What is the IQ of your Data Transformation System?

Giansalvatore Mecca[1] Paolo Papotti[2] Salvatore Raunich[3] Donatello Santoro[1,4]

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

ABSTRACT

Mapping and translating data across different representations is a crucial problem in information systems. Many formalisms and tools are currently used for this purpose, to the point that developers typically face a difficult question: "what is the right tool for my translation task?" In this paper, we introduce several techniques that contribute to answer this question. Among these, a fairly general definition of a data transformation system, a new and very efficient similarity measure to evaluate the outputs produced by such a system, and a metric to estimate user efforts. Based on these techniques, we are able to compare a wide range of systems on many translation tasks, to gain interesting insights about their effectiveness, and, ultimately, about their "intelligence".

Categories and Subject Descriptors: H.2 [Database Management]: Heterogeneous Databases

General Terms: Algorithms, Experimentation, Measurement.

Keywords: Data Transformation, Schema Mappings, ETL, Benchmarks.

1. INTRODUCTION

The problem of translating data among heterogeneous representations is a long-standing issue in the IT industry and in database research. The first data translation systems date back to the seventies. In these years, many different proposals have emerged to alleviate the burden of manually expressing complex transformations among different repositories.

However, these proposals differ under many perspectives. There are very procedural and very expressive systems, like those used in ETL [18]. There are more declarative, but less expressive schema-mapping systems. Some of the commercial systems are essentially graphical user-interfaces for defining XSLT queries. Others, like data-exchange systems, incorporate sophisticated algorithms to enforce constraints and generate solutions of optimal quality. Some systems are inherently relational. Others use nested data-models to handle XML data, and in some cases even ontologies.

In light of this heterogeneity, database researchers have expressed a strong need to define a unifying framework for data translation and integration applications [16, 6]. In fact, it would be very useful, given a task that requires to translate some input instance of a *source* schema into an output instance of the *target* schema, to have a common model to answer the following fundamental question: "*what is the right tool for my translation task?*"

Answering this question entails being able to compare and classify systems coming from different inspirations and different application domains. For this purpose, several benchmarks have been proposed [3, 29]. In this paper, we concentrate on an ambitious task that has not been addressed so far, i.e., we aim at measuring the *level of intelligence* of a data transformation system, in order to base the comparison upon this measure.

In our vision, the level of intelligence of the internal algorithms of a tool can be roughly defined as *the ratio between the* quality *of the outputs generated by the system, and the* amount of user effort *required to generate them*. In other terms, we want to measure, for each system, how much effort it takes to obtain results of the highest possible quality.

To make this rather general intuition more precise, we need several tools: (i) a notion of data-transformation system that is sufficiently general to capture a wide variety of the tools under exam, and at the same time tight enough for the purpose of our evaluation; (ii) a definition of the quality of a data translation tool on a mapping scenario; (iii) a definition of the user-effort needed to achieve such quality.

1.1 Contributions

We develop several techniques that contribute to give an answer to the question above.

(i) We introduce a very general definition of a *data-transformation system*, in terms of its input-output behavior; differently from earlier approaches that have focused their attention on the actual specification of the transformations, we see a system as a black box receiving as input some specification of the mapping task and an instance of the source schema, and producing as output an instance of the target schema; then, we analyze the system in terms of this input-output function.

(ii) We define the notion of *quality* of a data transformation tool on a given scenario as the *similarity* of the output instance wrt the *expected instance*, i.e., the "right" solution that the human developer expects for a given input. Notice that we allow nested data models and XML data, and therefore measuring the quality of an output imposes to compare two different trees, notoriously a difficult problem, for which high-complexity techniques are usually needed. We show, however, that for the purpose of this evaluation it is possible to define an elegant and very efficient similarity measure that provides accurate evaluations. This comparison technique is a much needed contribution in this field, since it is orders of magnitude

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

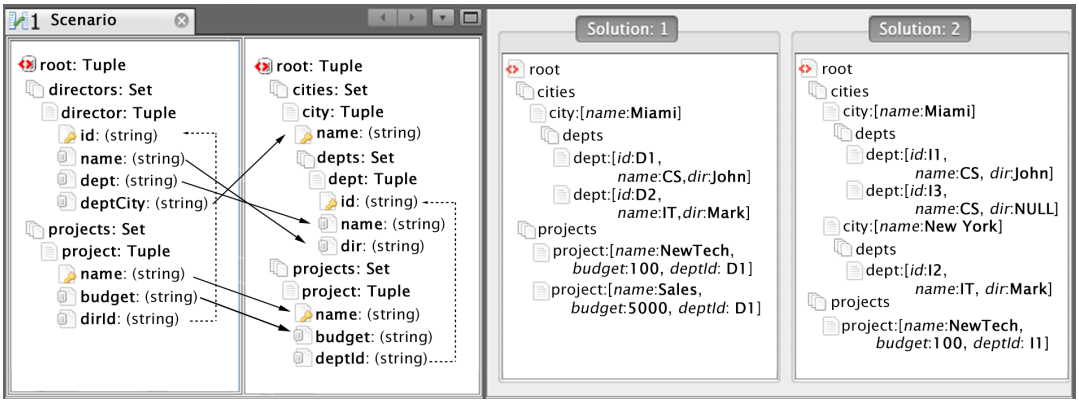

Figure 1: Sample Scenario in the GUI of an Open-Source Mapping System

faster than typical edit-distance measures, and scales up to large instances. In addition, it concretely supports the mapping improvement process, since it returns detailed feedback about mismatches between the two trees. By doing this, it helps users in understanding why their mapping is faulty, and proves much more effective than simple yes/no measures used in previous benchmarks.

(iii) Transformation systems typically require users to provide an abstract specification of the mapping, usually through some graphical user interface; to see an example, consider Figure 1, which shows the graphical specification of a mapping scenario. The figure shows the source and target schema with a number of *correspondences* used to provide a high-level specification of the mappings, as it is common in this framework. Based on the mapping specification, given a source instance the system generates an output, i.e., a target instance such as the ones shown on the right. Despite the fact that different tools have usually different primitives to specify the transformations, it is still possible to abstract the mapping specification as a *labeled input graph*; our estimate of the user effort is a measure of the size of an encoding of this graph inspired by the minimum description length principle in information theory [19]. We believe that this measure approximates the level of user effort better than previous measures that were based on point-and-click counts [3].

(iv) We develop a working prototype of our techniques, and use it to conduct a comprehensive evaluation of several data transformation systems. In this evaluation, we are able to gain a deeper insight about data transformation tools that exist on the market and in the research community, based on a novel graphical representation, called *quality-effort* graphs. More specifically, we fix a number of representative scenarios, with different levels of complexity, and different challenges in terms of expressive power. Then, we run the various tools on each scenario with specifications of increasing efforts, and we measure the quality of the outputs. The introduction of quality-effort graphs is a major contribution of this paper: it allows us to derive several evidences about how much intelligence the internal algorithms of a tool put into the solution, i.e., how fast the quality of solutions increases with respect to the increasing effort.

We want to make it clear that our primary goal is not comparing transformation systems in terms of expressiveness, as it has been done in previous benchmarks [3]. In fact, our approach has been conceived to be applicable to a wide variety of tools, which, as we discussed above, have rather different inspiration and goals. For example, in our evaluation, we consider both schema-mapping systems and ETL tools. It is well known that ETL tools are by far more expressive than schema-mapping systems, since they allow for a number of operations like pivoting, aggregates, rollups [18] that are not natively supported by other systems in our evaluation. As a consequence, in any application scenario in which these primitives are needed, the developer has very little choice. However, there are a number of transformation tasks for which tools of different kinds are indeed applicable, and therefore it makes sense to compare their level of effectiveness in carrying out these specific tasks. In fact, this work represents a concrete investigation of the trade-offs between declarative and procedural approaches, a foundational problem in computer science.

