Incrementally Improving Dataspaces Based on User Feedback[☆]

Khalid Belhajjame[*], Norman W. Paton, Suzanne M. Embury,
Alvaro A. A. Fernandes, Cornelia Hedeler

School of Computer Science, University of Manchester, Oxford Road, Manchester, UK

Abstract

One aspect of the vision of dataspaces has been articulated as providing various benefits of classical data integration with reduced up-front costs. In this paper, we present techniques that aim to support schema mapping specification through interaction with end users in a pay-as-you-go fashion. In particular, we show how schema mappings, that are obtained automatically using existing matching and mapping generation techniques, can be annotated with metrics estimating their fitness to user requirements using feedback on query results obtained from end users.

Using the annotations computed on the basis of user feedback, and given user requirements in terms of precision and recall, we present a method for selecting the set of mappings that produce results meeting the stated requirements. In doing so, we cast mapping selection as an optimization problem. Feedback may reveal that the quality of schema mappings is poor. We show how mapping annotations can be used to support the derivation of better quality mappings from existing mappings through refinement. An evolutionary algorithm is used to efficiently and effectively explore the large space of mappings that can be obtained through refinement.

User feedback can also be used to annotate the results of the queries that the user poses against an integration schema. We show how estimates for precision and recall can be computed for such queries. We also investigate the problem of propagating feedback about the results of (integration) queries down to the mappings used to populate the base relations in the integration schema.

Keywords: Dataspaces, Pay-as-you-go, User feedback, Data integration, Feedback propagation.

1. Introduction

The problem of data integration has been investigated for the past two decades with the aim of providing end users with integrated access to data sets that reside in multiple sources and are stored using heterogeneous representations [35]. The recent increase in the amount of structured data available on the Internet, due in significant measure to the Deep Web [36, 29, 37], has created new opportunities for using data integration technologies. Yet, in spite of the significant effort devoted to data integration, there seems to have been a limited impact in practice. By and large, data integration solutions are manually-coded and tightly bound to specific applications. The limited adoption of data integration technology is partly due to its cost-ineffectiveness [27]. In particular, the specification of schema mappings (by means of which, data structured under the source schemas is transformed into a form that is compatible with the integration schema against which user queries are issued) has proved to be both time and resource consuming, and has been recognized as a critical bottleneck to the large scale deployment of data integration systems [27, 38, 42].

To overcome the above issue, there have been attempts to derive schema mappings from information obtained using schema matching techniques [46, 16, 40, 15]. In their simplest form, matchings are binary relationships, each of which connects an element of a schema, e.g., a relational table in a source schema, to an element that is (predicted to be) semantically equivalent in another schema, e.g., a relational table in the integration schema. Schema matching techniques can be used as a basis for the generation of complex mappings that specify, for example, how the instances of one element of an integration schema can be computed by using the instances of two or more elements in source schemas [44, 56].

The mappings that are output by the above methods are based on heuristics. Therefore, many of them may not meet end user needs. Consider, for example, the case of Clio [44]. To specify complex mappings that involve two or more relations in the source schemas, these relations are combined using, for example, a join predicate that capitalizes on referential integrity constraints between the relations in question. While intuitive and useful, this approach does not guarantee that the mapping obtained meets the requirements of end users. The

[☆]The work reported in this paper was supported by a grant from the EPSRC.

[*]Corresponding author. Email address: khalidb@cs.man.ac.uk.

fact that the mapping that meets user requirements may not be generated is not due to faulty behaviour of the algorithm implemented by the Clio tool, but because the information provided by the matches used as inputs does not allow the correct mapping to be identified. This raises the question as to how the generated schema mappings can be verified.

A handful of researchers have investigated the problem of schema mapping verification [9, 13]. For example, the Spicy system provides functionalities for checking a set of mappings to choose the ones that represent better transformations from a source schema into a target schema [9]. To do this, instance-level data obtained using the mappings under verification are compared with instance-level data of the target schema, which are assumed to be available. The Spider system is another example of a tool for mapping verification [13, 4]. Specifically, the tool assists users in debugging schema mapping specifications by computing *routes* that describe the relationship between source and target data.

Using the above tools, the verification of schema mappings takes place before the data integration system is setup, which may incur a considerable up-front cost [23, 27]. In this paper, we explore a different approach in which alternative schema mappings co-exist, and are validated against user requirements in a pay-as-you-go fashion. Instead of verifying schema mappings before they are used, we assume that the data integration system is setup using as input schema mappings that are derived using mapping generation techniques. These mappings are then incrementally annotated with estimates of precision and recall [53] derived on the basis of feedback from end users. Our approach to mapping annotation is consistent with the dataspaces aim of providing the benefits of classical data integration while reducing up-front costs [27]. We do not expect users to be able to (directly) confirm the accuracy of a given mapping nor do we require them to give feedback based on the mapping specification [13]. Instead, the feedback expected from users provides information about the usefulness of the results obtained by evaluating queries posed using the generated mappings. Specifically, given a query that is issued by the user against the integration schema, a.k.a. global schema, it is reformulated in terms of the sources using the candidate mappings that express the elements of the integration schema in terms of the sources. Note that the reformulation phase may yield multiple queries, since the integration elements are likely to be associated with more than one candidate mapping. Reformulated queries are then evaluated by querying the sources. The user can then provide feedback by commenting on the tuples returned as a result of evaluating reformulated queries. Specifically, a feedback instance provided by the user specifies if a given tuple is expected or unexpected[1].

Given the feedback instances provided by the user, we then annotate the mappings. Specifically, we estimate the precision and recall of the mappings, given the results they return, based on the feedback supplied by the user. For example, consider a mapping m that is a candidate for populating a relation r in the integration schema. Based on user feedback that picks tuples that belong to r and tuples that do not, we estimate the precision and recall of the results retrieved using m. They are no more than estimates because we do not assume the user has complete knowledge of the correct extent to be returned and, therefore, do not ask the user to judge every tuple returned. In this paper, we report on an evaluation of the quality of the resulting mapping annotations for different quantities of user feedback. The feedback specified by users may be inconsistent with their expectations. For example, a user may mistakenly tag an expected tuple as a false positive. We investigate the impact inconsistent feedback may have on the quality of the computed mapping annotations.

Individual elements of the integration schema will frequently be associated with many candidate mappings. We consider a setting in which the candidate mappings are generated based on a large number of matches obtained using multiple matching mechanisms. Therefore, evaluating a user query using all candidate mappings incurs a risk of dramatically increasing the query processing time, and of obtaining a large collection of results, the majority of which are unlikely to meet user needs. We present a method that, given user feedback and user requirements in terms of precision and recall, selects the set of mappings that are likely to meet the stated requirements. Specifically, this method casts the problem of mapping selection as a constrained optimization problem, i.e., that of identifying the subset of the candidate mappings that maximize the recall (resp. precision) given a minimum threshold for the precision (resp. recall).

Mapping annotations may reveal that the quality of schema mappings is poor, i.e., that they have low precision and recall. We address this issue by refining schema mappings with a view to constructing *better*-quality mappings. The space of mappings that can be obtained through refinement is potentially very large. To address this issue, we present an evolutionary algorithm for exploring this space. Our approach to mapping refinement combines and mutates the candidate mappings to construct new mappings with better precision and recall.

As well as annotating, selecting and refining schema mappings, feedback can be used for annotating queries that a user poses against the integration schema. We show how estimates for precision and recall can be computed for such queries. Finally, we investigate the problem of propagating feedback about the results of an (integration) query down to the mappings used to populate the base

[1]As we shall see later in Section 2, a feedback instance can also be used to specify that a given attribute value, or combination of attribute values, is expected or unexpected. That said, in this article, we mainly consider tuple-based feedback instances that comment on given tuples.

relations involved in the query.

In summary, the contributions of this paper are:

1. An approach for incrementally annotating schema mappings based on user feedback (Section 3). This is to our knowledge the first study that investigates the use of feedback supplied by end users to annotate schema mappings with estimates of their precision and recall. We empirically assess the quality of the annotation computed for mapping annotation based on user feedback.
2. An analysis of the impact that inconsistent feedback may have on the quality of the annotations computed for schema mappings (Section 3.2). To do so, we empirically examine the error in mapping annotations due to inconsistent feedback.
3. A method for schema mapping selection (Section 4). Given a set of feedback instances supplied by the user, we present a method for selecting mappings whose results meet user requirements in terms of precision and recall. We also report the results of experiments that examine the effectiveness of the method used for mapping selection.
4. A method for refining schema mappings (Section 5). The refinement method is implemented by an evolutionary algorithm for mapping refinement. Using a set of mutation and cross-over operators, we show how user feedback can inform the construction of better quality mappings from an initial set of candidate ones. We also show, empirically, that better quality mappings can be constructed out of an initial set of candidate mappings using this refinement algorithm.
5. A study of the problem of integration query annotation (Section 6). We show how the feedback acquired for annotating schema mappings can be used to compute precision and recall estimates for queries posed over the integration schema, and we empirically examine the quality of these estimates. We also study the problem of propagating feedback about the results of (integration) queries down to the mappings used to populate the base relations in the integration schema.

Additionally, we present in Section 2 a model for capturing user feedback. We analyze and compare existing works to ours in Section 7 and conclude the paper in Section 8.

This paper is an extended version of a previous conference paper [7]. The material in Section 3.2, which investigates the annotation of schema mappings in the presence of inconsistent feedback is new, as is the material in Section 6, which investigates the problem of annotating integration queries and that of propagating user feedback. We also provide substantially increased coverage of related work in Section 7.

2. Candidate Mappings and User Feedback

We begin by introducing the notion of candidate mappings and by presenting the model for defining user feedback.

A data integration system is essentially composed of four elements, namely the schemas of the sources, the data sets to be integrated, an integration schema over which users pose queries, and schema mappings that specify how data structured under the schemas of the sources can be transformed and combined into data structured according to the integration schema [21]. For the purposes of this paper, we consider *global-as-view* mappings [11], which relate one element in the integration schema to a query over the source schemas. We also adopt the relational model for expressing integration and source schemas. We, therefore, define a schema mapping m by the pair $m = \langle r_i, q_s \rangle$, where r_i is a relation in the integration schema, and q_s is a relational query over the source schemas. We use *m.integration* to refer to r_i, and *m.source* to refer to q_s.

To respond to the need for rapid data integration, existing schema matching techniques can be used to produce the input for algorithms capable of automatically generating the mappings between the integration schema and the source schemas (e.g., [46, 15, 44, 40]). Multiple matching mechanisms can be used, each of which could give rise to multiple mapping candidates for populating the elements of the integration schema. To answer a user query uq, issued against the integration schema, each relation r_i involved in uq needs to be reformulated in terms of the source relations using a mapping candidate. This raises the question as to which mappings among the candidate mappings of r_i to use for answering a user query.

Candidate mappings may be labeled by scores that are derived from the confidence of the matches used as input for the generation of mappings (e.g., [18]). This suggests that the candidate mappings with the highest scores can be used for reformulating users' queries. However, since the confidences of matches, and therefore the scores of mappings, are computed based on heuristics, there is no guarantee that the mapping with the highest score reflects the true needs of end users [9, 24]. Moreover, in a data integration setting, (a sample of) the content of the integration schema is rarely available, and therefore instance-based matchers may not be an option to match the source schemas to the integration schema. Thus, the likelihood that the scores associated with the mappings are inaccurate can be higher than in situations in which the contents of the schemas to be matched are available, e.g., in data exchange [21].

In this paper, we use a different source of information for assessing candidate mappings, namely user feedback. In doing so, the user is not provided with a set of (possibly complex) mapping expressions; rather, s/he is given a set of answers to a query issued against the integration schema that was answered using one or more candidate mappings. To further illustrate the kinds of feedback that can be supplied, assume that the user issued a query to retrieve the tuples of the relation r_i in the integration schema, that was evaluated using one or more mappings that are candidates for populating r_i, and that the query results were displayed to the user. The user then examines and comments on the results displayed using the following kinds of feedback:

1. That a given tuple was expected in the answer.
2. That a certain tuple was not expected in the answer[2].
3. That an expected tuple was not retrieved.

The kinds of feedback we have just described are tuple-based in the sense that they comment on the correctness of the membership relation between tuples and the result set returned by a set of mappings. A finer grained form of feedback can also be supported. In particular, the user can indicate that a given attribute of r_i cannot have a given value. As in information retrieval [47], we assume that users provide feedback on a voluntary basis: they are not required to comment on every single result they are given, rather, they supply feedback on the results of their choice.

To cater for the types of feedback introduced above, we define a feedback instance uf provided by the user for a tuple:

$$uf = \langle AttV, r, exists, provenance \rangle$$

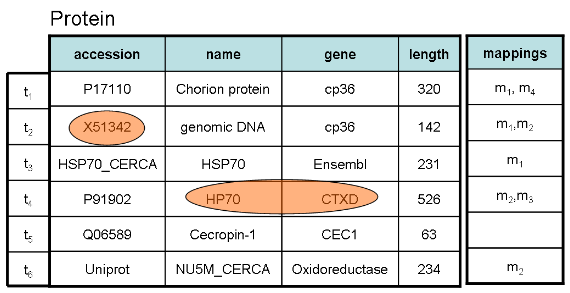
where r is a relation in the integration schema, $AttV$ is a set of attribute-value pairs $\langle att_i, v_i \rangle$, $1 \leq i \leq n$, such that $att_1, \ldots, att_n$ are attributes of r, and $v_1, \ldots, v_n$ are their respective values. $exists$ is a boolean specifying whether the attribute value pairs in $AttV$ conform to the user's expectations. To specify whether $exists$ is true or not, we assume the existence of a function $extent(r)$, that returns the set of tuples that belong to r in the user's conceptualization of the world. Note that $extent(r)$ is not available, rather we obtain incomplete information about the tuples that belong to $extent(r)$ through user feedback. $exists$ is true iff there is a tuple in $extent(r)$ in which the attributes $att_1, \ldots, att_n$ take the values $v_1, \ldots, v_n$, respectively. That is:

$$|t \in extent(r) \; s.t. \; \forall \; i \in \{1, \ldots, n\}, \; t[att_i] = v_i| \geq 1$$

where $|s|$ denotes the magnitude of the set s, and $t[att]$ denotes the value of the attribute att in the tuple t. $provenance$ specifies the origin of the attribute value pairs on which feedback was given. These could have been provided by the user, or obtained from the sources using one or multiple mappings. Therefore:

$$provenance \in \{\text{'}userSpecified\text{'}, Map\}$$

$provenance$ = 'userSpecified' means that the attribute value pairs $AttV$ are provided by the user. $AttV$ may also be retrieved from the sources in which case $provenance = Map$, where Map is the set of mappings that can be used to retrieve $AttV$ from the sources.



Protein

	accession	name	gene	length	mappings
t_1	P17110	Chorion protein	cp36	320	m_1, m_4
t_2	X51342	genomic DNA	cp36	142	m_1,m_2
t_3	HSP70_CERCA	HSP70	Ensembl	231	m_1
t_4	P91902	HP70	CTXD	526	m_2,m_3
t_5	Q06589	Cecropin-1	CEC1	63	
t_6	Uniprot	NU5M_CERCA	Oxidoreductase	234	m_2

Figure 1: Example of query results.

As an example, consider a life scientist who is interested in studying the proteome of the Fruit Fly. Given that data describing this proteome is stored across multiple bioinformatics sources (e.g., Uniprot[3], IPI[4] and Ensembl[5]), the scientist needs to access and combine data that belong to these sources. In doing so, the scientist prefers to use a given (integration) schema: this schema can be manually designed or automatically derived from a set of queries that are of interest to the scientist. Rather than attempting to manually specify the schema mappings between the source schemas and the integration schema, the scientist opts for a low upfront-cost option whereby candidate schema mappings are automatically derived. Once the mappings have been derived, the scientist issues a query to find the available proteins of the Fruit Fly. This query is evaluated using the mapping candidates for populating the *Protein* relation. Assume that the evaluation results are displayed as shown in Figure 1. The *mappings* column specifies the candidate mappings that were used for retrieving a given tuple. Note that, in general, this column would not be visible to the user; it is displayed in Figure 1 for ease of exposition only.

The user examines the results displayed and supplies feedback specifying whether they meet the requirements. For example, the feedback instance uf_1 given below specifies that the tuple t_1, which was retrieved using the mappings m_1 and m_4 is a *true positive*, i.e., meets the user's expectations.

$$\begin{aligned} uf_1 &= \langle AttV_1, Protein, true, \{m_1, m_4\} \rangle \\ AttV_1 &= \{\langle accession, \text{'}P17110\text{'} \rangle, \langle name, \text{'}Chorion\,protein\text{'} \rangle, \\ &\quad \langle gene, \text{'}cp36\text{'} \rangle, \langle length, \text{'}320\text{'} \rangle\} \end{aligned}$$

The user can also provide feedback specifying *false positives*, i.e., results that do not meet the requirements. For example, the feedback instance uf_2 below specifies that the accession of a protein cannot have the value $X51342$;

[2]This form of feedback is called negative relevance feedback in the information retrieval literature [47].

[3]http://www.uniprot.org
[4]http://www.ebi.ac.uk/IPI
[5]http://www.ensembl.org

indeed, this is a DNA accession, not a protein one. Similarly, the feedback instance uf_3 below specifies that the protein named *HP*70 does not belong to the *CTXD* gene.

$$uf_2 = \langle\{\langle accession, \text{'X51342'}\}, Protein, false, \{m_1, m_2\}\rangle$$
$$uf_3 = \langle\{\langle name, \text{'HP70'}\rangle, \langle gene, \text{'CTXD'}\rangle\}, Protein, false, \{m_2, m_3\}\rangle$$

In addition to true positives and false positives, the user can also specify false negatives, i.e., results that are expected by the user and are not returned. The tuple t_5 in Figure 1 is, for example, a false negative as specified by the following feedback instance.

$$uf_4 = \langle AttV_4, Protein, true, \text{'userSpecified'}\rangle$$
$$AttV_4 = \{\langle accession, \text{'Q06589'}\rangle, \langle name, \text{'Cecropin} - 1\text{'}\rangle, \langle gene, \text{'CEC1'}\rangle, \langle length, \text{'63'}\rangle\}$$

The following process can be used to specify feedback of the form illustrated above. The user begins by specifying if a given tuple is expected. If the tuple is annotated as unexpected, e.g., tuples t_2 and t_2 in the above example, then the user is invited to specify why the tuple is unexpected. To do so, the user specifies illegal attribute values, if any, i.e., values that do not belong to the domain of the attributes in question according to the user expectations. For example, the user can tag the value *X*51342 of the *accession* attribute of tuple t_2 as illegal. The user can also specify illegal combinations of legal attribute values. For example, the user can tag the combination of attributes values *HP*70 and *CTXD* in the tuple t_4, as illegal.

In what follows, given a mapping candidate *m* for populating a relation *r* in the integration schema, and given a set of feedback instances *UF* supplied by the user, we use $tp(m, UF)$, $fp(m, UF)$, $fn(m, UF)$, respectively, to denote the true positives, false positives and false negatives tuples of *m* given the feedback instances in *UF*.

3. Annotating Schema Mappings

We now show how candidate mappings can be labeled with annotations specifying their fitness to user expectations using feedback instances of the form described in the previous section.

We can annotate a schema mapping as either correct or incorrect using a simple annotation scheme: a mapping is correct iff it meets the needs of users, i.e., its source query returns all the expected answers, and does not return any unexpected ones. However, since the set of candidate mappings derived using matching algorithms is likely to be incomplete in the sense that it may not contain a mapping that meets user needs, there is a risk of annotating as incorrect all the candidate mappings for populating a given relation. We therefore opt for a less stringent annotation scheme that tags schema mappings with metrics specifying the degree to which they meet user requirements.

