Putting Context into Schema Matching

Philip Bohannon
Bell Labs, Lucent Technologies
bohannon@research.bell-labs.com

Eiman Elnahrawy *
Rutgers University
eiman@cs.rutgers.edu

Wenfei Fan †
University of Edinburgh & Bell Labs
wenfei@research.bell-labs.com

Michael Flaster
Bell Labs, Lucent Technologies
mflaster@research.bell-labs.com

ABSTRACT

Attribute-level schema matching has proven to be an important first step in developing mappings for data exchange, integration, restructuring and schema evolution. In this paper we investigate *contextual schema matching*, in which selection conditions are associated with matches by the schema matching process in order to improve overall match quality. We define a general space of matching techniques, and within this framework we identify a variety of novel, concrete algorithms for contextual schema matching. Furthermore, we show how common schema *mapping* techniques can be generalized to take more effective advantage of contextual matches, enabling automatic construction of mappings across certain forms of schema heterogeneity. An experimental study examines a wide variety of quality and performance issues. In addition, it demonstrates that contextual schema matching is an effective and practical technique to further automate the definition of complex data transformations.

1. INTRODUCTION

A *schema mapping* is a data transformation that, given an instance of a source schema, will produce an instance that conforms to a target schema while preserving the appropriate information content of the source. Finding schema mappings is a common task in a wide variety of data exchange and integration scenarios. A *schema matching* is a pairing of attributes (or groups of attributes) from the source schema and attributes of the target schema such that pairs are likely to be semantically related. In many systems finding such a schema matching is an early step in building a schema mapping. Even with some availability of domain expertise, the computation of a schema matching may not be easy since the task itself may be large, involving dozens of tables and thousands of attributes. The combined effort of understanding an unfamiliar schema and matching it to another is a substantial burden.

As a result, automated support for schema matching has received a great deal of attention in the research community (see [29] for a recent survey). In state-of-the-art schema matching systems, schema matches are discovered by considering a wide variety of evidence that may indicate a match, including similarity of data, similarity of schema and metadata information, preservation of constraints, and transitive similarity based on other known mappings (see [11, 20] for example). Once verified by the user, matches discovered by the schema matching process constitute a key input to the creation of schema mappings. In particular, the matches form the basis of constraints that should be upheld by a mapping – a valid mapping from source to target instances ensures that these constraints are enforced (see, for example, [24, 28]).

While schema matching, as described, may be challenging in itself, there are many cases where such matchings fail to capture information critical to the construction of a schema mapping. We illustrate this with an example.

Example 1.1: Consider the problem of finding a mapping between schemas $\mathcal{R}_S$ and $\mathcal{R}_T$ for the retail inventory tables shown in Figure 1. In the source table, $\mathcal{R}_S$.**inv**, information about books and CDs being sold by "Company S" is provided, and a type field indicates whether the object is a book or music. In the target schema, for "Company T", information about books and music are stored in separate tables.

$\mathcal{R}_S$.**inv**

id	name	type	instock	code	descr
0	leaves of grass	1	Y	0195128	hardcover
1	the white album	2	Y	B002UAX	audio cd
2	heart of darkness	1	N	0486611	paperback
3	wasteland	1	Y	0393995	paperback
4	hotel california	2	N	B002GVO	elektra cd

(a) Instance of source inventory table

$\mathcal{R}_T$.**book**

id	title	isbn	price	format
50	the historian	0316011770	15.57	hardcover
51	lance armstrong's war	0486400611	15.95	hardcover
52	• • •			

(b) Instance of target book table

$\mathcal{R}_T$.**music**

id	title	asin	price	sale	label
80	x&y	B0006L16N8	13.29	12.50	capitol
81	moonlight	B0009PLM4Y	13.49	9.99	sony
82	• • •				

(c) Instance of target music table

Figure 1: Example source and target instances

*Work performed in part while the author was an employee of Bell Labs.

†Supported in part by EPSRC GR/S63205/01, GR/T27433/01 and BBSRC BB/D006473/1.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

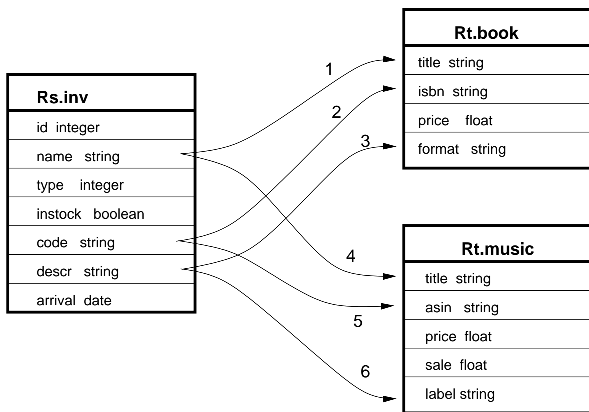

Figure 2: A traditional schema match for inv, books and music.

A traditional schema matching system might give (some subset of) the matches (numbered 1-6) between $\mathcal{R}_S$ and $\mathcal{R}_T$ shown in Figure 2. While this set of matches can form the basis of a schema mapping, it is ambiguous and clearly does not help the user discover the semantic distinction between the two target tables. □

In this paper, we introduce the notion of a *contextual schema match*, in which each match is annotated with a logical condition providing the context in which the match should apply. In this example, matches 1-3 might be annotated with the condition type= 1, while 4-6 should hold where type= 2. Equivalently, one might think of views being introduced into the source or target schema to reflect a common context for several attribute matches, as shown in Figure 3.

Whenever one or more inheritance relationships are implicit in data, a database designer must choose, based on application needs, between placing the sub-types in separate tables or in a common table. This is a common-form of schema heterogeneity [29]. Example 1.1 shows one case of this since "books" and "CDs" are sub-types of "inventory items", but other examples abound: in one school's database both faculty and teaching assistants may appear in a single employee table, while in another school's database separate tables may be used, and similarly for conference and journal papers in a bibliography, apartments and houses in a real-estate database, etc. Clearly, contextual schema matches directly eases the task of overcoming this form of schema heterogeneity.

Another important case where contextual matches are needed is that of "attribute normalization" in which separate rows of one table correspond to different attributes in the same row of another table, as illustrated below.

Example 1.2: Consider supplementing $\mathcal{R}_S$ shown in Figure 1 with the $\mathcal{R}_S$.**price** table appearing in Figure 4. A standard schema matching tool might find only the matching : ($\mathcal{R}_S$.**price**.price $\rightarrow$ $\mathcal{R}_T$.**music**.price). However, a contextual match is more helpful, in which this match is conditioned on ($\mathcal{R}_S$.**price**.prcode = "reg"). Ideally, a second match ($\mathcal{R}_S$.**price**.price $\rightarrow$ $\mathcal{R}_T$.**music**.sale) would be discovered based on the context ($\mathcal{R}_S$.**price**.prcode = "sale"). □

From these examples, it is clear that semantically correct conditions associated with matches increase the value of those matches to the user. The additional information helps the user construct a semantically correct mapping between R_S and R_T.

Contributions. The identification of contextual matching as an important extension of schema matching is our first contribution. The second contribution is to define a general framework for finding good contextual matches. A salient feature of this framework is that it treats schema matching largely as a black box, and thus can

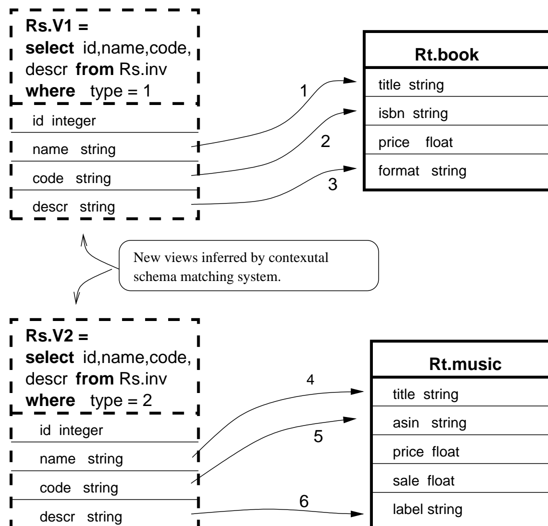

Figure 3: A contextual schema match for inv, books and music.

$\mathcal{R}_S$.**Price**

id	prcode	price
0	reg	14.95
1	reg	27.99
1	sale	24.99
2	reg	8.95
2	sale	8.45
3	reg	11.40
4	sale	12.25
4	reg	14.95

Figure 4: Price Table

be used with any (instance-based) schema matching technique.

Our third contribution is the development of techniques for identifying good candidate conditions. One technique, TgtClassInfer, uses a classifier built on target attribute values while a second, SrcClassInfer depends only on *internal* features of a source table to identify promising conditions by rating some attributes on their ability to *classify* values of other attributes.

Our fourth contribution is the definition of *filtering criteria* for contextual match conditions. These filtering criteria are important since there are many *possible* contextual matches, and it is critical that, in addition to finding semantically correct contextual matches, a contextual matching algorithm does not confuse the user with too many false positives.

