Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2013-022 September 10, 2013

Harvesting Application Information for Industry-Scale Relational Schema Matching

Nate Kushman, Fadel Adib, Dina Katabi, and Regina Barzilay

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Harvesting Application Information for Industry-Scale Relational Schema Matching

Nate Kushman Fadel Adib Dina Katabi Regina Barzilay
Massachusetts Institute of Technology

ABSTRACT

Consider the problem of migrating a company's CRM or ERP database from one application to another, or integrating two such databases as a result of a merger. This problem requires matching two large relational schemas with hundreds and sometimes thousands of fields. Further, the correct match is likely complex: rather than a simple one-to-one alignment, some fields in the source database may map to multiple fields in the target database, and others may have no equivalent fields in the target database. Despite major advances in schema matching, fully automated solutions to large relational schema matching problems are still elusive.

This paper focuses on improving the accuracy of automated large relational schema matching. Our key insight is the observation that modern database applications have a rich user interface that typically exhibits more consistency across applications than the underlying schemas. We associate UI widgets in the application with the underlying database fields on which they operate and demonstrate that this association delivers new information useful for matching large and complex relational schemas. Additionally, we show how to formalize the schema matching problem as a quadratic program, and solve it efficiently using standard optimization and machine learning techniques. We evaluate our approach on real-world CRM applications with hundreds of fields and show that it improves the accuracy by a factor of 2-4x.

1. INTRODUCTION

Modern enterprises store much of their important data using web-based applications backed by relational databases. These include both generic applications such as customer-relationship management (CRM) and enterprise resource planning (ERP), as well as customized applications that perform functions unique to an individual industry (e.g., e-commerce, inventory management, or health-record management applications). As a company's business needs change, it periodically migrates from using one application to using a different application to perform the same function; for example it may move from using Oracle's CRM application to using SAP's CRM application. This change may result from an acquisition, or just from the desire to shift from a legacy application to a more modern one that includes new desired functionality. When a company moves from using one application to another (e.g., switching from one CRM application to another), it needs to migrate all of its data from the relational schema of the old application to the relational schema of the new application. This process requires solving a *schema matching* problem, which builds a mapping from the fields of the old database to the fields of the new database. This mapping is then used to migrate the data. Despite major advances in schema matching [9, 22, 33, 23, 27, 25, 26, 15], matching large relational schemas still requires a significant manual effort, causing such migrations problems to cost hundreds of thousands or even millions of dollars [37].

Automating this process is desirable yet challenging. A typical CRM or ERP schema has a few hundred and sometimes thousands of fields [35]. Industry-specific applications may have even larger and more complex schemas. Matching large schemas is difficult since any column could (in theory) match with any other column. The problem is particularly acute for relational schemas because, unlike XML and ontology schemas, one cannot make accuracy-improving assumptions about the schema structure that constrain the set of possible matches. As a result, evaluations of fully-automated relational matching algorithms have typically been done over small relational schemas with a handful [27, 26] to a few dozen columns [25, 15].

In addition to schema size, two other challenges further complicate relational schema matching. First, some columns in the source database may map to multiple columns in the target database and vice versa, creating many-to-many relations, which require determining not just the best match for each field, but *all* of the appropriate matches. Second, some columns in the source database may have no matching columns in the target database and should be mapped to `NULL`. Most past work on automated schema matching focuses on one-to-one mappings [27, 25, 26], and only a few algorithms directly address such complex mappings [25, 15]. We have found however that, in large popular CRM applications [1, 2, 3, 4], the number of one-to-many and NULL mappings tends to exceed the number of simple one-to-one mappings.

This paper focuses on improving the accuracy of automated matching of large and complex relational schemas. Our key insight is the recognition that large database applications are designed to make the user interface (UI) and application dynamics intuitive to lay users. As a result, the UI and dynamics of two applications built for the same purpose often exhibit more consistency than the schemas themselves. To exploit this insight, we have built RSM (Relational Schema Matcher), a system that uses the consistency of the user interface to improve the performance of automatic (and semi-automatic) schema matching algorithms. RSM takes as input *empty* installations of the source and target databases. It systematically interacts with the user interface, while monitoring changes in the underlying database. This allows RSM to associate UI widgets and actions with the schema columns on which they operate.

RSM exploits this association to address the three practical challenges faced in large-scale relational schema matching:

- To improve matching accuracy in the face of large schemas, RSM exploits UI information, instead of just using schema information alone. For example, even when the schema column names of two fields do not match, the UI labels for their associated widgets often do.

- To identify potential one-to-many mappings, RSM recognizes that if updates from two conceptually different web pages are written into the same column in the source database, such a column incorporates two concepts, and hence may map to two columns in the target database (see Figure 2b).
- To identify potential NULL mappings, RSM recognizes that columns that cannot be modified in any way from the UI are often specialized or extraneous, and likely to map to NULL.

To integrate this information into the matching process, RSM introduces a new matching algorithm that formulates schema matching as an optimization problem. Specifically, the intuition underlying schema matching is that two columns that match each other are likely to show high similarity (e.g., have similar column names, similar data types, and so on). To capture this intuition, RSM formulates schema matching as a quadratic optimization, whose objective is to find the optimal match that maximizes the total similarity between the source fields and their matched target fields. The key feature of this formulation is that it incorporates not only local similarity measures (e.g., lexical similarity of two columns or their data types) but also global relational similarity measures (e.g., the relational graph structure of the source and target schemas), and jointly maximizes them within the same framework. RSM solves this optimization efficiently by combining standard machine learning and optimization techniques.

RSM builds on previous work but differs from it in two main ways. First, no past work has associated UI widgets and actions with the affected schema fields and used that information in schema matching. Existing approaches for matching web forms have used limited information from the UI [34, 21]. Their techniques, however, do not apply to our problem because they assume a large number of web forms, each with only a handful of fields [34]. In contrast, we focus on matching two relational schemas with hundreds of fields. Second, although past work has sometimes included relational information in the form of constraints [16] or as part of an iterative matching algorithm [26, 27, 18], our algorithm integrates both local and relational information into a joint optimization problem using the cosine similarity measure.

In our experiments, we compare RSM to Harmony, a state-of-art system that combines multiple matching algorithms to boost performance [28]. We test on two domains of differing complexity: four CRM applications with (on average) 300 columns each, and three publication tracking applications with about a dozen columns each. Similarly to past work, we evaluate matching accuracy using the F-measure.[1] We have the following findings:

- Leveraging information from the UI allows RSM to achieve F-measure gains of 2-4x over Harmony. Harmony is only able to achieve an F-measure of 0.39 on a version of the CRM dataset with only one-to-one mappings. Its F-measure falls to 0.18 when NULLs are added and 0.12 when one-to-many mappings are included. In contrast, RSM achieves an F-measure of 0.82 on the one-to-one CRM dataset, and an F-measure of 0.50-0.54 on the more complex datasets.
- Our optimization matching algorithm alone, without application information, provides significant gains over Harmony, leading to 1.3x gains in F-measure.
- The UI provides gains across multiple domains. In the simpler context of publication databases, RSM produces an F-measure of 0.90, and gains of more than 0.24 over Harmony.

Although RSM can provide significant gains, it does have some limitations. First, as with any automated schema matching system, the matches it generates are not perfect. We assume its output will be reviewed by a human who will generate the final matching. However, the improved accuracy provided by RSM can significantly reduce the required manual effort. Second, the machine learning component of RSM's matching algorithm does require a pair of already matched databases as training data to tune the parameters of the algorithm. Critically, however, RSM requires only one such matched database pair as training. We show in §5 that this training data does not have to come from either the source or target database, or even from a pair of databases in the same domain in order to obtain good results. In fact, when we evaluated RSM on the publication domain, we used parameters learned from training on the CRM domain, and found that performance is not affected. Thus, in practice, consultants or administrators performing database migrations should be able to obtain training data from previous migrations.

[1]F-measure is the harmonic mean of a test's precision and recall, and ranges from 0–1.