3.1. Cardinal Annotations

The quality of candidate mappings for populating a relation *r* can be quantified using precision and recall [53]. Of course, we cannot compute these metrics since they presuppose access to the extent of *r*, i.e., the set of tuples that belong to *r* in the users' conceptualization of the world. Notice, however, that the feedback instances supplied by users provide *partial* information about the extent of *r*. Specifically, they allow the identification of (some of the) true positives, false positives and false negatives of a given candidate mapping. Using this information, we can compute precision and recall *relative* to (the extent of *r* identified through) the feedback supplied by the user.

We adapt the notions of precision and recall [53] to measure the quality of a mapping. We define the precision of a mapping *m* relative to the feedback instances in *UF* as the ratio of the number of true positives of *m* given *UF* to the sum of true positives and false positives of *m* given the feedback instances in *UF*. That is:

$$Precision(m, UF) = \frac{|tp(m,UF)|}{|tp(m,UF)| + |fp(m,UF)|}$$

Similarly, the recall of a mapping *m* relative to the feedback instances in *UF* is defined as the ratio of the number of true positives of *m* given *UF* to the sum of true positives and false negatives of *m* given the feedback instances in *UF*. That is:

$$Recall(m, UF) = \frac{|tp(m,UF)|}{|tp(m,UF)| + |fn(m,UF)|}$$

We also compute an F-measure [53] relative to user feedback that combines both precision and recall, and use it for ranking candidate mappings. The relative F-measure of a mapping *m* w.r.t. the feedback instances in *UF* can be defined as:

$$F(m, UF) = \frac{(1+\beta^2) \times Precision(m,UF) \times Recall(m,UF)}{\beta^2 \times Precision(m,UF) + Recall(m,UF)}$$

where β is a parameter that controls the balance between precision and recall. For example, if β is *1* then precision and recall have the same weight.

Notice that the precision and recall are computed based on tuple-based feedback, that is feedback specifying true positive, false positive and false negative tuples. Our feedback model, however, provides users with a means for specifying feedback at a finer granularity, if they wish. In particular, they can specify that a given value of a relation attribute, or a combination of values of a subset of attributes, is expected or unexpected. Consider, for example, the *Protein* relation, the user provided feedback specifying that the value C51342 of the attribute *accession* is unexpected. Attribute-based feedback can be used to infer tuple-based feedback. For example, from the above feedback instance, we can infer that all the tuples of the protein relation in which the attribute *accession* takes the value *C*51342, are unexpected. Note, however, that when the attribute-based feedback instance specifies that a given attribute value is expected, we cannot infer any

Table 1: Information regarding the mapping annotation experiment.

Integration relation	FavoriteCity	MyReading
Datasource	Mondial	Amalgam
# of binary matches	15	37
# of mappings generated by the IBM data architect	12	15
# of mappings obtained by combining initial mappings	50	100
# of source relations involved in the mappings	12	17
Total # of tuples retrieved by the candidate mappings	2197	2138
# of "ground truth" tuples	100	200
% of expected tuples	4.55%	9.36%
% of unexpected tuples	95.54%	90.64%

feedback at the tuple level. For example, if the user provided a feedback specifying that the value $P17110$ of the *accession* attribute is expected, then we cannot infer any feedback at the tuple level. This is because expected and unexpected tuples may have that attribute value.

Given that relative precision and recall are computed based on *partial* knowledge about the user's conceptualization of the integration relation, the following question arises:

How much user feedback is required to approximate the real precision and recall, i.e., the precision and recall computed based on the true extent of the integration relation?

To investigate the above question, empirically, we used two datasets: the Mondial geographical database[6] and the Amalgam data integration benchmark[7]. We chose the Mondial and Amalgam datasets for the following reasons. The schema and contents of the two datasets are straightforward to understand: Mondial contains geographical information, and Amalgam contains bibliographical information. This helps when examining the quality of schema mappings. The two datasets are not synthetic, rather they are real. In particular, Amalgam is composed of 4 databases that were developed independently to cater for bibliographical information [45]. Finally, the two datasets, Mondial and Amalgam, are widely used in schema mappings literature for testing and validation purposes, e.g., [3],[9].

We created an integration relation $FavoriteCity(name, country, province)$ and specified binary matches between the attributes of this relation and the attributes of the source relations in the Mondial database. For example, the match $< FavoriteCity.name, Located.city >$ specifies that the attribute *name* of the *FavoriteCity* relation and the attribute *city* of the *Located* relation are semantically the same. Using different subsets of binary matches of this form, we then generated multiple candidate schema mappings to populate the *FavoriteCity* relation using IBM Infosphere Data Architect[8]. We also created an integration relation $MyReading(title, journal, volume, number)$, specified binary matches between the attributes of this relation and the attributes of the source relations of Amalgam, and generate candidate mappings to populate the *MyReading* relation using IBM Infosphere Data Architect. For the purposes of our experiment, we needed to increase the number of candidate mappings for *FavoriteCity* and *MyReading*. To do this, we randomly combined the mappings obtained using the IBM Infosphere Data Architect by using the union and intersection relational operators. As an example, we illustrate below the specification of three mappings for populating the *FavoriteCity* relation: the mappings m_1 and m_2 were generated by the IBM data architect, and the mapping m_{12} was generated by unioning the source queries of the mappings m_1 and m_2.

$$\begin{aligned} m_1 &= < FavoriteCity, \Pi_{name,countryprovince} Mondial.City > \\ m_2 &= < FavoriteCity, \Pi_{city,country,province} Mondial.Located > \\ m_{12} &= < FavoriteCity, \Pi_{name,countryprovince} Mondial.City \\ &\quad \cup \Pi_{city,country,province} Mondial.Located > \end{aligned}$$

The reader may notice that creating mappings by combining base mappings gives rise to correlated mappings. However, candidate mappings are often correlated in practice [8]. This is particularly the case when the mappings are generated based on binary matches, that do not necessarily specify how the relations in the sources should be combined to populate a relation in the integration schema, and therefore give rise to correlated candidate mappings. The reader may also wonder if the base mappings are privileged over combined mappings. This is not the case. As we will show later on in Section 5, it is possible to create better quality mappings by mutating and combining base mappings.
We then specified the ground truth extents for *FavoriteCity* and *MyReading* by randomly selecting a subset of the tuples returned by their respective candidate mappings. These extents serve two purposes. They allow the automatic generation of synthetic user feedback as well as computation of the "ground truth" against which the annotations computed based on user feedback are compared. Table 1 specifies the number of binary matches we specified, the number of mappings generated using the IBM Infosphere Data Architect, the number of source relations that participate in the mappings, the number of tuples retrieved by the candidate mappings, the number of "ground truth" tuples, and the percentage of expected and unexpected tuples. Notice that the percentage of expected tuples is low compared with the percentage of unexpected tuples. This is because, in practice, the majority of the tuples obtained based on mappings that are obtained automatically are likely to be unexpected.

[6]http://www.dbis.informatik.uni-goettingen.de/Mondial
[7]http://dblab.cs.toronto.edu/ miller/amalgam
[8]http://www-01.ibm.com/software/data/studio/data-architect

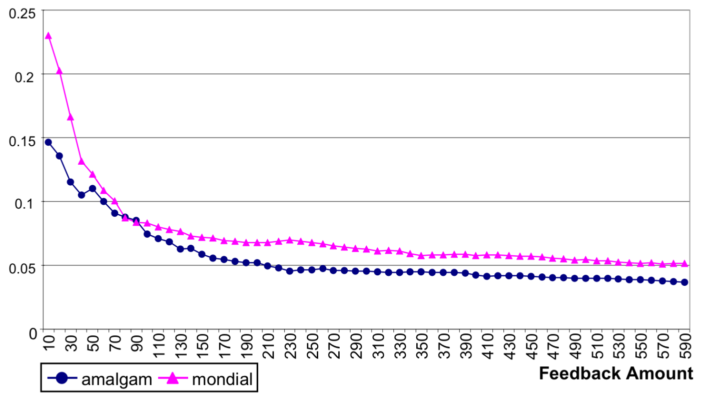


Figure 2: Average error in precision.

To annotate the mappings, we applied the following two-step procedure iteratively.

1. Generate feedback instances.
2. Compute the relative precision and recall of the candidate mappings given cumulative feedback.

At every iteration, we generated a set of feedback instances. For the purposed of this experiment we only consider tuple-based feedback. Specifically, we randomly generated a stratified sample of *10* annotated tuples out of the set of tuples returned by the candidate mappings. Stratified sampling ensures that the members of a population are first grouped into relatively homogeneous subpopulations before sampling. Stratified sampling is performed as follows. Given the ground truth which partitions the tuples retrieved by the candidate mappings into expected and unexpected, a sample of 10 tuples is randomly selected from that set of tuples retrieved by the candidate mappings, and for which feedback has not been provided yet, such that the ratio of expected (resp. unexpected tuples) in the sample is the same as the ratio of expected (resp. unexpected) tuples within the ground truth. This sampling method improves the representativeness of the feedback instances generated. It ensures that the number of expected and unexpected tuples in the sample is proportional to the number of expected and unexpected tuples in the result set obtained using all candidate mappings.

To measure the quality of the relative precision and recall, we repeated the annotation experiment described above multiple times, specifically *25* times[9], and computed the average error in precision and recall at every feedback iteration. The error in precision (resp. recall) is the difference between the relative precision (resp. recall) of a candidate mapping computed using supplied feedback and the "ground truth" precision (resp. recall).

Figure 2 illustrates the average error in precision, and Figure 3 illustrates the average error in recall. Note that the scale of the vertical axis is different in the two figures. The two figures show that when the user provided 10 feedback instances the error in both precision and recall are relatively high. For example, the error in precision is 0.23 and the error in recall is 0.22 for the mappings that are candidate to populate the *FavoriteCity* relation. The error drops significantly in the first few feedback iterations, specifically from 10 to 80 feedback instances. For example, the error in precision for the mapping candidates of the *FavoriteCity* relation drops from 0.23 to 0.08, and that of recall drops from 0.22 to 0.11. The error in precision for the mapping candidates of the *MyReading* relation drops from 0.14 to 0.07, and that of recall drops from 0.28 to 0.11. Note that 80 feedback instances is a relatively small number if compared with the number of tuples retrieved by the candidate mappings. It represents 3.64% of the total number of tuples retrieved by the candidate mappings of *FavoriteCity*, and 3.74% of the total number of tuples retrieved by the candidate mappings of *MyReading*. 80 feedback instances is also small if we consider the number of candidate mappings subject to annotation: *50* in the case of *FavoriteCity* and *100* in the case of *MyReading*.

Beyond 80 feedback instances, the error in precision and the error in recall decreases steadily as more feedback instances are provided, but this improvement incurs diminishing returns. For example, after collecting 600 feedback instances, the error in precision for the mapping candidates of *FavoriteCity* decreases from 0.08 to 0.05, and the error in precision for the mapping candidates of *MyReading* drops from 0.07 to 0.04. The same observation applies to the error in recall. 600 feedback instances is an important number as it represents 27.3% of the total number of tuples retrieved by the candidate mappings of *FavoriteCity*, and 28.1% of the total number of tuples retrieved by the candidate mappings of *MyReading*. This slow improvement can be explained by the fact that the smaller the error, the larger the number of feedback instances required to reduce it.

The above experiment shows that the quality of mapping annotations is incrementally improved as the user provides more feedback instances thereby reflecting the *pay-as-you-go* philosophy behind dataspaces.

As well as stratified sampling, we also ran the above experiment using a random sampling strategy, in which the tuples on which feedback was given, were randomly selected. The error in the precision and recall estimates computed for schema mapping in this case were similar to those obtained using the stratified sampling strategy. Using the above sampling strategies, the user provides feedback specifying both expected and unexpected tuples. In practice, however, it is possible that the user focuses on one particular class of feedback, i.e., expected tuples or unexpected. This raises the question as to what impact on annotation estimates this may have. Consider, for example, that the user provides feedback specifying only expected tuples. Using this kind of feedback has a negative impact on the precision estimates. Indeed, given the

[9]We chose to run the experiment for *25* times because it has been shown statistically to yield good estimations [54].

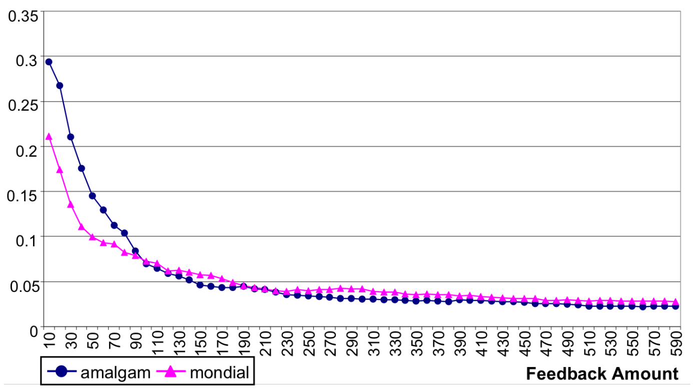


Figure 3: Average error in recall.

definition of precision, the precision of a given mapping takes the value 1, if the user identified expected tuples that are retrieved by that mapping, or undefined, otherwise. Consider now the case in which the user provides feedback specifying only unexpected results. Given the definition of recall, recall estimates cannot be computed for schema mappings. Moreover, the precision takes the value 0, if the mapping retrieves at least one false positive that is identified by the user, or is undefined, otherwise. The above analysis suggests that it is important for the user to provide feedback specifying both expected and unexpected results for the annotation estimates computed for schema mappings to be informative.

3.2. Inconsistent Feedback and its Impact on Mapping Annotations

We have, so far, assumed that the feedback instances supplied by users are consistent with their expectations. In practice, this assumption may not hold. Users may mistakenly provide feedback that contradicts their needs, e.g., a scientist who is looking for Fruit Fly proteins may tag a tuple representing a protein that belongs to that species as a false positive. Also, users may in error provide incorrect feedback, e.g., when they have partial knowledge of the domain of expected results.

It is worth mentioning that inconsistencies between feedback and user expectations may also be caused by changes in user expectations. For example, the user may decide to study the chicken proteome instead of the Fruit Fly proteome. This shift in expectations may cause inconsistencies between the feedback instances collected before the change and those collected subsequently. In this paper we do not consider inconsistencies stemming from changes in requirements, and focus on those caused by user mistakes.

Inconsistent feedback may have an impact on mapping annotations. In particular, we can anticipate that the error in precision and recall, in the presence of inconsistent feedback instances, will be larger than would otherwise be the case. In this section, we aim to assess the impact that inconsistent feedback instances have on mapping annotations. In particular, we aim to identify the amount/percentage of inconsistent feedback instances that can be tolerated before the annotations computed for mappings become uninformative, i.e., exhibit an unacceptable error level.

We investigated the above problem empirically by running an experiment for annotating the candidate mappings for populating *FavoriteCity*. In doing so, we introduced incorrect feedback and measured the error in precision and recall computed based on feedback with the ground truth annotation. Specifically, we ran the following procedure iteratively.

1. Generate a sample of feedback instances. We randomly generated *10* feedback instances by applying a stratified sampling to the set of tuples returned by the candidate mappings.
2. Introduce inconsistent feedback into the generated sample.
3. Compute the relative precision and recall of the candidate mappings given cumulative feedback.

Inconsistent feedback instances are introduced by altering a specific percentage p of the feedback instances generated in (1). To do so, we randomly select a subset S of the feedback generated. The size of S is specified using the percentage p. We then alter the feedback instances in S. If a feedback instance specifies that a tuple t is an expected result (resp. unexpected result), we substitute it with a feedback instance specifying that t is an unexpected result (resp. expected result).

To measure the effect that inconsistent feedback has on mapping annotations, we used as a metric the difference between the error observed in precision (resp. recall) when inconsistencies are present and the error in precision (resp. recall) recorded when all feedback instances are consistent with the expectations. We will refer to that difference in the rest of this section using the term "error due to inconsistent feedback".

We repeated the annotation experiment described above *25* times, and computed for each percentage value of inconsistent feedback, the average error in precision due to inconsistent feedback, shown in Figure 4, and the average error in recall due to inconsistent feedback, shown in Figure 5. Note that the scale of the vertical axis is different in the two figures.

As expected, Figures 4 and 5 show that the error in annotation due to inconsistent feedback increases as the percentage of inconsistent feedback instances increases. However, they also show that the annotation computed based on user feedback is reliable even in the presence of a significant amount of inconsistent feedback. Indeed, the charts show that 10% of inconsistent feedback introduces a small error, less than 0.02, in the annotations computed. This is further evidence that the pay-as-you-go approach is viable as a strategy for gradual improvement. Notice that the average error in precision is quite high when

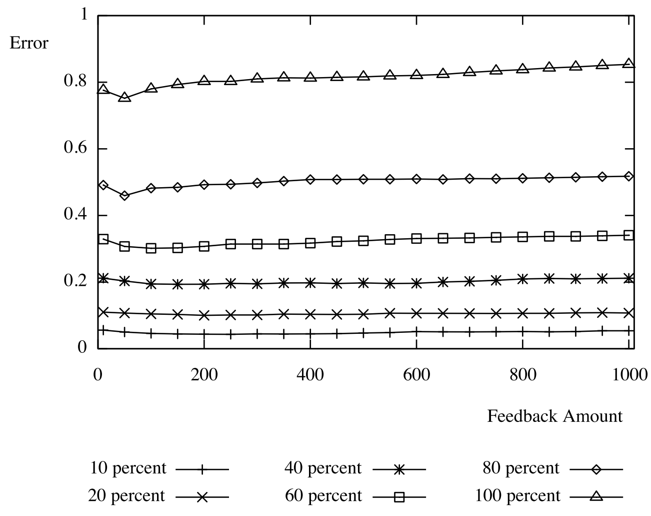


Figure 4: Error in precision due to inconsistencies in feedback.

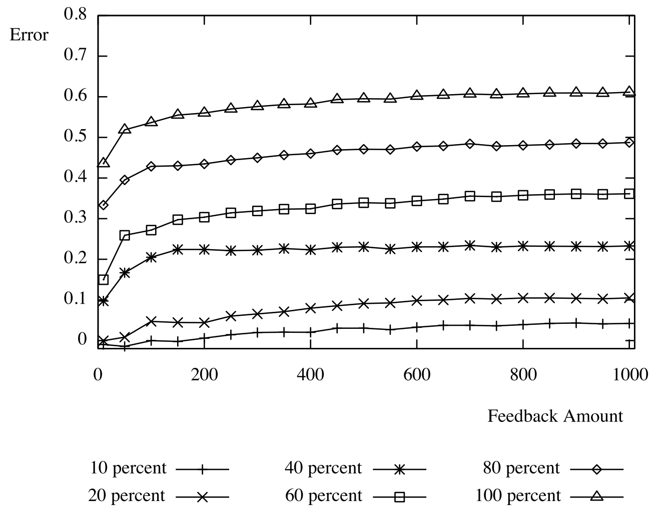


Figure 5: Error in recall due to inconsistencies in feedback.

all feedback instances, i.e., 100%, are inconsistent with user expectations. This is because the average "ground truth" precision of the mappings is low, it is equal to 0.14. Regarding the recall, the error when all feedback instances are inconsistent is not as high as for precision. This is because the average "ground truth" recall is higher, 0.34.

3.3. Ordinal Annotations

Another source of information that can be exploited for mapping annotation is the dependencies between the candidate mappings in terms of the tuples they retrieve from the sources. We capture these dependencies in the form of ordinal annotations that partially order the candidate mappings in terms of true positives and false positives. Consider, for example, two candidate mappings m_1 and m_2 for populating a relation r in the integration schema. We say that m_1 covers m_2 in terms of true positives iff m_1 retrieves all the true positives that are obtained using m_2 given the feedback instances in UF, i.e., $tp(m_2, UF) \subseteq tp(m_1, UF)$. We write $m_2 \leq_{tp}^{UF} m_1$. Similarly, m_1 covers m_2 in terms of false positives iff m_1 returns all the false positives that are obtained using m_2 given the feedback instances in UF, i.e., $fp(m_2, UF) \subseteq fp(m_1, UF)$. We write $m_2 \leq_{fp}^{UF} m_1$.