We next consider what happens when contextual matches are presented to an automated mapping generation algorithm like that of Clio [28, 14], in which the presence of contextual matches equates to the presence of *views* on the source or target table. In many cases, these views will be handled correctly by a Clio-style algorithm, but in general the joins necessary to construct the correct mapping for attribute normalization cannot be found with the standard rules.

Our fifth contribution is to extend the definition of the key-foreign key relationship so that *contextual key-foreign key* constraints are well-defined between views and either other views or base-tables. We also introduce rules for inferring some of these constraints based on the nature of the view.

Our sixth contribution is the definition of new join rules that, when added to Clio, allow automatic generation of a variety of mappings involving attribute normalization. We note that this extended framework would also allow Clio to better cope with a standard schema match in which views were intentionally included by

the user rather than inferred by a contextual schema matcher.

Our seventh and final contribution is an experimental study of contextual schema matching. Our study addresses both the quality of matches found and the performance of the different algorithms. We find that TgtClassInfer is somewhat more accurate than SrcClassInfer, but that for scaling to large schemas with dozens of attributes, SrcClassInfer performs better. We also investigate the ability of contextual schema matching to accomplish attribute normalization.

Some recent work has considered overcoming a variety of forms of schema heterogeneity automatically [13], but takes a different search approach that requires common data instances to be represented in both schemas. In contrast, contextual schema matching is a direct extension of existing schema match technology. While contextual schema matching is a natural idea, we are not aware of previous work addressing this topic. Much recent work has focused on matching in different data models, improving match quality by considering more fully the available information (e.g., [11, 20, 8, 16]) and maintaining mappings over time [22, 31]. In contrast, this work, like the inference of multi-attribute matches [29], expands the scope of schema matching in a way that allows greater automation of schema mapping than existing techniques.

Organization. In Section 2, we introduce our data model and review our standard (non-contextual) matching system. In Section 3, we introduce contextual schema matching and develop a design space of matching algorithms. In Section 4, we extend Clio-style schema mapping to better handle the conditions output by a contextual match. In Section 5, we present our experimental study on algorithms for contextual schema matching. In Section 6, we discuss the relationship to related work, and in Section 7 we conclude and discuss future work.

2. BACKGROUND

In this section we first present our data model and basic terminology used in the paper. We then briefly review the infrastructure for standard schema matching, which our algorithms extend.

2.1 Data and Match Model

We focus on the main idea of contextual schema matching, and illustrate it in the relational model, leaving other models, as well as inter-model matching, for future work.

Data Model. In our notation, a schema (for example, $\mathcal{R}_S$ or $\mathcal{R}_T$) is a collection of tables and views ranged over by the capital letter R (e.g. R_S, R_T). A table or view R has a set of attributes, $\mathsf{att}(R)$, represented by lower case letters a,s,t,l or h, or with the font AttributeName. An attribute has a type, type(a), drawn from (string, int, real, etc). Views considered in this paper are single-table selection views of the form $V_c =$ "select * from R where c", with selection condition c. Given an instance of R (a sample input), we use $v(R.a)$ to denote the bag of values associated with attribute a in the instance, a shorthand for "select a from R".

Matches. A *match* is a triple $(R_S.s, R_T.t, c)$, where $R_S \in \mathcal{R}_S$, $R_T \in \mathcal{R}_T$, and c is a Boolean condition. Intuitively, it says that the pairing between attributes $R_S.s, R_T.t$ makes sense if the condition c holds. A match is referred to as a *standard match* if c is a constant expression "true" and R_S and R_T are base tables; otherwise it is a *context match*. Thus a standard match is a special case of context match. Some prior work considers "complex matches" [29] in which multiple attributes are involved in a single match, but these are largely orthogonal to the issues in this paper, and we omit consideration of such matches to simplify our presentation.

A list of "accepted" matches is referred to as $\mathcal{L}$; usually these are matches with a sufficiently high confidence. Note that $\mathcal{L}$ is an extension of what is referred to as a *value correspondences* in the literature [14, 24, 28] to include matches to and from views.

Categorical Attributes. Informally, an attribute a is categorical if many of the values it takes are (intended to be) associated with several tuples in a full data set. In some cases, categorical attributes may be indicated by the schema, for example by a constraint limiting the value to a fixed set of choices. However, in the absence of such information and if only a sample of the data is available, the intent of attributes may be hard to discern. In this case, we consider an attribute, a, to be categorical if more than 10% of the values of a are associated with more than 1% of the tuples in our sample. In the case of small samples, at least two values must be associated with at least two tuples.

2.2 Context Complexity

The complexity of the conditions that can be associated with each view has a large effect on the difficulty of finding contextual matches. In developing matching schemes, it is useful to classify the complexity of the contextual match by the number of attributes mentioned in its condition: a k-condition on relation R mentions exactly k attributes of R. A "simple" where condition is of the form $a = v$ where a is an attribute and v is a constant, and is thus a 1-condition. Disjunctive, conjunctive, and general k-conditions generalize simple conditions in the usual way. The term "simple, disjunctive" refers to disjunctive 1-conditions, and can be written as $a \in \{v_1, v_2, \ldots, v_k\}$.

2.3 Standard Schema Matching

We now give an overview of our standard schema matching process. In fact, the approach we describe for contextual schema matching is independent of the standard matching technology used, but a full understanding of our algorithms' performance requires reference to the base matching algorithm's behavior. Like the systems described in, for example, [20, 11, 8], our base schema matching system employs a variety of matching algorithms, referred to as *matchers*, to compute similarity scores between a pair of attributes. These scores are weighted [8] and for a single matcher m and source attribute a, the distribution of scores to all target attributes are treated as samples of a normal distribution, allowing the raw scores given by m for a to be converted into *confidence scores* using standard statistical techniques. For a particular pair of attributes a and b, the confidences of all matchers are combined to compute the confidence of the match.

3. FINDING CONTEXTUAL MATCHES

In this section we describe issues that arise with determining contextual matches, as well as a space of approaches to the problem. For simplicity, we consider *source* contextual matches in the remainder of the paper, although it is generally straightforward to reverse the role of source and target tables to discover matches involving conditions on the target table.

A Strawman Approach. We introduce a number of issues that arise in contextual matching in terms of the following "strawman" approach to the problem: Consider a source relation R_S and a target relation R_T, and assume that a run of a standard match algorithm returned $\mathcal{L} = [M_1, \ldots, M_n]$ as the accepted matches between these relations. Let the average matcher score for M_i be s_i, and the confidence assigned to that score be f_i. For some match $M_i = (R_S.s, R_T.t, \mathsf{true})$ and some condition c, we want to know whether or not $(R_S.s, R_T.t, c)$ will improve M_i. To determine the

impact of adding the condition c, we restrict the sample of R_S based on c, and re-evaluate the quality of the match between this modified sample and $R_T.t$. Let f_c be the confidence of this new match, which can be estimated using the new score s'_i and the distribution of scores seen for $R_S.s$ across the sample. We refer to $\delta_c = f_c - f_i$ as the *improvement* in M_i for condition c. Now, given a space C of conditions on R_S (e.g. the simple equality conditions on categorical attributes of R_S), let c_i^+ be the condition, if any, that maximizes δ_c subject to $\delta_c > 0$. If c_i^+ exists, then it is likely that $(R_S.s, R_T.t, c_i^+)$ does better than M_i, and we use $(R_S.s, R_T.t, c_i^+)$ in place of M_i.

Significance. Unfortunately, the strawman approach will almost always find some condition which improves the match confidence, even when the original, unconditioned match is semantically correct but the new match is not. To see why, consider taking random subsets of the sample associated with $R_S.s$, and scoring the match (normalized for the size of the subset) with $R_T.t$. It is clear that there will always be a random subset that yields an above average score (unless all scores are equal). Thus, even if there are no semantically valid contextual matches between R_S and R_T, the strawman is still likely to find meaningless conditions for each match, and the output will be confusing to the user rather than helpful.

False Negatives. A secondary issue with the strawman approach is the potential for *false negatives*, i.e., semantically valid contextual matches that do not correspond to any of the matches $M_1 \ldots M_n$ returned by the original match algorithm. The potential for false negatives was illustrated by the sale attribute in Example 1.2.

Conditions as Views. Our first step to avoid insignificant matches is to set an *improvement threshold* for accepting a condition, so that just being above average is not sufficient. Unfortunately, setting a threshold for a single match condition is problematic. Even for a threshold that will only be crossed by a random attribute pair a small percent of the time, if enough random conditions are tried, there is still a significant chance that some will pass the threshold. On the other hand, it is also possible that a semantically correct condition will *not* pass the threshold, and hence be overlooked.

A better approach is to require that a single condition c be found for the whole table R_S, and count the total improvement across all of the individual matches. Based on the assumption that semantically valid conditions will improve several matches, it is more likely for a valid improvement to surpass a given threshold. For semantically invalid conditions, however, the improvement in different matches is less likely to be correlated, and hence the total improvement should be small.

Since the quality of a condition c needs to be evaluated across multiple attributes, it makes sense to think of conditions at the *table level* rather than at the *attribute level*. Accordingly, it is natural to think of each candidate condition c on table R_S as defining a 'select-only' view, V_c, on R_S. This terminology is convenient, as we can now talk about the quality of the match between V_c and some target table. Since some schemes consider a large number of candidate conditions, it is important to emphasize that views are *not* created in the DBMS storing $\mathcal{R}_S$ or $\mathcal{R}_T$ during the search process.