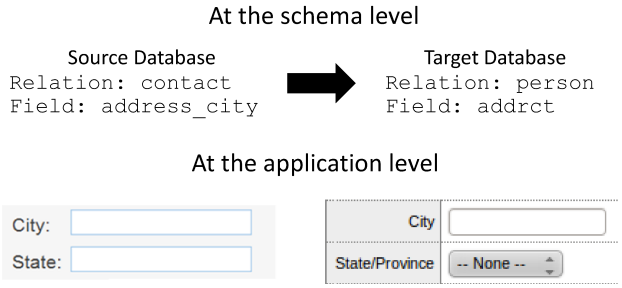


Figure 1: Example Showing UI Consistency. This example taken from matching Centric with Opentaps shows how UI labels can be more consistent than schema field names.

In summary, this paper makes the following contributions:

- It extracts previously unexploited information from the UI and the dynamics of the overlaying applications and demonstrates that this information is useful for matching large and complex relational schemas.
- It formulates automatic schema matching as a quadratic program that integrates both local information as well as global relational information. Then it solves the quadratic program through standard optimization and machine learning techniques.
- It implements and evaluates the design on real-world applications, and presents results for fully-automated matching of relational schemas with hundreds of columns, showing significant accuracy gains over existing approaches.

2. VALUE OF APPLICATION INFORMATION

Before presenting the details of RSM we first show that the application UI and the application dynamics contain information that improve schema matching effectiveness. To show this we asked a consultant to manually generate a ground-truth match for two large and popular CRM applications: Centric [2] and Opentaps [3]. While the examples and statistics in this section come from these two applications, we have obtained similar results when performing the same evaluation across the other CRM application pairs from Table 5 as well as the publication applications discussed in §4.

As we extract information from the application to use in schema matching, we use the term *feature* to refer to a property that helps match columns in the source schema with the corresponding columns in the target schema. For example, we can define a feature to be "exact name match" which is true only if the column name in the first schema matches the column name in the second schema. We

	% Having Partial UI Label Match	% Having Partial Schema Name Match
Among Ground-Truth Matches	67%	48%
Among Ground-Truth Non-Matches	3.2%	4.2%

Table 1: Consistency of UI Labels. This table shows that in general UI labels are more consistent than schema field names. For example, among all fields that should be matched to each other, 67% have at least a partial UI label match, while only 48% have a partial schema name match.

can measure how useful this feature is by computing how often the feature is satisfied among ground-truth matches vs. non-matches. A perfect feature will be satisfied in 100% of the true matches and 0% of the true non-matches. If a perfect feature exists, it can be used to perform schema matching with 100% accuracy since all one needs to do is to check the value of this feature for each pair of columns in the two schemas. In practice, however, no feature is perfect. Features are hints that are correlated with the correct match. Thus in addition to extracting features from the UI, RSM must figure out how to integrate the features together in a way that takes into account how well each one correlates with correct matches. The details of how the features are extracted, and then integrated to produce a matching are covered in section 3. We use the rest of this section to describe the most important features and show that they correlate well with correct matches.

2.1 UI Consistency

The most obvious source of matching information provided by the application comes from the UI widget(s) associated with a database field. More specifically, many database fields can be directly updated by entering data into a particular widget in a web form, and the attributes of these web form widgets can provide information useful for matching the associated database fields. For ease of exposition, in this section we assume that we know how to associate each widget to database fields. In §3.1, we explain in detail how we obtain this association.

(a) Consistency of UI Labels: The text label associated with each widget is the first feature that one notices when trying to match database fields via their associated UI widgets. For example, in Figure 1 it is clear that the two city widgets should be associated with each other because they both have the label `City`. It can also be seen from the figure that in this example the UI labels are more consistent than the underlying schema column names. But is this true in general?

To answer this question we consider "Partial Text Match", a common metric used to generate match features from a pair of text strings. The resulting feature is true if there is at least one "word" matches between the two text strings. As with past work, we clean the text strings by replacing non-alpha-numerical characters with white spaces, ignoring articles (e.g., "the", and "a"), and stemming the words before matching [5].

We then use the Partial Text Match metric to generate two different features, one from the UI labels associated with each database field, and one from the schema column names themselves. Table 1 compares the usefulness of these two features for predicting ground-truth matches. The table shows that "Partial UI Label Match" is true for 67% of the ground-truth matches, while "Partial Schema Column Name Match" is true only for 48% of the ground-truth matches. At the same time, only 3.2% of the true non-matches have a "Partial UI Label Match", while 4.2% of the true non-matches have a "Partial Schema Column Name Match". This analysis shows that in general UI labels are more consistent than schema column names, and hence incorporating them in schema matching can improve accuracy.

	Among Ground-Truth Matches	Among Ground-Truth Non-Matches
Widget Type	88%	37%

(a) Widget Type is a Weak Feature

	Among SELECT True Matches	Among SELECT Non-Matches
`SELECT` Widget Type	86%	16%
`SELECT` Partial Text Match	84%	3%
`SELECT` Full Text Match	29%	0.29%

(b) Performance by Widget Type

Table 2: Consistency of Other Widget Features. Table (a) shows that "having the same widget type" is a weak feature since it is satisfied by one third of the non-matches. In contrast, Table (b) shows that focusing on some specific widget types makes the feature more useful. Also adding attributes increases the feature's selectivity of true matches relative to non-matches.

Update Features
Its value only set when row is first added
Its value is only ever updated to the current Date/Time
Its value is updated whenever the row is updated
Context Features
Update affected by Current User context
Update affected by IP address of http connection
The value of the update depends on the previous update
The value of the update depends on the previous action

Table 3: Features Extracted From Application Dynamics. This lists features that are extracted for a database field based on the UI actions that cause the field to be updated.

(b) Consistency of Other Widget Information: The UI widgets also have many other features besides just the UI labels that can be leveraged for schema matching. Some of these are highlighted in Table 2. For example, in Table 2(a) we can see that the widget type is the same for almost all of the true matches. But since there are only a small number of widget types, the type is also the same for about a third of the non-matches, making this feature a weak indicator. We can however strengthen this indicator by combining the widget type with additional attributes. For example, if we look just at fields associated with `SELECT` (i.e., drop-down menu) widgets, then we can compute a partial text match feature over the menu values available for selection. Table 2b shows that this feature is true for almost all of the correct `SELECT-to-SELECT` matches, while it is true for only 3% of the `SELECT-to-SELECT` non-matches. Further filtering the `SELECT-to-SELECT` matches for at least one menu option with exact text match provides an even higher signal-to-noise ratio.

2.2 Application Dynamics

At first, it might seem that application information is only useful for matching columns that a user can directly update from the UI. Databases, however, have many columns that are generated automatically by the application. Simple examples of automatically

	App. Dynamics	Schema Name
Detect “Created By”	100%	21%
Detect “Modified By”	100%	28%
Detect “Creation Date”	100%	32%
Detect “Last Modified”	100%	18%

Table 4: Usefulness of UI Dynamics. This table shows that for certain types of fields the UI dynamics can identify matches more consistently than schema names. Specifically, we see that for created-by/modified-by and creation-date/last-modified-date fields, the UI can generate features which will always match these fields, while using the schema name provides a match in at most 32% of cases.

generated fields are “created/modified by” or “created/modified date”. Such columns cannot be directly associated with a particular widget in the UI. How can we use the application to help us match such fields?

For these types of fields, we exploit the dynamics of the application. Our system utilizes two aspects of the application dynamics. The first is the set of actions that cause a field to be updated. (As we explain in §3.1, we learn such actions by exploring the UI and observing underlying changes in the database.) Table 3 shows some examples of such features. These features would identify “created date” as a field that is only set upon addition and is set to the current date/time. Additionally, “modified date” would be recognized as a field that is set every time a table is touched and is always set to the current date/time.

The second category of dynamics relate to context. Here we would like to recognize actions that are context sensitive, i.e., where performing the same set of actions in a different context results in a different update to the database. By recognizing fields whose value is dependent on the current user and which are updated whenever a table is modified, we can identify “created by” fields and “modified by” fields.