Ordinal annotations are used as input to the mapping refinement process in Section 5.3.

4. Selecting Schema Mappings

An element of an integration schema is likely to be associated with many candidate mappings. We consider a setting in which the candidate mappings are generated based on a large number of matches obtained using multiple matching mechanisms. In this context, by evaluating a user query using all candidate mappings, there is a risk of significantly increasing the processing time of the user query, and obtaining a large collection of results, the majority of which are unlikely to meet user needs. We show in this section how the mapping annotations presented in the previous section can be employed for selecting the candidate mappings to be used for populating a given element in the integration schema.

4.1. Mapping Selection as an Optimization Problem

Not all users of dataspaces will have the same requirements in terms of precision and recall. Consider, for example, a dataspace providing access to proteomic data sources. A drug designer who issues queries to this dataspace may require high precision; the existence of false positives in query results may lead to further expensive investigation of inappropriate candidate drugs. On the other hand, an immunologist who is using the proteomic dataspace to identify the proteins responsible for an infection may tolerate low precision, since further investigation may potentially give rise to the identification of new proteins associated with the infection under investigation.

In order to tailor mapping selection to user requirements, we use a technique that aims to maximize the recall of the results while guaranteeing that their precision is higher than a given threshold λ_p, which can be specified by the user. The selection method can be formulated as a search problem that maximizes the following evaluation function:

$$eval_{recall}(V, UF, \lambda_p) = \begin{cases} 0, & \text{if } prec(V, UF) < \lambda_p \\ recall(V, UF), & \text{otherwise} \end{cases} \quad (1)$$

where $V = \langle b_1, \ldots, b_n \rangle$ is a vector of booleans that specifies the selected mappings: b_i is true iff the candidate mapping m_i is selected. *prec(V,UF)* and *recall(V,UF)* denote the precision and recall of the union of the results obtained using the mappings specified by V given the feedback instances in *UF*. An optimization problem is then to identify values for V (i.e., subsets of mappings) that maximize $eval_{recall}$ while satisfying the constraint $prec(V, UF) < \lambda_p$.

The user may also be interested in maximizing the precision of the results obtained, provided that the recall is higher than a given threshold, λ_r. As for the above case, this problem can be formulated as a search that aims to maximize the evaluation function obtained by swapping the roles of precision and recall in $eval_{recall}(V, UF, \lambda_p)$.

The above evaluation function poses a problem during the search for suitable values for V, in that it associates all vectors of mappings with a precision below λ_p with the same value, viz. zero. Therefore, a vector of mappings with a precision that closely misses λ_p is ranked equally as badly as a vector with zero precision. To overcome this problem, we use a function by Menascé and Dubey [41] for estimating utility in service-oriented architectures in order to return, for those vectors of mappings with a precision lower than the threshold λ_p, a decreasing value as the precision decreases, as follows:

$$eval_{recall}(V, UF, \lambda_p) = recall(V, UF) \times K_p \times C(prec(V, UF), \lambda_p) \quad (2)$$

where $C(x, \lambda_p)$ is a monotonic function that increases as x increases defined as:

$$C(x, \lambda_p) = \frac{1}{1 + e^{-\frac{x}{\lambda_p} \times 10 + 5}} - \frac{1}{1 + e^5},$$

and $K_p = \frac{1+e^5}{e^5}$ a scaling factor so that $lim_{prec(V,UF) \to \lambda_p}(eval_{recall}(V, UF, \lambda_p)) = recall(V, UF)$. For our purposes, the specific formulas used to compute K_p and C are not especially significant, but the resulting curve shape is. Figure 6 shows the graph of the function when the precision threshold λ_p is equal to 0.5 and the recall is equal to 1. As depicted in this figure, the value returned by the function increases as the precision does. It reaches the maximum value of 1 when the precision reaches the threshold λ_p. Crucially, for the optimization algorithm, in contrast with Equation (1), Equation (2) discriminates between different values where $prec(V, UF) < \lambda_p$, preferring values that are closer to λ_p over those that are further away.

For the purpose of solving this constrained optimization problem, we use the Mesh Adaptive Direct Search (MADS) approach [1], which is centered on a nonlinear search algorithm that is appropriate for solving black-box constrained optimization problems.

4.2. Experimental Evaluation

The optimization method for selecting schema mappings presented in Section 4.1, based on the evaluation function in equation (2), uses as input feedback instances to estimate the precision and recall of the results retrieved by a set of mappings. This raises the following questions:

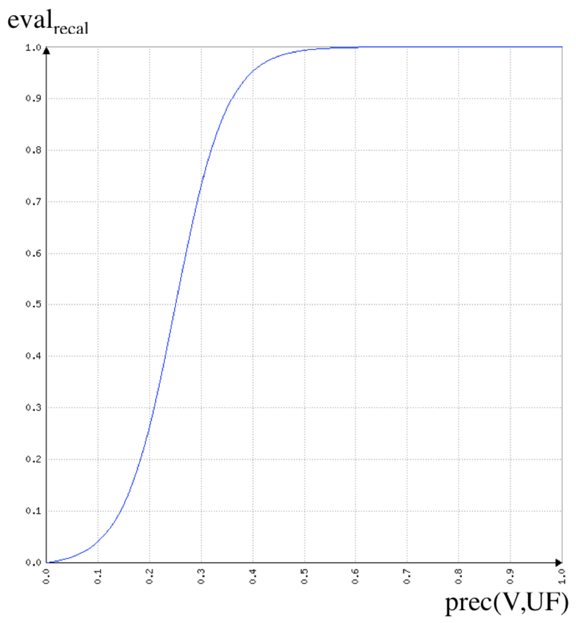


Figure 6: The graph of the evaluation function when the precision threshold is equal to 0.5 and the recall is 1.

Table 2: Precision and recall obtained using the mappings selected when the "ground truth" expectations of the user are known.

λ_p	0	0.2	0.5	0.7	1
Precision	0.09	0.28	0.57	0.7	1
Recall	1	0.82	0.72	0.28	0.04

- *What is the amount of feedback that is needed for the precision threshold set by the user to be respected by the results retrieved by the mappings selected based on that optimization method?*
- *What is the amount of feedback needed to select mappings that yield the same recall as the mappings selected with complete knowledge of user expectations.*

To answer the above questions, we used the mappings candidates to populate the *FavoriteCity* relation. Table 2 shows the precision and recall of the results obtained using the mappings selected based on the optimization method presented in Section 4.1 where the "ground truth" expectations of the user are known and the precision threshold varies between 0 and 1.

We then used the optimization method to select the mappings to be used to populate the *FavoriteCity* relation based on feedback supplied by the user. In doing so, we varied the following parameters: *i)* the number of feedback instances supplied by the user, and *ii)* the threshold set by the user. To do so, we applied the following procedure to the candidate mappings of *FavoriteCity* iteratively.

1. Generate a sample of feedback instances.

2. Select mappings that maximize the evaluation function $eval_{recall}$ in equation (2) with a precision threshold λ_p.
3. Compute the ground truth precision and recall of the results obtained using the mappings selected.

At every iteration, we generate a sample of *10* feedback instances using stratified sampling as described in Section 3.1. We then select the mappings that maximize the recall with a given precision threshold λ_p, i.e., the mappings that maximize the evaluation function $eval_{recall}$ in equation (2). To do this, we used NOMADm[10], an implementation of the Mesh Adaptive Direct Search algorithm [1].

We repeated the above experiment by changing the value of the precision threshold λ_p from *0* to *1*. The ground truth precision of the results obtained using the mappings selected at every feedback iteration appears in Figure 7, and their ground truth recall appears in Figure 8. Note that, for the sake of clarity in the figures, Figures 7 and 8 have scales of Precision Threshold inverted.

The analysis of the results shows that using a number of feedback instances that is greater or equal to 170 (which represents *7.72%* of the result set retrieved by the candidate mappings) the precision threshold is respected and the recall is the same as the recall of the results of the mappings selected with complete knowledge of the ground truth expectations.

During the earlier feedback iterations, the precision threshold is not always respected. For example, the precision threshold $\lambda_p = 1$ could not initially be satisfied. This is because none of the annotated mappings has a precision of *1* with the feedback collected at that point. Later on, when the cumulative number of feedback instances reached *150* (i.e., 6.81% of the result set retrieved by the candidate mappings) the only mapping with a precision equal to *1* was identified, thereby allowing the solver to satisfy the constraint specified by λ_p. For $\lambda_p = 0.7$, when the number of feedback instances supplied is lower than *170*, the precision of the results obtained using the mappings selected was below the threshold. When the number of feedback instances supplied reached *170*, the precision of the results obtained using the mappings selected was greater than the precision threshold of 0.7.

Regarding the recall, we notice that in the first feedback iteration the recall of the results obtained using the mappings selected for $\lambda_p \leq 0.5$ is lower that the recall obtained using the mappings selected based on complete knowledge of the ground truth expectations. As more feedback instances were provided, the recall reached the recall obtained using the mapping selected based on complete knowledge of the ground truth expectation. On the other hand, in the first feedback iterations, the recall obtained using the mappings selected for $\lambda_p \geq 0.7$ is greater than the recall obtained using the mappings selected based on complete knowledge of the ground truth expectations. At a first glance this seems to be odd. However, it can be explained by the fact that for $\lambda_p \geq 0.7$, the results obtained using the mappings selected in the first feedback iterations did not respect the precision threshold (see Figure 7).

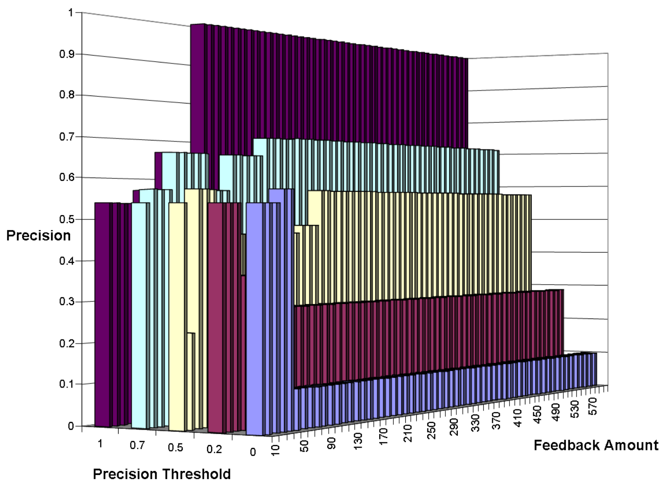


Figure 7: Precision of the results obtained using the mappings selected using the evaluation function $eval_{recall}$.

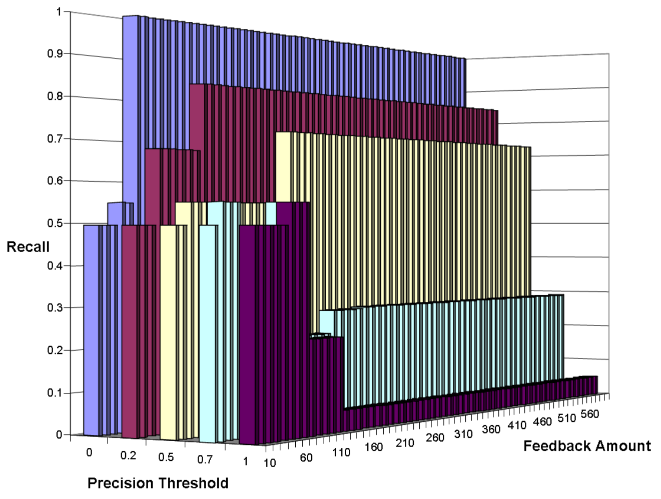


Figure 8: Recall of the results obtained using the mappings selected using the evaluation function $eval_{recall}$.

It is worth mentioning that, except for the case where λ_p equals *1*, the number of mappings required to reach the correct recall and precision ranges from *3* to *5* (see Table 3).

5. Refining Schema Mappings

Mapping annotations may reveal that the quality of the candidate mappings is poor. Specifically, they may indicate that the number of true positives obtained using the *best* mapping, i.e., the mapping with the highest F-measure, is small compared with the number of false positives. One response in this case is to try to improve

[10]http://www.gerad.ca/NOMAD/Abramson/nomadm.html

Table 3: Number of mappings selected when the recall reaches the maximum.

λ_p	0	0.2	0.5	0.7	1
Number of mappings selected	5	4	3	3	1

the quality of candidate mappings through refinement. In mapping refinement, one or more mappings are constructed out of existing ones taking into account available user feedback. We distinguish two kinds of mapping refinement: one that seeks to reduce the number of false positives, and one that aims to increase the number of true positives.

5.1. *Refining Mappings to Reduce the Number of False Positives*

A candidate mapping is refined by modifying the corresponding source query so as to reduce the number of false positives it returns. This requires results filtering. There are four operators in the relational algebra that allow filtering of the results, viz. join, selection, intersection and difference. We now explore how they can be used to reduce the number of false positives.

Join. Assume that in the user conceptualization of the world, the *Protein* relation in the integration schema is to be populated with tuples providing information about the proteins that belong to the Fruit Fly species, and consider the mapping ⟨*Protein*, *ProteinEntry*⟩, which maps *Protein* to *ProteinEntry* in the source schema. The source schema is illustrated in Figure 9. Using this mapping, the user will obtain both true positives and false positives, viz., proteins that belong to the Fruit Fly and proteins that belong to other species. The number of false positives returned may be reduced by joining the source query of the mapping with relations in the sources. To identify which source relations can be used for this purpose, we exploit information provided by the source schemas. For example, Figure 9 shows that *ProteinEntry* is connected to *FlyProteome* by a referential integrity constraint. This constraint can be used to reduce the number of false positives that would be obtained using the above mapping. It can be used to filter out proteins that belong to other species, e.g., mouse and chicken. The source query of the mapping obtained by this refinement is:

$$ProteinEntry \ltimes_{acc\,=\,id} FlyProteome$$

Here, the symbol $\ltimes$ denotes the semi-join. This style of refinement is based on input information that is readily available in the source schemas, viz., foreign key constraints. As well as foreign key constraints, it would be possible to refine mappings using associations that are inferred from the results obtained by matching the sources [51]. Using this kind of association, a mapping can be refined by joining elements across sources.

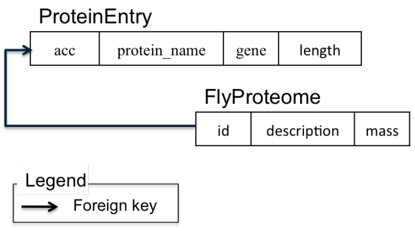


Figure 9: Source schema used for mapping refinement.

To specify which relations are to be joined with the source query of a given mapping, we use the notion of a *path*. A path p can be defined as a sequence:

$$p = (\langle R_1, att_1^{out}\rangle, \ldots, \langle R_i, att_i^{in}\rangle, \langle R_i, att_i^{out}\rangle, \ldots, \langle R_n, att_n^{in}\rangle)$$

where $n \geq 2$, and att_i^{in} and att_i^{out} are attributes of the relation R_i. Let m be a mapping and $\langle R_1, att_1^{out}\rangle$ an attribute that is involved in the source query of m (i.e., an attribute of a relation in the from clause of the query). To refine the source query of m using the path p, we use the function *constructJoinMapping*(m, p), which returns a mapping where query is obtained by joining the source query of m with the relations in p (except R_1) and projecting the values of the attributes of *m.source*:

$$constructJoinMapping(m, p) = \langle m.integration, m.source \ltimes_{jp_1} \ldots \ltimes_{jp_{n-1}} R_n\rangle$$

where jp_i = "$(R_i.att_i^{out} = R_{i+1}.att_{i+1}^{in})$", $1 \leq i \leq n-1$.

Note that only a subset of the paths that contain an attribute involved in the source query of the mapping m are useful for refinement; not every path reduces the number of false positives retrieved using m. Also, a path that reduces the number of false positives retrieved by the mapping m may reduce the number of true positives as well. Therefore, we use the F-measure to quantify whether the decrease in terms of false positives outweighs the loss in terms of true positives.

Difference. The number of false positives retrieved by a mapping m for populating r, a relation in the integration schema, may be reduced by applying the difference operator between the source query of m and a query q_s over the sources that is union compatible with *m.source*. To identify q_s, we exploit the fact that there are multiple candidate mappings for populating r. In particular, if there is a mapping m' which is a candidate for populating r that is known to return some false positives that are retrieved using the mapping m, then the source query of m' can play the role of q_s:

$$m.source \leftarrow m.source - m'.source$$

Intersection. If there is a mapping m' that is known to have true positives in common with the mapping m, then m can be refined by applying the intersection operator between the source query of m and the source query of m'. That is:

$$m.source \leftarrow m.source \cap m'.source$$

Note that this refinement may lead to a loss of true positives that are not retrieved by m'. To avoid this, we make use of ordinal annotations by using only the candidate mappings that are known to cover m in terms of true positives.

Selection. The number of false positives returned by a given candidate mapping m can be reduced by applying a selection to its source query:

$$m.source \leftarrow \sigma_C\ m.source$$

The problem is, of course, identifying a selection condition C that causes the number of false positives to be reduced. This condition can be specified based on the available user feedback. Assume, e.g., that the user is searching for proteins that belong to the Uniprot database[11]. If the available user feedback stipulates that the attribute *accession* of *Protein* cannot have the values *X51342* and *AA6513* then we can specify a selection condition that rules out all *accession* tuples containing either of these values. The downside of this technique is that it may result in predicates that overfit the available feedback, e.g., the number of conjuncts could be equal to the number of incorrect attribute values. To avoid this problem, it may be possible to use an external source of information for specifying selection conditions, e.g., ontologies that encode the application domains that are of interest to the user. An ontology is a set of concepts and relationships between them [26]. The concepts of a domain ontology are, in certain cases, associated with recognizers [19]. Typically, a recognizer is a regular expression using which it is possible to determine whether a given object is an instance of the concept in question [32]. If these recognizers are available, then they can be used to specify selection predicates for reducing the number of false positives. Assume, for example, an ontology that encodes the domain of bioinformatics, e.g., the myGrid ontology[12], and assume that this ontology contains a concept associated with the following regular expression: re = '$[A-N,R-Z][0-9][A-Z][A-Z,0-9][A-Z,0-9][0-9]$'. This expression specifies the format of Uniprot accession numbers. Only the true positive accession values match re. In other words, the concept associated with re captures the domain of the *accession* attribute. Therefore, in order to rule out the proteins that do not belong to the *Uniprot* database, we let C be a predicate that is true when the accession value matches the regular expression re.

5.2. Refining Mappings to Increase the Number of True Positives

To increase the number of true positives returned by a mapping m that is a candidate for populating a relation r, we union its source query with a query over the sources that is union compatible with *m.source* and known to retrieve true positives that are not returned by m. Here again, we use the fact that there are multiple candidate mappings for r and union *m.source* with the source query of a mapping m' that is known to retrieve true positives that are not retrieved by m:

$$m.source \leftarrow m.source \cup m'.source$$

Notice that the increase in true positives may be accompanied by an increase in false positives. Once again, we use the F-measure to establish whether the increase in terms of recall outweighs the decrease in terms of precision that may have occurred.

Relaxing a selection condition is one means that can be used for augmenting the number of true positives retrieved using m. Assume that the source query of m is of the form $\sigma_C\ q_s$. The number of true positives retrieved by m may be increased by replacing C with a less stringent condition C' such that $C \Rightarrow C'$. In its simplistic form, the relaxation consists in replacing C with $\top$, a predicate that always evaluates to *true*. In other words, $m.source \leftarrow q_s$.

