A Grammar-based Entity Representation Framework for Data Cleaning

Arvind Arasu
Microsoft Research
One Microsoft Way
Redmond, WA, USA
arvinda@microsoft.com

Raghav Kaushik
Microsoft Research
One Microsoft Way
Redmond, WA, USA
skaushi@microsoft.com

ABSTRACT

Fundamental to data cleaning is the need to account for multiple data representations. We propose a formal framework that can be used to reason about and manipulate data representations. The framework is declarative and combines elements of a generative grammar with database querying. It also incorporates actions in the spirit of programming language compilers. This framework has multiple applications such as parsing and data normalization. Data normalization is interesting in its own right in preparing data for analysis as well as in pre-processing data for further cleansing. We empirically study the utility of the framework over several real-world data cleaning scenarios and find that with the right normalization, often the need for further cleansing is minimized.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Design, Algorithms, Experimentation

Keywords

Data Cleaning, Entity Resolution, Deduplication

1. INTRODUCTION

It is well-understood that the primary source of poor data quality is the fact that real world entities lack a unique representation. For example, the US state of California could be represented using three different strings: *California*, *Calif*, *CA*; other representations (e.g., *Callifornia*, *Calfornia*) can occur unintentionally due to typographic errors. The representation of an entity could be a string as in the examples above, but more generally, it could be a structured record. Figure 1 shows five example author records derived from a publication database, where each author record contains a name and an affiliation. These five records, although textually different, correspond to just two well-known authors.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

Id	*Author*	*Affiliation*
1	Jeffrey D. Ullman	Department of Computer Science, Stanford University, Stanford, CA, USA
2	Jeff Ullman	Stanford University, Stanford CA 94305
3	J D Ullman	Stanford Univ., Stanford, Calif.
4	M. Stonebraker	M.I.T.
5	Michael Stonebraker	Department of EECS and Laboratory of Computer Science, M.I.T., MA 02139, Cambridge, USA

Figure 1: Sample Author Records

Id	*ProductName*
1	ThinkPad T43 Centrino PM 750 1.8GHz/512MB/40GB/14.1"TFT
2	ThinkPad T43 Centrino PM 740 1.7GHz/512MB/40GB/14.1"TFT
3	Lenovo ThinkPad T43 Notebook (1.7GHz Pentium M Centrino 740, 512MB, 40 GB, 14.1TFT)

Figure 2: Sample Product Records

A fundamental problem in data cleaning is that of determining whether or not two representations are *duplicates*, i.e., correspond to the same real-world entity. This problem forms the core of well-known data cleaning primitives such as *record matching* and *deduplication*. Most of the current approaches [3, 6, 7, 8] use textual similarity, typically quantified using a *similarity function* such as edit distance and cosine similarity, to determine if two representations are duplicates. The basic premise behind these approaches is that duplicate representations are textually similar, and this approach is indeed useful for catching most kinds of typographic errors. However, textual similarity can often be misleading: two representations of the same entity could be highly dissimilar textually (e.g., the author records 4 and 5 of Figure 1) and, conversely, two representations that are textually very similar could correspond to different entities. The latter point is illustrated in example (laptop) product records shown in Figure 2. The records with ids 1 and 2 are textually similar—they differ in just two characters—but correspond to different laptops with different processors. On the other hand, the records with ids 2 and 3 correspond to laptops with the same configuration and can therefore be considered duplicates.

The motivation for our work comes from the observation

that variations in representations of entities arise in *complex* ways that are hard to capture using a black-box similarity function, motivating the need for a more sophisticated approach. For illustration, consider an affiliation such as those in the *Affiliation* column of Figure 1. Variations in the representation of an affiliation can arise at several levels: First, variations arise from the presence or absence of details such as department information, university information, and location information that comprise an affiliation. Second, variations arise from the different possible orders in which these details are specified. Third, each detail itself can be considered as a sub-entity and, recursively, variations in the representations of these sub-entities contribute to overall variations. At the "leaf-level" entities, variations arise due to abbreviations and other kinds of aliasing, e.g., *MIT* and *Massachusetts Institute of Technology*. Such aliasing also occur at the word level: for example, & and *and*, and *Techn.* and *Technology*. Typographic errors can also be considered as variations at the word or multi-word level. Finally, there are variations at the character level due to accented characters.

Some limitations of black-box similarity functions are addressed by recent prior work [2], which proposes a programmatic approach where *string transformations* that specify, for example, that `Bob` and `Robert` are synonymous can be provided as explicit input. Two string representations are considered similar (and therefore potential duplicates) if they can be made textually similar by applying some transformations. There are two limitations of this approach: First, it does not understand the internal "structure" of the representation of an entity, thereby missing out on rich contextual information necessary to determine if an alias or synonym is meaningful. For example, it is meaningful to abbreviate `University` to `U` when dealing with university names (e.g., `U of Calif Berkeley`) but not within street names (e.g., `U Ave` for `University Ave`). The second limitation of this approach arises from the fact that similarity of two strings can only *increase* by adding transformations. This approach is therefore not useful in handling the scenario of Figure 2 where two strings that are textually similar should *not* be matched.