Table 4 shows that this information is quite useful in practice. Specifically, we can see that the UI features always correctly recognize the appropriate field, while utilizing the schema field names only provides a correct match 18-32% of the time. One can similarly use application dynamics to identify other automatically generated fields, such as last login ip address and last login date/time.

2.3 One-To-Many

As noted earlier, we often encounter a single field in the source database that maps to multiple fields in the target database, and vice-versa. An example of this can be seen in Figure 2a, where the target database has separate tables for contacts and users, while the source database stores both of these in the same table with an additional type field indicating whether a particular record refers to a contact or a user. In order to correctly match such one-to-many fields, we need a feature that indicates which fields in the source database is likely to map to multiple fields in the target database.

As can be seen in Figure 2b, the UI provides a strong hint of when a field may be mapped one-to-many. Specifically, the figure shows that though the source schema has only one field that stores both users and contacts, at the UI level, this field is updated by two different web pages, one for users and the other for contacts. Said differently, while an application may condense the database by storing multiple concepts in the same table, it still tries to avoid confounding the two independent concepts at the UI level. We can leverage this feature to identify possible one-to-many mappings. Specifically, if a source field can be updated from multiple different web pages, or even multiple different widgets on the same web page, it is likely that the field combines multiple concepts. This

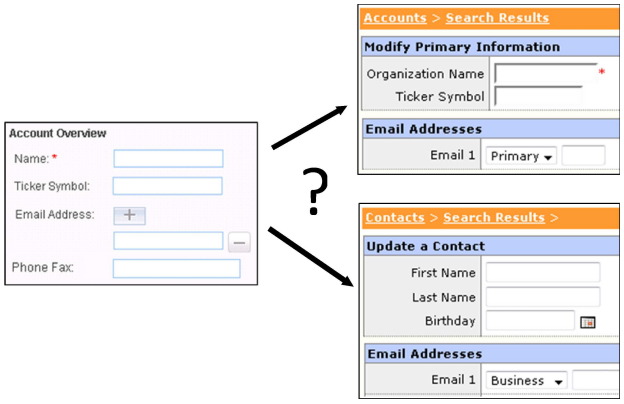


Figure 3: Example Useful Relational UI Feature. The UI label of Email Address here maps equally well to both the e-mail address on the accounts page, and the e-mail address on the contacts page. Thus, to map this field correctly, we must recognize that the widget labeled “Ticket Symbol” will map only to the top page, and so it is likely that the e-mail address widget also maps to that page.

indicates it may map to multiple fields in another database. In our CRM data-set we find that this feature correctly detects 70% of the widgets that should be mapped one-to-many, while only 5% of those that are not one-to-many have this feature. While this means that the feature is not perfect, recall that features are hints and only need to be highly correlated with the true match.

2.4 NULLs

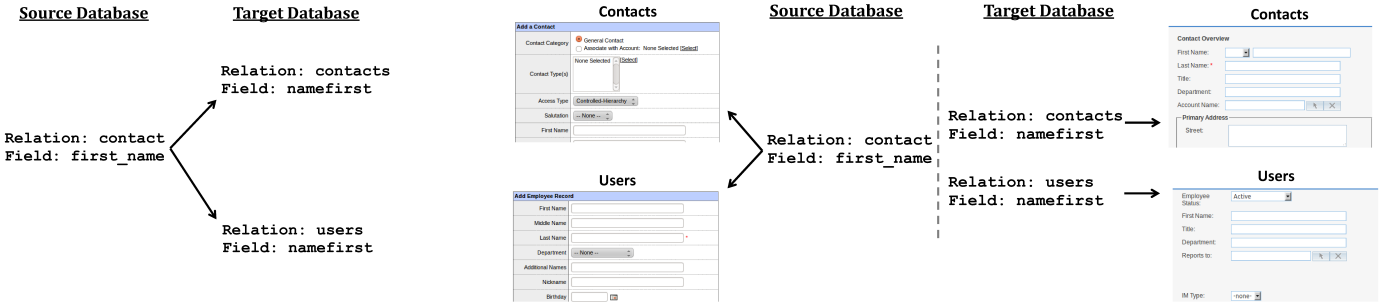
NULL mappings arise in two different cases. The majority come from fields that are simply unused. These may be vestigial fields which are no longer used by the current version of the application, fields reserved for future expansion, fields accessed only by a module of the application the company does not have, fields used for debugging, or specialized fields added by the company using the SQL interface. The remainder of the NULLs mappings come from fields that simply do not have an analog in the other database. For example one of the CRM applications has fields for “longitude” and “latitude” associated with each address, while the other CRM applications do not.

We can use the UI to recognize some (though not all) of these NULLs. Specifically any field whose value cannot be modified by any UI action is likely to be a NULL mapping. This includes both entire tables that cannot be modified or augmented as well as individual fields whose value cannot be set to anything besides the default value. We find that in our CRM application about 50% of the NULL mappings can be correctly recognized using this feature alone, which significantly reduces the confounding effect of NULL mappings.

2.5 Relational Features

Individual columns should not be mapped independently of each other. For example, columns that are in the same table in the source database are likely to map to columns in the same table in the target database. We call features like this that leverage the schema structure *relational features*. Just as one can extract relational features from the schema structure, one can also extract relational features from the UI structure.

Consider the example in Figure 3 where the label “Email Address” in the source application has a partial match with the “Email 1” label which appears on two different web pages in the target application, each associated with a distinct database field. Which of



(a) Source/Target Schemas

(b) Source/Target UIs

Figure 2: One To Many. We can take advantage of information from the UI to detect that a single field in one database may map to multiple fields in another database. Specifically, we can see here that Application 1 stores users and contacts in the contact table, whereas Application 2 has two separate tables. Thus, contact.first_name matches to multiple fields in Application 2. We can recognize that contact.first_name can be updated from two distinct UI pages, indicating that such a one-to-many mapping is likely.

the two target fields should the source field be matched with? Intuitively, we would like to match the "Email Address" source field with the top "Email 1" target field based on the fact that they are both on a web page which also contains a widget with the label "Ticker Symbol". More generally, fields associated with widgets on the same web page in one application are likely to match to fields with widgets on the same web page in the other application. This feature is analogous to the schema-level feature of two columns appearing in the same table in both schemas. We found that turning on relational UI featured increases the F-measure of RSM by 0.10.

3. RSM'S DESIGN

Our system contains two main components. The first component interacts with the database application in order to extract both the application-based features and the schema-based features. The second component is the matching algorithm which combines machine learning techniques with an optimization framework in order to utilize the extracted features to produce an optimal match. This section covers the details of each of these components in turn.

3.1 Feature Extraction Engine

To include features extracted from the application's UI and dynamics into schema matching, we first need to associate UI widgets and actions with the database fields that they operate on (e.g., associate a textbox with the database column it writes, or associate UI dynamics with automatically generated fields like "created by"). Once we have performed this association, we can generate a feature for each pair of fields in the source and target databases; example features include "same UI widget type", or "UI label similarity".[2] As noted in the previous section, field pairs with higher feature values have a higher probability of matching in the two schemas. Some of these features come from the application while the rest are extracted directly from the schema. Below we focus on how we extract application and UI features and associate them with the right fields in the database. Extracting features from the schema is a standard process, which is done in the same manner as in previous work [26, 18, 33]. Table 6 in the Appendix lists all of the features we extract.

Our feature extraction engine takes as input an empty test installation of the source and target database applications. This installation is different from the operational database application, and, hence RSM can safely perform updates and deletes in this test installation without interfering with the live application. All feature extraction operations are done on the test installation and hence can be performed by a third party who does not have access to the actual database.