5.3. Exploring the Space of Refined Mappings

The space of mappings that can be obtained by refinement is very large. The refinement operations can potentially be composed and recursively applied. For example, a refined mapping can be obtained by unioning the source query of a candidate mapping with the query obtained by joining the source query of another candidate mapping with other source relations. Also, a query obtained by joining the source query of a candidate mapping with some source relations can be joined with other source relations. This raises the question as to how to explore the space of potential mappings that can be constructed by refinement, with the objective of discovering the *best* possible mapping(s). An exhaustive enumeration of all potential mappings is likely to be too expensive. In this section, we present an evolutionary algorithm for exploring the space of refined mappings in bounded time.

Evolutionary algorithms are heuristics for solving combinatorial optimization problems [43]. The literature is rich with combinatorial optimization algorithms [8]. We chose to use an evolutionary algorithm because it is a population-based approach: unlike point-based algorithms, which explore a single solution at a time, population-based algorithms explore multiple competing solutions at a time. This feature fits our purpose since we aim to explore an initial set of candidate mappings aiming to derive a new set of better candidate mappings. Figure 10 presents the algorithm used for refining candidate mappings. The mappings given as input are iteratively refined. At each iteration, a subset of candidate mappings is selected *(line 2)*, viz., the mappings with an F-measure greater than a given threshold. This threshold can be specified either manually or based on the F-measure of the initial set of candidate mappings, e.g., the

[11]http://www.uniprot.org
[12]www.mygrid.org.uk

threshold can be set as the F-measure of the top mapping in the initial set. Variation operators are then applied to the selected mappings in order to derive new ones. To effectively and efficiently explore the space of mappings obtained by refinement, two variation operators are used, viz., a *cross over* and a *mutation* operator *(lines 3 and 4)*. The cross over operator is used to construct new mappings by combining the "good parts" of existing mappings. The mutation operator, on the other hand, is used to avoid the premature convergence towards a sub-optimal solution by diversifying the space of candidate mappings to be explored [8].

The mappings constructed using cross over and mutation are then annotated with respect to the available feedback *(line 5)*, before the *next generation* of candidate mappings for the next iteration is selected *(line 6)*. Multiple schemes can be used for selecting candidate mappings. For example, the top-k mappings in terms of F-measure can be used in the next iteration. Note, however, that a mapping with a low F-measure can be crucial for constructing the best mapping(s): this is the case, e.g., of a mapping that retrieves true positives (or false positives) that are not retrieved by other mappings. Because of this, we also use, in addition to the top-k mappings, a minimal subset of mappings that cover all the true positives and false positives identified through user feedback.

The process presented above is repeated until a termination condition is met, e.g., when the time allocated for refinement expires, or when a candidate mapping with an F-measure higher than a given threshold, e.g., 0.95, is discovered.

```
Algorithm RefineMappings
Inputs  Map : A set of candidate mappings
        UF: A set of user feedback instances
Outputs Map : A set of candidate mappings
Begin
1  While Termination condition not met Do
2      S_Map ← SelectMappings(Map)
3      M_Map ← MutateMappings(S_Map)
4      C_Map ← CrossOverMappings(S_Map, Map, UF)
5      AnnotateMappings(M_Map ∪ C_Map, UF)
6      Map ←
          SelectNewGeneration(Map ∪ M_Map ∪ C_Map)
7  Return Map
End
```

Figure 10: Evolutionary Algorithm for Refining Candidate Mappings.

Mutating a Candidate Mapping. A mapping is mutated by applying the join or selection relational algebra operators to its source query. Figure 11 illustrates the subroutine used for mutating schema mappings. Given a mapping m, a path p that originates from an attribute of a relation that is involved in the source query of m is selected *(line 2)*. The mapping obtained by joining the source query of m with the relations in p is added to the set of mutated mappings *(line 3)*. Note that the path p is selected randomly among candidate paths. However, this selection, as we shall see next, avoid paths that are known from previous iterations not to improve the F-measure.

The number of paths that originate from an attribute in the source query of m can be large and even infinite if we consider cyclic paths. To reduce the number of paths to be explored, we exploit the dependencies between paths. To illustrate this idea, consider the path:

$$p = (\langle R_1, att_1^{out}\rangle, \ldots, \langle R_i, att_i^{in}\rangle, \langle R_i, att_i^{out}\rangle, \ldots, \langle R_n, att_n^{in}\rangle)$$

and let p' be a path that originates from the same attribute as p and covers p, i.e., p' contains the sequence of attributes in p as follows:

$$p' = (\langle R_1, att_1^{out}\rangle, \ldots, \langle R_n, att_n^{in}\rangle, \langle R_n, att_n^{out}\rangle, \ldots, \langle R_k, att_k^{in}\rangle)$$

The source query of the mapping m'_r, obtained by mutating m using p', is equivalent to the join of the source query of the mapping m_r, constructed by mutating m using p, with a query over the sources. Specifically:

$$constructJoinMapping(m, p').source = \\ constructJoinMapping(m, p).source \bowtie_{jp_{n+1}} \ldots \bowtie_{jp_k} R_k$$

where $jp_i, n \le i \le k-1$, is a predicate of the form: $R_i.att_i^{out} = R_{i+1}.att_{i+1}^{in}$.

Therefore, if m_r does not retrieve any true or false positives (in which case it is of no relevance to the refinement process), then m'_r will not retrieve any true or false positives. We exploit this property in order to reduce the number of paths to consider in subsequent iterations. Specifically, if the annotations computed by the main algorithm (Figure 10, line 5) show that the mapping m_r does not return any true or false positives, then none of the paths that cover p, including p, are used for mutating m in subsequent iterations. As well as m, these paths will not be used in the next iterations for mutating any mapping m' with a source query contained within the source query of m. The above behavior is ensured by the function *getPath* (Figure 11, line 2).

Mappings can also be mutated by applying a selection predicate to their source queries. If an ontology θ_{domain} that describes the application domain of the integration schema is available, and a selection predicate *prec* can be derived based on user feedback *(lines 4,5)*, then a mapping that is obtained by augmenting the source query of m with a selection predicate *prec*, is added to the set of mutated mappings *(line 6)*.

Combining Candidate Mappings. Two mappings are crossed over by applying the union, intersection or difference relational operators to their source queries. Figure 12 shows the algorithm used for crossing over candidate mappings. Given a mapping that is provided as input and a cross over operator, e.g., union, we identify candidate mappings that can act as recombination mappings, a.k.a. neighbors. To identify the neighbors of a

```
Algorithm MutateMappings
Inputs   S_Map : A set of candidate mappings
Outputs  M_Map : A set of mutated mappings
Begin
1  For Each $m \in S\_Map$ Do
2      $p \leftarrow getPath(m)$
3      Add $constructJoinMapping(m, p)$ To $M\_Map$
4      $prec \leftarrow getPredicate(m, \theta_{domain})$
5      If $(prec \neq null)$
6          Add $\langle m.integration, \sigma_{prec} m.source \rangle$ To $M\_Map$
7  Return $S\_Map$
End
```

Figure 11: Algorithm for Mutating Mappings.

mapping *m*, we use the ordinal annotations associated with candidate mappings (see Section 3). Consider, e.g., the case of union. This operator is used to increase the number of true positives returned by a mapping *m*. The neighbors of *m* w.r.t. union are, therefore, the mappings that return true positives that are not retrieved by *m*. We reduce the set of neighbor mappings by considering only the mappings that are not covered by others in terms of true positives. That is:

$$union\_neighbors(m, Map, UF) = \{m_i \neq m \in Map \; s.t. \; (tp(m_i, UF) - tp(m, UF) \neq \emptyset) \; and \; (\nexists \, m_j \in Map, \; m_i <^{UF}_{TP} m_j)\}$$

Unlike union, the intersection and difference operators are used for reducing the number of false positives returned by a given mapping. The neighbors of a mapping *m* w.r.t. intersection are those that cover *m* in terms of true positives but not in terms of false positives (see below). The queries obtained by constructing the intersection of the source query of each of the neighbor mappings with the source query of *m*, retrieve all the true positives obtained using *m* and a subset of the false positives obtained using *m*. In other words, it allows an increase in precision without a reduction in the recall.

$$intersection\_neighbors(m, Map, UF) = \{m_i \neq m \in Map \; s.t. \; (m \leq^{UF}_{TP} m_i) \; and \; (m \nleq^{UF}_{FP} m_i)\}$$

Similarly, the neighboring mappings of *m* w.r.t. difference are the mappings that cover *m* in terms of false positives but not in terms of true positives. That is:

$$difference\_neighbors(m, Map, UF) = \{m_i \neq m \in Map \; s.t. \; (m \leq^{UF}_{FP} m_i) \; and \; (m \nleq^{UF}_{TP} m_i)\}$$

The query obtained by applying a difference operator to the source query of *m* and the source query of one of its neighbors m_i, i.e., $m.source - m_i.source$, does not return any false positive obtained using *m*. Note, however, that such queries may return a subset of the true positives obtained using *m*, i.e., the recall of the resulting mapping may be lower than that of *m*.

```
Algorithm CrossOverMappings
Inputs   S_Map : A set of candidate mappings
         Map : A set of candidate mappings
         UF : A set of feedback instances
Outputs  C_Map : A new set of mappings
Begin
1  For Each $m \in S\_Map$ Do
2  $Union\_Map \leftarrow union\_neighbors(m, Map, UF)$
3  For Each $u\_m \in Union\_Map$ Do
4      Add $\langle m.integration, m.source \cup u\_m.source \rangle$ To $C\_Map$
5  $Inter\_Map \leftarrow intersection\_neighbors(m, Map, UF)$
6  For Each $i\_m \in Inter\_Map$ Do
7      Add $\langle m.integration, m.source \cap i\_m.source \rangle$ To $C\_Map$
8  $Diff\_Map \leftarrow difference\_neighbors(m, Map, UF)$
9  For Each $d\_m \in Diff\_Map$ Do
10     Add $\langle m.integration, m.source - d\_m.source \rangle$ To $C\_Map$
11 Return $C\_Map$
End
```

Figure 12: Algorithm for Combining Mappings.

Note that some of the mappings obtained by mutation and cross-over may have lower quality than the candidate mapping used as input. These mappings will be found unfit to remain in the population when the candidate mappings to be explored in the next iteration of refinement are selected (Figure 10, line 6).

5.4. Experimental Evaluation

The approach described above for mapping refinement raises the following questions: *Can mapping refinement improve the quality of initial candidate mappings, and, if so, at what cost, i.e., what is the amount of user feedback required?*

To answer the above questions, we conducted an experiment in which the candidate mappings for *FavoriteCity* are refined using the *RefineMappings* algorithm in the light of user feedback. Specifically, we iterated over the process listed below until the F-measure of the top mapping constructed through refinement reaches the maximum, i.e., *1*.

1. Generate *10* feedback instances.
2. Annotate the set of candidate mappings.
3. Refine candidate mappings using the *RefineMappings* algorithm.

Regarding the *RefineMappings* algorithm, we chose the following setup. At every iteration in the algorithm, the three mappings with the best F-measure are selected for constructing new *offspring* mappings (Figure 10, line 2). The algorithm iterates until the population of mappings remains unchanged for *10* consecutive iterations (Figure 10, line 1). If a newly constructed mapping returns the same result set as an existing mapping, then it is removed (Figure 10, line 6). We also considered the general case in which no domain ontology that captures the domain of the source schemas is known, and, therefore, the selection

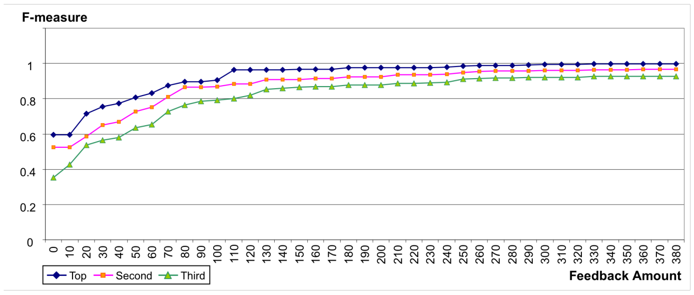


Figure 13: Average F-measure of the top, second and third mapping obtained by refinement.

conditions used for mutating the mappings are derived based on the tuples annotated through the feedback.

We repeated the above experiment *10* times. In each repetition, we specified the set of correct tuples based on a mapping that was randomly created by mutating and combining candidate mappings using the selection, join, intersection, union and difference relational operators.

The results of this experiment are shown in Figure 13, which shows the F-measure of the top, second and third mappings constructed through refinement at every feedback iteration averaged over the *10* runs.

The figure shows that the quality of the top mappings improves substantially during the first few feedback iterations. The average F-measure of the top mapping increases from 0.58 to 0.8 after the 5th feedback iteration (i.e, after collecting 50 feedback instances, which represents 2.27% of the result set retrieved by the candidate mappings). It then increases to 0.9 after collecting 80 feedback instances (i.e., 3.63% of the result set retrieved by the candidate mappings). On the other hand, the number of feedback instances required to reach the maximum F-measure value, i.e., *1*, is important. Specifically, *360* feedback instances, i.e., 16.36% of the result set retrieved by the candidate mappings, were needed for the F-measure of the top mapping to reach the maximum value. This observation can be explained as follows. If the F-measure of a mapping is high, i.e., close to *1*, then the number of expected results that are not returned by the mapping and the number of unexpected results returned by the mapping is small. Therefore, it is very likely that the feedback supplied does not cover the few expected tuples that the mapping does not return or the few unexpected tuples that the mapping does return. Because of this, the opportunities for improving the quality of such a mapping are few in principle.

The F-measure of the second and third mappings follows a pattern similar to that of the top mapping, and the above observation applies to them as well (see Table 4).

Figure 14 shows the number of offspring mappings that were created at each iteration. The figure shows that the number of offsprings is small in the first feedback iteration. This is because not all candidate mappings were annotated at this stage due to the small amount of feedback, i.e., 10 feedback instances. The number of off-spring mappings then increases until it reaches 48. This number then starts decreasing, due to the fact that the number of off-spring mappings that could improve the F-measure and that were not identified in earlier feedback iterations decreases.

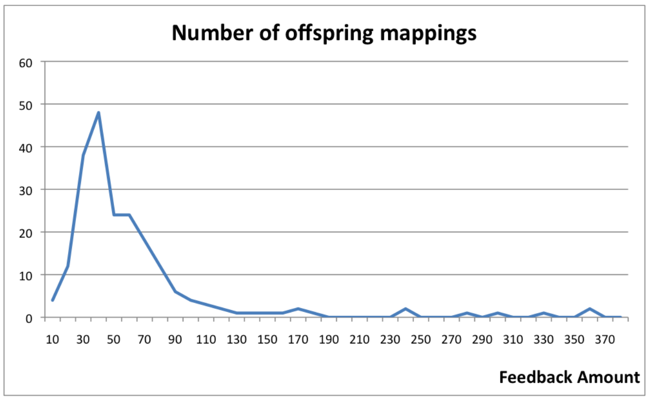


Figure 14: Number of offspring mappings.

Table 4: Average percentage of the result set that needed to be annotated by feedback to reach a given F-measure.

	top mapping	2^{nd} mapping	3^{rd} mapping
F-measure $\geq$ 0.7	0.9	2.27	3.18
F-measure $\geq$ 0.8	2.27	3.18	4.99
F-measure $\geq$ 0.9	3.63	5.9	11.36
F-measure $\geq$ 0.95	4.99	11.81	NA
F-measure $\geq$ 0.99	13.17	NA	NA
F-measure = 1	16.36	NA	NA

In summary, this experiment shows that refinement can construct good quality mappings in a pay-as-you-go fashion: the more feedback instances that are provided, the better the mappings constructed. The experiment also shows that refinement is more cost effective during the early feedback iterations. The quality of the mapping constructed improves substantially during the first feedback iterations, whereas the amount of feedback required to reach an F-measure that is close to the maximum value was, in this example, almost four times larger for only a small increment in the F-measure.

6. Annotating Integration Queries

To annotate schema mappings, we have so far considered that, given an element of the integration schema, the user provides feedback commenting on the membership of tuples to that element. In practice, however, the user is more likely to provide feedback on results to queries s/he issued. The process of evaluating an integration query can be viewed as a two step process, as illustrated in Figure 15. The query issued by the user against the integration

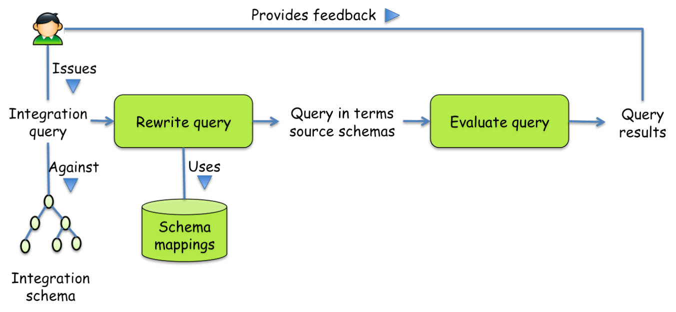


Figure 15: Query evaluation.

schema is first rewritten in terms of the constructs in the source schemas. To do so the query is unfolded using schema mappings. The query obtained is then evaluated, and the results are returned to the user. The user can then provide feedback specifying whether the result tuples meet the requirements intended by the query s/he issued.

In this section, we investigate two problems: we show how estimates for precision and recall can be computed for queries that the user issues against an integration schema, and show how the feedback the user supplies, given the results of a query, can be propagated down to the mappings used to populate the base relations in that query.

6.1. Propagating Feedback for Annotating Integration Queries

Just like schema mappings, integration queries can be annotated using precision and recall estimates that provide the user with insights into the quality of query results. In this respect, we propagate the feedback about the relations in an integration schema to queries over that schema. For a range of query operators, viz., selection, projection, join, intersection and union, we now show how the set of true positives, false positives and false negatives can be derived for an integration query given the feedback accumulated for the base relations involved in that query.

Selection. Consider the following selection query $q = \sigma_c R$, where $R = (a, b)$[13] and c is a Boolean predicate. To compute the precision and recall for q, we need to identify the set of true positives, false positives and false negatives obtained when evaluating q. To do so, we exploit existing feedback about the extent of the relation R. The true positives of R that satisfy the selection predicate c are also true positives for the query q, i.e.:

$$tp(q, UF) = \{t \in tp(R, UF) \text{ s.t. } holds(t, c)\}$$

where $holds(t, c)$ is a Boolean predicate that is true iff the tuple t satisfies the condition c.

[13]Note that we use simple binary relational tables in this section for explanation purposes, and that the technique presented for propagating feedback works for relational table schemas of any arities.

Table 5: Example of tuples annotated based on user feedback.

	name	country	feedback
t_1	Manchester	UK	true positive
t_2	Cardiff	Wells	false positive
t_3	Manchester	Morocco	false positive
t_4	Dublin	Ireland	false negative
t_5	London	UK	false negative

Similarly, the false positives (resp. negatives) of R that satisfy the selection predicate c are false positives (resp. negatives) for the query q, i.e.:

$$fp(q, UF) = \{t \in fp(R, UF) \text{ s.t. } holds(t, c)\}$$

$$fn(q, UF) = \{t \in fn(R, UF) \text{ s.t. } holds(t, c)\}$$

Projection. Consider the following projection query $q = \Pi_a R$, where $R = (a, b)$ is a relation in an integration schema, and let UF be the set of feedback instances that the user provided about the extent of R. If the feedback instances in UF are attribute-based, by which we mean feedback that annotates the values of the individual attributes of R, then we can derive true positives, false positives and false negatives for q as follows:

$$\begin{aligned} tp(q, UF) = \{ \langle x \rangle \text{ s.t. } \exists\, uf \in UF, \ (uf.AttV = \{\langle a, x \rangle\}\}) \\ \wedge (uf.Table = R) \ \wedge \ (uf.exists = true) \\ \wedge (uf.provenance \ != \ \text{'}UserSpecified\text{'})\} \end{aligned}$$

$$\begin{aligned} fp(q, UF) = \{ \langle x \rangle \text{ s.t. } \exists\, uf \in UF, \ (uf.AttV = \{\langle a, x \rangle\}\}) \\ \wedge (uf.Table = R) \ \wedge \ (uf.exists = false) \\ \wedge (uf.provenance \ != \ \text{'}UserSpecified\text{'})\} \end{aligned}$$

$$\begin{aligned} fn(q, UF) = \{ \langle x \rangle \text{ s.t. } \exists\, uf \in UF, \ (uf.AttV = \{\langle a, x \rangle\}\}) \\ \wedge (uf.Table = R) \ \wedge \ (uf.exists = true) \\ \wedge (uf.provenance \ = \ \text{'}UserSpecified\text{'})\} \end{aligned}$$

If, on the other hand, the feedback instances in UF are tuple-based, in the sense that they do not annotate the values of individual attributes but rather entire tuples, then we can only derive true positives for q. We can derive neither false positives nor false negatives for the query q.