Closely related to our problem of dealing with representational variations is the well-studied problem of *segmentation* [4, 20, 24]. Segmentation is the problem of partitioning a string into its constituent components: for example, given an affiliation string, segmentation techniques can identify substrings corresponding to the department, university name, and location. However, segmentation by itself does not deal with representational variations: segmentation can be used to recognize that `MIT` and `Massachusetts Institute of Technology` are both university names, but cannot be used to reason that they refer to the same university. Also, segmentation techniques often rely on external tables such as a list of known university names for the affiliation example above. We run into the same problem of multiple representations when using external tables since, for example, the representation of a university name in the external table could be different from its representation in an affiliation being segmented. Again, similarity functions have been used to address this problem in this setting [9], and we face the same kinds of problems we illustrated earlier.

Id	*FName*	*LName*	*Univ*
1	Jeffrey	Ullman	Stanford University
2	Jeffrey	Ullman	Stanford University
3	J	Ullman	Stanford University
4	M	Stonebraker	Massachusetts Institute of Technology
5	Michael	Stonebraker	Massachusetts Institute of Technology

Figure 3: Transformed output records for input records of Figure 1

Our Contribution

We propose a rich, declarative framework for reasoning with and manipulating representations of entities. In terms of overall functionality, our framework is a *programmable* module that can be programmed to transform "dirty" input records to one or more "clean" output records with consistent representation of entities and subentities. For example, we can program the framework to transform the input records of Figure 1 to records such as those shown in Figure 3. (The records of Figure 3 represent idealized, hand generated output with the sole purpose of illustrating the overall functionality of our framework.) Note that the output records are not simple segmentations of the input records: the first name `Jeff` has been normalized to `Jeffrey` in record id 2 and all non-university information has been stripped out of the affiliation, and university names have been normalized.

The transformed output records can then be used for a variety of data cleaning tasks:

1. They can be used as input for other data cleaning operations such as record matching. We believe that in many record matching settings, we can use the framework to significantly simplify subsequent record matching and reduce its reliance on complex similarity functions. In our running example, once we have transformed the input records of Figure 1 to those in Figure 3, we can identify potential duplicate authors using a combination of simple string functions (to handle name initials) and equi-joins.

2. They can be used for rich querying: For example, we can pose simple group-by aggregation queries over the transformed author table of Figure 3 to compute the number of publications affiliated with each university, to determine the most prolific author, and so on.

Our framework is purely programmatic, meaning that it has no built-in data cleaning knowledge. At a high level, a *program* for our framework consists of a *generative grammar*, described using context-free grammar rules, that specifies how input records are "generated." Given an input record, the framework parses the record using the grammar. Each rule of the grammar has an associated execution logic called *action*, which is evaluated whenever the rule participates in parsing. The actions associated with the rules that participate in parsing a record do the actual work of transforming an input record to its corresponding output record(s). The combination of generative grammar and actions provides the programmer a powerful ability to access and manipulate parts of an input record.

An important feature of our framework is the ability to implicitly specify a large number of grammar rules using datalog style queries. This feature allows us to easily exploit large amounts of domain knowledge within our framework.

As other authors have observed earlier [21], we can compile large amounts of relatively clean, structured data from online resources such as Wikipedia, Wiktionary, and DBLP (for publication domain). In fact, there already exists collections with millions of clean records and facts [14, 21]. We will illustrate using detailed case studies how we can leverage such data for reasoning about entity representations within our framework.

Outline

The rest of the paper is organized as follows: Section 2 motivates the high-level design of our framework using a simple illustrative example. The framework itself is formally specified in Section 3. Section 4 describes implementation aspects of the framework. Section 5 presents detailed case studies of our framework for two data cleaning tasks. Section 6 discusses various aspects of our framework. We cover related work in Section 7 before concluding.

2. OVERVIEW

As mentioned in Section 1, our framework provides a declarative method for reasoning with and manipulating entity representations. We now use a simple example to illustrate the kinds of variations that can occur in entity representations, and use this to motivate the basic design aspects of our framework. Consider the following affiliation string from Figure 1:

> *Department of computer science, Stanford University, Stanford, CA*

Even without introducing any typographical errors, there are a number of ways of representing the same affiliation. There are several ways of specifying the department information. For example, we can shorten the string *department* to *dept.* We can shorten *computer science* to *CS*, *comp. sci.*, or *computer sci.* We can represent the department information using a different order as *computer science department.* Similarly, we can represent *Stanford University* using a several variations. For example, we can shorten *university* to *univ* or even drop it altogether. Finally, the state of California can be represented in different ways such as *CA*, *California*, and *Calif.* We make the following observations about these variations:

1. Most of these variations are *orthogonal* to each other. We can combine any variation of the department, any variation of the university, and any variation of the state to represent the same affiliation.