To perform the extraction, RSM interacts with the application, at two levels: the UI level and the database level. At the UI level, it interacts through the application's web interface using a browser plug-in. This plug-in can walk the DOM tree[3] of a web page to extract features from the web page itself, as well as type into or click on any of the nodes in the DOM tree. At the database level, each time the plug-in performs an action on the web interface, the system parses the query log to determine both which fields in the database were read or updated, and the values of these fields. From the query log, it also extracts information indicating the tables in which records are added or deleted, and the values of these records.

The feature extraction system has three main components:

(a) The Crawler: This component randomly walks the interface clicking on UI widgets and links in an attempt to find novel web pages that it has never seen before. Periodically, the crawler resets itself to one of the pages it has seen before. This reset is biased toward web pages that have not been fully explored to ensure that all areas of the UI are discovered. Additionally, it biases its clicks towards widgets and links with text it has not seen before.

(b) The UI-DB Linker: This component takes all the web pages discovered by the crawler that contain editable widgets (i.e. INPUT, SELECT and TEXTAREA elements) and explores these pages more deeply. This is done using two subcomponents. The first subcomponent updates the editable widgets. For SELECT, radio and check-box widgets such edits are straightforward. For text-box and textarea widgets we must generate text which will be accepted by the application. For example, the application may refuse to update an e-mail address unless it contains an "@" symbol. Thus, the update component generates text values in 5 different ways: 1) by modifying the text that is already in the text-box, 2) by pulling a value from some field in this database or another database, 3) by pulling a value from some other web page, 4) by randomly generat-

[2]Note that the information we extract directly from the UI, such as the label, or widget type is associated with a single field in one database. However in order to perform the matching, we need to compute the similarity of a pair of fields. So the raw UI information is used to generate features over pair of fields. For example, UI labels are used to generate a feature measuring the UI label similarity between two fields.

[3]The DOM tree is the hierarchical representation of the objects in an HTML page.

ing text, or 5) by deleting the existing text and leaving the text-box empty. The second subcomponent clicks on the various buttons and links on the page to find the equivalent of the **Submit** button. More generally, it tries each of the buttons/links in turn until it finds one which results in some modification to the database. Once the Submit action (or sequence of actions) has been discovered, it edits each of the widgets on the web page in various different ways to determine both which fields in the database are modified, and the values to which they are modified. If, through this process, it discovers web pages it has not seen before then these pages are fed back into the first sub-component, which will thoroughly explore them.

(c) The Feature Generator: This component takes all of the information acquired by the first two components and uses it to generate a set of features for each potential match, i.e., for each pair of fields where the first field is from the source database and the second is from the target database. Two types of features are generated. The first type is widget-based features. To generate these features, the system utilizes information about which fields are updated when a given widget is modified to match fields to widgets, and vice-versa. For each widget, it then extracts features, such as the UI label, from the web page containing the widget, and associates these features with the relevant fields. The second type of features are application dynamics features. These features are for auto-generated fields whose modification is not directly linked to a widget.

The feature value associated with matching a particular source field, i, with a particular target field, j, measures the similarity between the two fields. For example, the "Widget type" feature is "1" if the two fields have the same widget type, and zero otherwise. For all text comparisons we generate 4 types of features: 1) edit distance, 2) word edit distance, 3) TF-IDF weighted word edit distance, and 4) unique word match. For schema text comparisons, word breaks are generated on all non-alpha-numeric characters (e.g., underscore). All word comparisons are also done on stemmed words using the Porter stemmer [5]. These methods for computing similarity features are standard and match what is done in past work [17].

3.2 Matching Algorithm

Automatic schema matching algorithms are based on the intuition that two columns that match each other are likely to show high similarity (i.e., they tend to have similar field names, similar data types, and so on). We want to formalize this intuition as an optimization problem that determines the match that maximizes the similarity between the source fields and their matched target fields. To do this, we need to compute a similarity score for the candidate match of each field in the source database with each field in the target database. Many different features may contribute to this score, such as how well the fields' UI labels match, as well as whether the widget types match. As we saw in §2, these features vary widely in how useful they are for predicting correct matches. Thus, we would like to integrate them together in a way that appropriately accounts for the usefulness of each feature. To do this, we first normalize the features so that their values are between zero and one, then we assign to each feature f, a weight θ_f that accounts for its usefulness. Given these weights, we can formulate the matching problem as an optimization whose objective is to find the match that maximizes the total weighted similarity between the source fields and their matched target fields. This optimization is parameterized by the feature weights, θ_f's.

In §3.2.1, we first describe how we model the matching as a quadratic optimization problem, and then in §3.2.2 we describe how we use machine learning techniques to learn the parameters (i.e., feature weights θ_f's) of this quadratic program from training data.

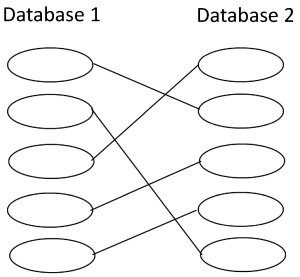


Figure 4: Bipartite Match. Each node on the left side represents a field in the source database, and each node on the right side represents a field in the target database

3.2.1 Schema Matching as an Optimization Problem

The formulation of schema matching into an optimization problem is complicated by three factors: one-to-many mappings, the presence of relational features, and NULL mappings. If we ignore these complications, and assume that every field in the source schema maps to exactly one field in the target database, the optimization problem can be formulated as a standard bipartite matching, as shown in Figure 4. Standard bipartite matching problems can be efficiently and optimally solved with the Hungarian algorithm (Kuhn-Munkres) [24]. Formulated as a linear program this looks like:

$$\begin{aligned} \underset{c_{i,j}}{\text{argmax}} \quad & \sum_{i,j} c_{i,j} \sum_f \theta_f \times M_{i,j}^{(f)} \\ \text{subject to} \quad \forall i \quad & \sum_j c_{i,j} = 1 \\ \forall j \quad & \sum_i c_{i,j} = 1 \end{aligned} \tag{1}$$

where $c_{i,j} \in \{0, 1\}$ and represent whether node i on the left side is mapped to node j on the right side. Additionally, $M_{i,j}^{(f)}$ represents the value of feature f between nodes i and j, and θ_f represents the weight of this feature. The $c_{i,j}$ constraints ensure that a one-to-one match is generated. In the rest of this section we discuss how we extend this model to handle one-to-many fields, relational features and NULL mappings.

3.2.1.1 One-to-Many.

As discussed in §2.3, we can utilize information from the UI to detect when a given field on one side may to map to multiple fields on the other side. We capture this information in our linear program by loosening the constraints on the $c_{i,j}$, which constrain them to sum to 1. For example, if we determine from the UI that field i on the left side can map to P_i different fields on the right side, then the constraint becomes: $\forall i \sum_j c_{i,j} = P_i$.

This modified formulation can be trivially integrated into the Hungarian algorithm by simply duplicating the nodes that we want to map one-to-many. As demonstrated in Figure 2, we duplicate a field if it can be updated from multiple different web pages, or even multiple different widgets on the same web page. Specifically, we duplicate it once for each unique widget from which it can be updated.

Note that this may cause unnecessary duplication, but such du-

plication is harmless. Specifically, consider a modified version of Figure 2, in which the target database, just like the source database, stores users and contacts in one field that gets updated from two UI web pages. In this a case, there would be no one-to-many mapping. Our system, however, would duplicate both the source and target fields. But such a duplication is benign since the optimization will match both the original fields and their duplicates to each other. To see why this is true, recall that the duplicates have identical features to the original fields. Thus, if the optimal solution matches the original fields to each other, it will also mach their duplicates. We can simply prune out such double mappings at the end, ensuring that the superfluous duplications are harmless.

3.2.1.2 Relational Features.

In §2.5 we motivated the idea that two fields in the source database that are associated with widgets on the same web page are likely to match to fields in the target database that are also associated with widgets on the same web page. There are many examples of these features, such as two fields whose widgets are near each other on the web page in one application are likely to match to fields which are near each other on the web page in the other application, and two fields that are in the same table on one application are likely to match to fields in the same table on another application. We call such features relational features.