Using the framework proposed in this paper, we conduct a systematic evaluation of data transformation tools. We strongly believe that this evaluation provides precious insights in the vast and heterogeneous world of transformation systems, and may lead to a better understanding of its different facets.

- On one side, it may be the basis for a new and improved generation of benchmarks that extend the ones developed so far [3, 29]. Besides providing a better guide to data architects, this would also help to identify strong and weak points in current systems, and therefore to elaborate on their improvements.
- In this respect, our framework represents an advancement towards the goal of bringing together the most effective features of different approaches to the problem of data translation. As an example, it may lead to the integration of more sophisticated mapping components into ETL workflows [9].
- Finally, as it will be discussed in the following sections, it provides a platform for defining test scenarios for data exchange systems, another missing component in the mapping ecosphere.

1.2 Outline

The paper is organized as follows. Section 2 introduces the notion of a transformation system. The quality measure is introduced in Section 3, its complexity in Section 4. User efforts are discussed in Section 5. We introduce ways to define a scenario and to select gold standards in Section 6. Experiments are reported in Section 7. Related works are in Section 8, conclusions in Section 9.

2. TRANSFORMATION SYSTEMS

In our view, a *data-transformation system* is any tool capable of executing *transformation scenarios* (also called *mapping scenarios* or *translation tasks*). Regardless of the way in which transformations are expressed, in our setting these scenarios require to translate instances of a source schema into instances of a target schema.

The transformation system is seen as a black box, of which we are only interested in the input-output behavior, as shown in Figure 2.

For the purpose of our evaluations, we fix a set of *translation tasks*. A *translation task* is defined in terms of a quadruple $\{S, T, I_S, I_e\}$, where: (i) S is the source schema; (ii) T is the target schema; (iii) I_S is an input instance, i.e., a valid instance of S; (iv) I_e, the *expected output*, is an instance of T generated by applying the desired transformation to I_S.

Notice that the source and target schema may be either explicit, or implicit, as it happens in many ETL tasks. To better express the intended semantics of the transformation, it is possible that also some specification of the mapping, M_e, is given, in a chosen language. This, however, is not to be intended as a constraint on the way in which the transformation should be implemented, but rather as a means to clarify to developers the relationship between the expected output, I_e, and the input, I_S, in such a way that $I_e = M_e(I_S)$.

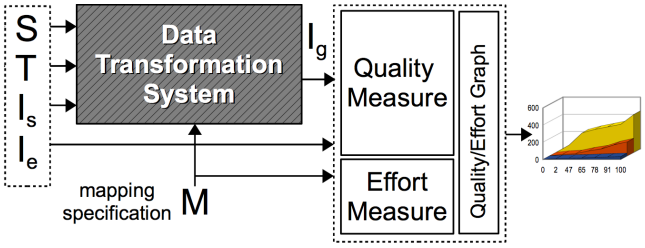

Figure 2: Architecture of the Evaluation Framework

We want to emphasize the high level of generality with which the various components of the architecture have been chosen.

First, as it was discussed in the previous section, we encode the behavior of a system in terms of its input-output function. By doing this, we allow ample space to the different formalisms that are commonly used to express the semantics of the transformation and the mapping specification, M_e; it is in fact perfectly fine to specify the mapping as a query Q in a concrete query language – say XQuery – as STBenchmark does [3]. As an alternative, the transformation can be expressed as a set of embedded dependencies (tgds and egds) [10], as it happens in schema-mapping and data-exchange works. To be even more general, it is also possible to assume that a procedural specification of the mappings is provided, as it happens in some ETL tools.

Second, we are not constraining the primitives offered by the systems under evaluation, nor the way in which users express the transformation. This can be done through a GUI, or – going to the opposite extreme – even by manually writing a query in an executable query language.

Finally, the data model according to which S and T are constructed is a generic one. For the purpose of this paper, we adopt a nested-relational data model, as it will be detailed in the following Section, that provides a uniform representation for most of the structures typically found in concrete data models – primarily relational and, to some extent, XML. However, as it will be clear in the following Section, any data model based on collections, tuples of attributes and relationships is compatible with our framework.

A distinctive feature of our approach is that we assume that a "gold standard", I_e, has been fixed for each scenario. We discuss in detail how this can be done in Section 6. For the time being, we want to emphasize that this is the basis of our evaluation. More specifically, as shown in Figure 2, assume a translation task $\{S, T, I_S, I_e\}$ is given, and we need to evaluate a transformation system *TS*. We proceed as follows:

- we use the primitives offered by *TS* to express the desired transformation; this gives us a specification M - possibly in a different formalism or query language wrt the expected one, M_e - that is supposed to have the same input-output behavior;
- we run M on the input instance, I_S, to generate the output, a target instance I_g;
- then, we measure the quality achieved by the system by comparing I_g to our expected output, I_e. If $I_g = I_e$, then *TS* achieves 100% quality on that specific translation task. Otherwise, the quality achieved by *TS* is the measure of the similarity between I_g and I_e;
- once the quality has been measured, we use the techniques in Section 5 to measure the user-effort, and generate the quality-effort graph.

3. THE QUALITY MEASURE

As discussed in the previous Section, our idea is to evaluate the quality of a tool by measuring the similarity of its outputs wrt a fixed, expected instance that has been selected in advance using one of the methods that will be introduced in Section 6.

Since we adopt a nested relational data model, our instances are trees, as shown in Figure 1. While there are many existing similarity measures for trees, it is important to emphasize that none of these can be used in this framework, for the following reasons:

(i) We want to perform frequent and repeated evaluations of each tool, for each selected scenario, and for mapping specifications of different complexity. In addition, we want to be able to work with possibly large instances, to measure how efficient is the transformation generated by a system. As a consequence, we cannot rely on known tree-similarity measures, like, for example, edit distances [7], which are of high complexity and therefore would prove too expensive.

(ii) The problem above is even more serious, if we think that our instances may be seen as graphs, rather than trees, as it will be discussed in the following paragraphs; we need in fact to check key/foreign-key references that can be seen as additional edges among leaves, thus making each instance a fully-fledged graph. Graph edit distance [13] is notoriously more complex than tree edit distance.

(iii) Even if we were able to circumvent the complexity issues, typical tree and graph-comparison techniques still would not work in this setting. To see this, consider that it is rather frequent in mapping applications to generate synthetic values in the output – these values are called *surrogate keys* in ETL and *labeled nulls* in data-exchange. In Figure 1, values $D1, D2, I1, I2, I3$ are of this kind. These values are essentially placeholders used to join tuples, and their actual values do not have any business meaning. We therefore need to check if two instances are identical up to the renaming of their synthetic values. We may say that we are rather looking for a technique to check *tree or graph isomorphisms* [12], rather than actual similarities.

It can be seen that we face a very challenging task: we need to devise a new similarity measure for trees that is efficient, and at the same time precise enough for the purpose of our evaluation.

In order to do this, we introduce the following key-idea: since the instances that we want to compare are not arbitrary trees, but rather the result of a transformation, we expect them to exhibit a number of regularities; as an example, they are supposedly instances of a

fixed nested schema that we know in advance. This means that we know: (a) how tuples in the instances must be structured; (b) how they should be nested into one another; (c) in which ways they join via key-foreign key relationships.