To illustrate this, consider Table 5, which shows tuples in the relation $City(name, country)$. Given annotations as exemplified in Table 5, we can derive true positives of the projection query $\Pi_{country} City$. If a tuple t is known to be a true positive given UF, then $t[country]$ is a true positive for the query $\Pi_{country} City$. For example, the first tuple t_1 is a true positive, and so is the value of the country attribute, 'UK'. If on the other hand a tuple t is known to be a false positive given UF, then $t[country]$ can be either a true positive or a false positive for $\Pi_{country} City$. For example, both the tuples t_2 and t_3 in Table 5 are false positives. The value of the $Country$ attribute in t_2, 'Wells', is a false positive, whereas the value of the same attribute in t_3, '*Morocco*', is a true positive. Similarly, if a tuple t is known to be a false negative given UF, then $t[country]$ can be either a true positive or a false negative for $\Pi_{country}City$. For example, both t_4 and t_5 in Table 5 are false negatives. The value of the *Country* attribute in t_4, '*Ireland*', is a false negative, whereas the value of the same attribute in t_5, '*UK*', is a true positive.

The reader may wonder why we do not infer that a value is a false negative when all annotated tuples of this value are false negatives, e.g., *Ireland*. The reason is that the set of annotated tuples provides only a partial description of user expectations. In the case of the attribute value *Ireland*, for example, there may exist a non-annotated tuple $\langle Galway, Ireland\rangle$ that is unexpected according to the ground truth, in other words, false positive or true negative, in which case, the attribute value *Ireland* can be a false negative or a true negative. The same applies for the attribute value *UK*: we cannot conclude that it is a true positive despite the fact that all annotated tuples that this value appears in are true positive or false negative tuples. There may exist a non annotated tuple, e.g., $\langle York, UK\rangle$ that is unexpected, i.e., false positive or true negative, in which case the attribute value *UK* can be a true positive or a false positive. Note, however, that if we have complete knowledge of the ground truth expectations as to which tuples are expected, then we can make the above inferences.

Given the above analysis, it follows that we cannot derive false positives or false negatives for projection queries given tuple-based feedback. All we can safely derive are the following inclusion constraints:

1. $\{\langle x\rangle s.t. \langle x, -\rangle \in tp(R, UF)\} \subseteq tp(q, UF)$
2. $fp(q, UF) \subseteq \{\langle x\rangle s.t. \langle x, -\rangle \in fp(R, UF)\}$
3. $fn(q, UF) \subseteq \{\langle x\rangle s.t. \langle x, -\rangle \in fn(R, UF)\}$

The first constraint states that if a tuple t is known to be a true positive for R, then $t[a]$ is a true positive for the projection query q. The second constraint states that if a value x is a false positive for the projection query q, then there is a false positive tuple t for R in which the a attribute takes the value x. Indeed, the value x cannot be part of a true positive or a false negative tuple. The third constraint states that if a value x is a false negative for the projection query q, then there is a false negative tuple for the R relation in which the attribute a takes the value x.

Given the above inclusion constraints, we can derive lower bounds on the precision and recall of q. Consider the following cardinalities:

$$L_{tp} = |\{\langle x\rangle s.t. \langle x, -\rangle \in tp(R, UF)\}|$$
$$U_{fp} = |\{\langle x\rangle s.t. \langle x, -\rangle \in fp(R, UF)\}|$$
$$U_{fn} = |\{\langle x\rangle s.t. \langle x, -\rangle \in fn(R, UF)\}|$$

The inequalities illustrated below define lower bounds on the precision and recall of q:

$$\frac{L_{tp}}{L_{tp}+U_{fp}} \leq precision(q, UF)$$
$$\frac{L_{tp}}{L_{tp}+U_{fn}} \leq recall(q, UF)$$

Instead of computing the lower bounds for precision and recall, we can compute estimates for these metrics. We may do so by predicting the number of false positives of q in $\{\langle x\rangle s.t. \langle x, -\rangle \in fp(R, UF)\}$, and the number of false negatives of q in $\{\langle x\rangle s.t. \langle x, -\rangle \in fn(R, UF)\}$.

A tuple t of $R(a, b)$ is false positive if:

i $t[a]$ is a false positive,

ii $t[b]$ is a false positive, or

iii $t[a]$ is a true positive and so is $t[b]$, however, there is no expected tuple in R that combines $t[a]$ and $t[b]$.

If we assume that (*i*), (*ii*) and (*iii*) have the same probability of occurrence, then we can expect that $\frac{1}{3}$ of the values in $\{\langle x\rangle s.t. \langle x, -\rangle \in fp(R, UF)\}$ are false positives for q, and that the rest of values, i.e., $\frac{2}{3}$, are true positives for q.

Similarly, a tuple t of $R(a, b)$ is false negative if:

i $t[a]$ is a false negative,

ii $t[b]$ is a false negative, or

iii $t[a]$ is a true positive and so is $t[b]$, however, there is no expected tuple of R that combines $t[a]$ and $t[b]$.

If we assume that (*i*), (*ii*) and (*iii*) have the same probability of occurrence, then we can expect $\frac{1}{3}$ of the values in $\{\langle x\rangle s.t. \langle x, -\rangle \in np(R, UF)\}$ are false negatives for q, and that the rest of values, i.e., $\frac{2}{3}$, are true positives for q.

Therefore, under the above assumptions, the precision and recall of q can be estimated as follows:

$$\begin{aligned} precision(q, UF) &= \frac{tp(q,UF)}{tp(q,UF) + fp(q,UF)} \\ &= \frac{(L_{tp} + \frac{2}{3}U_{fp} + \frac{2}{3}U_{fn})}{(L_{tp} + \frac{2}{3}U_{fp} + \frac{2}{3}U_{fn}) + \frac{1}{3}U_{fp}} \\ &= \frac{L_{tp} + \frac{2}{3}(U_{fp} + U_{fn})}{L_{tp} + U_{fp} + \frac{2}{3}U_{fn}} \end{aligned}$$

$$\begin{aligned} recall(q, UF) &= \frac{tp(q,UF)}{tp(q,UF) + fn(q,UF)} \\ &= \frac{(L_{tp} + \frac{2}{3}U_{fp} + \frac{2}{3}U_{fn})}{(L_{tp} + \frac{2}{3}U_{fp} + \frac{2}{3}U_{fn}) + \frac{1}{3}U_{fn}} \\ &= \frac{L_{tp} + \frac{2}{3}(U_{fp} + U_{fn})}{L_{tp} + U_{fn} + \frac{2}{3}U_{fp}} \end{aligned}$$

Join. Consider the join query $q = R \bowtie_{(b=c)} S$, where $R(a, b)$ and $S(c, d)$ are two relations in the integration schema. In what follows, we show how the true positives, false positives and false negatives of q can be derived from the true positives, false positives and false negatives of the base relations R and S. For exposition purposes, we assume the existence of relations $R_{tp}(a, b)$, $R_{fp}(a, b)$ and $R_{fn}(a, b)$ that log the tuples of R that are true positives, false positives and false negatives, respectively. Similarly, we consider the existence of relations $S_{tp}(b, c)$, $S_{fp}(b, c)$ and

$S_{fn}(b, c)$ that log the tuples of S that are true positives, false positives and false negatives, respectively.

The true positives of q is the set of tuples obtained by joining the true positives of R with the true positives of S, i.e.:

$$R_{tp} \bowtie_{(b=c)} S_{tp}$$

A false positive of q is the result of joining a false positive of R with a false positive of S, or a false positive of R with a true positive of S, or a true positive of R with a false positive of S. The false positives of q can, therefore, be defined as follows.

$$(R_{fp} \bowtie_{(b=c)} S_{fp}) \cup (R_{fp} \bowtie_{(b=c)} S_{tp}) \cup (R_{tp} \bowtie_{(b=c)} S_{fp})$$

A false negative of q is the result of joining a false negative of R with a false negative of S, a false negative of R with a true positive of S, or a true positive of R with a false negative of S. Therefore, the false negatives of q can be defined as:

$$(R_{fn} \bowtie_{(b=c)} S_{fn}) \cup (R_{fn} \bowtie_{(b=c)} S_{tp}) \cup (R_{tp} \bowtie_{(b=c)} S_{fn})$$

Intersection. Consider the intersection query $q = R \cap S$, where R and S are intersection-compatible relations. A tuple t is a true positive for q iff it is a true positive for both R and S. Therefore, the true positives of q can be derived from the true positives of R and S using the following set intersection.

$$(R_{tp} \cap S_{tp})$$

A tuple t is a false positive for q iff it is a false positive for both R and S, or a false positive for R and a true positive for S, or a true positive for R and a false positive for S. Therefore, the false positives of q can be defined as:

$$(R_{fp} \cap S_{fp}) \cup (R_{fp} \cap S_{tp}) \cup (R_{tp} \cap S_{fp})$$

A tuple t is a false negative for q iff it is a false negative for both R and S, or a false negative for R and a true positive for S, or a true positive for R and a false negative for S. Therefore, the false negatives of q can be defined as:

$$(R_{fn} \cap S_{fn}) \cup (R_{fn} \cap S_{tp}) \cup (R_{tp} \cap S_{fn})$$

Union. Consider the union query $q = R \cup S$, where R and S are union-compatible relations.

A tuple that is a true positive of R or S is a true positive for q. Also, a tuple that is a false positive for R (resp., S) and false negative for S (resp., R) is a true positive for q. Therefore, the true positives of q can be defined as:

$$(R_{tp} \cup S_{tp}) \cup (R_{fp} \cap S_{fn}) \cup (R_{fn} \cap S_{fp})$$

A tuple that is a false positive for R (resp., S), and that is neither a true positive nor a false negative for S (resp., R), is a false positive for q. Therefore, the false positives of q can be defined as:

$$(R_{fp} - (S_{tp} \cup S_{fn})) \cup (S_{fp} - (R_{tp} \cup R_{fn}))$$

A tuple that is known to be a false negative for R (resp., S), and that is neither true positive nor false positive for S (resp., R), is a false negative for q. Therefore, the false negatives of q can be defined as:

$$(R_{fn} - (S_{tp} \cup S_{fp})) \cup (S_{fn} - (R_{tp} \cup R_{fp}))$$

6.2. Experimental Evaluation

We have shown in Section 3, through empirical evaluation, that schema mappings can be usefully annotated as to their precision and recall in a pay-as-you-go fashion by using feedback provided by end users. In this section, we report on the results of an experiment that we conducted to see whether the same applies to annotating integration queries, the base relations of which are populated using candidate schema mappings. In particular, we seek to determine whether a small amount of feedback suffices to obtain a small error in the precision and recall computed for integration queries. The parameters of the experiment are the amount of feedback provided for annotating the candidate mappings, the precision and recall of the candidate mappings, and the kind of integration query, specifically, selection, projection, join, intersection and union queries.

For the purposes of the experiment, we used the following two relations *FavoriteCity(name, province, country)* and *VisitedCity(name, province, country)*. We created for each of these relations three candidate mappings with different precisions and recalls. We also specified integration queries, that we shall present later.

For each integration query, we repeated the following two-step procedure iteratively.

1. Generate 10 feedback instances for *FavoriteCity* and 10 feedback instances for *VisitedCity*. For selection and projection queries, we use *FavoriteCity*. As with the experiment in Section 3, we used a stratified method for feedback sampling.
2. Compute the precision and recall of the query in question. In doing so, we consider all possible permutations of the candidate mappings for populating the base relations *FavoriteCity* and *VisitedCity*.

We repeated the above experiment 25 times and averaged the error obtained in precision and recall. In the following, we analyze the results obtained for each kind of query.

Selection. We used the selection query below to assess the quality of precision and recall estimates:

$$\sigma_{(country='USA')} FavoriteCity$$

The chart in Figure 16 illustrates the error in precision for the above query, and the chart in Figure 17 shows the error in recall. The charts plot the mean and min/max

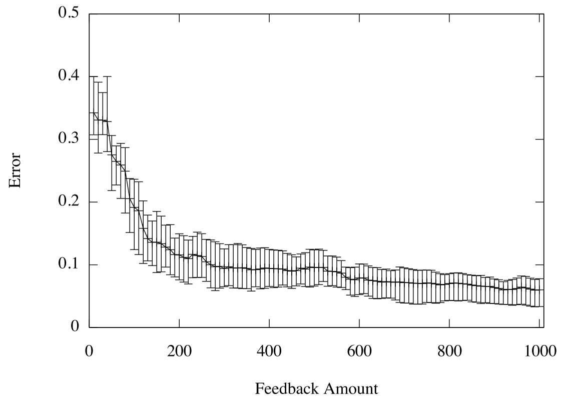


Figure 16: Error in precision for selection query.

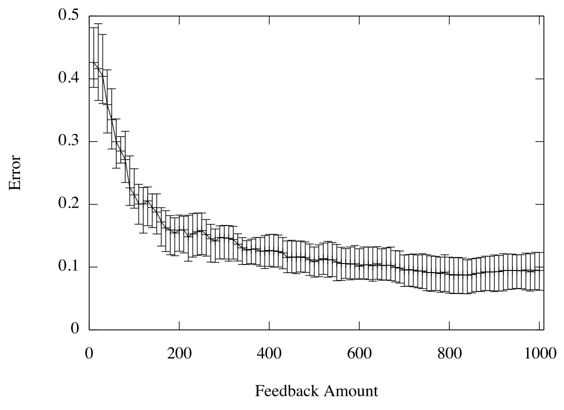


Figure 17: Error in recall for selection query.

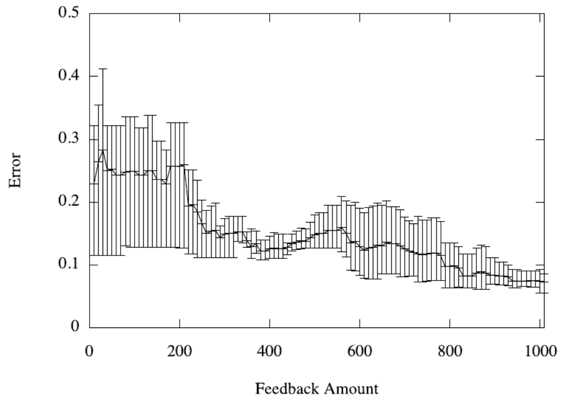


Figure 18: Error in precision for selection query with high selectivity.

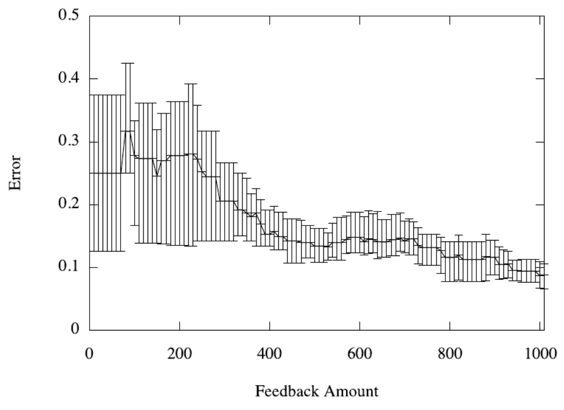


Figure 19: Error in recall for selection query with high selectivity.

error bars across the different mapping permutations[14]. Initially, the error for both precision and recall is high. However, the error decreases substantially after the first feedback iterations and keeps decreasing as the number of feedback instances increases. Specifically, to reach an average error of 0.2 in precision, 130 feedback instances were needed, which represents 4.3% of the size of the resultset returned by the candidate mappings of *FavoriteCity*. To reach the same error in recall, 160 feedback instances, that is 5.33% of the size of the result-set retrieved by the candidate mapping, were needed.

To examine if the selectivity of the query has an impact on the quality of precision and recall estimates, we rerun the above experiment using a selection query that has a higher selectivity, which is illustrated below. The chart in Figure 18 illustrates the error in precision for such query, and the chart in Figure 19 shows the error in recall. The two charts show that selectivity does indeed have an impact on the quality of the precision and recall estimates. In particular, the error does not decrease in a pay-as-you-go fashion in the first iterations. Moreover, we notice that the variability of the error is high. This can be explained by the fact the number of feedback instances that are used to compute the precision of the query is very small due to the selectivity. The same analysis applies to the case of recall.

$$\sigma_{(country\ ='Italy')}FavoriteCity$$

Projection. We used the projection query below to assess the quality of precision and recall estimates.

$$\Pi_{province}FavoriteCity$$

The chart in Figure 20 illustrates the error in precision for the above query, and the chart in Figure 21 shows the error in recall. The error for both precision and recall is small. However, the rate at which the error decreases is low. This can be explained by the following facts. Firstly, the initial error is already small. Secondly, given the nature of the projection, multiple feedback instances on the base table can coalesce into a single feedback instance for the projection query. Consider for example two tuples of *FavoriteCity* that have the same value for the *province*

[14]All the charts in the rest of this section plot the mean error and the min/max error bars.

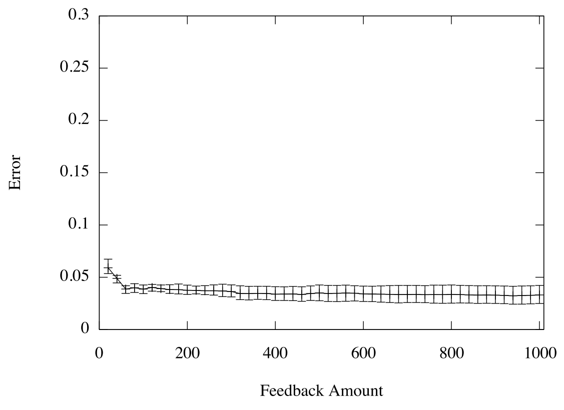


Figure 20: Error in precision for projection query.

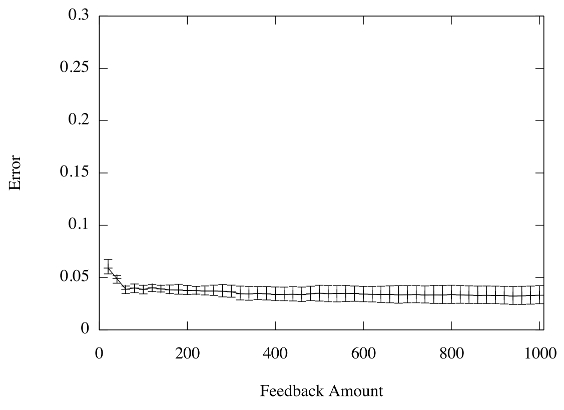


Figure 21: Error in recall for projection query.

attribute. If the two tuples are known to be true positives, then we can only derive a single true positive for the projection query.