Relational features cannot be directly integrated into our optimization the same way as the existing features (which we call local features), because local features quantify the affinity between two fields in *different* databases, while relational features quantify the affinity between two fields in the *same* database, e.g., these two fields are in the same table, associated with the same web page, etc. To utilize these relational features we need to use them to somehow generate a measure of the affinity between two fields in different databases. Specifically, we would like to encode into the optimization our intuition that if two fields in database 1 are related in some way, they are likely to match to two fields in database 2 that are related in that same way.

To see how this is done, let us focus on the relational feature: "being on the same web page". We will encode this feature into two binary matrices, S for database 1 and T for database 2. $S_{i,k}$ is 1 if the widget associated with field i in database 1 is on the same page as field k, and similarly for T and database 2. So if we look just at field i in database 1, and field j in database 2, then i is likely to match to j if all of the other fields on the same page as i are matched to fields on the same page as j. So if we assume that we have an existing matching c as defined above, then we can say that the affinity between i and j from this feature is proportional to:

$$\sum_{k,l} c_{k,l} \left(S_{i,k} \times T_{j,l}\right) \tag{2}$$

Said differently, if we look at each pair of fields k, l where k is in database 1 and l is in database 2, and k is a match to l, i.e., $c_{k,l} = 1$, then we would like to increase the affinity between i and j if i is on the same page as k and j is on the same page as l. However, if we use this unnormalized summation, then if i and j happen to be on a web pages with many of widgets then they will end up with a much higher affinity than two fields that are on web pages with only a few widgets. Thus, we would like to normalize this score by the number of widgets on the same page. It is not quite this simple, however, because i's web page may have a different number of widgets than j's web page. So instead we need to normalize by a suitable average of these two numbers. Thus, we choose to normalize by the product of the Euclidean norms. The reasoning behind this choice is as follows. If we view $S_{i,*}$ as a binary vector indicating the fields in database 1 that are on the same web page as i, and $T_{j,*}$ as a vector indicating which fields in database 2 are on the same web page as j then the affinity measure becomes simply the cosine similarity between these two vectors. Formally, this is:

$$A_{i,j} = \frac{\sum_{k,l} c_{k,l} \times \left(S_{i,k} \times T_{j,l}\right)}{\sqrt{\sum_n (S_{i,n})^2} \times \sqrt{\sum_m (T_{j,m})^2}} \tag{3}$$

In practice, many such relational features, g, exist; similar to the local features, we would like to learn a weight parameter θ_g for each one. Note, that each feature will have its own S and T matrix, $S^{(g)}$ and $T^{(g)}$, and so (3) becomes:

$$A^{(g)}_{i,j} = \frac{\sum_{k,l} c_{k,l} \times \left(S^{(g)}_{i,k} \times T^{(g)}_{j,l}\right)}{\sqrt{\sum_n (S^{(g)}_{i,n})^2} \times \sqrt{\sum_m (T^{(g)}_{j,m})^2}} \tag{4}$$

Additionally, instead of focusing on just local features, and just relational features we would like to appropriately trade them off relative to each other. Thus, we integrate the relational features into our original optimization by incorporating (4) into (1) to obtain:

$$\begin{aligned} \underset{c_{i,j}}{\text{argmax}} \sum_{i,j} c_{i,j} \left(\left(\sum_f \theta_f M^{(f)}_{i,j} \right) + \left(\sum_g \theta_g A^{(g)}_{i,j} \right) \right) \\ \text{subject to } \forall i \sum_j c_{i,j} = P_i \\ \forall j \sum_i c_{i,j} = Q_j \end{aligned} \tag{5}$$

3.2.1.3 Optimizing.

While our original formulation, (1), was easy to optimize, (5) is more difficult to optimize. Specifically, the $c_{i,j}$ at the start of (5) is multiplied by the $c_{k,l}$ inside of $A_{i,j}$ turning our linear program into a quadratic program. Unfortunately, this quadratic program is an instance of the quadratic assignment program whose exact solution is known to be NP-hard[12]; thus, we must resort to an approximation.

We use an iterative algorithm, similar in nature to past work [27, 18], except that each iteration of our algorithm improves the global objective score. Specifically, in each iteration v we calculate a new assignment, $c^*(v)$. We initialize the algorithm by calculating $c*(0)$ based on equation (1) above, (i.e., we ignore the relational features), and for each iteration we calculate:

$$A^{(g)}_{i,j}(v) = \frac{\sum_{k,l} c^*_{k,l}(v) \times \left(S^{(g)}_{i,k} \times T^{(g)}_{j,l}\right)}{\sqrt{\sum_n (S^{(g)}_{i,n})^2} \times \sqrt{\sum_m (T^{(g)}_{j,m})^2}} \tag{6}$$

Then for each iteration v we compute the new assignment:

$$c^*(v) = \underset{c_{i,j}}{\text{argmax}} \sum_{i,j} c_{i,j} \left(\left(\sum_f \theta_f M^{(f)}_{i,j} \right) + \left(\sum_g \theta_g A^{(g)}_{i,j}(v-1) \right) \right)$$

More generally, in each iteration we compute the value of the quadratic term using the optimal assignments from the previous iteration. Thus, each iteration becomes a basic bipartite match problem, which

we can again solve with the Hungarian algorithm. We find that in practice this approximation converges in only a few iterations, and provides good results.

3.2.1.4 *NULL Mappings.*

In our current formulation, every field in database 1 is always mapped to a field in database 2. This is the correct thing to do if there are no NULL mappings. But this formulation will cause RSM to incorrectly map all fields that should map to NULL. Past work has handled this problem by choosing a minimum match threshold, and pruning out all matches below the threshold. Such an approach is suboptimal in our system due to the relational features. With relational features, fields are no longer matched independently from each other. Simply pruning out the weak matches at the end will not account for the fact that these incorrect matches have also improperly impacted other matches during the iterations of the algorithm.

The solution to this problem is to incorporate the threshold into the optimization itself. Specifically, instead of just pruning out weak matching nodes after the optimization has converged, we prune at each iteration. Say that we would like to prune all matches whose score is below α. We start with a very low threshold in the initial mapping so nothing is pruned out. We run the iterative optimization algorithm from §3.2.1.3 until convergence. We then increase the threshold slightly, and rerun until convergence. We continue increasing the threshold and rerunning like this, until the minimum score in the resulting match is greater than α. We find that in practice it works well to set our step size to one-tenth of the difference between the initial threshold and α, ensuring the approximation algorithm is run no more than 10 times. This iterative pruning ensures that matches that do not meet the threshold are not only eliminated, but also prevented from propagating wrong information to other nodes in the final match.

3.2.2 *Learning Feature Weights*

So far we have described how to produce an optimal mapping given a set of feature weights and a pruning threshold α. Here, we discuss how to learn those parameters. Specifically, we would like to learn those parameters from training data that consists of a set of database pairs for which we have the correct mapping (or "gold mapping") indicating the list of field pairs that do match together.

For purposes of parameter estimation only, we treat our problem as a standard binary classification problem [11]. The classifier classifies each pair of fields i, j as a match or a non-match, where i comes from database 1, and j comes from database 2. If database 1 has n fields and database 2 has m fields, we generate $n \times m$ training samples. If there are g matches in the gold data, then g of these samples will be positive examples, and the rest of them of them will be negative examples. Thus, intuitively, for each field i in database 1 we generate one positive training sample for the pair i and the field j to which it is mapped in the gold annotation, and $m-1$ negative training samples for the matching of i to every other field in database 2 besides j.

The major complication that arises from this treatment of the problem is the handling of relational features. Since the classifier treats each i, j pair independently, its not entirely clear what to use for $c_{k,l}$ when computing $A_{i,j}$ in equation (4) when computing the relational features for each data point. Since we are only doing this at training time, however, we can utilize the gold annotations which specify the correct answer. Thus, we just use the c from the gold annotations (i.e. $c_{i,j}$ is 1 if i is matched to j in the gold, and 0 otherwise) to compute each of the relational features. This allows us to utilize an out-of-the-box classifier at training time.