We design our similarity metric by abstracting these features of the two trees in a set-oriented fashion, and then compare these features using precision, recall and ultimately F-measures to derive the overall similarity. In the following paragraphs, we make this intuition more precise.

3.1 Data Model

We fix a number of *base data types*, τ_i – e.g., string, integer, date etc. – each with its domain of values, $dom(\tau_i)$, and a set of *attribute labels*, $A_0, A_1 \ldots$. A *type* is either a base type or a *set* or *tuple* complex type. A *set type* has the form $\mathsf{set}(A : \tau)$, where A is a label and τ is a tuple type. A *tuple type* has the form $\mathsf{tuple}(A_0 : \tau_0, A_1 : \tau_1, \ldots, A_n : \tau_n)$, where each A_i is a label and each τ_i is either a base type or a set type. A *schema* is either a set or a tuple type. Notice that schemas can be seen as (undirected) trees of type nodes. In the following, we will often blur the distinction between a schema and the corresponding tree.

Constraints may be imposed over a schema. A *constraint* is either a functional dependency or an inclusion constraint – i.e., a foreign key – defined in the usual way [1]. Both schemas in Figure 1 are constructed according to this data model. It can be seen that the source schema is relational, the target schema is nested.

Let us first formalize the notion of an *instance* of a schema in our data model, as a tree that may contain constants and *invented values*. More specifically, for each base type, τ_i, we consider the constants in the corresponding domain $dom(\tau_i)$. We also consider a countable set of special values, NULLS, typically denoted by $N_0, N_1, N_2 \ldots$ that are called *placeholders* – but in other terminologies have been called *labeled nulls* and *surrogates* – which we shall use to invent new values into the target when required by the mappings. An instance of the base type, τ_i, is a value in $dom(\tau_i) \cup$ NULLS. Instances of base types are also called *atomic values*.

Instances of a tuple type $\mathsf{tuple}(A_0 : \tau_0, A_1 : \tau_1, \ldots, A_n : \tau_n)$ are (unordered) tuples of the form $[A_0 : v_0, A_1 : v_1, \ldots, A_n : v_n]$, where, for each $i = 0, 1 \ldots, n$, v_i is an instance of τ_i. Instances of a set type $\mathsf{set}(A : \tau)$ are finite sets of the form $\{v_0, v_1, \ldots, v_n\}$ such that each v_i is an instance of τ. An instance of a schema is an instance of the root type. Sample instances can be found in Figure 1 ($D1, D2, I1, I2, I3$ are placeholders); from those examples it should be apparent that, like schemas, also instances can be seen as undirected trees. In the following, we shall often refer to tuple nodes in an instance simply as "tuples".

As it is common in nested data model, we assume a *partitioned normal form (PNF)* [26], i.e., at each level of nesting we forbid two tuples with the same set of atomic values. In light of this, we also forbid tuples whose atomic values are all placeholders.

3.2 Identifiers

Given a mapping scenario as defined in Section 2, we consider the target schema, T, and the expected solution, I_e. The features we associate with a target instance are *tuple* and *join identifiers*.

Tuple identifiers are string encodings of paths going from the root to tuple nodes. To introduce them, we shall first introduce a function $\mathsf{enc}()$ that we recursively apply to nodes. Given a node n in an instance tree, I, we denote by $\mathsf{father}(n)$ the father of n, if it exists; for the root node, n_{root}, $\mathsf{father}(n_{root})$ is a special, dummy node, $\perp$ such that $\mathsf{enc}(\perp)$ equals the empty string. Then, the $\mathsf{enc}()$ function is defined as follows:

- if n is an instance of a set node $\mathsf{set}(A : \tau)$, then $\mathsf{enc}(n) = \mathsf{enc}(\mathsf{father}(\mathsf{n})).A$;
- if n is an instance of a tuple node $[A_0 : v_0, A_1 : v_1, \ldots, A_n : v_n]$, then $\mathsf{enc}(n) = \mathsf{enc}(\mathsf{father}(\mathsf{n})).[A_{i_0} : \mathsf{enc}(v_{i_0}), \ldots, A_{i_k} : \mathsf{enc}(v_{i_k})]$ where $v_{i_0} \ldots v_{i_k}$ are the atomic values in the tuple, and A_{i_0}, $\ldots A_{i_k}$ appear in lexicographic order;
- if n is an instance of a base type τ, then $\mathsf{enc}(n) = value(n)$, where $value(n)$ equals n if n is a constant in $dom(\tau)$, or the string $null$ if n is a placeholder in NULLS.

Join identifiers are strings encoding the fact that the same placeholder appears multiple times in an instance. More specifically, given two tuples t_1, t_2 in an instance with atomic attributes A, B, respectively, they are said to be *joinable* over attributes A, B if the same placeholder N appears as the value of attribute A in t_1 and of attribute B in t_2. If $t_1 = t_2$, i.e., we consider the same tuple twice, then we require that $A \neq B$, i.e., A and B must be different attributes.

We are now ready to define the identifiers associated with an instance. Given an instance I of the target schema T, we define two different sets of strings associated with I. The *tuple ids*:

$$\mathsf{tids}(I) = \{\mathsf{enc}(t) \mid t \textit{ is a tuple node in } I\}$$

and the *join ids*:

$$\mathsf{jids}(I) = \{\mathsf{enc}(t_1).A = \mathsf{enc}(t_2).B \mid t_1, t_2 \textit{ are tuples in } I \textit{ joinable over } A, B \textit{ and } \mathsf{enc}(t_1).A, \mathsf{enc}(t_2).B \textit{ appear in lexicographic order}\}$$

Tuple and join identifiers for the instances in Figure 1 are reported in Figure 3. It is worth noting that the computation of tuple identifiers requires special care. As it can be seen in the figure, we keep the actual values of placeholders out of our identifiers, in such a way that two instances are considered to be identical provided that they have the same tuples and the same join pairs, regardless of the actual synthetic values generated by the system.

3.3 Instance Quality

Based on these ideas, whenever we need to compute the quality of a solution generated by a system, I_g, we compare I_g to the expected output, I_e by comparing their identifiers. More specifically: we first compute the tuple and join ids in I_e, $\mathsf{tids}(I_e)$, $\mathsf{jids}(I_e)$. Then, we compute the actual ids in I_g, $\mathsf{tids}(I_g)$, $\mathsf{jids}(I_g)$, and measure their precision and recall wrt to $\mathsf{tids}(I_e)$, $\mathsf{jids}(I_e)$, respectively, as follows:

$$p_{tids} = \frac{|\mathsf{tids}(I_g) \cap \mathsf{tids}(I_e)|}{|\mathsf{tids}(I_g)|} \quad r_{tids} = \frac{|\mathsf{tids}(I_g) \cap \mathsf{tids}(I_e)|}{|\mathsf{tids}(I_e)|}$$

$$p_{jids} = \frac{|\mathsf{jids}(I_g) \cap \mathsf{jids}(I_e)|}{|\mathsf{jids}(I_g)|} \quad r_{jids} = \frac{|\mathsf{jids}(I_g) \cap \mathsf{jids}(I_e)|}{|\mathsf{jids}(I_e)|}$$

As it is common, to obtain the distance between I_g and I_e, we combine precisions and recalls into a single *F-measure* [31], by computing the harmonic means of the four values as follows:

$$\mathsf{distance}(I_g, I_e) = 1 - \frac{4}{\frac{1}{p_{tids}} + \frac{1}{r_{tids}} + \frac{1}{p_{jids}} + \frac{1}{r_{jids}}}$$

Figure 3 reports the values of precision and recall and the overall F-measure for our example.