2. Some of these variations have a fairly general structure that is not specific to the above affiliation. For example, we can derive the variations for a different university such as *Oxford University* by replacing *Stanford* with *Oxford.* Similarly, we can derive variations for a different department, for example, by replacing *Computer Science* by *Electrical Engineering*, and using corresponding abbreviations for *Electrical* and *Engineering.*

3. In contrast to Observation 2, some variations are specific to a particular entity. These include, for example, variations in representations of *California* and word level variations (e.g., *tech* and *technology*). We need external domain knowledge to know about these variations.

The first observation above suggests that a *generative grammar* is a natural and concise way of representing all the variations. Figure 4 illustrates this for the above affiliation. The second observation suggests the possibility of using a single generative grammar to represent the variations for a large number of different affiliations. The third observation suggests that any framework for capturing variations should provide a flexible way of incorporating external domain knowledge.

The design of our framework is based on the above observations. Informally, our framework allows a programmer to declaratively specify using an "augmented" generative grammar how variations in entity representations arise. The augmented generative grammar has aspects of database querying which can be used to easily incorporate domain knowledge. As we will see shortly, a knowledge of how variations in entity representations arise can be used to *normalize* the variations to simplify subsequent data cleaning.

3. FRAMEWORK

Functionality

The framework takes a record as input and outputs one or more records. The output records can be *complex* with nested attributes. This feature is useful, for example, to extract a *set* of authors from a citation string. The framework associates a positive real number *weight* with each output record. A natural interpretation of the functionality of the framework is to view the input record as a "dirty" record and the set of output records as possible "clean" records with weights capturing the confidence in a clean record. By design, smaller weights reflect greater confidence.

Program

Our framework is *programmatic*, meaning how an input record is transformed to output records is specified using a declarative program. A program is an *augmented context free grammar (CFG)*. It is a collection of triples of the form $\langle R, P, A\rangle$, where R is a *grammar rule*, P a *predicate*, and A an *action*. We define these terms shortly. We call the triple $\langle R, P, A\rangle$ an *augmented rule* or simply a *rule* when unambiguous.

A grammar rule is similar to a standard CFG rule: It has a head and a body. The head is a single nonterminal. The body is a sequence of nonterminals, terminals, and a third kind of symbol called *variables*. Terminals are basic components of text such as words and punctuation. We represent nonterminals using angular brackets (e.g., $\langle name\rangle$), terminals using single-quoted strings (e.g., 'Jeff'), and variables using uppercase letters (e.g., X). As we describe shortly, grammar rules are used to parse an input record. For purposes of parsing, we assume that each attribute in the schema of input records is associated with a unique nonterminal that serves as the *start symbol* for parsing.

A predicate is the body of a datalog rule. It is a conjunction of (nonnegated) *atomic predicates*. An atomic predicate is either an *extensional database (EDB)* [22] predicate (input relation) or a built-in predicate. Every variable in an augmented rule $\langle R, P, A\rangle$ is constrained to occur at least once in an EDB predicate of P. The last constraint is analogous to the *limited variable* constraint to make datalog rules safe [22].

An action is a function that takes zero or more records as input and produces a record as output. The arity of action

⟨All⟩	:	⟨CS Dept⟩ ⟨Stanford University⟩ 'Stanford' ⟨California⟩
⟨CS Dept⟩	:	⟨Dept⟩ 'of' ⟨CS⟩ | ⟨CS⟩ ⟨Dept⟩
⟨Dept⟩	:	'Department' | 'Dept'
⟨CS⟩	:	⟨Computer⟩ ⟨Science⟩
⟨Computer⟩	:	'Computer' | 'Comp'
⟨Science⟩	:	'Science' | 'Sci'
⟨Stanford University⟩	:	'Stanford' ⟨University⟩ | 'Stanford'
⟨University⟩	:	'University' | 'Univ'
⟨California⟩	:	'California' | 'Calif' | 'CA'

Figure 4: A generative grammar to capture variations

A in an augmented rule $\langle R, P, A\rangle$ is the number of nonterminals in the body of R. To consistently specify actions we assume that each nonterminal is associated with a fixed schema. The output schema of action A is the schema corresponding to the nonterminal that forms the head of grammar rule R, and the schema of the input records of A correspond to the schemas of the nonterminals in the body of R. We deal mostly with actions that are *projection* functions. In a projection function, the output record is a projection of the attributes of the input records. We represent projection functions as a sequence of assignment operations, where each operation has an attribute of the output record for its lhs and an attribute of an input record or a constant/variable for its rhs. The definition of an action can involve variables; we define the meaning of variables shortly.