As in standard classification, we let the classifier run until it converges to the feature weights and the pruning threshold that are most consistent with the gold mapping. We tried the two most popular classifiers, an SVM classifier [6] as well as a maximum entropy classifier [30]. We found that the SVM worked better in all cases; thus the results we report in §5 are based on training with the SVM classifier.

Database Pair	One-to-one Matches	One-to-Many Matches	NULL Matches	Total Matches
Centric-Sugar	100	87	143	330
Sugar-Opentaps	63	118	118	299
Opencrx-Centric	75	67	168	310
Opentaps-Opencrx	53	32	150	235
Centric-Opentaps	69	44	165	278
Sugar-Opencrx	71	91	134	296

Table 5: CRM Applications. This shows the six CRM database pairs used in our experiments along with the number of gold matchings of each type. Note that for each pairing, the number of one-to-many matches and the number of NULL matches differ depending on which database is considered the source and which is considered the target. So we report in the table the averages over the two directions.

3.3 Implementation Details

Our implementation includes both a component for feature extraction and a component for the optimization algorithm. Our implementation of the feature extractor is fairly simple, and we seed it with the set of web pages containing editable widgets, and a list of keywords for the submit buttons (in our case only "Save" and "Submit"). The optimization algorithm is implemented using NumPy[7] combined with a third party Hungarian algorithm library [24]. The SVM training is done using SVM Light [6].

4. EXPERIMENTAL SETUP

Our goal is to evaluate whether the use of application information in database integration can allow it to work on industry-scale problems. Thus, we evaluate on four real world CRM databases: Centric [2], Sugar [1], Opentaps [3], and Opencrx [4].

(a) Dataset. For our evaluation dataset, we generate all six distinct pairings of the four databases as shown in Table 5. We have hired a consultant to generate the gold matchings for each of the pairings. We directed the consultant to focus on the core CRM functionality and ignore the extraneous functionality included with some of the applications. As shown in the table, the resulting gold match has many one-to-many mappings and the number of NULL mappings is even more than that of one-to-one mappings. While different applications may have different ratios of these three match types, we believe that the existence of many complex NULL and one-to-one mappings is representative of real world applications. In particular, all four CRM applications have evolved over a long period of time which has led both to many vestigial fields which are simply unused, as well as many idiosyncratic datafields that are unique to an individual database (such as associating a longitude and latitude with all addresses).

We choose this evaluation dataset for two main reasons. First, we could not use the benchmarks from past work because they have included only very small relational schemas, or tree-based XML

schemas, neither of which is representative of industry scale relational databases. Additionally, none of the existing benchmarks included overlying applications. Secondly, we wanted to work with real world applications to be as representative of the industry setting as possible. Given that CRM applications are one of the most popular categories of large-scale database-backed applications in use today, they presented a particularly relevant application domain for testing our system.

(b) SVM Training. As we discussed in §1 our algorithm needs only a single matched database pair for training. Thus, in all of our evaluations we train using only a single matched database pair. More specifically, we evaluate the performance on each matched pair by training on just the single matched pair containing the other two databases (i.e., we never train on examples from the tested databases and never use more than a single pairing for training). For example when testing on the Sugar to Centric mapping we would train on just the Opentaps to OpenCRX mapping. We assume in practice that database migration consultants would have training data that comes from migrations they have done in the past.

(c) Domains. In order to ensure that our results apply to other applications besides CRM, we also evaluate on a set of three paper publication database systems. Two of these come from different research groups in our lab, and the third is the publicly available Refbase [8]. We choose this domain because of ready availability of sample applications. The gold mappings were also generated by a consultant, and the evaluation was done the same way.

(d) Metrics. Similarly to past work, we report matching performance in terms of the F-measure. The F-measure is a popular measure of a test's accuracy. It is computed as the harmonic mean of the test's precision (i.e., the percentage of correct matches in the test's output) and its recall (i.e., the percentage of total correct matches discovered by the test). Specifically, for a given schema pair, let c be the number of elements in the two schemas for which a match is predicted and the predicted match is correct. Let n the total number of predicted matches, and m the total matches in the gold standard. The precision p is c/n, or the fraction of elements with correct predictions, and the recall r is the fraction of matches in the gold mapping that were correctly predicted, or c/m, and the F-measure, f, is the harmonic mean of the two:

$$f = \frac{2pr}{p+r} \tag{7}$$

F-measure balances precision and recall. An increase in F-measure indicates a better ability to predict correct matches and also identify non-matches. We report the F-measure averaged over multiple runs of all matching tasks in a domain.

(e) Compared Systems. We compare the following three systems:

- **RSM-full:** The full version of our system as described in section §3.
- **RSM-schema-only:** The same as the full version of our system except that it does not use any of the features generated from the UI.
- **Harmony:** Harmony [28] is the schema matching component of the OpenII [33] data integration system. It uses a vote merger to combine several different match voters each of which identifies match correspondences using a different strategy. In addition, it includes a structural match component that is an implementation of similarity flooding [27]. In our evaluation, we configure Harmony to use all of its match voters. In order to output a matching, Harmony requires the user to set a threshold score, whereby all matches above this threshold are included. To determine a suitable threshold, we tried two different approaches. First, we set the threshold to zero as previous work has done [28, 14]. Second, we first run Harmony on a given pairing and find the threshold that produces the optimal F-measure on this pair; consequently, we use this threshold when testing Harmony on the other pair of databases. The pairs are chosen in exactly the same manner as the test and train sets for RSM. We found that the second approach gave better results in all test scenarios; thus, our results in §5 are reported according to this method.

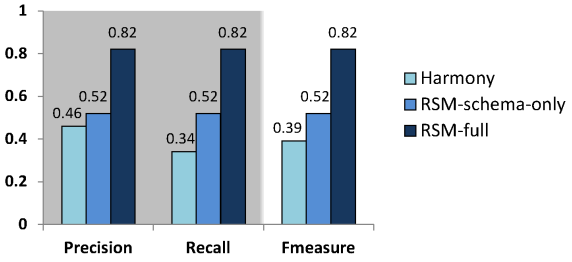


Figure 5: Basic One-to-One CRM Averages. This shows the results for the basic CRM one-to-one dataset averaged across all six database pairs. We can see that RSM-full achieves a 2x f-measure gain over Harmony, and a 1.6x gain over RSM-schema-only.

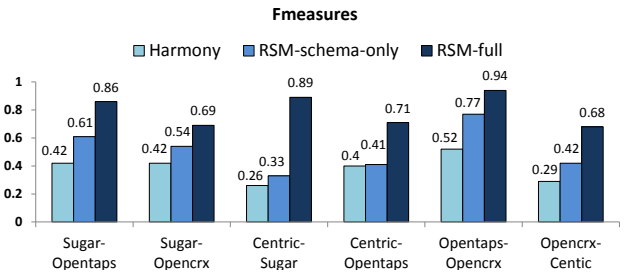


Figure 6: CRM One-to-one By Database Pair. This shows the F-measure for the one-to-one CRM dataset broken down by database pair. For each pair there are two scenarios depending on which database is the source and we report the average over these two scenarios. From this we can see that RSM-full generates gains of between 1.7x and 3.5x over Harmony and between 1.2x and 2.7x over RSM-schema-only.

5. RESULTS

As we discussed earlier, real world database integration brings both the challenge of scale as well as two additional challenges: many-to-one mappings and NULL mappings. Thus, in addition to evaluating in the basic one-to-one scenario we also separately evaluate with NULL mappings, and with many-to-one mappings. We find that RSM-full provides significant improvement in all three scenarios, leading to an F-measure gain of 2-4x over Harmony, and 1.6-2.2x over RSM-schema-only.

5.1 One-to-One

The simplest matching scenario is one in which each field in the source database matches to exactly one field in the target database, and vice versa. In this section, we evaluate RSM's performance in such one-to-one scenarios.