3.1 Finding Contextual Matches

Our overall approach to finding contextual matches is shown as the algorithm ContextMatch in Figure 5. We now walk through this algorithm and discuss how the issues identified above are handled. Alternative implementations of the functions called by ContextMatch are developed in the remainder of this section.

The goal of the algorithm is to assemble in $\mathcal{M}$ a collection of high quality source contextual matches. To accomplish this, algorithm ContextMatch considers each source table R_S in turn at line 2, and generates a list of prototype matches between R_S and tables in the target schema by taking the output of a standard schema match algorithm, StandardMatch. In this call, the quality threshold imposed by StandardMatch on returned matches is shown as a parameter τ. Note that by reducing τ, we can reduce the risk of false negatives at the cost of greater time spent in ContextMatch.

```
Algorithm ContextMatch (𝓡_S, 𝓡_T)
Input:     Source and target schemas 𝓡_S, 𝓡_T with associated sample data.
Constants: a confidence threshold τ, improvement threshold ω,
           and boolean control parameter, EarlyDisjuncts
Output: A set of contextual matches 𝓜
1.  𝓜 = ∅;
2.  for R_S ∈ 𝓡_S do
3.      RL := ∅; // candidate matches for R_S
4.      M := StandardMatch(R_S, 𝓡_T, τ);
5.      C := InferCandidateViews(R_S, M, EarlyDisjuncts);
6.      for c ∈ C do
7.          V_c := R_S where c;
8.          for m ∈ M s.t. m is from R_S do
9.              m' := m with R_S replaced by V_c;
10.             s := ScoreMatch(m');
11.             RL := RL ∪ {(m', s)};
12.     𝓜 := SelectContextualMatches(𝓜, RL, ω, EarlyDisjuncts);
13. return 𝓜;
```

Figure 5: Algorithm ContextMatch

Next, based on some underlying space of contextual conditions (usually the 1-conditions of the source tables), a set of candidate view conditions, C, is generated by the function InferCandidateViews. We consider a variety of possible implementations for InferCandidateViews in Section 3.2 below. Further, the behavior of InferCandidateViews with regard to disjunctive conditions is controlled by EarlyDisjuncts, as described in Section 3.3. Note that no conditions will be returned if $\mathcal{M}$ is empty.

In lines 8-11, the algorithm evaluates contextual matches based on each of the conditions in C. The routine ScoreMatch evaluates the quality of each match m' between V_c and R_T. This function is provided by a standard instance-based schema matching system, but considers only the subset of sample data for R_S meeting c. Note that a match is scored only if it is a conditional version of one of the matches returned by StandardMatch. Finally, SelectContextualMatches is called to determine the matches (and thus the conditions) to return to the user as the output of the schema matching process. This is described in more detail in Section 3.4.

3.2 Selecting Views to Evaluate

We next describe our condition pruning policy as defined by our implementation of InferCandidateViews. We begin by introducing a naive approach, and then present three novel techniques designed to filter out conditions that are unlikely to improve match scores. The key idea behind all three techniques is that a family of views should be created whenever the data values of some non-categorical attributes are *well-classified* by a categorical attribute.

3.2.1 A Naive Approach

The NaiveInfer approach infers views on a schema as follows: for every table R, and every categorical attribute $l \in \mathsf{Cat}(R)$, a set of views is created. For *simple* context, a view V_i = "select * from R where $l = v_i$" is added to the set of returned views for every value v_i of l in the sample data. If simple-disjunctive views are considered, a set of views is created for every partitioning of the v_i values. This approach may work reasonably well for simple context if there are relatively few categorical attributes.

Algorithm ClusteredViewGen (R, trainingData, testingData)

Input: Mutually exclusive sets of tuples from table R
Output: A set of well-clustered view families

1. for each $h \in \text{NonCat}(R), l \in \text{Cat}(R)$
2. $(C_h, apriori)$:=doTraining (h,l,trainingData);
3. $c := doTesting(\text{testingData}, h, l, C_h)$;
4. $(\mu, \sigma) := C_{\text{Naive}}(apriori)$;
5. if $\Phi(\frac{c-\mu}{\sigma}) > \mathsf{T}$
6. add new ViewFamily(l) to $\mathcal{M}$;
7. return $\mathcal{M}$;

Figure 6: Algorithm ClusteredViewGen

3.2.2 Well-Clustered View Families

We define a *view family* $F = (R, l, \{V_i\})$, as a set of select-only views $\{V_i\}$, based on mutually exclusive boolean conditions over only one attribute, l. A view family F effectively partitions the tuples of a table R into a set of views based on values of a categorical attribute l. For example, in Figure 1(a), a view family on the attribute type would consist of two views, dividing the tuples based on whether type= 1 or type= 2. Intuitively, F is of higher quality if the tuples are *well-clustered*, that is, if for some other attribute h, the $t.h$ values are more similar within a view than between views. In our example, this means that other attributes, like code or descr, are more similar for tuples with the same value for type than for tuples with different values.

We formalize this intuitive quality into a metric based on machine-learning techniques for categorization (see, for example, [30]), and illustrate how it is applied to find good candidate conditions in Algorithm ClusteredViewGen of Figure 6. On line 1, the algorithm considers a non-categorical attribute h, and a categorical attribute l, taking on values $v_1, v_2, \ldots v_\gamma$ in the sample data. The idea of the algorithm is to consider the values taken by h to be "documents" to be classified, the v_i values in our sample data to be classification labels, and the tuples to be the "expert assignment" of labels to documents. Two subsets of the sample tuples of R are then considered, one for *training* and one for *testing*. In doTraining at line 2, a single-label classification function C_h is developed based only on the training data. (The implementation of C_h differs, but one might think of a Naive Bayes classifier on tokens or Q-grams.)

Next, in doTesting, the function C_h is presented with unseen testing data, and the quality of C_h is assessed in a standard way as the combined, micro-averaged, precision and recall of C_h according to the standard F_β function [30, 18] with $\beta = 1$. If the quality is good enough, as described below, we consider the family of views $\{V_i \mid i \in [1, \gamma]\}$ (where V_i is conditioned on $l = v_i$) to be well-clustered, and add it to the return list at line 6.

Score Significance. Precision and recall are traditionally used as a quality metric for the *classifier* C_h, compared to other classifiers, but we instead score the extent to which h is correctly classified by l. We need to determine not only if C_h classifies the testing data correctly, but also if the number of correct classifications are *significant*. Accordingly, we consider the null hypothesis that there is in fact no correlation between h and l, and instead values of l are just chosen randomly, proportional to their frequencies in the training data. We then compare C_h to a naive classifier, C_{Naive}, which always chooses the most common value of l, denoted by v^*, as the label, regardless of h. The number of correct classifications by C_{Naive}under the null hypothesis is a binomial distribution, with $p = \frac{|v^*|}{n_{train}}$. Its expected score μ is $n_{test}p$, and its standard deviation σ is $\sqrt{n_{test}p(1-p)}$ (where n_{train} and n_{test} are the training and testing sizes, respectively). The likelihood of the null hypothesis is therefore $1 - \Phi(\frac{c-\mu}{\sigma})$ (where Φ is the normal Cumulative Distribution Function, CDF). If the inverse of this likelihood is above a threshold T (typically 95%), we accept the alternative hypothesis that l can be predicted by h, and include simple conditions on l as candidate matches.

Algorithm createTargetClassifier ($D, \mathcal{R}_T$)

Input: Domain D ("int", "string", "text", ...)
Output: A classifier C_T^D based on target attributes

1. C_T^D := new classifier over D
2. for each $R_T \in \mathcal{R}_T, a \in \text{att}(R_T)$
3. if type(a) compatible with D
4. for each tuple $t \in R_T$
5. C_T^D.teach($t.a$,"R_T.a");
6. return C_T^D;

Figure 7: Algorithm createTargetClassifier

We next present two ways to define C_h, leading to two algorithms for inferring well-clustered view families: SrcClassInfer, and TgtClassInfer. (A third technique based on clustering was also evaluated, but its performance was similar to SrcClassInfer and we omit it for brevity.)

3.2.3 Inferring Candidate Views with SrcClassInfer

Our first approach is the simplest one naturally suggested by the previous discussion: namely, train a classifier C_h on the values of h in the source table. Specifically, for each tuple t in the training data, C_h is trained on $t.h \rightarrow t.l$. If h is a text attribute, a standard Naive Bayesian classifier is used, with the values tokenized into 3-grams. If h is a numeric attribute, a statistical classifier is used instead.

3.2.4 Inferring Views with TgtClassInfer

Unlike SrcClassInfer, TgtClassInfer attempts to classify elements of h based on information gleaned from the *target* tables. It tags individual data values in $\mathcal{R}_S$ based on attributes of $\mathcal{R}_T$ to which it is most similar. For example, assume it recognizes that some titles in $\mathcal{R}_S$.**inv** are most similar to titles in $\mathcal{R}_T$.**book**. It will tag those tuples with **Book.Title**. Furthermore, assume it recognizes that other elements of the same column are most similar to titles in $\mathcal{R}_T$.**music**. It will tag those tuples with **Music.Title**. doTraining will then attempt to learn an association between these tags and the categorical attributes in l. While we only consider inferring views on source relations, this approach can be used to infer views on target relations by simply reversing the roles of $\mathcal{R}_S$ and $\mathcal{R}_T$.