Join. We used the equi-join query below to assess the quality of precision and recall estimates:

$$FavoriteCity \bowtie_{name} VisitedCity$$

Figure 22 shows the error obtained for precision. We distinguish two phases in this chart. In early feedback iterations, from point A to point B in the chart, the error in precision increases as the amount of feedback does. This can be explained by the following. The error in precision we compute is averaged over 25 executions of the experiment described above. Initially, i.e., when the number of feedback instances is 20, we were only able to compute the precision for a few executions (specifically, 3 executions). The join of these executions produced only one annotated tuple. This tuple was a true positive when the ground truth annotation of the join was high, and one false positive when the ground truth precision was low. As the number of feedback instances increased, the number of executions for which the join produced an annotated tuple increased too. However, not all executions followed

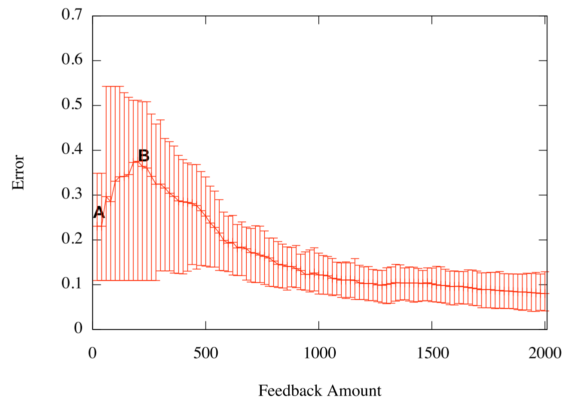


Figure 22: Error in precision for join query.

the same pattern; the join of some executions produced a false positive even though the ground truth precision is high, or a true positive when the ground truth precision is low. Because of this, we observe an increase in the error in precision from point A to point B. After point B, the join in most executions begins to produce a more representative collection of annotated tuples, and the average error in precision starts decreasing as the number of feedback instances increases as expected in a the pay-as-you-go approach.

The chart in Figure 22 also shows that the average error in precision is high compared with the error obtained for selection. For example, to reach an average error of 0.2, 600 feedback instances were needed (i.e., 10.2% of the resultset returned by the candidate mappings for *FavoriteCity* and *VisitedCity*). This can be explained by the fact that among the feedback instances generated for *FavoriteCity* and *VisitedCity*, only a small proportion satisfies the join condition, and, therefore, are used to compute the precision for the join query.

Figure 23 shows the error obtained for recall. The error in recall follows a similar pattern to that for precision. The error in recall increases in the first feedback iterations. It then starts decreasing as the number of feedback instances increases. As for precision, the error in recall is high and 580 feedback instances (i.e., 9.8% of the result-set returned by the candidate mappings for *FavoriteCity* and *VisitedCity*) were needed to reach an average error of 0.2 in recall.

Intersection. We used the intersection query below to assess the quality of precision and recall estimates:

$$FavoriteCity \cap VisitedCity$$

Figure 24 shows the error obtained for precision. As for join, in the first feedback iterations, the average error in precision does not decrease, which can be explained by the fact that a very small fraction of the result produced by intersection was annotated. Note, however, that while the average error does not decrease, we notice

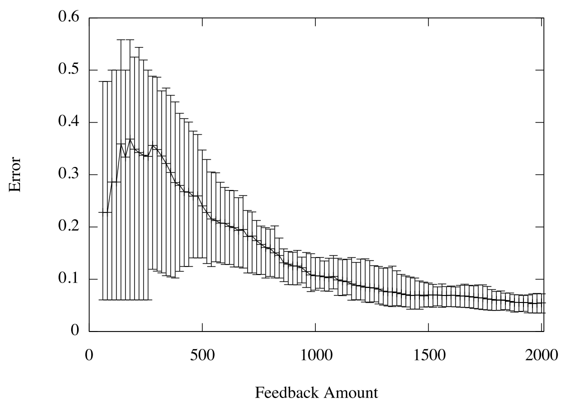


Figure 23: Error in recall for join query.

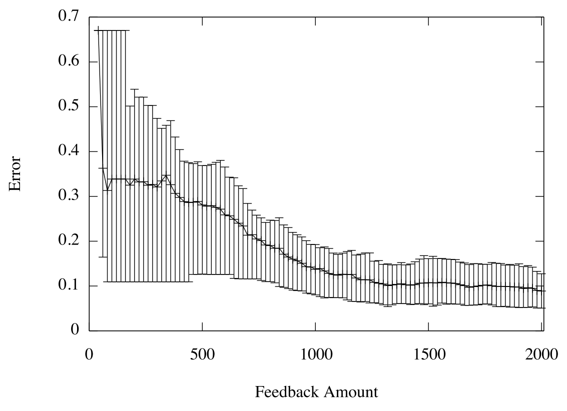


Figure 24: Error in precision for intersection query.

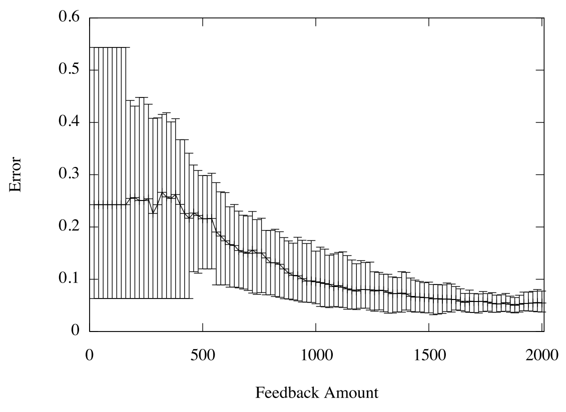


Figure 25: Error in recall for intersection query.

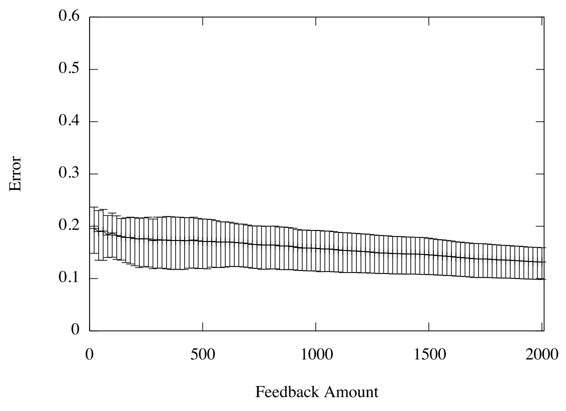


Figure 26: Error in precision for union query.

that the variability in the error decreases as more feedback is accumulated. For example, the maximum error is 0.67 when the number of feedback instances accumulated is 100. That error decreases to 0.48 when the number of feedback instances accumulated reaches 380. At later stages, the average error starts decreasing as the number of feedback instances increases. The same observation applies to recall. Figure 25 shows the error in recall.

Union. We used the union query below to assess the quality of precision and recall estimates:

$$FavoriteCity \cup VisitedCity$$

Figure 26 shows the error obtained for precision. Compared with join and intersection, the error in precision is small, and decreases as the number of feedback instances increases. Figure 27 shows the error obtained for recall. Similar to precision, the initial error in recall is small compared with that of join and intersection, and decreases as the number of feedback instances increases. This can be explained by the fact that, unlike join and intersection, all feedback instances provided at the level of the base relations are propagated up to the union query.

To further investigate the behavior of the error in the estimates computed based on propagated feedback, we ran an experiment using queries that contain 2 occurrences of the same binary relational operator.

Query with 2 occurrences of join. We used the join query below to assess the quality precision and recall estimates:

$$FavoriteCity \bowtie_{name} VisitedCity \bowtie_{name} OtherCity$$

where *OtherCity(name,province,country)* is a relation that we created for experimentation purposes.

Figure 28 shows the error obtained for precision. The precision, and therefore the error in precision, could not be estimated in early feedback iterations due to the absence of annotated result tuples. The precision could only be computed when the number of feedback instances reaches 520. Once the precision could be computed using propagated feedback, the error does not decrease straightaway. Indeed, the error in precision gets worse before it gets better. Figure 29 shows the error obtained for recall. As with precision, a large number of feedback instances, specifically 520, were needed before the recall could be computed. A further number of feedback instances were needed before the error in recall starts decreasing. This negative result is

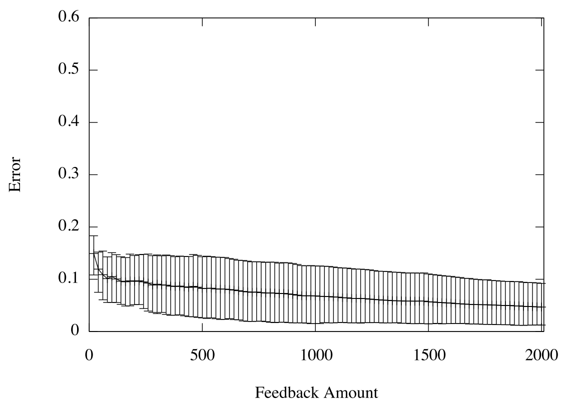


Figure 27: Error in recall for union query.

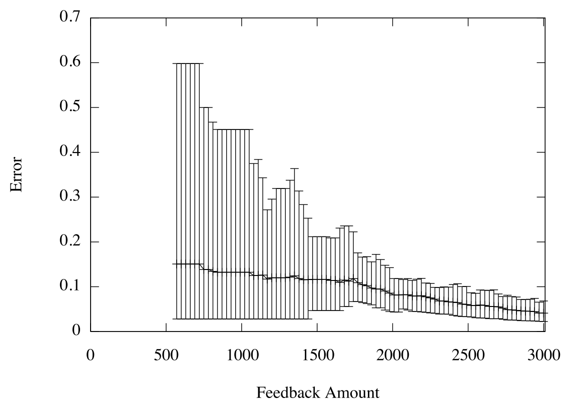


Figure 29: Error in recall for query with two join operators.

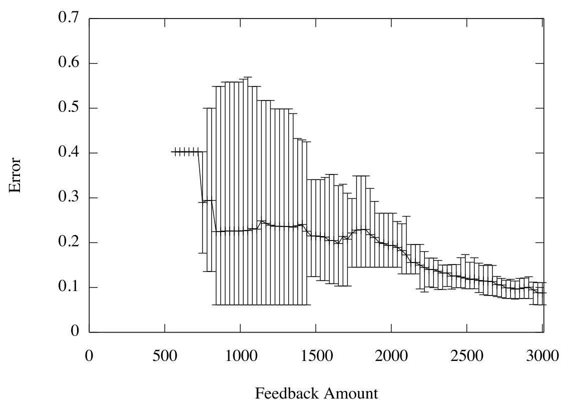


Figure 28: Error in precision for query with join operators.

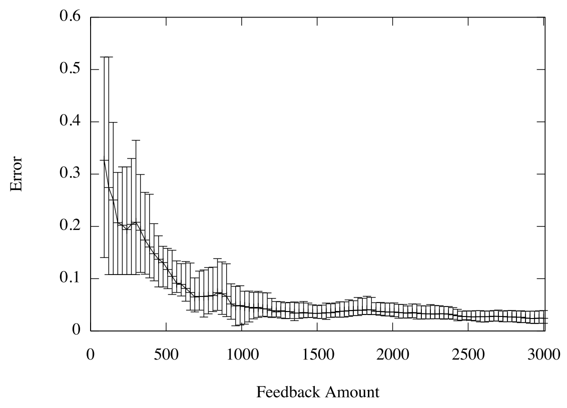


Figure 30: Error in precision for query with two join operators having a low selectivity.

due to the high selectivity of the join, which implies that only a small proportion of the feedback supplied is propagated up to the join results. To confirm the validity of this hypothesis, we ran an experiment for annotating the join query specified below. This query has a low selectivity: every tuple in each of the three relations, *FavoriteCity*, *VisitedCity* and *OtherCity*, is used to construct at least one tuple in the query results.

$$FavoriteCity \bowtie_{province} VisitedCity \bowtie_{province} OtherCity$$

Figure 30 shows the error obtained for precision, and Figure 31 shows the error obtained for recall. The results illustrated by these charts confirm our hypothesis. The precision can be computed using a small number of feedback instances, specifically 90 feedback instances. Furthermore, the error decreases in a pay-as-you-go fashion. The same analysis applies for recall.

Query with 2 occurrences of intersection. We used the intersection query below to assess the quality precision and recall estimates:

$$FavoriteCity \cap VisitedCity \cap OtherCity$$

Figure 32 shows the error obtained for precision, and Figure 33 shows the error obtained for recall. The error in precision and recall could not be computed during the first iterations. Notice, also, that once the precision could be computed, the error does not decrease in a pay-as-you-go fashion. This can be explained by the fact the number of feedback instances that are used to compute the precision of the intersection at this stage is very small. The same analysis applies to the case of recall. As for the case of join, this negative result is due to the high selectivity of the intersection.

Query with 2 occurrences of union. We used the union query below to assess the quality precision and recall estimates:

$$FavoriteCity \cup VisitedCity \cup OtherCity$$

Figure 34 shows the error obtained for precision, and Figure 35 shows the error obtained for recall. The error in both precision and recall could be computed using a small number of feedback instances. Moreover, it decreases in a pay-as-you-go fashion. This can be explained by the fact that, unlike join and intersection, all feedback instances provided at the level of the base relations can be propagated up to the union query.

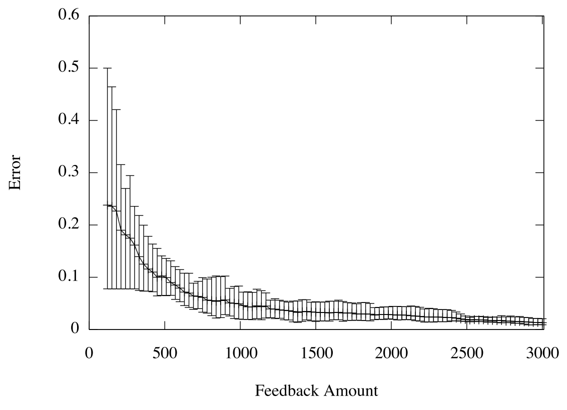


Figure 31: Error in recall for query with two join operators having a low selectivity.

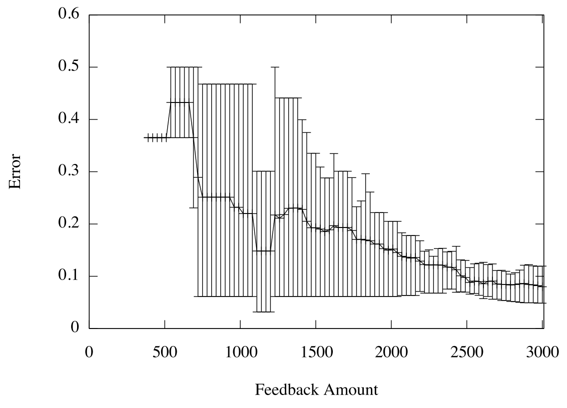


Figure 32: Error in precision for query with intersection operators.

In summary, the experiment showed that the pay-as-you-go can be applied to projection and union queries. On the other hand, the experiment showed that one needs to pay relatively more in the case of selection, join and intersection queries. In particular, if the selectivity is high then the improvement takes longer to be felt because, with few feedback iterations, the error in both precision and recall is high and increases due to the small number of feedback instances that can be propagated from the base tables to the query. In a second stage, when a representative collection of feedback instances are propagated up to the query, the error in precision and recall begins to decrease as the number of feedback instances increases.

These results suggest that the precision and recall estimates computed for projection and union queries can, in general, be trusted even with small numbers of feedback instances. On the other hand, users should anticipate high errors in the estimates computed for selection, join and intersection queries having high selectivities until a larger amount of feedback is made available.

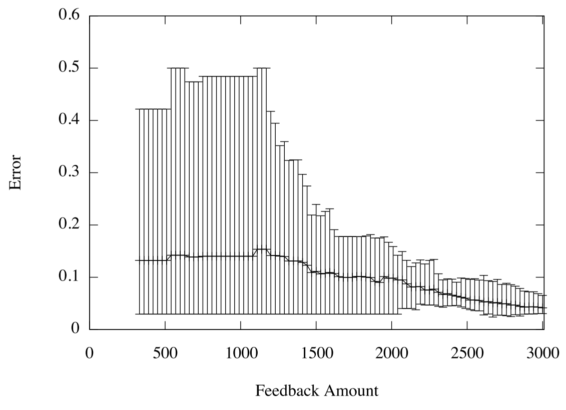


Figure 33: Error in recall for query with two intersection operators.

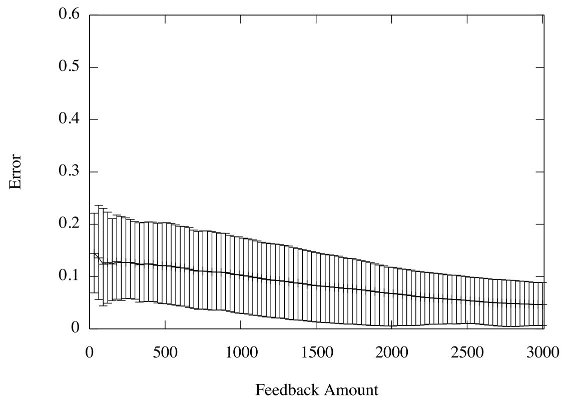


Figure 34: Error in precision for query with union operators.

6.3. *Propagating Feedback From Query Results to Schema Mappings*

We have, so far, focused on annotating the (global-as-view) mappings used to populate base relations in an integration schema. In doing so, we assumed that end users will provide feedback about the membership of tuples to those relations. In practice, however, end-users are more likely to be willing to provide feedback about the results of queries that they issued against the integration schema. With this in mind, in this section we study the problem of propagating feedback about the results of an (integration) query down to the mappings used to populate the base relations involved in that query. For each relational operator, we analyze the possibility of propagating true positives, false positives and false negatives annotations.

Selection. Consider the selection query $q = \sigma_c R$, where $R = (a, b)$[15] and c is a boolean predicate. A feedback instance specifying a true positive, false positive or false negative tuple for q gives rise to a true positive, false positive or

[15]Note that we use simple binary relational tables for explanation purposes and that the propagation technique presented in this section can be applied to any relational table.

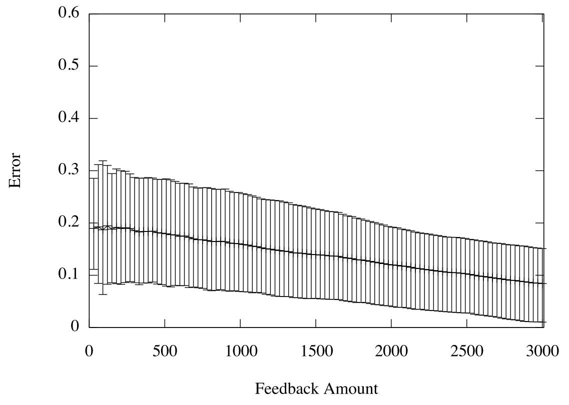


Figure 35: Error in recall for query with two union operators.

Table 6: Propagation of feedback on query results down to feedback on base relations.

	True positive	False positive	False negative
Selection	√	√	√
Projection	√*	√*	√*
Join	√	×	√
Intersection	√	×	√
Union	×	√	×

√: Feedback can be propagated.
×: Feedback cannot be propagated.
⋆: The feedback propagated is not tuple-based; it comments on values of the attributes projected out by the query.

false negative tuple, respectively, for the mapping used for populating the R relation.

Projection. Consider the projection query $q = \Pi_a R$, where a is an attribute of the R relation. We cannot propagate feedback specifying true positive, false positive or false negative tuples for q into true positive, false positive or false negatives tuples for the mapping used to populate R. All we can safely infer are true positive, false positive or a false negative values for the projected attribute a of the R relation.

Join. Consider the join query $q = R \bowtie S$, where $R = (a, b)$ and $S = (c, d)$ are two relations in the integration schema. A feedback instance specifying a true positive tuple for q gives rise to a true positive tuple for R and a true positive for S. A feedback instance specifying a false negative tuple t for q gives rise to a true positive or a false negative for R and a true positive or false negative for S. To illustrate this, consider that the tuple t was obtained by joining the tuple t_R in R and the tuple t_S in S. If t_R is retrieved by the mapping that populates R, then t_R is a true positive, otherwise, it is a false negative. Similarly, if t_S is retrieved by the mapping that populates S, then t_S is a true positive, otherwise, it is a false negative. Note that, given that t is a false negative, at least one of the two tuples t_R or t_S, if not both, is a false negative.