Method. To create a one-to-one scenario from our dataset, we simply remove all database fields that match to NULL in the gold match, and for each field matched one-to-many we include only

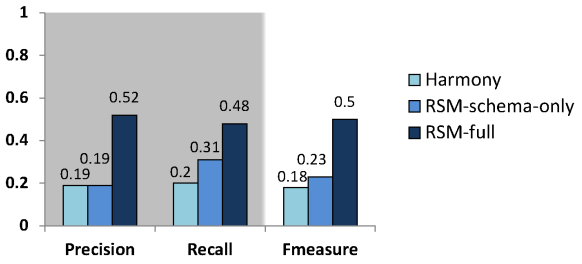


Figure 7: NULL Mapping CRM Averages. This shows the average results for the CRM dataset when NULL mappings are included. We can see that RSM-full achieves additional gains from its ability to use the UI to detect fields that are likely to map to NULL. Specifically, its gains have improved to 2.7x over Harmony and 2.2x over RSM-schema-only.

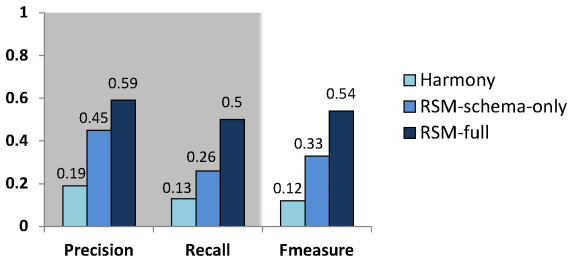


Figure 8: Many-to-One Mapping CRM Averages. This shows the average results for the CRM dataset when many-to-one mappings are included. RSM-full improves its gains in this scenario from its ability to use the UI to detect which fields are likely to map to multiple fields. This results in gains of 4.5x over Harmony, and 1.6x over RSM-schema-only.

one of its matches.

Results. We can see from Figure 5 that in this scenario RSM-full correctly matches over 80% of the fields, resulting in an average gain of 2x over Harmony, and an average gain of 1.6x over RSM-schema-only. When we break this down by database pair in Figure 6, we can see that on a per pair basis, RSM-full's gains over Harmony vary between 1.6x and 3.4x. This wide variation in the gain comes largely from differences in the quality of the underlying schema names. For example, both Sugar and OpenCRX have relatively clean schemas with most of their table names containing full words with underscores between them, e.g., "first_name". In such cases, the use of UI features provides less benefit. However, when even just one of the two schemas has low quality names, the UI can provide significant benefit. This can be seen by the more than 3x F-measure gain achieved when testing on Sugar and Centric, since the Centric schema tends to use short and non-descriptive column names.

Finally, in one-to-one mode, RSM has the same precision and recall since in this case there is no duplication and each source field is mapped to exactly one field in the target database.

5.2 NULLs

As we discussed in §2.4 we find that many fields in the source database do not map to anything in the target database. We call such fields *NULL fields*. In this section we evaluate RSM's performance on these NULL fields.

Method. To test the effect of NULL matches we augment the one-to-one CRM dataset used in the previous section with all database fields that match to NULL.

Results. We can see from Figure 7 that when we include NULL fields, RSM-full's gains over both of the baselines increases. As discussed in §2.4, this increased gain results from the fact that about half of the NULL fields come from fields which are simply unused in the source database. These NULL fields may be vestigial, reserved for future use, or simply used only for debugging. These fields are difficult for traditional techniques to handle because without information from the UI, it is quite hard to detect which fields are unused. RSM, however, is able to use the UI to identify the fields that can be set via the user interface, giving it an indication of used vs. unused fields, and improving matching performance.

5.3 One-to-Many

In addition to NULL mappings, practical scenarios also have many-to-one mappings where a single field in the source database may map to multiple fields in the target database. This section evaluates RSM's performance on such one-to-many mappings.

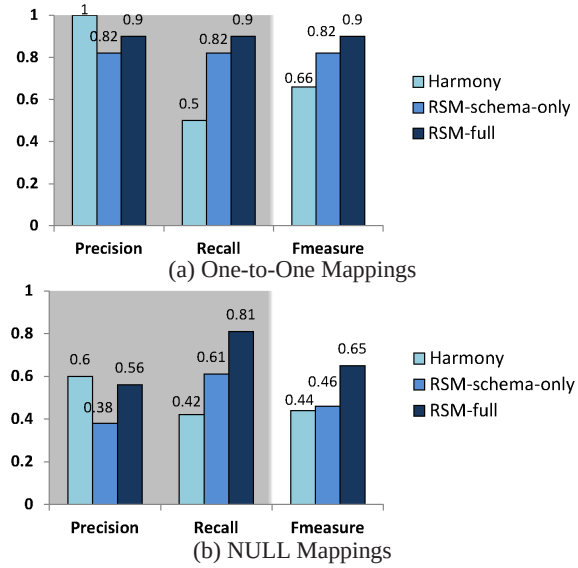


Figure 9: Publication Domain. This shows the average results for testing on the basic one-to-one Publication dataset (a), as well as the Publication dataset with NULLs (b). These results were generated by training RSM on database pairs from the CRM dataset. We can see that RSM works well even if trained on data from a different domain as RSM-full still achieves gains of 1.3x over Harmony and 1.1x over RSM-schema-only.

Method. To test the effect of one-to-many mappings we augment our original one-to-one dataset by including *all* matches for fields mapping one-to-many.

Results. As we can see from Figure 8, RSM-full again has higher gains in this scenario than in the basic one-to-one scenario. This increased gain results from the fact that without the use of the UI, there is no good indication of which source database fields should be mapped to multiple target database fields. As we discussed in §2.3, however, the UI provides a very strong indication of which source fields may map to multiple target fields.

5.4 Publication Domain

So far all of the results that we have discussed come from the CRM domain. In this section, we show that RSM's improvements also apply in other domains. Furthermore, we show that RSM need

not be trained on a database pair from the same domain, rather a database pair from any rich UI domain is sufficient.

Method. We train RSM on a random database pair from the CRM domain just as before, but we test it on the dataset from the paper publication domain discussed in §4. We repeat for a different choice of the CRM training pair and report the average results. We test two scenarios: without NULLs where we remove all NULL mappings, and with NULLs where we include the NULL mappings. Note that our publications dataset did not have any many-to-one mappings so we did not test this scenario.

Results. We can see from Figure 9 that RSM provides substantial gains in this scenario as well, confirming that its gains do generalize to other domains, and its training does not need to be domain specific (as long as it is trained on a UI rich application like CRM). To further ensure that RSM does not need to be trained on database pairs from the same domain, we also tested RSM on the publication dataset when it was trained on database pairs from this same dataset. We found the results to be exactly the same as those in Figure 9, confirming that training on the same domain does not improve the results.

We do note, however, that it is important for RSM to be trained a rich UI domain such as the CRM domain. Specifically it would not be appropriate to train on the publications domain, which has only a very simple UI, and then test on the CRM domain, which has a much richer UI. This is because when RSM is trained on a simple UI domain, any feature not exhibited by that domain would get a weight of zero. This prevents RSM's matching algorithm from utilizing these features, even when testing on a rich UI domain in which those features are present. In contrast, if RSM is trained on a rich UI domain and it puts weight on features that do not appear in the test domain (because it has a simpler UI), then the weights on these unobserved features will simply be ignored, and thus cannot have a negative impact on the resulting matching.

5.5 Computation Time

In this section, we show that RSM's runtime is reasonable.

Method. We run RSM-full on the full dataset for each database pair, and we time how long it takes during the training phase, as well as how long it takes for the optimization to complete. We also measure the runtime of Harmony on the same dataset.

Results. We find that in all cases RSM-full takes less than 1 minute to solve the optimization for a single database pair, while Harmony's runtime is approximately 30 seconds. Additionally, training RSM takes less than 30 seconds in all cases. Further, we note that training can easily be performed offline thus not affecting RSM's runtime in practice. Lastly, our current implementation prioritizes flexibility over performance, so a more performance focused implementation could further reduce these runtimes.