We want to emphasize the fact that our technique nicely handles placeholders. Consider for example instance $I_e = \{R(a, N_1), S(N_1, b)\}$, where a, b are constants, and N_1 is a placeholder. Any instance that is identical to I_e up to the renaming of placeholders –

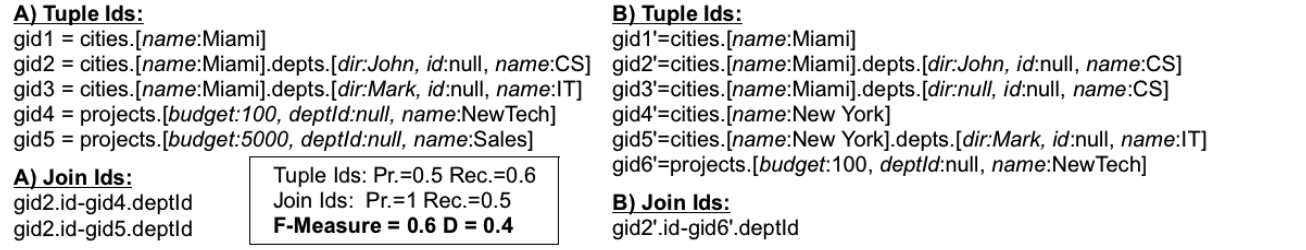

Figure 3: Comparing instances. Instance A is the expected output, B is the generated output

like, for example, $I_e = \{R(a, N_2), S(N_2, b)\}$ – has distance 0 wrt I_e. On the contrary, instances with different constants, and/or additional/missing tuples, are considered different and have distance greater than 0.

Notice also that our approach also allows us to easily detect the actual differences with respect to the expected output, i.e., tuples/surrogates that were expected and were not generated, and unexpected tuples/surrogates. Consider for example tuple ids; we define the set of *missing tuples* and the set of *extra tuples* as follows (here $-$ is the set-difference operator):

$$\mathsf{missingTuples}(I_g, I_e) = \mathsf{tids}(I_e) - \mathsf{tids}(I_g)$$
$$\mathsf{extraTuples}(I_g, I_e) = \mathsf{tids}(I_g) - \mathsf{tids}(I_e)$$

Similarly for *missing joins* and *extra joins*:

$$\mathsf{missingJoins}(I_g, I_e) = \mathsf{jids}(I_e) - \mathsf{jids}(I_g)$$
$$\mathsf{extraJoins}(I_g, I_e) = \mathsf{jids}(I_g) - \mathsf{jids}(I_e)$$

When reported to users, these sets represent a precious feedback, since they clearly point out what are the tuples and surrogates that cause mismatches between the expected solution and the one generated by a system. In other words, our similarity measure provides two different comparison indicators: the first one is a number measuring the overall similarity of the two instances; the second one is a detailed list of "edits" (tuple additions, tuple deletions, surrogate replacements) that should be applied to the instances to make them equal. In this respect, it is very similar to a traditional edit distance measure, as it will be discussed in Section 4.

4. COMPLEXITY AND EXPRESSIBILITY

Since our quality measure is based on the idea of comparing instances, we believe it is important to explore its relationship to other known tree-comparison techniques. As we have already noticed, the problem we deal with is at the crossroads of two different problems: the one of computing tree similarity (like, for example, tree edit distances), and the one of detecting graph isomorphisms (due to the presence of placeholders). In this section, we establish a number of results that relate the complexity and expressibility of our framework to those of other known techniques. In particular, we show that our technique is orders of magnitude faster than some of the known alternatives.

Assume that we need to compare two instances, of n_1, n_2 nodes, t_1, t_2 tuple nodes, and p_1, p_2 placeholders, respectively. Notice that $t_i < n_i, p_i < n_i$. Usually $p_i \ll n_i$. On the contrary, while the number of tuples is strictly lower than the number of nodes, for large instances t_i is of the same order of magnitude as n_i, since tuples in sets are the primary factor of multiplicity for the instance tree. Therefore, in the following, we shall approximate t_i by n_i. Let us call n the maximum value of n_1, n_2. To establish a complexity bound, let us analyze the various steps of the algorithm.

(i) As a first step, our algorithm computes tuple identifiers. This can be done by visiting the instance and keeping track of the labels and identifiers of the visited nodes. This step has therefore a cost of $O(n_1 + n_2)$ and generates $t_1 + t_2$ identifiers.

(ii) Then, we generate join pairs. For each placeholder, during the visit we also keep track of the identifiers of the tuples it appears in. To generate join pairs, we need to combine these identifiers in all possible ways. If we call o_{max} the maximum number of occurrences of a placeholder in one of the instances, we generate $n_{max} = \binom{o_{max}}{2}$ identifiers at most for each placeholder. Note that o_{max} is usually quite low, in the order of 2 or 3; moreover, it typically depends on the mapping and it is independent from n. As a consequence, we shall approximate the number of join identifiers by $O(p_1 + p_2)$.

(iii) Finally, in order to compare the two instances and compute the quality measure, we need to intersect the two identifier sets. To do this, we can use a sort-merge algorithm, with a cost of $O(t_1 log(t_1) + t_2 log(t_2))$ to compare tuple identifiers, and $O(p_1 log(p_1) + p_2 log(p_2))$ to compare join identifiers. Since, as discussed above, $p_i \ll n_i$ and we can approximate t_i by n_i, we have a total cost of $O(n_1 log(n_1) + n_2 log(n_2))$. The overall time cost is therefore lower than $O(nlog(n))$.

We are interested in comparing this bound with those of other comparison techniques. In order to do this, we shall discuss two different cases.

Case 1: General Case Let us first discuss the case in which the two instances may contain placeholders. In this case, we find it useful to formalize the relationship between our instances, which we have defined in Section 3 as undirected trees, and their graph counterpart. Given a schema T, and an instance I of T, the *instance graph* associated with I contains all nodes and edges in I. In addition it contains an additional edge between each pair of distinct leaf nodes labeled by the same placeholder N.

Based on this, we are not looking for *identical* instance graphs, but rather *isomorphic* instance graphs, i.e., instance graphs that are identical up to the renaming of placeholders. We say that two instance graphs G_1, G_2 are *isomorphic* if there is a bijective mapping, h, between the nodes of G_1 and G_2 such that: (i) for each pair of nodes n_1, n_2 in G_1 there exists an edge between n_1 and n_2 if and only if there exists and edge between $h(n_1)$ and $h(n_2)$ in G_2; (ii) in addition, the mapping h preserves the labels of non-placeholder nodes, i.e., if n is labeled by a constant v, then also $h(n)$ is labeled by v. We can state the following soundness property for our algorithm:

THEOREM 4.1. *Given two instances I_1, I_2, then* $\mathsf{distance}(I_1, I_2) = 0$ *if and only if the corresponding instance graphs G_1, G_2 are isomorphic.*

Notice that the general problem of computing graph isomorphisms is known to be in *NP*, and only high complexity algorithms are currently known for its solution [12]. This makes these techniques hardly applicable in practice. Our technique, on the contrary, runs with an $O(nlog(n))$ time bound, and therefore easily allows the comparison of large instances. The improvement in the complexity bound is not surprising, since the problem we concentrate on is a simplified variant of the general graph-isomorphism problem. It is, however, remarkable that for this specific instance of the problem such a significant speed-up is achievable.