On the other hand, we cannot propagate feedback specifying false positive tuples for q down to R and S. To see the reason why, consider the case of tuple that is annotated as a false positive for q. Such a tuple can be constructed by joining a false positive tuple of R with a false positive tuple of S, a true positive tuple of R with a false positive tuple of S, or a false positive of R with a true positive of S. Because of this ambiguity, we cannot deduce any feedback for R or S when the user identifies a false positive in q results.

Intersection. Consider the intersection query $q = R \cap S$, where R and S are intersection-compatible relations. Similar to join queries, a feedback instance specifying a true positive tuple for q gives rise to a true positive tuple for R and a true positive for S. A feedback instance specifying a false negative tuple t for q gives rise to a true positive or a false negative for R and a true positive or false negative for S. To illustrate this, consider that the tuple t was obtained using the tuple t_R in R and the tuple t_S in S. If t_R is retrieved by the mapping that populates R, then t_R is a true positive, otherwise, it is a false negative. Similarly, if t_S is retrieved by the mapping that populates S, then t_S is a true positive, otherwise, it is a false negative. As for join queries, given that t is a false negative, at least one of the two tuples t_R or t_S, if not both, is a false negative.

On the other hand, we cannot propagate feedback specifying false negative tuples for q down to R and S. A false positive tuple can be constructed by intersection of a false positive tuple of R with a false positive tuple of S, a true positive tuple of R with a false positive tuple of S, or a false positive of R with a true positive of S. Because of this ambiguity, we cannot deduce any feedback for R or S when the user identifies a false positive in q results.

Union. Consider the union query $q = R \cup S$, where R and S are union-compatible relations. A feedback instance specifying a false positive tuple t for q may give rise to a false positive for R and/or a false positive for S. Specifically, t is a false positive for R (resp. S) if it is retrieved by the mapping that populate R (resp. S).

On the other hand, feedback specifying true positives and false positives for a union query cannot be propagated down to the base relations. Consider that the user provided a feedback instance specifying a true positive tuple for q. Such a tuple can be a true positive for R and S, a true positive for R and a false positive S, a false for R and a true positive for S. Because of this ambiguity, the feedback cannot be propagated to R or S. The same analysis applies to false negatives.

The results of the above analyses are summarized in Table 6. The table shows that the majority of feedback can be propagated down to the base relation, with the exception of false positives in the case of join and intersection queries, and true positives and false negatives for union

Table 7: Queries used for propagating of feedback on query results down to feedback on base relations.

	Query
selection	$\sigma_{country='GB'} FavoriteCity$
selection	$\sigma_{country='MA'} FavoriteCity$
selection	$\sigma_{country='AL'} VisitedCity$
selection	$\sigma_{country='E'} VisitedCity$
join	$FavoriteCity \bowtie_{country} VisitedCity$
join	$FavoriteCity \bowtie_{city} VisitedCity$
intersection	$FavoriteVity \cap VisitedCity$
union	$FavoriteVity \cup VisitedCity$

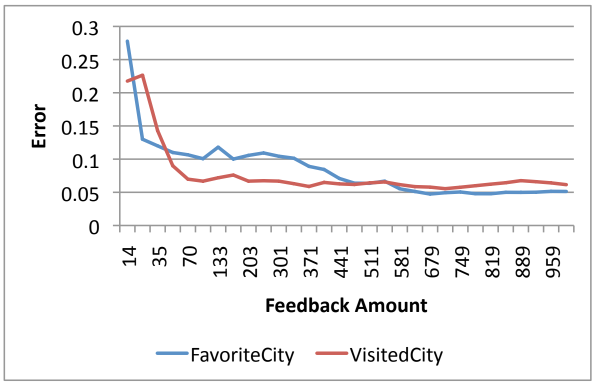


Figure 36: Error in precision.

queries. To overcome this limitation, a finer grained feedback acquisition process can be used when the query is a join, intersection or union. For example, if the user annotates a tuple t obtained by a union query, $R \cup S$, as a false positive, the system can ask the user whether t is a false positive for R, S or both.

To examine the degree to which the feedback given on query results are useful in the annotation of schema mappings that populate the base relations, we ran an experiment using the query workload in Table 7[16].

To annotate the mappings that are candidate to populate the *FavoriteCity* and *VisitedCity* relation, we applied the following procedure iteratively.

1. For each query in Table 7, generate randomly one feedback instance annotating a tuple returned by the query.
2. Propagate, when possible, the feedback generated in the previous step to the base relations: *FavoriteCity* and *VisitedCity*.
3. Compute the relative precision and recall of the candidate mappings given cumulative feedback for *FavoriteCity* and *VisitedCity*.

We then computed the average error in the precision and recall estimates for the candidate mappings of *FavoriteCity* and *VisitedCity*. Figure 36 illustrates the average error in precision for the candidate mappings, and Figure 37 illustrates the average error in recall. The two figures show that the quality of mapping annotations is incrementally improved as the user provides more feedback instances. In particular, the error in precision drops significantly in the early feedback iterations. Specifically, Figure 36 shows that when the user provided 100 feedback instances, the average error in the precision estimate for *FavoriteCity* drops from 0.27 to 0.1, and that of *VisitedCity* drops from 0.23 to 0.08. Similarly, Figure 37 shows that when the user provided 100 feedback instances, the average error in the recall estimate for *FavoriteCity* drops from 0.33 to 0.15, and that of *VisitedCity* drops from 0.31 to 0.05. As the number of feedback instances increases, the improvement in the precision and recall estimates diminishes. For example, we observe little change in the precision estimate for *VisitedCity* beyond 100 feedback instances.

The reader may wonder why the error in precision and recall estimates follows the pay-as-you-go model, even when certain kinds of feedback instances are not propagated down for certain kinds of queries. For example, true positives and false negatives cannot be propagated down to the base relations for union queries (see Table 6). This can be explained by the following. The fact that the union query does not allow for propagating true positive and false negatives tuples is somewhat counter-balanced by the fact that queries such as selection, join and intersection allow for propagating those kinds of feedback. Similarly, the fact that join and intersection queries do not allow for propagating false positive tuples is counter-balanced by the fact that selection and union queries allow for propagating false positives. Therefore, the main lesson that can be learned from the above experiment is that the pay-as-you-go philosophy is applicable as long as the query workload whereby feedback instances are propagated, contains a mixture of different kinds of queries. Also, a single item of feedback for join and intersection results in two items of feedback at the mapping level.

An integration schema is, generally, composed from multiple relations. A user may provide feedback instances annotating tuples from each relation in the integration schema. This form of feedback may be expensive when the number of relations that compose the integration schema is large. To partially address this problem, the method for propagating feedback presented in this section can be exploited. Specifically, by providing feedback on queries that are issued against multiple relations in the integration schema, the feedback the user provides on the results of such queries can, for certain kinds of queries (see Table 6), be propagated to the integration relations involved in that query. A second solution that can be adopted exploits the constraints defined between the

[16]For the purpose of this experiment, we considered only queries that allow propagation of tuple-based feedback, in other words, we did not consider projection queries.

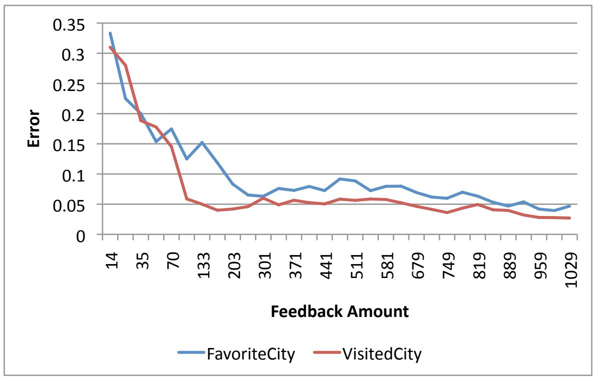


Figure 37: Error in recall.

relations in the integration schema. Indeed, the relations that compose an integration schema can be related with some dependencies, e.g., they can be connected using referential integrity constraints. Such constraints can be used to propagate feedback given on one relation to other relations in the integration schema. To illustrate this, consider two relations r_i and r_j, and consider that there is a referential integrity constraint that connects the attribute a_{r_j} of r_j to a key attribute k_{r_i} of r_i. Consider now that the user annotated a value v of the attribute a_{r_j} as expected. Given the referential integrity constraint, we can infer that the value v is expected for the attribute k_{r_i} of r_i. Conversely, if the user annotated the value v' of the attribute k_{r_i} as unexpected, then we can infer that the value v' is unexpected for the attribute a_{r_j}. Furthermore, we can infer that all tuples of r_j in which the attribute a_{r_j} takes the value v' are unexpected.

Note that feedback can be propagated for queries that involve multiple operators. Consider, for example, the query below, which involves join and selection operators. Such a query allows propagation of true positive and false negative feedback down to the base tables: *FavoriteCity* and *VisitedCity*. This is because selection allows the propagation of all kinds of feedback, and join allows the propagation of true positives and false negatives (See Table 6). This observation also applies to queries involving selection and intersection. Similarly, queries involving union and selection operators allow propagation of false positives, since the union operator only supports the propagation of that kind of feedback.

$$\sigma_{country='GB'} FavoriteCity \bowtie_{country} \sigma_{country='GB'} VisitedCity$$

On the other hand, queries involving union and join (or union and intersection) do not allow feedback propagation. Consider, for example the query illustrated bellow. Such a query does not support the propagation of any kind of feedback down to the base tables. This is because the kinds of feedback that can be propagated by union and join do not overlap. The former allows propagation of false positives, whereas the second supports the propagation of true positives and false negatives.

$$\begin{array}{c}(FavoriteCity \bowtie_{country} VisitedCity)\\ \cup \\ (OtherCity \bowtie_{country} PopularCity)\end{array}$$

7. Related Work

In this section, we present and analyze existing proposals and compare them to ours. In doing so, we focus our attention on two kinds of proposals: proposals that fall under the dataspace vision, and data exchange and integration proposals that aim to facilitate the design of schema mappings.

7.1. Dataspaces

The vision of dataspaces has been articulated as providing various of the benefits of classical data integration with reduced up-front costs. In particular, a dataspace system is distinct from a classical data integration system in the following way. Before any services can be provided, a data integration system requires a semantic integration stage to identify relevant data sources and specify the precise relationship between user requirements and the contents of the data sources. Differently, a dataspace system provides services from the start. Initially, a dataspace provides few guarantees on the quality of the services it provides. However, the quality of those services is incrementally improved given user inputs. In this paper, we described an approach in which we assumed that the integration schema together with a set of candidate schema mappings are given, and showed how those mappings can be annotated and refined over time to identify the mappings that meet user requirements. While the techniques we presented in this paper are in the spirit of dataspaces, they represent a point in the broad space of dataspace solutions, which includes aspects such as bootstrapping dataspaces to provide users with services from the start [49], providing users with the means for querying heterogeneous data sources in the absence of schema mappings, using e.g., keyword search [52, 51], indexing dataspaces [17], profiling dataspaces in the absence of schema information [30]. In this section, we analyze and compare these proposals to ours.

7.1.1. Bootstrapping and Improving Dataspaces

Sarma *et al.* [49] proposed an approach for automatically setting up a dataspace. Specifically, they show how probabilistic mediated schemas and associated probabilistic schema mappings can be automatically generated based on the similarity scores between attributes of data sources. A probabilistic mediated schema is a set of mediated schemas, each of which is associated with a probability specifying the likelihood that the schema correctly captures the domain of the sources. Similarly, a probabilistic schema mapping describes a probabilistic distribution of possible mappings between a source and a mediated schema. The authors show how the set of mediated schemas defined by a probabilistic mediated schema can be consolidated within a single mediated schema that can be exposed to the user. While the focus of the proposal by Sarma *et al.* [49] is on automatically setting up a dataspace with little or no cost, our work focuses on the improvement phase in the dataspace life-cycle, and is therefore complementary to the work by Sarma *et al.* [49]. In particular, the quality of the set of mappings specified by a probabilistic schema mappings can be measured while the system is used by soliciting feedback from the user, as we illustrated in this paper. Also, the quality of those mappings can be improved through refinement. On the other hand, our work can be extended to help users identify among the set of schemas specified by a probabilistic mediated schema, the one that best captures the requirements of the user. To this end, the feedback model introduced in this paper will need to be revised and extended.

Dong *et al.* [18] investigated the problem of data integration with uncertainty. In particular, they studied the problem of query answering against integration schema using probabilistic mappings. They showed that there are two possible semantics for such mappings: *by-table* semantics, which assumes that there exists a correct mapping, albeit unknown, and *by-tuple* semantics, which assumes that the correct mapping may depend on the particular tuple in the source data. We note that the semantics that we consider in our work fits the *by-tuple* semantics. As pointed out by Dong *et al.* [18], a critical issue with probabilistic mappings is the presence of a reliable source of probabilities. In our work, we took a different approach. We did not assume that the candidate mappings are associated with probabilities, but rather tried to gain knowledge about their fitness through feedback. In this respect, we note that the form of feedback that we introduced in our work can be used as a source for computing schema mapping probabilities.

Building on the work by Dong *et al.* [18], Gal *et al.* [25] investigated the semantics of aggregate queries when such queries are evaluated against uncertain schema mappings. Specifically, they studied three possible semantics for aggregate queries: i) under the *range semantics*, the query returns an interval within which the aggregate is guaranteed to be found, ii) using the *distribution semantics*, the query returns a set of possible values, each associated with a probability value, finally iii) *expected value semantics* is used when the user wishes to get a single value. Gal *et al.* [25] show that the above semantics combine with the *by-table* and by-tuple semantics in six ways, and proposed algorithms that can be used to efficiently process aggregate queries under the six semantics. In our work, we did not consider aggregate queries, however, the work by Gal *et al.* raises the following interesting questions: *What kinds of feedback are suitable to provide against aggregate query results with the purpose of annotating and refining underlying uncertain schema mappings?* Rather than simply specifying that a given count (resp. average or max) value is false, the user may provide additional information specifying, for example, that the value returned is too small or too large. Also, *How can the schema mappings be annotated and refined based on feedback given on aggregate query results?* The above questions bring out interesting research problems that need to be investigated further.

DeRose *et al.* [14] investigated the problem of building community portals. Given a set of relevant sources (web pages) and an integration schema that is specified by the user, the user applies plans, which consist of a set of information extraction and integration operators to populate the integration schema given the contents of the underlying sources. Although the approach followed by DeRose *et al.* is similar to ours in the sense that they assume that the integration schema is given, their proposal is different from ours as it requires the user to specify the plans that need to be applied to populate the integration schema. Indeed, the proposal by DeRose *et al.* is aimed at developers who wish to build community portals as opposed to end users who utilize such portals.

7.1.2. Annotating Schema Mappings Using Feedback

The work by McCann *et al.* [39] is similar to ours; they developed a community-based approach that solicits feedback from the community members. The objective is, however, different from ours. Their aim is to use feedback in order to inform the schema matching operation. In doing so, the feedback is used to assess the matches between attributes in two schemas. For example, user feedback can be used to verify the data type of an attribute (e.g., month), or the validity of a domain constraint (e.g., the value of an attribute is always less that the value of another). In contrast, we seek to assess the quality of executable mappings that are candidates for populating the elements of an integration schema.

Feedback can also be solicited in a way that maximizes the benefits the user can draw. For example, Jeffery *et al.* [31] developed a decision-theoretic framework for specifying the order in which candidate mappings can be confirmed by soliciting feedback from users with the objective of providing the *most benefit* to a dataspace. Our proposal is different from this work in the following respects. Firstly, Jeffery *et al.* assume that a mapping is either *correct* or *incorrect*. As we mentioned earlier, if the initial set of candidate mappings is not complete, i.e., does not contain a candidate mapping that meets the *exact* expectations of users, all the candidate mappings will ultimately be found to be incorrect. Because of this, we opted for a finer-grained annotation scheme that orders candidate mappings and labels them with metrics specifying their precision and recall. Secondly, Jeffery *et al.* do not specify the means by which feedback instances are collected. In our proposal, we showed how feedback can be provided by users by examining the results to queries that they issued, and which were evaluated using the candidate mappings. Thirdly, Jeffery *et al.* did not address the problems of mapping selection or refinement, whereas we have.

7.1.3. Profiling and Indexing Dataspaces

Salles *et al.* [48] investigated the use of hints, which they termed trails, as a mechanism for pay-as-you-go information integration. The core idea is that trails can be used as a lightweight mechanism to declaratively define relationships between loosely integrated data sources or between user requirements, specified in the form of keywords, and the underlying data sources. As mentioned by Salles *et al.*, such trails can be specified by administrators or domain experts who are knowledgeable of the structure and semantics of the underlying data sources. The form of feedback that we presented in this paper does not require knowledge of the structure or semantics of the underlying data sources for the user to be able to provide feedback. We also note that the form of feedback that we have investigated in this paper can be used as input for the creation or refinement of trails. Indeed, just like with mapping refinement, one can envisage a refinement operation that, given initial trails created when bootstrapping the dataspace, creates better quality trails in the light of feedback supplied by the user.

Dong and Halevy [17] investigated indexing support for dataspaces. In particular, they explored several extensions to inverted lists to capture structure, and showed how such extensions can be used to answer predicate queries and neighborhood keyword queries. Predicate queries allow the user to specify keywords as well as simple structural requirements, whereas neighborhood keyword queries explores association between data items. The indexing techniques developed by Dong and Halevy can be useful when refining schema mappings. In particular, where the user identifies false negative tuples (or attribute values) that are expected, and are not retrieved by any of the candidate mappings, then the keyword queries of the form investigated by Dong and Halevy can be used to detect the source relations that contain such tuples (attribute values). Those relations can be used to construct new mappings that retrieve those expected tuples (attribute values). For example, if the user identify that the tuple $\langle Manchester, GB\rangle$ is a false negative for the $City(name, country)$ relation in the integration schema, then indexing support can be used to identify the sources that contain such information.

Howe *et al.* developed a system called Quary [30], which provides capabilities for profiling dataspaces. Specifically, it provides techniques for discovering structural characteristics and properties of data sources with limited or no explicit schema information. This can be used for example, to assess the suitability of a data source for a given task or application. It can also be used to verify conditions that are required in order to add a given capability over the data source(s). Using Quary, the user may discover that instances of a given concept are always connected to the instances of another concept, or that a functional dependency holds between the instances of two concepts. In the context of our work, Quary can be used to discover structural characteristics and regularities of the integration schema by exploring the feedback supplied by the user. For example, the analysis of feedback may suggest the presence of a foreign key that is not explicitly enforced in the integration schema. Also, using Quary, users may be able to identify opportunities for generalizing feedback. For example, by analyzing the feedback instances provided by the user, it may transpire that all the tuples of the *FavoriteCity* relation in which the attribute *Country* takes a given value *GB* are false positives.

Kot *et al.* [34] investigated the problem of updates propagation based on schema mappings. The idea is that users can update relations in the underlying databases, by inserting, deleting or modifying tuples. Such updates are then propagated to other relations using pre-defined schema mappings. User input considered by Kot *et al.* are to a certain extent similar to the feedback we considered in this paper, for example, a tuple insertion (resp. deletion) can be seen as identifying a false negative (resp. false positive), and a tuple modification can be seen as identifying a false positive and providing a false negative. The objective of user input in the proposal by Kot *et al.*, is however different from the role that feedback plays in our proposal. Kot *et al.* considers that schema mappings are correct, and they focus on propagating updates when violation of those schema mappings occurs. Note, however, that our work can benefit from the idea of updates propagation investigated by Kot *et al.*. In particular, feedback provided on a given integration relation can be used to infer feedback on other integration relations when the relationships between the relations that constitute the integration schema are encoded in the form of schema mappings. Using this approach, for example, a false negative for a given relation r in the integration schema can be inferred given a true positive tuple that was identified for another integration relation r'.