6. RELATED WORK

There is a rich literature on schema matching, which falls into three areas:

(a) Automatic Matching Paradigms: RSM differentiates itself from past work primarily based on the source of information used for matching. Past work on automatic schema matching has utilized three sources of information for matching: the schema, the data instances in the database, and usage statistics [19, 29]. Schema-based paradigms rely primarily on the structure of the schema or ontology and the properties of its attributes [26, 27, 25]. Instance-based techniques, on the other hand, observe that existing database entries (instances) may convey additional information about the columns or fields they belong to and thus may be used to aid the matching process [36, 18]. Last, usage-based techniques exploit access patterns extracted from query logs to improve the results of schema matching [19, 29]. In contrast to all of this past work, RSM utilizes information mined directly from the user interface of an overlying application. This represents a fourth source of information that has not been utilized by past work.

In addition to this new source of information, RSM also differs from past work in the algorithm it uses for matching. The most similar past work has used machine learning to weight the importance of various sources of match information, just as RSM does [21, 18, 16, 25, 15]. Other related techniques have employed iterative algorithms to propagate similarity measures between neighboring fields [18, 27]. While the algorithm used in RSMis similar in flavor to those in prior work, it differs in that it models the matching problem as a quadratic program, which integrates both local and global relational features. It then solves this quadratic program to produce an optimal match.

(b) Large-Scale Schema Matching: One of the major challenges in schema matching is developing matching techniques that can scale to real-world applications [17, 32]. Much of the prior research in this direction has focused on speeding up the matching process [31, 10, 18]. In contrast, RSM is focused on improving the matching accuracy for large-scale schemas. Furthermore, most prior work on large matching problems is most effective for hierarchically structured schemas such as XML documents and ontologies, and is less effective on non-hierarchical relational schemas. For instance, COMA++ [9] partitions source and target schemas into similar fragments and then matches elements in corresponding fragments. While such partitioning into mutually exclusive fragments works well for hierarchical schemas, the lack of structural elements (beyond tables) in relational schemas renders such techniques less effective in conventional RDBMSs.

(c) User Interface and User Interaction: Past research has leveraged the user interface in schema matching problems for two purposes: matching web forms and integrating user feedback. In web form matching, the goal is to simultaneously perform a joint matching of the widgets on a large number of web forms [20, 21, 34, 13]. This work is based on the heuristic that field names which appear together on one web form are less likely to match to each other when appearing in different web forms. The effectiveness of this technique, however, is dependent on the availability of a sufficiently large number of databases (20-70) which each have a relatively small number of attributes (3-7). Thus, even the authors themselves [34] assert that the employed techniques are ineffective for matching large-scale relational schemas where there is relatively little training data, and hundreds of fields to match.

The other use of the UI in past work is to provide a GUI interface to allow users to inspect and correct schema matchings [9, 28, 33, 22] or to involve them in the matching process by providing hints for the matcher [10]. This work is complementary with ours in that RSM is focused on improving the *automated* match quality. Specifically, RSM can be utilized by these GUI tools to provide a high quality initial match which will significantly reduce the effort required by the user.

7. CONCLUSION

Automatic schema matching has made giant leaps over the past fifteen years; however, fully automating the matching process remains elusive for industry-size relational databases. In this paper, we have proposed RSM, which provides an additional dimension for improving this matching process by leveraging application information. We have shown that RSM achieves F-measures as high

as 0.82 for one-to-one matchings on databases with hundreds of columns, and gains of 2-4x over Harmony, an existing state-of-the-art approach. Furthermore, RSM develops methods to specifically tackle many-to-one and null mappings, again through utilizing application information. We have also shown that RSM is generic and capable of achieving high matching accuracies on one domain when given only training data from a different domain. In the future, we would like to develop RSM into a system that is capable of stipulating human feedback on specific matches through a user interface, and of using this feedback to ramp up its matching accuracy.

8. REFERENCES

[1] http://www.sugarcrm.com/crm/.
[2] http://www.concursive.com/.
[3] http://www.opentaps.org/.
[4] http://www.opencrx.org/.
[5] http://tartarus.org/martin/PorterStemmer/.
[6] http://svmlight.joachims.org.
[7] http://numpy.scipy.org.
[8] http://www.refbase.net/.
[9] D. Aumueller, H.-H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with coma++. In *SIGMOD*, 2005.
[10] P. A. Bernstein, S. Melnik, and J. E. Churchill. Incremental schema matching. In *VLDB*, 2006.
[11] C. M. Bishop. Pattern recognition and matchine learning, 2007.
[12] R. E. Burkard. Quadratic assignment problems. In *European J. Operational Research*, 1984.
[13] K. C.-C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web, 2004.
[14] N. W. P. A. A. F. Chenjuan Guo, Cornelia Hedeler.
[15] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. imap: discovering complex semantic matches between database schemas. In *ACM SIGMOD*, 2004.
[16] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, 2001.
[17] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 2005.
[18] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology matching: A machine learning approach. In *Handbook on Ontologies in Information Systems*, 2003.
[19] H. Elmeleegy, M. Ouzzani, and A. Elmagarmid. Usage-based schema matching. In *SIGMOD*, 2011.
[20] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *SIGMOD*, 2003.
[21] B. He, K. C.-C. Chang, and J. Han. Discovering complex matchings across web query interfaces: A correlation mining approach, 2004.
[22] M. A. Hernandez, R. J. Miller, and L. M. Haas. Clio: a semi-automatic tool for schema mapping.
[23] IBM. Ibm infosphere data architect.
[24] H. W. Kuhn. The hungarian method for the assignment problem, 1955.
[25] J. Madhavan, P. Bernstein, K. Chen, A. Halevy, and P. Shenoy. Corpus-based schema matching. In *In ICDE*, pages 57–68, 2003.
[26] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with cupid. In *In The VLDB Journal*, 2001.
[27] S. Melnik, H. Garcia-molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. 2002.
[28] P. Mork, L. Seligman, A. Rosenthal, J. Korb, and C. Wolf. Journal on data semantics xi. chapter The Harmony Integration Workbench. Springer-Verlag, 2008.
[29] A. Nandi and P. A. Bernstein. Hamster: Using search clicklogs for schema and taxonomy matching.
[30] K. Nigam. Using maximum entropy for text classification. In *Workshop on Machine Learning for Information Filtering*, 1999.
[31] E. Rahm, H.-H. Do, and S. Massmann. Matching large xml schemas. *SIGMOD Rec.*, 2004.
[32] A. Rosenthal, L. Seligman, and S. Renner. From semantic integration to semantics management: case studies and a way forward. *SIGMOD Rec.*, 33:44–50, December 2004.
[33] L. Seligman, P. Mork, A. Halevy, K. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick. Openii: an open source information integration toolkit, 2010.
[34] W. Su, J. Wang, and F. Lochovsky. Holistic schema matching for web query interface. In *EDBT 2006*.
[35] I. SugarCRM. Sugar community edition schema diagrams version 5.2, 2009.
[36] J. Wang, J.-R. Wen, F. Lochovsky, and W.-Y. Ma. Instance-based schema matching for web databases by domain-specific query probing. In *VLDB*, 2004.
[37] N. Yuhanna. Forrester report: Simpler database migrations have arrived, 2011.

APPENDIX

Schema-based	
Local	Number of matching words in column name
	Do column names have a matching word which is unique in both databases
	Number of matching words in column name, tf-idf weighted
	Number of matching words in table name
	Number of matching words between column name and table name
	Same Schema Type
	Both Foreign Keys
Relational	Same Table
UI-based	
Local	Number of matching words in UI label
	Do UI labels have unique matching word
	Number of matching words in UI label, tf-idf weighted
	Do widget types match
	Separate binary feature for each pair of widget types
	Does column have associated widget
	Both have value set when row is added
	Both have value only updated to current Date/Time
	Both have value updated whenever row is updated
	Both have update affected by *current user* context
Relational	Widgets are on the Same Page
	Widgets are in the Same Section

Table 6: Features. The full list of features used in our implementation of RSM˙