TgtClassInfer maintains separate classifiers C_T^D for every basic type D (e.g., "int", "string", etc.). These classifiers are created by the procedure createTargetClassifier ($D, \mathcal{R}_T$) (Figure 7), which also trains them on the data in the target schemas. (Note that this is different from the training that occurs in ClusteredViewGen.) These classifiers essentially attempt to guess the column that any given sample should appear in. For example, if the target schema contained the tables **Book** and **Music**, each having only a single attribute **Title**, then a classifer C_T^{String} would be created which, when given a String, would return either **Book.Title** or **Music.Title**.

The next step is to apply these classifiers to the source table. In doTraining in ClusteredViewGen, TBag($R_S.h, R_S.l$) is created by collecting the bag of pairs (g, v) for each tuple t in trainingData, s.t. g is the tag C_T^D.classify($t.h$), and $v = t.l$. In other words, given a tuple t, doTraining attempts to learn an association between $t.l$ and C_T^D.classify($t.h$). For example, if we consider tuple 2 in $\mathcal{R}_S$.**inv** with type= 1 and name= "lance armstrong's war", and C_T^D.classify("lance armstrong's war") returns **Book.Title**, then the pair (**Book.Title**, *2*) will be included in TBag(R_s.name,R_s.type).

In doTesting, TgtClassInfer must return a category v_i for every $t.h$ provided to it. It accomplishes this in the following manner. Given TBag, we define accuracy and precision in the usual way between g and v, by treating each (g,v) pair as an occurrence and computing $\mathsf{acc}(g,v) = P(g|v)$ and $\mathsf{prec}(g,v) = P(v|g)$. We combine these two to produce $\mathsf{score}(g,v) = \mathsf{acc}(g,v) \cdot \mathsf{prec}(g,v)$. We then compute the "best" categorical attribute value, $\mathsf{bestCAT}(g)$, as the value v that maximizes $\mathsf{score}(g,v)$. Ties are broken in favor of the value v that is more common. In the case of a continued tie (or when g was never encountered during training), an arbitrary categorical value is selected for g and returned. Given this infrastructure, during doTesting, the overall TgtClassInfer classifier will return:

$$v_i = \mathsf{bestCAT}(C_T^D.\mathsf{classify}(t.h))$$

3.3 Handling Disjunctive Context

Another dimension to finding matching conditions is the treatment of disjunction. The simplest approach is to exclude disjunctive conditions from condition selection, and then to union together the high-scoring conjunctive views. An alternative approach is to allow conditions with disjunctions in C, and attempt to find the single best condition. This alternative approach is *early disjunction* handling, and is taken when EarlyDisjuncts = true. Because the number of disjunctive conditions grows exponentially in the cardinality of the categorical data, it is critical to prune the disjunctive conditions that are considered.

We now describe our techniques for extending SrcClassInfer and TgtClassInfer for EarlyDisjuncts to infer well-clustered views on *disjunctive* simple conditions; that is, conditions of the form $S.l \in \{v_1, v_2, \ldots, v_k\}$. Our algorithm is a simple extension of ClusteredViewGen based on *errors* in classification, and is not shown. Errors take the form (v, v') where for some tuple t, $t.l = v$, but the classifier returned v' when presented with $t.h$. False positives and false negatives are not distinguished, so (v', v) is grouped together with (v, v'). To build disjunctive conditions, we simply note the pair (v, v') that appears most often as an error during testing (after normalizing for the frequency of v and v'). We then consider *merging* v and v'; that is, we replace all occurrences of either value in $S.l$ with a new token pair (v, v'). We then repeat the training and testing process, and if the new view family containing the view $t.l = v$ or $t.l = v'$ has high quality, it is added to the return list. Regardless of whether a new view family is successfully formed, this process is repeated until either there are no errors during testing, or there are no more categorical values to merge. While it may be worthwhile to consider other techniques, this approach works well and is quite efficient.

3.4 Selecting Contextual Matches

The set of contextual matches $\mathcal{M}$ returned from ContextMatch will likely be large. There may be many matches returned from StandardMatch (depending on the value of τ), and then the number of those matches are then multiplied by the number of views created by InferCandidateViews. In order to avoid overwhelming the user with possible match candidates, we implement SelectContextualMatches to attempt to find a small, coherent subset for presentation to the user.

We consider two techniques for SelectContextualMatches. The simplest technique is to find the single match with highest confidence for every target attribute. Note that this technique will allow a target table to have matches from many different source tables. We call this technique MultiTable.

Instead of selecting the best matches on a per-attribute basis, a second approach is to select the best matches on a *per-table* basis. This technique, called QualTable, considers each target table R_T in turn. It selects the source table R_S which maximizes the total confidences of all matches in $\mathcal{M}$ that are between R_S and R_T. It then considers each candidate view V_c of R_S. If the total confidences in the matches between V_c and R_T improves the base confidence between R_S and R_T by at least an *improvement threshold* ω, then V_c is used instead of R_S. If multiple candidate views improve the base confidence by at least ω, then the set of views selected is based on EarlyDisjuncts. Selecting multiple candidate views is analogous to disjuncting over those views. If EarlyDisjuncts = true, disjunctive conditions are allowed in the view, and so only the single best candidate view is selected. If EarlyDisjuncts = false (also referred to as LateDisjuncts), then all candidate views that exceed ω are selected. (Note that while ω affects both EarlyDisjuncts and LateDisjuncts, it has a much larger role for LateDisjuncts.) Finally, the matches between the selected views and the target tables are returned.

Strawman Revisited. We note that the strawman approach to contextual matching described previously can be obtained in this framework by using NaiveInfer for InferCandidateViews, and MultiTable for SelectContextualMatches.

3.5 Handling Conjunctive Conditions

The algorithms so far have handled simple conditions and disjunctive conditions. However, conjunctive conditions might also be involved in contextual matches. For example, the **Books** target table might instead be semantically **Non-fiction-Books**, and the correct match from the inventory table might require a condition of the form "type=1 and fiction=0". The algorithms as given so far cannot find this condition. Handling conjunctive conditions is potentially a problem, due to the obvious exponential explosion in the number of conditions that must be considered.

To deal with this, we do not attempt a NaiveInfer that enumerates conjunctive conditions. Rather, we take a heuristic approach that assumes that a high-quality k-condition has at least one high-quality $k-1$-sub-condition. The idea is to run ContextMatch repeatedly. At stage i, the algorithm will have found views involving i-conditions. At stage $i+1$, we restrict InferCandidateViews so that (a) only views created during the i'th run are considered as base tables that might need partitioning, and (b) when generating candidate subsets of a "base table" V_c, only attributes *not in c* are allowed to participate in the partitioning. In the example given above, the correct condition can be found in the second iteration as long as one of the sub-conditions, either "type=1" or "fiction=0" is found in the first iteration. While this algorithm can potentially find k-conditions for any k, we strongly hypothesize that very few, say 2 or 3, iterations will be practically useful.

4. EXTENDING SCHEMA MAPPING FOR VIEWS

In this section, we address the challenges to standard schema mapping raised by the presence of contextual matches in $\mathcal{L}$, the output of the schema matcher. We treat such matches as introducing select-only *views* into the schema mapping process. To this end we extend the schema mapping approach of Clio [14, 24, 28] to accommodate contextual conditions (views). Since a fundamental concept in Clio is the formation of logical tables based on key-foreign key relationships, the definition of these relationships and rules for building logical tables must be extended to handle views. In Section 4.2, we propose a form of foreign keys to capture semantic connection between views and base tables. We provide inference rules for constraint propagation analysis. In Section 4.3, we pro-

pose new rules for joining semantically related attributes in views and base tables. As a result of these rules, we are able to correctly infer mapping queries involving *attribute normalization*, demonstrating an important synergy between contextual schema matching and the subsequent schema mapping process.

4.1 Standard Schema Mapping (Clio)

Schema mapping is concerned with turning value correspondences (matches) into mapping queries from source tables to target tables. Formally, the schema mapping problem can be stated as follows: Given a collection $\mathcal{L}$ of matches $(R_S.a, R_T.b, \mathsf{conf})$, source schema $\mathcal{R}_S$ and target schema $\mathcal{R}_T$, find a mapping $\mathsf{map} : \mathcal{R}_S \rightarrow \mathcal{R}_T$ such that for any instance I of $\mathcal{R}_S$, $\mathsf{map}(I)$ is an instance of $\mathcal{R}_T$.

In the absence of contextual conditions, the problem of finding schema mapping based on value correspondences has been well studied. In particular, it constitutes one of the key modules of Clio (see, *e.g.,* [14, 24, 28]). Central to generating a schema mapping is how to join tables together based on schema matches found and their semantic associations, such that semantically related attributes can be mapped from source to target in groups to preserve their logical connections. Previous approaches depend on foreign keys to guide the semantic associations (joins).