Case 2: Absence of Placeholders It is interesting to evaluate the performance of our algorithm also in the case in which instances do not contain placeholders. In this case, the instance graph coincides with the instance tree, and the notion of isomorphism degrades into

the usual one of equality. It would therefore be possible to apply one of the known tree-similarity algorithms, like, for example, tree-edit distances, to compute the quality of solutions.

The tree-edit distance measures the similarity of a tree T_1 with respect to a tree T_2 by counting the number of operations that are needed to transform T_1 into T_2. In our framework, we concentrate on two operations: node insertions and node deletions. We call these $insert(T_1, T_2)$, $delete(T_1, T_2)$.

In order to state our result, we need to introduce an additional data structure, called the *tuple-reduced tree* associated with an instance I. The latter is obtained by taking the induced subtree of I corresponding to set and tuple nodes, and then by relabeling tuple nodes by the corresponding tuple identifiers in I. In essence, such subtree is obtained by discarding nodes corresponding to atomic attributes, and by embedding their values into tuple labels (Figure 1 shows the tuple-reduced trees of the two instances).

We can state the following result, which shows that our algorithm correctly computes the tree edit distance of tuple-reduced trees.

THEOREM 4.2. *Given two instances I_1, I_2 without placeholders, then* $\mathsf{distance}(I_1, I_2) = 0$ *if and only if the corresponding tuple-reduced trees T_1, T_2 are identical. Moreover, the set of extra tuples (missing tuples) detected by the algorithm is equal to the set of node insertions (node deletions, respectively) computed by the tree-edit distance over T_1, T_2, i.e.,* $\mathsf{missingTuples}(T_1, T_2) = insert(T_1, T_2)$, $\mathsf{extraTuples}(T_1, T_2) = delete(T_1, T_2)$.

Notice that the best algorithms [7] to compute edit distances have an upper bound of $O(n_1 n_2)$. Better bounds can be achieved for ordered trees, but this is clearly not the case we consider, since we do not assume any order of appearance of tuples inside sets. Our bound improves this for the particular instance of the problem we concentrate on. Other works [4] have reported bounds similar to ours, but, unlike ours, results computed by these algorithms cannot be related to tree edit distances.

5. ESTIMATING USER EFFORTS

Now that we have formalized our quality metric, we need to introduce a way to measure user efforts. This is needed in order to compute our quality-effort graphs.

Previous works have used point-and-click counts to do this [2]. However, click-counts are often unreliable since they are heavily influenced by GUI layout choices and, even more important, by the level of expertise of users. On the contrary, to estimate user-efforts, we measure the complexity of the mapping specification by means of an information-theoretic technique. We model the specification complexity as an *input graph* with labeled nodes and labeled edges. This model is general enough to cover a very broad range of approaches to data transformations.

More precisely, the *input graph* is an undirected graph $G = (N, E)$, where N is a set of nodes, and E is a set of edges. Nodes are partitioned in two groups: *schema nodes*, and *additional nodes*. Given a graphical specification of a transformation, we build the corresponding input-graph as follows: (i) every element in the source and target schemas is a schema node in the graph; (ii) arrows among elements in the GUI become edges among nodes in the graph; (iii) a tool may provide a library of graphical elements – for example to introduce system functions – that are modeled as additional nodes in the graph; (iv) extra information entered by the user (e.g., manually typed text) is represented as labels over nodes and edges.

We report in Figure 4 the input-graph for a sample scenario specified using a commercial mapping system (this scenario will be discussed in Section 6.1). In this example there are 31 nodes for the source and target schemas; arrows are drawn among schema nodes to specify logical correspondences. The white boxes are graphical elements on the GUI that specify functions used to manipulate source values; there are 5 functions in this scenario that generate additional nodes in the graph.

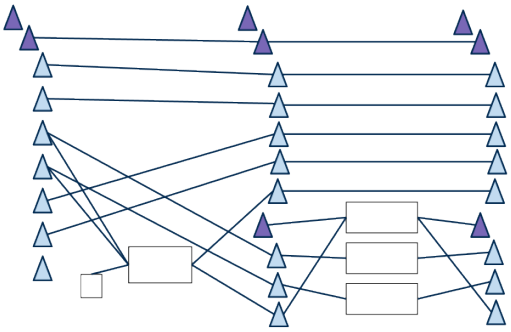

Figure 4: Sample input graph.

We measure the size of such graphs by encoding their elements according to a *minimum description length* technique [19], and then by measuring the size in bits of such description, with the following algorithm:

- as a first step, we assign a unique id to each node in the graph, and compute the minimum number of bits, b_n, needed to encode node ids. Our example uses a total of 36 nodes, so that we need 6 bits for their encoding. Therefore $b_n = 6$ bits;
- next, we measure the size of the encoding of nodes in the graph; the source and target schema nodes are considered as part of the input and therefore are not counted in the encoding. On the contrary, we encode the additional function nodes. To do this, we build an encoding of the corresponding function, by assigning a binary id to each function in the library. Based on the size of the library of functions provided by the tool (approximately 150), every function node in our example requires an additional 8 bits for its encoding; therefore, encoding additional nodes requires $5 * (6 + 8)$ bits;
- then, we measure the encoding of edges; each edge in the graph is encoded by the pair of node ids it connects, with a cost of $2b_n$ bits; in our example, the graph contains 26 edges (without labels) that we shall encode by $2 * 6$ bits;
- finally, node and edge labels are treated as arbitrary strings, and therefore are encoded in ASCII; in this example, one of the graph nodes must be labeled by a single char, for which we need 8 bits.

The specification complexity is therefore given by the following sum of costs: $(5 * (6 + 8)) + (25 * (2 * 6)) + (6 + 8) = 384$ bits. With the same technique we are able to measure the size of the specification needed by different systems, and compare efforts.

We want to stress that this representation is general enough to accomodate very different visual paradigms. To give another example, consider Figure 5. It shows the input graph for a complex ETL workflow. In the graph, each oval represents a workflow step. The schemas and intermediate recordsets used in the various steps are encoded as nodes, with their correspondences as edges.

Our abstraction can also be used to model purely textual queries. In this case, the input graph degenerates into a single node, labeled with the textual specification of the query.

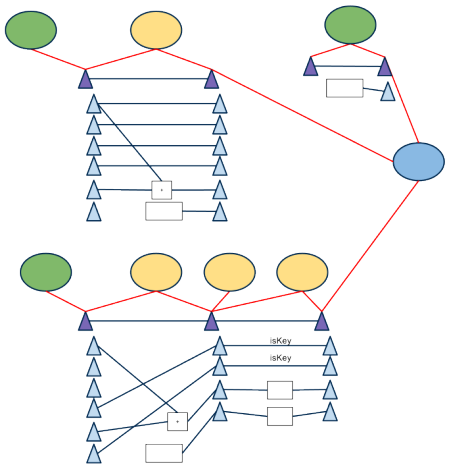

Figure 5: Input graph for an ETL workflow.

6. DESIGNING SCENARIOS

Based on the techniques introduced in Sections 3 and 5, we have conducted a comprehensive experimental evaluation based on several transformation scenarios. Before getting to the details of these experimental results, we want to discuss how it is possible to design scenarios in this framework.

Designing a scenario amounts to choosing a source and target schema, and an input-output function, i.e., a set of source instances given as inputs, and a set of target instances considered as expected outputs, i.e., as the gold standard. It can be seen that the critical point of this process consists in deciding what the expected output should be when some input is fed to the system.

6.1 Selecting the Gold Standard

In this respect, our approach is very general, and allows one to select expected outputs in various ways. It is in principle possible to craft the expected output by hand.