7.1.4. Authoring Integration Queries over Dataspaces

Assisting end users in authoring queries over multiple sources is another area that is related to our work. An example is the Q system, which, given a set of keywords specified by the user, infers a collection of candidate structured queries over the underlying data sources [52, 51]. Similar to our approach, the Q system uses as input feedback provided by end users about the results obtained using the candidate queries. However, the solution adopted in the Q system is different from ours. They use feedback to rank candidate queries, whereas the annotations that we compute estimate the precision and recall of candidate mappings, thereby opening the door to tailorable selection, i.e., one that seeks to meet specific user requirements in terms of these criteria. In addition to ranking candidate mappings, these annotations allow the quality of each candidate mapping to be measured. Also, the assumptions underlying the *Q* system are different from ours. Using the *Q* system, it is assumed that a candidate query returns few answers, and that feedback about a given tuple can be applied to all the tuples of the same query. We do not make this assumption in our work: the same candidate mapping can produce tuples that meet users expectations (i.e., are true positives), and tuples that do not (i.e., are false positives). In addition, we allow, and derive benefits, from false negatives that the user is able to supply.

Another proposal that seeks to help users author queries over multiple sources is that by Cao *et al.* [12]. Cao *et al.* developed a method for exploring semi-structured data collections [12] with the objective of identifying queries that are relevant to the user. In doing so, they seek two kinds of feedback from end users: *soft* feedback and *hard* feedback. Soft feedback is used for ranking candidate queries, whereas hard feedback is used to rule in, or rule out a candidate query. Candidate queries are ranked, with the top query being the one that meets to the highest degree the user needs that were learned through supplied feedback. While the problem tackled in our paper is similar, in the sense that we seek to identify the mapping (or query) that meet users needs, the means we use for collecting feedback is fundamentally different. In the work, by Cao *et al.*, the user is required to provide feedback by examining query specifications. In contrast, in our work, the user provides feedback by commenting on results obtained using the candidate mappings, i.e., we assume a setting in which the user is not necessarily familiar with the sources and their schemas, and as such may not be able to provide feedback that requires examining mapping specifications.

Sattler *et al.* [50] developed VIbE, a system that assists designers in information integration tasks by providing them with the means for querying data sources using query-by-example (QBE) style queries. In the light of the specification of such queries and the results they retrieve, the designer specifies the mappings that reconcile the heterogeneities between data sources. The approach taken by the authors of this work is similar to ours in the sense that query results are used to judge the fitness of a mapping. However, different from our proposal, in VIbe, it is up to the designer, who is assumed to be able to manually specify and refine the mappings.

7.2. Data Exchange and Integration

Schema mappings are central to data exchange and integration. In data exchange, they are used as a means to transform data structured according to a source schema into data structured according to a target schema. In data integration, schema mappings are used to reformulate queries issued against the integration schema into queries that are expressed in terms of the underlying source schemas. In this section, we present data exchange and integration proposals that aim to facilitate the design, to refine and to verify schema mapping specifications, and compare them to our work.

7.2.1. Designing and Refining Schema Mappings

Our proposal for designing and refining schema mappings is inspired to some extent by the work by Yan *et al.* [57] and Alexe *et al.* [3], in that we use information provided by the source schemas. However, their approach is different from ours. In particular, in the above proposals, the mappings that do not meet users expectations are considered incorrect, and are ruled out. In contrast, in our proposal, refinement does not seek to remove "bad" candidate mappings: we allow for the co-existence of all candidate mappings, including those that are known to return false positives. This means that refinement operation seeks to create new better-quality candidate mappings from existing ones by iteratively mutating and crossing over the mappings in the initial set of mappings derived using existing generation techniques (e.g., [44, 56]).

As well as the above proposals, a number of researchers investigated the issue of designing and refining schema mappings using data examples [2, 6, 22]. For example, Alexe *et al.* [2, 6] developed a system for generating mapping specifications based on data examples that the user provides. Initially, the user provides a set of data examples, where a data example is a pair (I,J), where I is an instance of the source schema and J is an instance of the target schema. A schema mapping that fits the data examples is generated. The user can then refine the mapping specifications by modifying the data examples s/he provided or add new ones.

Our work differs from the proposal by Alexe *et al.* [2, 6] in the following points. Firstly, as stated by Alexe *et al.* [2], the intended users of their system are mappings designers who wish to design a schema mapping over a pair of source and target relational schemas. In particular, the users of the system developed by Alexe *et al.* [2] are expected to be able to detect that a schema mapping needs to be refined by examining the mapping specification, and to modify existing data examples in a way that yields the desired refinement. Using our approach, users do not have to examine mapping specifications. They do not have to specify data examples either. Rather, they are simply asked to identify expected and unexpected tuples within the integration schema. Secondly, the system developed by Alexe *et al.* [2] assumes that the user is familiar with the structure and is knowledgeable of the semantics of the source and the target schemas, and is able to provide for data structured according to the source schema, the corresponding data structured according to the target schema. The techniques that we presented in this paper are intended for users who are not familiar with the structure and semantics of the source schemas, but are able to comment on tuples of the target schema.

Fletcher and Wyss [22] developed a system called Tupelo for automatically defining schema mappings by iteratively restructuring the source schema until the resulting schema is the same as the target schema. This process is driven by data examples that are specified by the user. Tupelo considers a subset of the structural transformation defined by the Federated Interoperable relational Algebra [55]. The mapping discovery process implemented is similar to mapping refinement that we proposed in this paper, in the sense that the discovery (or refinement) of mappings is driven by a heuristic that seeks to improve the quality of the mapping. There are however fundamental differences between the mapping discovery process implemented by Tupelo and the mapping refinement presented in this paper. Firstly, like Alexe *et al.* [2], the mapping discovery process in Tupelo is driven by data examples that specify instances of the source schema and the corresponding instances structured according to the target schema. These data examples are provided by the user who is assumed to know the semantics of the source and target schemas, and is able to specify the structure of the same data content according to the source and target schema. In our setting, the user is supposed to know the semantics of the integration (target) schema, but does not have to be familiar with the semantics of the source schemas. Secondly, the data examples provided by the user in Tupelo are assumed to uniquely characterize the schema mapping that is expected by the user. These data examples are called by the authors of the Tupelo system *critical instances*. Specifying critical instances is, as acknowledged by the authors of Tupelo, a hard task that requires deep knowledge as well as the specificities of the source and target schemas. The authors of Tupelo briefly mentioned the possibility of generating critical instances using techniques developed for record linkage. However, in the context of data integration, often, the integration schema is a virtual one in the sense that it is not associated with instances. Because our approach is targeted towards users who are not able to specify critical feedback instances that can uniquely be used to specify the mapping, the user provides feedback specifying both expected as well as unexpected results. This is an important feature: negative feedback specifying unexpected tuples allows reduction of the space of undesired solutions during the search. Thirdly, the mapping discovery process in Tupelo assumes that it is always possible to derive the schema mapping that exactly captures user requirements based on data examples. In certain cases, this process may fail to locate the ideal mapping, either because the content of the sources does not allow such a mapping to be derived, or because the mapping language is not expressive enough to allow for the derivation of such a mapping. Using Tupelo in such cases, the process of mapping discovery simply fails. Instead, using our approach, the refinement process attempts to identify the best mappings, in terms of F-measure, in the light of the feedback provided by the user.

Also related to our work, is the MapMerge operator developed by Alexe *et al.* [5]. Given a set of mappings, which are expressed as second order tuple generating dependencies [21], between one (or more) source schema and a target schema, MapMerge correlates those mappings in a meaningful manner. In particular, the MapMerge operator ensures that duplicates tuples are not created within the target schema, and that the associations between tuples in the source and target schemas are preserved. In our work, we confine ourselves to global as view mappings, using which every element in the integration (target) schema is associated with a query over the sources that can be used to populate such a relation. Our work can benefit from the MapMerge operator. In particular, the mappings that are selected using our method to populate the different relations in an integration schema can be correlated using MapMerge. Also, MapMerge can be used to check that the feedback provided by the user on different relations of the integration schema, are consistent with the constraints defined between those relations.

7.2.2. Verifying Schema Mappings

While schema mappings can be automatically derived using existing mapping generation techniques [44, 56], the mappings output by these techniques may not conform to users expectations. Some researchers attempted to address the issue of mapping verification within the context of data exchange. Chiticariu *et al.* [13] proposed a debugger for understanding and exploring schema mappings. To do this, they compute, and display on request, the relationships between source and target data with the schema mapping in question. Bonifati *et al.* proposed Spicy [9, 10], a system for verifying the quality of mappings between a source and target schema. To verify a collection of schema mappings, their source queries are issued against the source schema and the results obtained are compared with instances from the target schema, the contents of which are assumed to be available. The results of this comparison are meant to identify incorrect mappings and to suggest to designers which mappings are likely to be accurate. Using the above tools, the verification of schema mappings takes place before the data integration system is setup, potentially incurring a considerable up-front cost [23, 27]. In contrast, our proposal falls under the dataspaces vision, since we seek to annotate and refine the candidate mappings as the data integration proceeds incrementally.

7.2.3. Soliciting Feedback on Universal Solutions

In this paper, we considered the case in which the user provides feedback on tuples in which all attributes are bound to constant values. This approach was also adopted in other proposals, e.g., the work on authoring integration queries by Talukdar et al. [7]. A second approach that can be adopted when soliciting feedback is the use of universal solutions.

A universal solution exhibits properties that favor them over other solutions when providing feedback. Firstly, a universal solution captures exactly what the mappings specify: it captures no less and no more than what the mapping specifications state [21]. Therefore, providing feedback on tuples that belong to a universal solution, allows the user's attention to focus on tuples that are an exact reflection of the candidate mapping specification. Secondly, the feedback given on tuples that belong to a universal solution can be propagated to tuples that belong to other solutions. To illustrate this, consider that T_i is a relation in the integration schema T, and consider that the user provides feedback annotating a tuple t of the universal solution J (for a given candidate mapping). Given the feedback supplied by the user, we can infer feedback on tuples that are obtained in other solutions for that candidate mapping. Specifically, for every solution J', we can make the following inference:

If t is expected in J, then $h(t)$ is expected in J'.

The above inference follows from the definition of a universal solution [21]. Unfortunately, we cannot propagate false positives from the universal solution to other solutions. In particular, if t is unexpected in J, then we cannot infer that $h(t)$ is unexpected in J', unless J' is homomorphically equivalent to J'.

Note that while universal solutions exhibit good properties as discussed above, it assumes that the user is knowledgeable of the semantics of the target schemas, and are able to specify universal solutions. In dataspaces in general different categories of user may contribute to pay-as-you-go-integration. Those who are knowledgeable with the semantics of the sources schemas and the target schema will be in a position to specify universal solutions that capture the semantics of the mappings that they want. However, the approach that we described in this paper, is primarily targeted towards users who are familiar with the semantics of the target schema, but not that of the source schema, and do not have the necessary skill to be able to directly specify universal solutions.

8. Conclusions

In this paper, we explored the use of feedback supplied by end users for annotating, selecting and refining schema mappings in the context of dataspaces. We showed how schema mappings can be incrementally annotated with metrics that estimate the precision and recall of the results they retrieve based on feedback supplied by end users. The results of evaluation exercises showed the effectiveness of our solution. They demonstrated that it follows the *pay-as-you-go* philosophy: the more feedback the user supplies, the better the quality of the estimates computed. Empirical evaluation showed that mapping annotation is more cost effective in early feedback iterations (in that it suffers from diminishing returns), and that inconsistencies in feedback (as expected) may lead to an increase in the error of the estimates computed.

We presented a method for selecting mappings for populating an element of the integration schema that is responsive to user needs. This method casts the problem of mapping selection as a constrained optimization problem that we solve using the mesh adaptive direct search algorithm. Ours is, to the best of our knowledge, the first proposal that tackles the problem of schema mapping selection in this way. The experiment showed that the method for selecting schema mappings is effective: a small number of feedback instances allows the selection of the schema mappings that would have been chosen had the complete set of expected tuple results been known.

We also showed how better quality mappings can be constructed from an initial set of candidate mappings through refinement by using an evolutionary algorithm. The pay-as-you-go approach to gradual improvement seems to be applicable to mapping refinement. The more feedback used as input, the better the quality of the mappings obtained through refinement. Also, as for mapping annotation, mapping refinement is more cost effective in early feedback iterations.

Finally, we investigated the problem of annotating queries issued over an integration schema by propagating feedback given on the base relations that those queries rely on. Overall, the lesson is that the lower the query selectivity, the smaller the error in the precision and recall estimated for the query.

The mapping strategies that are reported in the paper have been incorporated within a toolkit for managing data integration systems, called DSToolkit [1,2], throughout their life cycle, from their initialization to their improvement and maintenance. In particular, DSToolkit provides users with the means to select the data sources they wish to integrate, to match their schemas, to create schema mappings based on the matches identified, and to annotate, select and refine such mappings using the methods presented in the paper.

References

[1] M. A. Abramson, C. Audet, and J. E. Dennis. Nonlinear programing with mesh adaptive direct searches. *SIAG/Optimization Views-and-News*, 17(1):2–11, 2006.

[2] B. Alexe, B. T. Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–144, ACM, 2011.

[3] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19. IEEE, 2008.

[4] B. Alexe, L. Chiticariu, and W. C. Tan. Spider: a schema mapping debugger. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1179–1182. ACM, 2006.

[5] B. Alexe, M. A. Hernández, L. Popa, and W. C. Tan. MapMerge: Correlating Independent Schema Mappings. In *Proceedings of the 36th International Conference on Very Large Data Bases*, pages 81–92. ACM, 2010.

[6] B. Alexe, P. G. Kolaitis, and W. Chiew Tan. Characterizing schema mappings via data examples. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 261–272. ACM, 2010.

[7] K. Belhajjame, N. W. Paton, S. M. Embury, A. A. A. Fernandes, and C. Hedeler. Feedback-based annotation, selection and refinement of schema mappings for dataspaces. In *Proceedings of the 13th*

International Conference on Extending Database (EDBT 2010), pages 573–584. ACM, 2010.

[8] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[9] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema mapping verification: the spicy way. In *EDBT*, pages 85–96. ACM, 2008.

[10] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. The Spicy system: towards a notion of mapping quality. In *SIGMOD*, pages 1289–1294. ACM, 2008.

[11] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. In *Inf. Syst.*, 29(2), pages 147–163, 2004.

[12] H. Cao, Y. Qi, K. S. Candan, and M. L. Sapino. Feedback-driven result ranking and query refinement for exploring semi-structured data collections. In *Proceedings of the 13th International Conference on Extending Database Technology, Lausanne, Switzerland (EDBT 2010)*, pages 3–14. ACM, 2010.

[13] L. Chiticariu and W. C. Tan. Debugging schema mappings with routes. In *VLDB*, pages 79–90. ACM, 2006.

[14] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building Structured Web Community Portals: A Top-Down, Compositional, and Incremental Approach. In *VLDB*, pages 399–410. ACM, 2007.

[15] R. Dhamankar, Y. Lee, A. Doan, A. Y. Halevy, and P. Domingos. imap: Discovering complex mappings between database schemas. In *SIGMOD*, pages 383–394. ACM, 2004.

[16] H. H. Do and E. Rahm. Matching large schemas: Approaches and evaluation. *Inf. Syst.*, 32(6):857–885, 2007.

[17] X. L. Dong, and A. Y. Halevy. Indexing Dataspaces. In *SIGMOD*, pages 43–54. ACM, 2007.

[18] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. *VLDB J.*, 18(2):469–500, 2009.

[19] D. W. Embley, D. M. Campbell, Y. S. Jiang, S. W. Liddle, Y.-K. Ng, D. Quass, and R. D. Smith. Conceptual model based data extraction from multiple-record web pages. *Data Knowl. Eng.*, 31(3):227–251, 1999.

[20] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[21] R. Fagin, P. G. Kolaitis, R. J. Miller, L. Popa, and W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

[22] G. H. L. Fletcher and C. M. Wyss. Data Mapping as Search. In *EDBT*, pages 95–111. Springer, 2006.

[23] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.

[24] A. Gal. Why is schema matching tough and what can we do about it? *SIGMOD Record*, 35(4):2–5, 2006.

[25] A. Gal, M. V. Martinez, G. I. Simari, and V. S. Subrahmanian Aggregate Query Answering under Uncertain Schema Mappings. *ICDE*, pages 940-951. IEEE, 2009.

[26] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[27] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of dataspace systems. In S. Vansummeren, editor, *SIGACT-SIGMOD-SIGART*, pages 1–9. ACM, 2006.

[28] A. Y. Halevy, A. Rajaraman, and J. J. Ordille. Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea (VLDB 2006)*, pages 9–16. ACM, 2006.

[29] B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the deep web. *Commun. ACM*, 50(5):94–101, 2007.

[30] B. Howe, D. Maier, N. Rayner, and J. Rucker. Quarrying dataspaces: Schemaless profiling of unfamiliar information sources In *Proceedings of ICDE Workshops*, pages 270–277. IEEE Computer Society, 2008.

[31] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, pages 847–860. ACM, 2008.

[32] R. Kaschek and S. Zlatkin. Where ontology affects information systems. In *ISTA*, pages 35–46. GI, 2003.

[33] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75. ACM, 2005.

[34] L. Kot and C. Koch. Cooperative Update Exchange in the Youtopia System. In *PVLDB*, 2(1), pages 61–75, 2009.

[35] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 233–246. ACM, 2002.

[36] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350, 2007.

[37] H. A. Mahmoud and A. Aboulnaga. Schema clustering and retrieval for multi-domain pay-as-you-go data integration systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010)*, pages 411–422. ACM, 2010.

[38] R. McCann, B. K. AlShebli, Q. Le, H. Nguyen, L. Vu, and A. Doan. Mapping maintenance for data integration systems. In *VLDB*, pages 1018–1030, 2005.

[39] R. McCann, A. Kramnik, W. Shen, V. Varadarajan, O. Sobulo, and A. Doan. Integrating data from disparate sources: A mass collaboration approach. In *ICDE*, pages 487–488. IEEE CS, 2005.

[40] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128. IEEE CS, 2002.

[41] D. A. Menascé and V. Dubey. Utility-based QoS brokering in service oriented architectures. In *ICWS*, pages 422–430. IEEE CS, 2007.

[42] G. Montiel-Moreno, G. Vargas-Solar, and J. Luis Zechinelli-Martini Modelling autonomic dataspaces using answer sets. *In Inteligencia Artificial*, 48: 3–14, 2010.

[43] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, December 2004.

[44] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[45] R. J. Miller, Daniel Fisla, Mary Huang, David Kymlicka, Fei Ku, and Vivian Lee. The Amalgam Schema and Data Integration Test Suite. Accessible at www.cs.toronto.edu/ miller/amalgam, 2001.

[46] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[47] I. Ruthven and M. Lalmas. A survey on the use of relevance feedback for information access systems. *Knowl. Eng. Rev.*, 18(2):95–145, 2003.

[48] M. A. V. Salles, J. P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. *VLDB*, 663–674, ACM, 2007.

[49] A. D. Sarma, X. Dong, and A. Y. Halevy. Bootstrapping pay-as-you-go data integration systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 861–874. ACM, 2008.

[50] K. U. Sattler, S. Conrad, G. Saake. Interactive example-driven integration and reconciliation for accessing database federations. *Inf. Sys.*, 28 (5), pages 393–414, 2003.

[51] P. P. Talukdar, Z. G. Ives, and F. Pereira. Automatically incorporating new sources in keyword search-based data integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA (SIGMOD 2010)*, pages 387–398. ACM, 2010.

[52] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *PVLDB*, 1(1):785–796, 2008.

[53] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[54] R. S. Witte and J. s. Witte. *Statistics*. Horcourt College Publishers, 2002.

[55] C. M. Wyss and E. L. Robertson. Relational languages for metadata integration *ACM Trans. Database Syst.*, 624-660, 2005.

[56] L. Xu and D. W. Embley. A composite approach to automating direct and indirect schema mappings. *Inf. Syst.*, 31(8):697–732, 2006.

[57] L.-L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, pages 485–496, 2001.