Although Clio handles a more powerful nested relational model [28], we refer to a restriction of this method to relational data as "the Clio approach", which we now summarize. In a nutshell, a mapping $\mathsf{map}()$ in (relational) Clio is the collection of individual $\mathsf{map}_{(\mathcal{R}_S, R_T)}()$ queries for each target table $R_T \in \mathcal{R}_T$. For a particular R_T, $\mathsf{map}_{(\mathcal{R}_S, R_T)}()$ is constructed as a union of *logical tables* as follows: (a) With respect to a target table R_T and a source table R_S, a logical table $R_{S,T}$ is formed as R_S plus a set of other tables that also have matches to R_T and that are *reachable* from R_S in $\mathcal{R}_S$ using a pattern of joins defined by certain *semantic association* rules. (b) It interprets value correspondences as inter-schema inclusion dependencies from the source to the target. (c) For each logical table $R_{S,T}$ it defines a query $\mathsf{map}_{(R_S, R_T)}()$ that, given any instance tuple t_S of R_S, generates a tuple t_T of R_T by mapping the value of the attributes in R_S to the value of the corresponding attributes in R_T via the related inclusion dependencies. For those attributes in t_T that are not related to any attributes in t_S via the inclusion dependencies (*i.e.,* these values in the target are not represented in the source), Skolem functions are used to generate non-null values based on the known values of t_T mapped from t_S. For those attributes in t_S that are not mapped to any attributes in t_T (*i.e.,* these source values are not represented in the target) their values are omitted. (d) Finally, $\mathsf{map}_{(\mathcal{R}_S, R_T)}()$ is defined as the *union* over *all* logical tables $R_{S,T}$ of source attributes with value correspondences targeting R_T.

Clio adopts two semantic association rules to group attributes together: (a) attributes from the same table are associated with each other; and (b) attributes from different tables are associated using *foreign keys*, *i.e.,* if table R_1 is in a logical table, and if there is a foreign key in a table R_1 referencing another table R_2, then R_2 can be added with an an *outer-join* between the foreign key and the key is conducted to group attributes in R_1 and R_2 together. The constraints are either declared in the definition of the schema, or discovered using constraint mining tools. In order to identify all meaningful joins, the closure of these constraints is computed using an extension [7] of the standard chase method (see, *e.g.,* [2]). While the chasing process may not terminate for the class of constraints involved, the Clio experience [14] verified that it is an effective method in practice.

4.2 Constraint Propagation from Base Tables to Views

While constraints are declared or discovered at the base-table level, to determine joins between views and/or base tables one needs keys and foreign keys on views. This requires reasoning about constraint propagation from base tables to views. We now tackle the following question: how to derive keys and foreign keys on views from their base table counterparts?

We start with a brief review of keys and foreign keys. We then introduce a new form of constraints in response to the contextual conditions in views. Finally, we investigate propagation analysis.

Keys and foreign keys. Consider a relational schema $\mathcal{R}$ and a class Σ of constraints defined on $\mathcal{R}$. Let R_1, R_2 be two tables in $\mathcal{R}$. The constraints in Σ have the following forms.

(a) Key: $\phi = R_1[X] \rightarrow R_1$, where $X \subseteq \mathsf{att}(R_1)$. The key holds on an instance I_1 of R_1 if for any tuples t_1, t_2 in I_1, if $t_1[X] = t_2[X]$, then $t_1 = t_2$, *i.e.,* the X attributes of a tuple t uniquely identify t in I_1.

(b) Foreign key: $\varphi = R_2[Y] \subseteq R_1[X]$, where Y is a list of attributes in $\mathsf{att}(R_2)$, and X is a list of attributes in $\mathsf{att}(R_1)$ and is a key of R_1. The foreign key holds on instances I_1, I_2 of R_1, R_2 if for any tuple t_2 in I_2, there exists a tuple t_1 in I_1 such that $t_2[Y] = t_1[X]$; in other words, the Y attributes of t_2 reference the t_1 tuple in I_1.

Contextual constraints on views. For constraints on views we extend the definitions of keys and foreign keys given above by allowing R_1, R_2 to be either base tables or views, *e.g.,* we allow a foreign key of a view referencing a base table or view. Furthermore, we introduce a notion of contextual foreign keys.

Let V_1 be a view defined on R_1 via a SP query Q_1. A *contextual foreign key* of V is an expression $V_1[Y, a = v] \subseteq R[X, b]$, where (a) Y is a list of attributes in $\mathsf{att}(V_1)$, (b) a is an attribute in R_1 but is not in $\mathsf{att}(V_1)$ (*i.e.,* a is not on the projection list of Q_1), (c) $a = v$ is the selection condition of Q_1, (d) R is either a base table or a view and (e) $[X, b]$ is a key of R. The constraint holds on instances I_1, I of R_1, R if for any tuple t_1 in $Q_1(I_1)$, there exists a tuple t in I such that $t_1[Y] = t[X]$ and $t[b] = v$. In other words, the Y attributes of V_1 augmented with a constant v as the value of a is a foreign key referencing R tuples. As will be seen shortly, contextual constraints are important for semantic association of attributes from views and base tables. No previous work has studied this form of constraints.

Example 4.1: Consider a schema $\mathcal{R}_S$ consisting of base tables:

student(<u>name: string</u>, email: string, address: string),
project(<u>name: string, assignt: int</u>, grade: char, instructor: string),

where keys are underlined. The *project* relation indicates that a student (*name*) gets *grade* for a project *assignt* under the *instructor*. Suppose that our schema matcher finds that *assignt* ranges over $[0, 9]$ and defines views V_i on *project* via SP query Q_i, for $i \in [0, 9]$:

select name, grade from project where assignt = i

then V_i*[name, assignt = i]* $\subseteq$ *project[name, assignt]* is a contextual foreign key of view V_i referencing the base table *project*. Note that *assignt* is not an attribute of the view V_i. □

Constraint propagation. Consider a collection $\mathbf{V}$ of views defined on $\mathcal{R}$ via SP queries $\mathbf{Q}$. We say that a constraint ψ on the views $\mathbf{V}$ is *propagated from* Σ *via* $\mathbf{Q}$, if for any instance I of $\mathcal{R}$, if I satisfies Σ then $\mathbf{Q}(I)$ satisfies ψ.

The *constraint propagation problem* is to determine, given Σ, $\mathcal{R}$ and $\mathbf{Q}$, whether a key or (contextual) foreign key is propagated from Σ via $\mathbf{Q}$. While the problem has been studied for functional and multi-value dependencies (see, *e.g.,* Chapter 10 of [2] and recently [15]), to the best of our knowledge it has not been considered for keys and (contextual) foreign keys of the above forms. Unfortunately, the theorem below tells us that the propagation analysis of keys and foreign keys is beyond reach.

Theorem 4.1: *The key and foreign key propagation problem is undecidable for views defined in terms of* SP *queries.* □

Proof sketch: This is verified by reduction from the implication problem for relational keys and foreign keys, for which the undecidability was established in [12]. □

In light of this negative result we use two methods to find constraints on views propagated from base tables. (a) We employ constraint mining tools on sample data to discover keys and (contextual) foreign keys on views, as Clio does for finding keys and foreign keys on base tables. (b) We propose a set of sound (but by no means complete) inference rules for the propagation analysis.

We next give some example inference rules (due to the space constraint we omit most of the rules). Let V_1 be a view defined on R_1 via a SP query Q_1, and X be attributes in $\mathsf{att}(V_1)$. Then the following tells us that X is a key of V_1 under certain conditions.

Contextual propagation. If $R_1[X,a] \rightarrow R_1$, *i.e.,* $[X,a]$ is a key on R_1 and $a = v$ is the selection condition of Q_1, then $V_1[X] \rightarrow V_1$, *i.e.,* the values of X attributes suffice to uniquely identify a V_1 tuple.

Assume that R is either a view or a base table, and that $R[Y] \rightarrow R$ is a key. Then X is a (contextual) foreign key of V_1 referencing Y of R if one of the following conditions holds.

View-referencing. If $R_1[X] \rightarrow R_1$, $X \subseteq \mathsf{att}(V_1)$, $a \in X$, the selection condition of Q_1 is $a = v_1$ or $\ldots$ or $a = v_n$, and the domain of a is exactly $\{v_1, \ldots, v_n\}$, then $R_1[X] \subseteq V_1[X]$.

Contextual constraint. If $R_1[X,a] \rightarrow R_1$ and the selection condition of Q_1 is $a = v$, then $V_1[X, a = v] \subseteq R_1[X,a]$ is a contextual foreign key of V_1 referencing R_1.

Example 4.2: Recall the schema $\mathcal{R}_S$ from Example 4.1. For each view V_i, V_i*[name]* $\rightarrow V_i$ can be derived from the set of keys on $\mathcal{R}_S$ using the rule *contextual propagation*, and the contextual foreign keys given in Example 4.1 are derived using the rule *contextual constraint*. Furthermore, if *project[name]* $\subseteq$ *student[name]* is a foreign key on $\mathcal{R}_S$, then one can derive V_i*[name]* $\subseteq$ *student[name]* as a foreign key of the view V_i referencing base table *student*, using a rule *FK-propagation* (omitted). □

4.3 Semantic Associations of Attributes

We next study how to group together attributes in different views and/or base tables based on their logical relations. This is important for generating schema mapping queries. To this end we propose new semantic association rules beyond those used in Clio.