A more typical way – as it was done, for example, in [2] – would be to express the input-output function as a query Q written in a concrete query language, say SQL or XQuery. In this case, for a given input I_S, the expected output would be $I_e = Q(I_S)$. For more complex workflows, one would need to choose a reference system, design the transformation using the system, compute the result, and take this as the expected output.

This approach has the disadvantage of forcing designers to express all of the details of the intended transformation, thus leaving very limited space to explore the variants supported by alternative tools. As an example, it requires to devise a precise strategy to generate surrogates – for example by using Skolem functions [17].

Data exchange theory [10, 11] represents an elegant alternative. More precisely, it provides a clean theoretical framework to state two essential elements of the description of a scenario: (a) a semantics of the transformation in terms of logical expressions; (b) a clear notion of optimal solution for each mapping scenario and source instance.

A mapping is specified using dependencies, expressed as logical formulas of two forms: *tuple-generating dependencies* (tgds) and *equality-generating dependencies* (egds). *Source-to-target tgds* (s-t tgds) are used to specify which tuples should be present in the target based on the tuples that appear in the source. *Target tgds* and *target egds* encode foreign-key and key constraints over the target.

As an example, consider the *vertical partition* scenario of ST-Benchmark [2]. This scenario takes a single, non-normalized table, *SourceReaction*, and splits it into two tables, *Reaction, ChemicalInfo*, joined via a key-foreign key reference. In doing this, it introduces a surrogate key.[1] Notice that the input graph in Figure 4 refers exactly to this scenario.

The intended transformation can be formally expressed in data exchange as follows:

$$m_1.\ \forall e, n, d, q, c, o, r : \textit{SourceReaction}(e, n, d, q, c, o, r) \rightarrow \exists F : \textit{Reaction}(e, n, c, o, F) \wedge \textit{ChemicalInfo}(d, q, F)$$

$$t_1.\ \forall e, n, c, o, f : \textit{Reaction}(e, n, c, o, f) \rightarrow \exists D, Q : \textit{ChemicalInfo}(D, Q, f)$$

$$e_1.\ \forall d, q, f, f' : \textit{ChemicalInfo}(d, q, f) \wedge \textit{ChemicalInfo}(d, q, f') \rightarrow (f = f')$$

Here, m_1 is a *source-to-target tgd* that states how the target tables should be materialized based on the source data. Dependency t_1 is a *target tgd* stating the referential integrity constraint over the target tables. Finally, e_1 is a *target egd* expressing the fact that the first two attributes are a key for the *ChemicalInfo* table. Notice how, by using existential variables, the tgds express the requirement that a placeholder is used to correlate the target tables, without actually providing any technical details about its generation.

A data exchange problem may have multiple solutions on a given source instance, with different degrees of quality. The *core universal solution* [11] is the "optimal" one, since, informally speaking, it is the smallest among the solutions that preserve the mapping. A nice property of the core solution is that there exist polynomial algorithms [11, 15, 30], some of which have been proven to be very scalable [23] to generate the core.

In light of this, a natural way to design scenarios would be to express the semantics of the transformation as a set of logical dependencies, and to pick the core universal solution as the expected output for a given source instance. In our experiments, whenever this was possible, we used this approach. We want, however, to remark that this is not the only alternative, as discussed above, nor it is imposed by the method.

6.2 Test Scenarios

This discussion suggests another promising facet of our evaluation technique. In fact, it shows that it can be used as the basis for a *regression-test tool* for schema-mapping systems whose algorithms are under development. Suppose, in fact, that we need to test the algorithms of a new system, currently under development. We may proceed as follows: (a) fix a set of mapping scenarios expressed as sets of logical dependencies; (b) use an existing system – for example [21, 22] or [30] – to generate the core universal solution; (c) run the system under evaluation to generate the output, and measure its distance from the expected output. Then, in case of errors, use the feedback to correct the translation algorithms.

In essence, this is an alternative evaluation process with respect to the one that we have discussed so far, in which there is less emphasis on quality-effort trade-offs, and more on correctness. To the best of our knowledge, this is the first proposal towards the development of test tools for schema-mapping systems.

7. EXPERIMENTAL RESULTS

The techniques presented in the paper have been implemented in a working prototype using Java; the prototype has been used to perform a large experimental evaluation. To start, we show how the proposed quality measure scales well up to very large instances and outperforms existing techniques. Then, we use our framework to compare several data-translation systems on a common set of scenarios and discuss the results. All the experiments have been

[1] We are actually considering a variant of the original scenario in [3] that has no key constraints.

conducted on a Intel Xeon machine with four 2.66Ghz cores and 4 GB of RAM under Linux.

7.1 Scalability of the Quality Measure

To test the scalability of our quality measure on instances of large size, we used the instance provided for the *Unnesting* scenario in STBenchmark [3]. The original instance contains 2.2 millions nodes and has a size of 65 MB. We generated a modified version of the same file by randomly introducing errors, in such a way that the original and the modified instance had a similarity of 90% according to our quality metrics. Then, we extracted smaller subtrees from this complete instance, in order to obtain a pool of instances of increasing size, varying from a few hundreds to 2 million nodes.

Since the nesting scenario does not require the generation of surrogates, we were able to compare experimentally our measure against an implementation of a tree-edit distance algorithm for ordered trees [5]. We tested ordered instances because of the lack of implementations of the tree edit-distance algorithm for unordered ones. We remark, however, that in the general case of unordered trees the complexity of the tree edit-distance computation is even higher.

We compared times of execution to compute our quality measure, the tree edit-distance on the original trees, and the tree edit-distance on the more compact tuple-reduced trees introduced in Section 4. We call this hybrid measure TED-TRT.

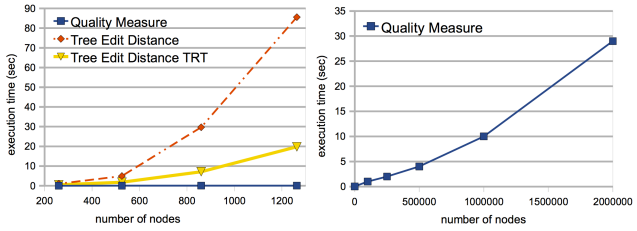

Figure 6: Execution times with files of increasing size.

Figure 6 summarizes the experimental results in terms of execution times and instance sizes. We experimentally confirmed the results reported in Section 4. More specifically, as stated in Theorem 4.2, the distance measured using our quality measure was identical to the one measured using the TED-TRT distance. In addition, the computation was orders of magnitude faster. In the graph on the left hand side, we show the comparison of our measure against the tree edit-distance on small instances. On the right, we show times only for our quality measure, since the tree edit-distance implementation did not scale to large instances. On the contrary, our comparison algorithm scaled nicely: it computed the distance between two files of approximately 65 MB in less than 30 seconds, thus confirming the low complexity bound established in Section 4.

7.2 Comparison of the Systems

In the spirit of evaluating representative systems from different perspectives, we have included the following tools: (i) an open-source schema-mapping research prototype [21, 22]; (ii) a commercial schema-mapping system; (iii) a commercial ETL tool.

In addition, to discuss the relationship of our technique to ST-Benchmark, we have also evaluated the performances of a different schema-mapping tool [25] on some of the scenarios. It is worth mentioning that we also considered the idea of including in our evaluation the open-source OpenII data integration tool [28]. We found out that, while promising, the data translation module of the current version of OpenII is still in a rather preliminary state, and therefore decided to exclude it from the evaluation.

