple cases constraints interact in such a way that alternating the execution of mappings and the repairing of data quality constraints does not terminate. In addition, when the pipelining process terminates, the quality of the solutions computed by this procedure are quite poor, and definitely worse than those generated by LLUNATIC.

b. Solutions as Upgrades Data-repairing algorithms in the literature [3] try to repair a dirty database by making the smallest number of changes to its cells. There are various minimality conditions for repairs, and repairing algorithms are centered around these notions. Consider our example in Figure 1. Tuples $t_5 = \langle 222, \textit{L. Lennon}, \textit{122-1876} \ldots \rangle$, $t_6 = \langle 222, \textit{L. Lennon}, \textit{000-000} \ldots \rangle$ in the target violate the functional dependency d_1 : $\textsf{SSN}, \textsf{Name} \rightarrow \textsf{Phone}$ over table Customers, i.e., the two phone cells generate a conflict. There are many possible ways to repair the database. For example, we may change cell t_6.Phone to value *122-1876*; as an alternative, we may change cell t_5.Phone to value *000-0000*.

However, these are not the only possible repairs. For example, we might decide to change both cells to a new number, say *555-6789*. This repair, however, it is not minimal, since it requires to change two different cells, while a single cell-change is sufficient the ones above. There are many other minimal repairs, even for this very simple example. One of these changes t_5.SSN to a new SSN, say *333* (similarly for t_6.SSN). These are called "backward repairs", since they falsify the premise of the dependency instead of enforcing its conclusion.

Traditionally, as long as repairs involve a minimal number of changes, they are considered as equally acceptable. However, our example above shows that the minimality criterion is rather weak, and algorithms often choose a repair arbitrarily. LLUNATIC is based on a different philosophy. Its semantics is centered around the notion of an *upgrade*: a repair is a solution as long as it *improves* the quality of the original database. Improvements cannot be made arbitrarily. On the contrary, a change to the database is considered an upgrade only when it unequivocally improves the data.

To specify when changes are actually upgrades, users can declaratively specify *preference rules*. Consider our example above; notice that confidence values are associated with phone numbers (see Figure 1.a). Then, it is natural to state a preference rule saying that a value of a Phone-cell should be preferred to another as soon as its confidence is higher. In Llunatic this is easily done by a few clicks that identify Conf as the *ordering attribute* for Phone. Given this rule, not all repairs above are equally acceptable. More specifically, changing t_6.Phone to value *122-1876* is an upgrade of the target, while the opposite is not.

Ultimately, by specifying preference rules, Llunatic users may specify a *partial order* over the repairs of a dirty database. In turn, this yields an elegant notion of a solution: it is a minimal upgrade of the original database that satisfies the constraints.

c. Effective User-Interaction When no preference rule is available, Llunatic does not make arbitrary choices, and rather marks conflicts so that users may resolve them later on. Conflicts are marked using special values called *lluns*. A llun is a distinguished symbol, L_i, distinct from nulls and constants, that is introduced whenever there is no clear way to upgrade the dirty database by changing a cell to a new constant.

Users will be asked to consider again tuples t_5, t_6 above, and focus on the conflict on the credit-card numbers, 781658 for t_5, 781659 for t_6. In this case we have no clear preference strategy. Therefore, the only acceptable upgrade to the database according to the semantics of Llunatic is to change both t_5.CC# and t_6.CC# to a llun L. Here, L represents a value that may be either 781658 or 781659, or even a different constant as long as it ensures that it resolves the conflict and upgrades the quality of the database. Unfortunately, this value is currently unknown, and only an expert user may identify it, possibly at a later time.

Lluns are different from the ordinary variables used in previous approaches, due to their relationship to cell-groups. Recall that cell-groups are the building blocks used by Llunatic to specify repairs. They not only specify how to change the cells of the database, but also carry full provenance information for the changes. Based on this combination, Llunatic offers powerful features to collect user-inputs. In fact, we will show how users may easily stop the chase to provide inputs. They may pick up a node in the chase tree, consult its history in terms of changes to the original database, inspect the lluns that have been introduced, and analyze the associated cell groups. Based on this, informed decisions are taken in order to remove lluns and replace them with the appropriate constants, or discard unwanted repairs.

During the demonstration, we plan to convey to attendees two main notions. The first one is the fact that lluns and cell groups, together, provide a natural and effective basis to support users in their choices. The second one is that by appropriately providing inputs it is possible to drastically prune the size of the chase tree. Figure 3 shows one of our experiments: we run the chase for the same scenario several times, each with an increasing number of inputs provided by the user. With no user inputs, the chase tree counts over 130 leafs, i.e., possible solutions. With as little as 10 inputs, the tree collapses to a single solution. We will reproduce this example during the demonstration, to show how small quantities of inputs from the user may significantly prune the size of the chase tree, and therefore speed-up the computation of solutions.

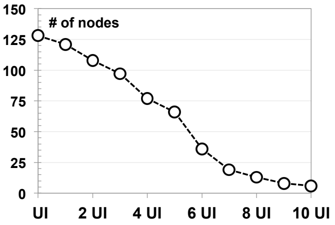


Figure 3: Impact of user-inputs on the size of the chase

d. Pay-As-You-Go Data Cleaning At the core of the system stands a powerful disk-based chase engine, capable of generating parallel-chase trees and multiple solutions.

As it is well known, the chase is a principled algorithm that plays a key role in many important database-related problems: data exchange, computing certain answers, query minimization, query rewriting using views, and constraint implication. Given its generality, Llunatic can be effectively used in all of these settings.

On real-life examples, it is crucial to guarantee good computation times. Llunatic strives to provide the best compromise between an exploration of the space of repairs that is more systematic and thorough that previous algorithms, and a good scalability on large mapping and cleaning problems. A key feature, in this respect, are *cost managers*. Cost managers are predicates over the chase tree that a user may specify in order to accept or refuse nodes. They can be used to implement different heuristics to prune the chase tree. For example, a *forward-only* cost manager accepts only forward changes and discards those that contain backward ones; a *maximum-size* cost manager accepts new branches in the chase tree as long as the number of leaves is less than N.

During the demonstration we plan to show how to obtain a good compromise between the quality of the solutions and the execution times by working with cost managers and their parameters, user-specified preference rules, and user-inputs, which, together, give a fine-grained control over the solution-generation process.

5. REFERENCES

[1] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, pages 541–552, 2013.

[2] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[3] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool, 2012.

[4] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The Llunatic Data-Cleaning Framework. *PVLDB*, 6(9):625–636, 2013.

[5] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, pages 232–243, 2014.

[6] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning: the Llunatic Way. Technical Report TR 1-2014 – University of Basilicata http://www.db.unibas.it/projects/llunatic/, 2014.

[7] D. Loshin. *Master Data Management*. Knowl. Integrity, Inc., 2009.

[8] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *PVLDB*, 4(12):1438–1441, 2011.

[9] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings: Scalable Core Computations in Data Exchange. *Inf. Systems*, 37(7):677–711, 2012.

[10] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.