Along the same lines as Clio, we associate (a) attributes from the same base table or view, (b) attributes from different tables or views that are related via an outer-join on foreign keys, which are derived by the propagation analysis and mining given above. However, additional association rules need to be considered in order to capture important schema mapping in the presence of contextual conditions in the view definitions, as illustrated by the example below.

Example 4.3: Consider a target relational schema $\mathcal{R}_T$

projs(name: string, assign$_0$: int, grade$_0$: char, ..., assign$_9$: int, grade$_9$: char).

Here the *projs* relation groups, in the same tuple, different *assign*s of the same student by *name*. Suppose that the contextual schema matcher finds a conditioned partition that maps V_i*.name* to *projs.name* and V_i*.grade* to *projs.grade*$_i$, for $i \in [0,9]$, where V_i is the view given in Example 4.1. Intuitively, to map the source data in the views to the target, one needs to group together the ten views by (outer-) joins on the key *name*. However, this grouping cannot be derived by using the two association rules given above since there are no foreign keys involved between those views. □

This motivates us to use the following association rule:

(**join 1**). Suppose that V_1, V_2 are views defined in terms of SP queries Q_1, Q_2 on the *same attributes* of *the same base table R*, *i.e.,* Q_i is select Y from R where $a = v_i$, for $i \in [1,2]$ and $v_1 \neq v_2$. If via propagation analysis we can derive, for $i \in [1,2]$, (a) keys $V_i[X] \rightarrow V_i$ and (b) (contextual) foreign keys $V_i[X, a = v_i] \subseteq R'[Z]$ for some relation R' and $Z \subseteq \mathsf{att}(R')$, then we group attributes of V_1, V_2 together via join between V_1 and V_2 on the key X. In a nutshell, the propagated constraints ensure that it is to associate *different properties* of *the same object*.

Example 4.4: From rule (**join 1**) and the constraint propagation analysis of Example 4.2, we derive a mapping from the joins of the ten views V_i to the target table $\mathcal{R}_T$ as described in Example 4.3. □

For views defined on *different* attributes of the same table, we need the following association rule.

(**join 2**). Suppose that V_1, V_2 are views defined in terms of SP queries Q_1, Q_2 on *different attributes* of *the same base table R*, *i.e.,* Q_i is select Y_i from R where $cond_i$, where Y_1 and Y_2 are not the same set of attributes. If via propagation analysis we can derive (a) keys $V_i[X] \rightarrow V_i$ for $i \in [1,2]$, where X is a subset of both Y_1 and Y_2, (b) (contextual) foreign keys $V_i[X, a = v] \subseteq R'[Z]$ for some relation R' and $Z \subseteq \mathsf{att}(R')$, and moreover, (c) $cond_1$ and $cond_2$ are the same condition $a = v$, then we group attributes of V_1, V_2 together via join between V_1 and V_2 on the key X. Here condition (c) is to avoid associating properties of different objects.

Example 4.5: Recall the schema $\mathcal{R}_S$ of Example 4.1 and consider a different set of views U_i on $\mathcal{R}_S$ for $i \in [0,9]$:

select name, instructor from project where assign = i

The rule (**join 2**) tells us that the join of V_i and U_i on *name* is meaningful. However, it is not logical to join V_i and U_j if $i \neq j$. □

(**join 3**). Suppose that V_1 is a view and R is either a view or a base table. If $V_1[Y, a = v] \subseteq R[X, b]$ is a contextual foreign key, then we can group attributes of V_1 and R together via an outer-join from V_1 to R on the equality between X of V_1 and Y of R, with $b = v$. This is a mild extension of Clio join rules, and it is based on contextual foreign keys instead of foreign keys.

The association rules of Clio are also used in our system to group attributes of views and/or base tables via outer-join on foreign keys. Observe that (**join 1**), (**join 2**) and (**join 3**) are introduced to capture the contextual conditions (views) that are not encountered in Clio. With the inclusion of these rules, the output of contextual schema matching can be smoothly incorporated in Clio to not only find simple contextual mappings, but also to infer sophisticated attribute normalizations.

5. EXPERIMENTAL RESULTS

In this section, we present the results of an experimental study of contextual schema matching. We investigate the impact of the different implementation options and parameters described in Section 3 on the quality and performance of contextual schema matching. We also investigate the ability of this process to find mappings involving attribute normalization in an experiment based on Example 4.3. We conduct our experiments on real-world inventory data scraped from web-sites and on artificially generated data about assignment grades.

Algorithms. We evaluate the space of implementation strategies outlined in Section 3. We evaluate different values of τ and ω, discussed in Sections 3.1 and 3.4, respectively. In other experiments, we use τ = .5 and ω = 5. We consider the three algorithms for *view inference* discussed in Section 3.2, NaiveInfer, SrcClassInfer and TgtClassInfer, and two algorithms

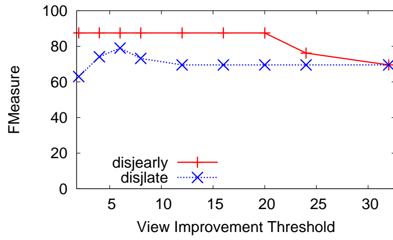

Figure 8: Setting ω for Aaron

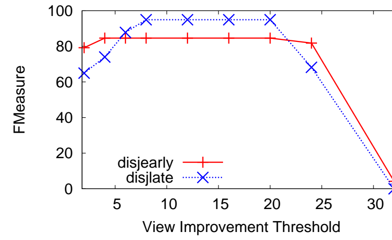

Figure 9: Setting ω for Barrett

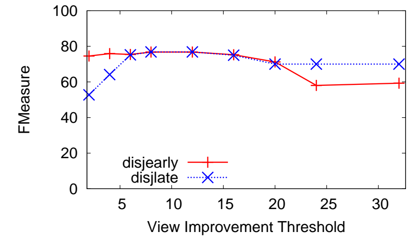

Figure 10: Setting ω for Ryan

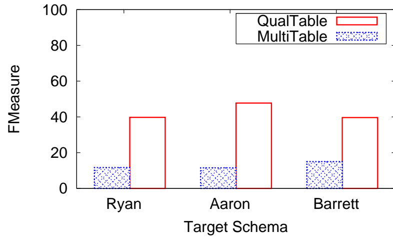

Figure 11: Strawman Performance

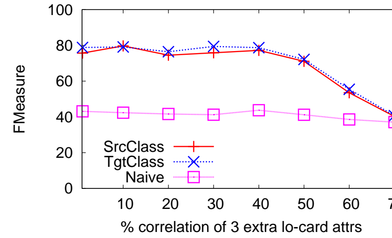

Figure 12: Varying ρ with EarlyDisj

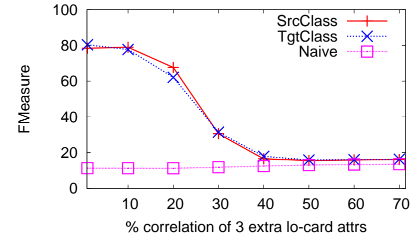

Figure 13: Varying ρ with LateDisj

for SelectContextualMatches: QualTable and MultiTable. Finally, we consider the EarlyDisjuncts and LateDisjuncts policies that control when disjuncts are considered, as discussed in Section 3.3. (Experimental evaluation of conjunctive conditions is left as future work.)

Inventory Data. We use publicly-available example schemas as the basis of our experiments. A variety of schemas for a retail database similar to Example 1.1 are described in [20] and are available at [32]. The schemas were created by students in database courses: the *Colin_Bleckner* schema contains a combined item file with all items and their attributes, while most others put book and music items into separate tables. We formed the Retail data set by using *Colin_Bleckner* as the source, and one of the latter schemas (*Ryan_Eyers*, *Aaron_Day*, or *Barrett_Arney*) as the target schema.

Colin_Bleckner has a single low cardinality attribute, ItemType. It contains data like "book", "music album", "music single", etc, which is indicative of whether the row is a book or a CD. To slightly complicate this schema, we add an additional low cardinality attribute *StockStaus*, ranging over "Low", "Normal" and "High". We replace the relatively small hand-made samples (tens of records) that come with the schemas with data scraped from real-world commercial web-sites, plus some name data taken from the Illinois Semantic Integration Archive [10].

Scraped records for books are given an item type of Book, and similarly for CDs. However, we allow *expansion* of the cardinality of ItemType in order to make the contextual matching problem harder. We use γ to represent the total number of Book and CD labels in ItemType. For example, with $\gamma = 4$, music items are randomly assigned either CD1 or CD2, while book items are randomly assigned Book1 or Book2, and so on for larger (even) values. When not specified, $\gamma = 4$ is used.