To conduct the evaluation, we selected twelve transformation tasks from the literature, with different levels of complexity. The selected tasks can be roughly classified in three categories:

(i) basic mapping-operations, taken from STBenchmark [3]: *Copy* (Cp), *Denormalization* (Dn), *Vertical Partition* (VP), *Key Assignment* (KA), *Value Management* (VM), and *Nesting* (Ne);

(ii) advanced mapping operations, taken from [23, 21]: *Minimization* (Mi), *Fusion* (Fu), *Nested Keys* (NK); these scenarios require the management of more complex target constraints wrt those above;

(iii) typical ETL tasks: *Aggregation* (Ag) is a simplified version of the *line* workflow from an ETL benchmark [29], while *AccessLog* (AL) and *CreditCards* (CC) are taken from the http://cloveretl.com/-examples Web page.

Of these twelve scenarios, five required the introduction of surrogates; two use nested models, the others are relational. For each scenario we have identified the gold standard, that is, the desired target solution for the given translation task and source instance. As discussed in Section 6, expected solutions were identified in such a way that they contained no unsound or redundant information [11]. Then, we tried to generate the expected output with each of the tools, and measured efforts and quality.

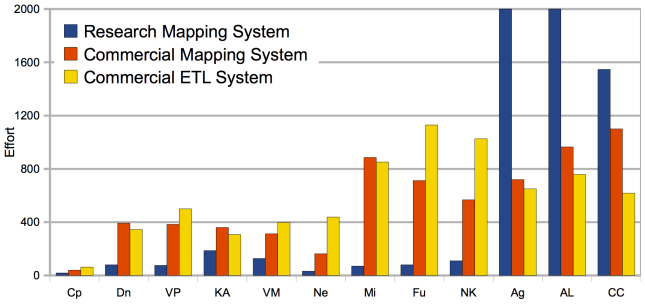

Figure 7: Effort needed by the systems to obtain 100% quality in the various scenarios.

Notice that the three tools under exam use fairly different strategies to compute the transformations: the research mapping systems generate SQL or XQuery code, the commercial mapping system typically generates XSLT, while the ETL workflow requires the internal engine in order to produce results. Nevertheless, our general view of the transformation process permits their comparison. We also want to mention that, while it is not in the scope of this paper to compare the systems in terms of scalability, all of the systems in our experiments scaled well to large instances.

Results are shown in Figure 7 and 8. More specifically, Figure 7 reports the effort needed by the various systems to obtain 100% quality in the various scenarios, while Figure 8 shows the quality-effort graphs. There are several evidences in the graphs that we want to highlight. Let us first look at Figure 7. As a first evidence, we note that the research mapping tool required considerably less effort than the commercial counterparts on the basic and advanced mapping tasks. On these tasks, the ETL tool was the one requiring the highest effort to compute the requested transformations. However, we also note that the situation is reversed for the ETL-oriented tasks. Notice how the commercial mapping system had intermediate performances. This suggests that these tools are progressively evolving from the schema-mapping ecosphere into fully-fledged ETL tools.

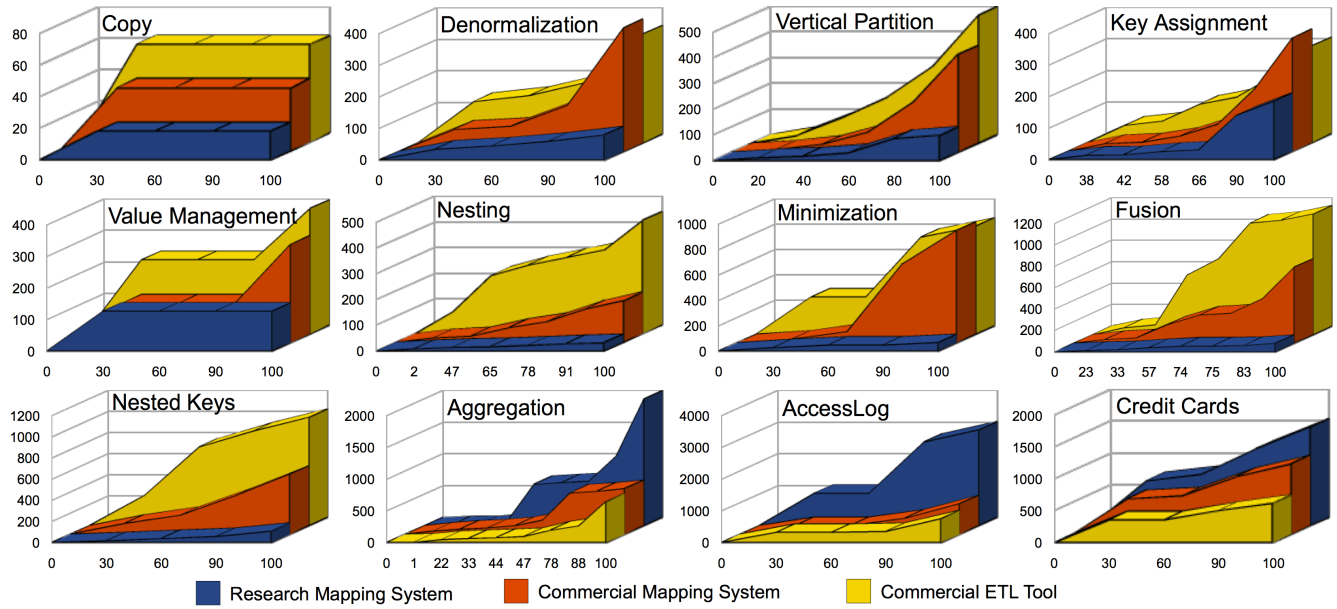

Figure 8: Quality-effort graphs. For each scenario, the smaller the area, the higher is the IQ of a transformation tool.

To get more insights about this, let us look at the quality-effort graphs in Figure 8. The Figure shows how much effort is needed to get a certain level of quality with a given system. Recall that one of our aims is that of measuring the "level of intelligence" of a tool, as its quality/effort ratio. From the graphical viewpoint, this notion of IQ can be associated with the area of the graph delimited by the effort-quality function: such area can be taken as a measure of the progressive effort needed by a tool to achieve increasing levels of quality in one experiment. The smaller the area, the higher is the IQ of a system.

We can observe that the research mapping tool handles in a more natural way some complex operations, like nesting values or data fusion and redundancy removal. But, as soon as the task at hand becomes more procedural, like in the key-assignment scenario or when typical ETL-like operations such as aggregations are required, this advantage becomes less visible or it is completely lost. This is apparent in the *Aggregation*, *AccessLog*, and *CreditCard* scenarios – three rather typical data warehousing transformations. For these tasks, while the effort needed to compute the transformation in commercial systems was in line with those of previous scenarios, in the case of the research mapping tool the cost was enormously increased by the need of manually changing the generated SQL code in order to introduce the needed aggregates. In fact, there is no declarative way of expressing aggregates in data exchange yet.

In fact, our experiments confirm the intuition that the sophisticated declarative algorithms introduced in recent years in schema-mappings research may really provide some advantage in terms of productivity to the data architect. However, this advantage is somehow confined to the typical scope of applicability of schema-mappings. When users want to deal with more complex scenarios, i.e., transformations requiring a rather fine-grained manipulation of values, the adoption of more procedural paradigms brings some advantages.

We strongly believe that these results clearly confirm the need for a new strategy for developing data-transformation tools, which brings together the best of both worlds, in the spirit of [9]. While the expressive power of procedural ETL tools is necessary to properly handle the wide range of transformations that a data architect typically faces, still there are a variety of mapping tasks – ranging from conjunctive queries, data-fusion and instance minimization, to management of functional dependencies and nested constraints – for which research mapping tools provide building blocks that may represent a very powerful addition to commercial transformation systems.