Grades data. Our second data set is an artificially generated data set meant to correspond to the test scores of 200 students on 5 exams. In the source schema (*grades_narrow*), there are three columns - name, examNum and grade. In the target schema (*grades_wide*), there is a name column and 5 $grade_i$ columns. In order to map these schemas properly, data values in the examNum column of the source schema must be promoted to attributes in the target schema. The grade data is generated randomly for each schema, so that the mean and standard deviation σ of each exam i is the same in each schema, but the actual scores are not. The mean of exam i is fixed at $40 + 10(i - 1)$, while σ is varied. Clearly, as σ gets larger, the matching task gets more difficult as the number of overlapping values between sets grows. The correct mapping for this data involves creating a view on the source table for every value of examNum.

Evaluating Accuracy. For these data sets, accuracy is evaluated against the correct mappings determined by manual inspection of the source and target schemas and designating some attribute-level matches as correct and the others as incorrect. Only edges originating from views are considered – all others are ignored.

Once the algorithm is run, the set of contextual matches selected is compared to the manually determined correct matches. *Accuracy* is then computed as the percentage of the correct matches found, and *precision* as the percentage of matches found that are correct. *FMeasure* is a single number commonly used for combining accuracy and precision. It is equal to $\frac{2*acc*prec}{acc+prec}$.

Experimental Setting. All algorithms are implemented in Java and run on a 2.8Ghz Pentium 4 processor with 8GB of main memory. All data and summaries fit in main memory. Sufficient experiments are run to ensure that each data point had low variance (between 8 and 200 random partitions of the sample data), and the results are averaged.

5.1 Varying ω under EarlyDisjuncts, LateDisjuncts

In the discussion of SelectContextualMatches in Section 3.4, the *improvement threshold* parameter ω was introduced. In general, for any algorithm and source and target schemas, there is a range of ideal ω values that achieve maximal schema matching performance, which we refer to as ω^+. Figures 8, 9, and 10 show the effect of varying ω on the matching performance, as measured by *FMeasure*, under both EarlyDisjuncts and LateDisjuncts. The plateaus in the graphs correspond to the optimal range ω^+. The shorter flat pieces in the curves for LateDisjuncts clearly demonstrate that the range ω^+ is narrower than EarlyDisjuncts. Hence, LateDisjuncts is more sensitive to ω than EarlyDisjuncts.

5.2 MultiTable vs. QualTable

Next we consider the two different algorithms for SelectContextualMatches. Recall that MultiTable will select the best match regardless of the source, while QualTable

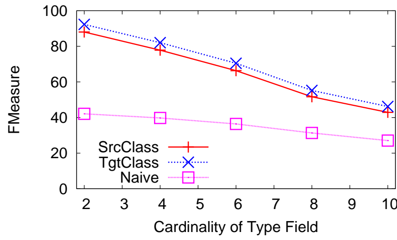

Figure 14: FMeasure of LateDisjuncts

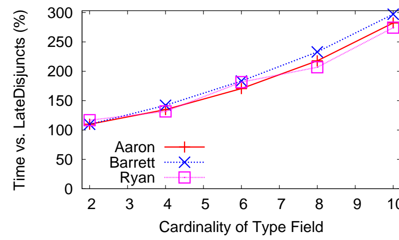

Figure 15: Runtime of EarlyDisjuncts

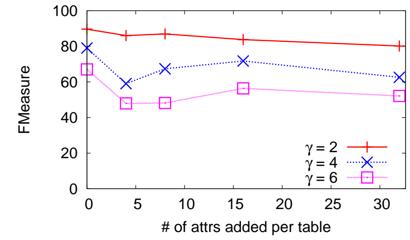

Figure 16: Scaling accuracy

will find the best set of matches coming from a consistent table (or set of views). MultiTable consistently performs significantly worse than QualTable. Figure 11 shows the difference in FMeasure between MultiTable and QualTable, using NaiveInfer for InferCandidateViews. Because of its poor performance, MultiTable is not considered further.

5.3 Adding Correlated Attributes

In these experiments, we modify *Colin_Bleckner* by adding 3 additional low cardinality attributes which have the same domain as ItemType (i.e., CD1, CD2, Books1 and Books2). We vary the correlation ρ between the new attributes and ItemType. For low correlations, these are essentially random categorical attributes. For high correlations, these attributes are chameleons of ItemType, and views based on them will be approximately equivalent to views based on ItemType. But we still consider any matches involving them to be errors. This is analogous to, for example, an OldItemType and NewItemType. While OldItemType and NewItemType are likely to be correlated, the user would consider mappings that used OldItemType to be errors.

As ρ increases, the number of (erroneous) views discovered increases, as expected. These extra views increase the time it takes to do schema matching. But it is interesting to note that when using EarlyDisjuncts, these extra views do not "fool" the schema matcher until ρ becomes very high. In other words, for intermediate values of ρ, many erroneous views are introduced, but the schema matcher with EarlyDisjuncts can still determine that views based on the original ItemType yield better matches, and hence the accuracy of the schema matching is not affected. These results are shown in Figure 12. When using LateDisjuncts, however, FMeasure degrades much more quickly (Figure 13). SrcClassInfer and TgtClassInfer behave similarly in both cases, and both perform significantly better than NaiveInfer.

5.4 Varying Cardinality

Here we experimentally vary the cardinality, γ, of ItemType from 2 to 10. We find that TgtClassInfer has a slightly higher FMeasure than SrcClassInfer throughout the range for both EarlyDisjuncts and LateDisjuncts, while NaiveInfer is significantly worse. When $\gamma = 2$, both EarlyDisjuncts and LateDisjuncts perform the same. As γ increases, while the FMeasure of EarlyDisjuncts remains constant (not shown), the FMeasure of LateDisjuncts degrades. This demonstrates the weakness of relying on ω for determining disjunct size. Figure 14 shows the FMeasure for LateDisjuncts on target *Ryan_Eyers*. The runtime for EarlyDisjuncts increases exponentially as γ increases, however, while the runtime of LateDisjuncts only increases linearly. The runtime of EarlyDisjuncts (relative to the runtime of LateDisjuncts) is shown in Figure 15.

5.5 Varying Schema Size

In these experiments, we vary the size of the schemas by adding n non-categorical attributes to every table. Furthermore, for every table which has a categorical attribute, we add $\frac{n}{4}$ categorical attributes. The extra non-categorical attributes are populated with random data from an unrelated real estate table, while the extra categorical attributes are populated with data from the same domain as the existing categorical attribute.

Figure 16 shows the FMeasure as the schemas get larger for 3 different values of γ, using Ryan_Eyers as the target. For slightly increased schema sizes, mismatches tend to be caused by non-categorical attributes. As the schema size increases further, the non-categorical attributes (all drawn from the same domain) tend to match with each other, reducing that type of error. With more categorical attributes introduced, however, there are more frequent errors involving incorrect candidate views. For a given schema size, as γ increases, the number of tuples in each candidate view decreases, making it more likely that a random candidate view will look appealing. We hypothesize that more sample data will allow the schema matcher to better maintain its accuracy as extra attributes are added. As shown in Figure 17, TgtClassInfer runs much slower than SrcClassInfer as the size of the schema increases. TgtClassInfer is slightly more accurate than SrcClassInfer, and both are significantly superior to NaiveInfer (not shown).

5.6 Varying Sample Size

For these experiments, we vary the size of the Retail table. We find that SrcClassInfer and TgtClassInfer perform similarly. When there are fewer tuples in the source Inventory table, it is less likely that InferCandidateViews will find the correct candidate views. As the size grows, however, accuracy increases. The performance of TgtClassInfer is shown in Figure 18.

5.7 Grades results

We now describe the result of attribute-normalization experiments using the Grades data set. For these experiments, we implement ClioQualTable, which modifies QualTable to include the join rules discussed in Section 4.3. Keys are inferred based on sample data. Because the names are unique within each of the Grade candidate views, the views can be joined together in a logical manner on Name via rule (**join 1**). The accuracies of the different algorithms for data sets with varying σ are shown in Figure 19. For low σ, both SrcClassInfer and TgtClassInfer with ClioQualTable generate a very accurate set of matches. As σ increases, accuracy decreases, as expected.

Despite the fact that NaiveInfer always presents more candidate views to the schema matcher than do the other algorithms, for many values of σ its accuracy is worse. Because the TgtClassInfer and SrcClassInfer essentially provide another filter to remove unnecessary views (generating approximately 25% of the possible views), they outperform NaiveInfer for a large range of σ. For high values of σ, however, SrcClassInfer and TgtClassInfer do not infer the correct candidate views consistently, and they are outperformed by NaiveInfer.

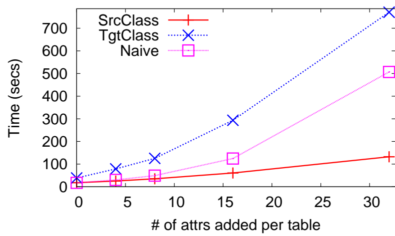

Figure 17: Scaling time

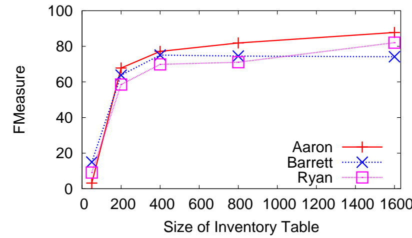

Figure 18: TgtClassInfer**, varying size**

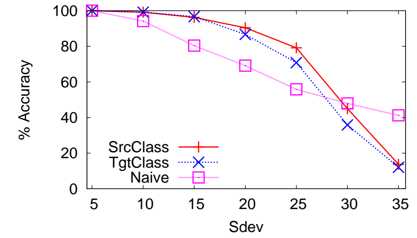

Figure 19: Grades **Accuracy**

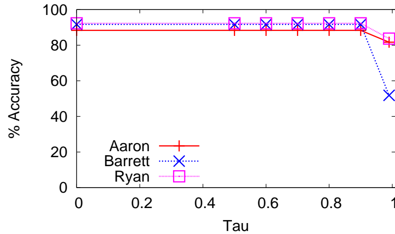

Figure 20: Inventory **sensitivity to** τ

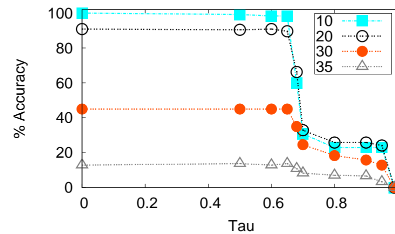

Figure 21: Grades **sensitivity to** τ

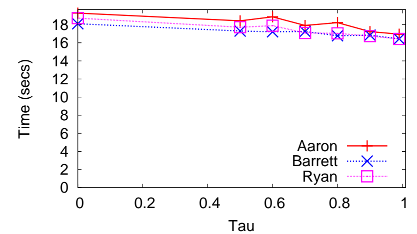

Figure 22: Inventory **runtime vs** τ

5.8 Changing the Match Pruning Policy

In these experiments, we vary τ and consider the effects on both Grades and Inventory results. Increasing τ increases the amount of pruning, and hence decreases the number of matches returned from StandardMatch. We find that for the Inventory tests, the overall accuracy remains constant with relatively high values of τ (Figure 20). Pruning does not degrade accuracy because all attributes in the inventory base table match with high confidence to appropriate attributes in both the Book and Music target tables, even before the base table has been split into its appropriate contexts. In the Grades example, however, the matches between the base table and the target table are more tenuous. Raising τ above 0.65 causes the accuracy to decrease significantly, as shown in Figure 21.

Finally, while the runtime decreases as τ increases (Figure 22), this effect is small compared to the overall runtime of the system. A τ of 0.5 seems to be a reasonable compromise of improved runtime without any sacrifice in accuracy.

5.9 Discussion

Our experiments demonstrate that contextual schema matching can effectively find conditional subsets and perform attribute normalization. We demonstrate a variety of tradeoffs between accuracy and performance. We find that EarlyDisjuncts is effective, relatively insulated from the setting of ω. However, for higher cardinality categorical data, its runtime is greater. LateDisjuncts scales more effectively, but is more dependent on ω being set correctly. Finally, both TgtClassInfer and SrcClassInfer perform significantly better than NaiveInfer. TgtClassInfer yields slightly higher accuracy than SrcClassInfer, but its runtime is greater.

In conclusion, the highest accuracy can be obtained by using EarlyDisjuncts with TgtClassInfer, while faster performance with reasonable accuracy can be obtained using LateDisjuncts with SrcClassInfer.

6. RELATED WORK

To our knowledge, no previous work has considered inferring conditional matches between a portion of one table and another, the topic of this paper.

In general a wide variety of schema matching techniques have been developed to integrate data represented in different models, for example relational, ER, object-oriented [4, 6, 17, 19, 27], and XML [11, 21, 23, 25, 26, 5] (see [29] for a recent survey). For example, Cupid [21] is a generic system that encompasses a variety of techniques such as linguistic analysis, structural matching and context dependencies. COMA [9] and its successor COMA++ [5] rely on fragment-based matching for XML data that is fragmented and then the matching algorithms are deployed over the fragments. TransScm [26] considers schema mappings based on schema matching. It uses a semi-automatic mechanism to match highly similar schemas.

Clio [14, 24, 28] also focuses on deriving schema mappings from schema matching and is discussed in Section 4.1. In [24], the authors mention that the user might impose selection conditions as part of information integration. We propose the inference of such conditions. Our schema mapping mechanism is an extension of its Clio counterpart in several aspects, as discussed in Section 4.

An alternative approach to handling schema heterogeneity in schema matching is reported in [13], in which a set of schema evolution operators [1] (normalizing, denormalizing, horizontal/vertical partitioning, renaming, etc.) is considered, and schema mapping is viewed as a search-problem in the space of schemas induced by applying sequences of these operators to one of the schemas to be matched. This approach has the advantage of handling more sophisticated scenarios, and Example 4.3 is drawn from [13]. However, the existing work is limited by the requirement that example transformed data be provided as input (the so-called “Rosetta-stone” principle). For unfamiliar target schemas, this requirement may not be realistic. Also the search space of schema transformations may be exceedingly large. Contextual schema matching and mapping can be seen as a step toward extending traditional schema matching/mapping techniques to handle more general transformations as considered in this work.

The use of multiple learners to infer mappings between a source schema and a target schema has been proposed in LSD [11], iMAP [8], and COMA [9]. The effectiveness of such an approach lies in that the learners handle different types of information. The overall accuracy of the system thus improves when mappings predicted by the different learners are combined. The StandardMatch used in our experiments follows this general approach.

We note that our view inference problem might be seen as an instance of *projective clustering* (see, e.g., [3]). We plan to investigate applying algorithms from this area in future work, but note that

most current projective clustering work focuses on metric domains.

7. CONCLUSION AND FUTURE WORK

We have introduced *contextual schema matching*, in which the schema matching system infers certain views of source and/or target tables as part of the schema matching process. We have investigated a class of simple and disjunctive contextual conditions that cover many practical examples in which matches are meaningful only under these conditions, and developed novel algorithms for inferring views to characterize these conditions. In response to the increased ambiguity introduced by inferred views, we have extended the *schema mapping construction* techniques of Clio [14, 24, 28], by introducing a novel form of constraints for views, inference rules for constraint propagation from base relations to views, and new join-group semantic association rules. We have presented the results of an extensive experimental study on the performance and quality of our techniques, concluding that contextual matching supports the further automation of schema mapping in the face of (a) horizontal partitioning of schemas and (b) promotions of data values to attributes.

Significant future work is suggested by this approach on both the theoretical and practical fronts. In particular, views on the target schema should be handled and more complex view conditions may be necessary, for example, if both horizontal partitioning and attribute promotion are encountered by the same match. This would lead to more involved views and add even more challenge to the automated inference of such views. Another issue is the need for a new form of constraints to accommodate complex contextual conditions and attribute promotion and associated semantic-association rules. A third issue concerns the interaction between constraints on the target and contextual matches, which calls for a systematic method to assure that contextual schema mapping does not violate the target constraints. We also plan to investigate inter-model contextual schema matching, namely between XML and relational model schemas.

8. REFERENCES

[1] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *TCS*, 62(1-2), 1988.

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

[3] C. Aggarwal and P. Yu. Finding generalized projected clusters in high dimensional spaces. In *SIGMOD*, 2000.

[4] V. Athitsos, M. Hadjieleftheriou, G. Kollios, and S. Sclaroff. Query-sensitive embeddings. In *SIGMOD*, 2005.

[5] D. Aumueller, H.-H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with COMA++. In *SIGMOD*, 2005.

[6] S. Castano, V. D. Antonellis, and S. D. C. di Vimercati. Global viewing of heterogeneous data sources. *TKDE*, 13(2):277–297, 2001.

[7] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 1999.

[8] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: discovering complex semantic matches between database schemas. In *SIGMOD*, 2004.

[9] H. Do and E. Rahm. COMA - a system for flexible combination of schema matching approaches. In *VLDB*, 2002.

[10] A. Doan. Illinois semantic integration archive.

[11] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, 2001.

[12] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368–406, May 2002.

[13] G. H. L. Fletcher and C. M. Wyss. Relational data mapping in MIQIS (demo). In *SIGMOD*, 2005.

[14] L. Haas, M. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.

[15] Q. He and T. W. Ling. Extending and inferring functional dependencies in schema transformation. In *CIKM*, 2004.

[16] J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, 2003.

[17] L. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – a language for interoperability in relational multi-database systems. In *VLDB*, 1996.

[18] D. D. Lewis and W. A. Gale. A sequential algorithm for training text classifiers. In *ACM Int'l Conf. on Research and Development in Information Retrieval*, 1994.

[19] W.-S. Li and C. Clifton. SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl Eng*, 33(1):49–84, 2000.

[20] J. Madhavan, P. A. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *ICDE*, 2005.

[21] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *VLDB*, 2001.

[22] R. McCann, B. AlShebli, Q. Le, H. Nguyen, L. Vu, and A. Doan. Mapping maintenance for data integration systems. In *VLDB*, 2005.

[23] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, 2003.

[24] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, 2000.

[25] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[26] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, 1998.

[27] L. Palopoli, D. Sacca, and D. Ursino. Semi-automatic semantic discovery of properties from database schemas. In *IDEAS*, 1998.

[28] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web data. In *VLDB*, 2002.

[29] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[30] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[31] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, 2003.

[32] U. Washington. Schema matching samples. "http://www.cs.washington.edu/homes/jayant/corpus".