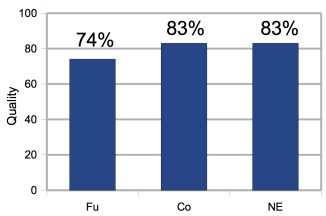

Figure 9: Comparison to STBenchmark.

As a final remark, we think it is useful to put in perspective the results reported in [3] using STBenchmark. STBenchmark uses a coarse-grained yes/-no style of evaluation to measure the performance of a mapping system on a given scenario. Our technique gives a better understanding of the performances of a system, especially in those cases in which it is not capable of fully capturing the semantics of a transformation.

We considered the mapping algorithm used in the STBenchmark evaluation [25], and evaluated its performance on three advanced mapping scenarios. The table in Figure 9 shows the maximum quality that we were able to obtain with such a system by using the GUI only, i.e., without manually changing the SQL or XQuery code (operation, which, as discussed has very high costs in our metrics). It can be seen that in three of the scenarios the system failed to achieve 100% quality. Differently from the yes/no output of STBenchmark, our comparison technique provided a detailed account of the results obtained by the algorithm in these cases. This is a further evidence that our evaluation framework may help to improve the design of benchmarks, and to gain better insights about the effectiveness and limitation of tools.

8. RELATED WORKS

Data translation tasks in enterprise settings are very often tackled using *Extraction-Transform-Load* (or *ETL*) tools, where transformations are defined by using sequences of building blocks in a rather procedural fashion. Representative examples of ETL sys-

tems can be found on the market (e.g., Oracle Warehouse Builder or IBM Information Server) (www.oracle.com/technetwork/developer-tools/warehouse, www-01.ibm.com/software/data/integration) and in the open-source community (e.g., Clover ETL or Talend Open Studio) (www.cloveretl.com, www.talend.com).

Different generations of research mapping systems have been developed in the last ten years. A first generation of schema-mapping systems [24, 25] has been followed first by an intermediate generation [30, 23], focused on the quality of the solutions, and then by a second generation [21, 22], which is able to handle a larger class of scenarios. In addition, Altova Mapforce and Stylus Studio (www.altova.com/mapforce, www.stylusstudio.com) are examples of commercial mapping systems.

Several works have studied the issue of quality in mapping-systems with a different focus, either to check desirable properties [27], or to rank alternative mappings [8].

The analysis of the information-integration and business-intelligence market is the subject of a large number of studies by business consulting firms [14]. However, these reports are centered around features and functionalities of the various systems, and do not rely on a quality evaluation metric such as the one developed in this paper. Closer to our approach, there exist some early benchmarks, designed both for ETL tools [29, 32, 20] and schema-mapping systems [3], which provide a basis for evaluating systems in the respective areas only. They mainly concentrate on expressibility, by introducing representative, small scenarios [3, 29], or on the efficiency evaluation [20]. With respect to measuring user efforts, existing solutions (such as [3]) rely on a simple metric based on the count of user actions (e.g., the number of clicks and the number of typed characters) in defining the transformation. Their main limitation, however, is that they fail in answering the main question addressed in this paper.

9. CONCLUSIONS

In this paper, we introduce a number of novel techniques that significantly improve the ones used in previous works. Our quality measure, coupled with the information-theoretic effort measure, enables the introduction of a new tool, called quality-effort graph, to study the effectiveness of a data transformation system.

This evaluation framework provides a clear perception of the level of intelligence of a data transformation tool, and ultimately measures how productive it is for a given scenario. For the specific problem of data-translation, it represents a concrete measure of the trade-off between declarative and procedural approaches.

In addition, we have shown that the technique is very scalable, despite the fact that we deal with a rather difficult problem, i.e., comparing possibly isomorphic graph-like structures.

We believe that this technique sheds some light on the right approach to solve data-integration problems: transforming and integrating data is a multi-faceted problem that requires a combination of state-of-the-art techniques, bringing together the expressibility of ETL tools and the declarative algorithms of schema-mapping research. Coupling together these approaches is a challenging but very promising research problem.

Acknowledgments The authors would like to thank Angela Bonifati and Yannis Velegrakis for the many helpful discussions on the subject of this paper.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. *PVLDB*, 1(2):1468–1471, 2008.

[3] B. Alexe, W. Tan, and Y. Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *PVLDB*, 1(1):230–244, 2008.

[4] N. Augsten, M. Bohlen, and J. Gamper. Approximate Matching of Hierarchical Data Using pq-Grams. In *VLDB*, pages 301–312, 2005.

[5] A. Bernstein, E. Kaufmann, C. Kiefer, and C. Bürki. SimPack: A Generic Java Library for Similiarity Measures in Ontologies. Technical report, Department of Informatics, University of Zurich, 2005.

[6] P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, pages 1–12, 2007.

[7] P. Bille. A Survey on Tree Edit Distance and Related Problems. *TCS*, 337:217–239, 2005.

[8] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema Mapping Verification: The Spicy Way. In *EDBT*, pages 85 – 96, 2008.

[9] S. Dessloch, M. A. Hernandez, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.

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

[11] R. Fagin, P. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *ACM TODS*, 30(1):174–210, 2005.

[12] F. Fortin. The Graph Isomorphism Problem. Technical report, Department of Computer Science, University of Alberta, 1996.

[13] X. Gao, B. Xiao, D. Tao, and X. Li. A Survey of Graph Edit Distance. *Pattern Analysis & Application*, 13:113–129, 2010.

[14] Gartner. Magic Quadrant for Data Integration Tools. http://www.gartner.com/technology/, 2011.

[15] G. Gottlob and A. Nash. Efficient Core Computation in Data Exchange. *J. of the ACM*, 55(2):1–49, 2008.

[16] L. M. Haas. Beauty and the Beast: The Theory and Practice of Information Integration. In *ICDT*, pages 28–43, 2007.

[17] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*, pages 455–468, 1990.

[18] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit*. Wiley and Sons, 2004.

[19] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[20] T. A. Majchrzak, T. Jansen, and H. Kuchen. Efficiency evaluation of open source etl tools. In *SAC*, pages 287–294, 2011.

[21] B. Marnette, G. Mecca, and P. Papotti. Scalable data exchange with functional dependencies. *PVLDB*, 3(1):105–116, 2010.

[22] 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(11):1438–1441, 2011.

[23] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings. In *SIGMOD*, pages 655–668, 2009.

[24] R. J. Miller, L. M. Haas, and M. A. Hernandez. Schema Mapping as Query Discovery. In *VLDB*, pages 77–99, 2000.

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

[26] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM TODS*, 13:389–417, October 1988.

[27] G. Rull Fort, F. C., E. Teniente, and T. Urpí. Validation of Mappings between Schemas. *Data and Know. Eng.*, 66(3):414–437, 2008.

[28] L. Seligman, P. Mork, A. Halevy, K. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick. OpenII: an Open Source Information Integration Toolkit. In *SIGMOD*, pages 1057–1060, 2010.

[29] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziovara. Benchmarking etl workflows. In *TPCTC*, pages 199–220, 2009.

[30] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan. Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries. *PVLDB*, 2(1):1006–1017, 2009.

[31] C. J. Van Rijsbergen. *Information Retrieval*. Butterworths (London, Boston), 1979.

[32] L. Wyatt, B. Caufield, and D. Pol. Principles for an etl benchmark. In *TPCTC*, pages 183–198, 2009.