Example 1. Figure 5 shows a sample program for processing names. The program has 9 augmented rules. The grammar rules, predicates, and actions of each augmented rule are shown in separate columns. All the predicates contain a single EDB predicate. The tables for the referenced EDB predicates are shown in Figure 6. To illustrate our notation for actions, consider the action associated with rule 1. This action takes 4 records as input and produces a record with two attributes, *fname* and *lname*, as output. The value of the *fname* attribute comes from the *value* attribute of the second input record (*2.value*) and the value of the *lname* attribute comes from the *value* attribute of the fourth input record (*4.value*). □

Semantics

Informally, given a program G and an input record r, the framework *parses* the attributes of r using grammar rules in G. Actions are then evaluated along the nodes of the *parse tree* to construct an output record. There could be multiple ways of parsing r leading to multiple output records. It is useful to view the grammar rules as specifying how input records are "generated," as illustrated by the following example.

Example 2. The program in Figure 5 specifies a grammar for generating people names. (This grammar is for illustrative purposes only and is not comprehensive.) Rule 1 specifies that a name can be generated by concatenating a prefix, firstname, middlename, and lastname. Rule 2 gives an alternate way of generating names, by concatenating lastname, firstname, and middlename, with a comma between the lastname and firstname. Rules 3-5 describe how a firstname can be generated. Rule 3 handles the case where the firstname is an initial. Rule 4 illustrates the use of a variable. It indicates that the firstname can be any value in the second column of the *FNames* table. Rule 4 is a shorthand

Id	*SourceId*	*Rule*	*Action*
1	4	⟨fname⟩ → Andrew	*value* = Andrew
2	4	⟨fname⟩ → John	*value* = John
3	5	⟨fname⟩ → Andy	*value* = Anderson
4	5	⟨fname⟩ → Andy	*value* = Andrew
5	8	⟨lname⟩ → Smith	*value* = Smith
6	9	⟨prefix⟩ → Dr	
		...	...

Figure 7: Expanded Program for Program in Figure 5

for a large number of rules of the form: $\langle fname\rangle \rightarrow Andrew$, $\langle fname\rangle \rightarrow John$, and $\langle fname\rangle \rightarrow Mary$. More generally, variables and predicates represent a shorthand notation for specifying a large number of rules and actions. Similarly, Rules 6-9 specify how middlenames and lastnames are generated. □

To present the formal semantics of our framework, we first define an *expanded* program G' for a given program G. This definition formalizes the meaning of variables and predicates. We then define how input records are parsed using the expanded program G', and finally define how actions are evaluated to produce the output records.

The expanded program G', like G, is a collection of augmented rules. However, it does not contain variables and its predicates are empty (trivially true). To construct G', we consider each augmented rule $\mathcal{R} = \langle R, P, A\rangle$ in G and enumerate all possible assignments of constant values to variables in $\mathcal{R}$ so that the predicate P evaluates to true. For each assignment, we include as part of G' the rule $\langle R', true, A'\rangle$ obtained by substituting variables in $\mathcal{R}$ with the corresponding constant values.

Example 3. Figure 7 shows part of the expanded program (without the predicates) for the program in Figure 5. For each augmented rule in Figure 7, the *Source Id* column contains the *Id* of the rule in Figure 5 from which the rule was generated. For example, the first rule in Figure 7 is generated by the substitutions: $I = 1$, $F = Andrew$, $G = M$. Note that two augmented rules can have the same grammar rule and differ only in their actions (e.g., rules with Ids 3 and 4 in Figure 7). □

Given a program G, we associate zero or more parses with each input record r. A parse of r is defined as a *derivation* for each attribute of r using grammar rules in the expanded program G' of G. The derivation for an attribute starts with the start symbol for the attribute, defined earlier. Note

Id	*Rule*	*Predicate*	*Action*
1	⟨name⟩ → ⟨prefix⟩$_1$ ⟨fname⟩$_2$ ⟨mname⟩$_3$ ⟨lname⟩$_4$		*fname = 2.value*; *lname = 4.value*
2	⟨name⟩ → ⟨lname⟩$_1$ ',' ⟨fname⟩$_2$ ⟨mname⟩$_3$		*fname = 2.value*; *lname* = 1.*value*
3	⟨fname⟩ → ⟨letter⟩$_1$ '.'		*value = 1.value*
4	⟨fname⟩ → F	FNames (I, F, G)	*value = F*
5	⟨fname⟩ → N	NickNames (I, N, F, G)	*value = F*
6	⟨mname⟩ → M	LNames (I, M)	
7	⟨mname⟩ → ⟨letter⟩ '.'		
8	⟨lname⟩ → L	LNames (I, L)	*value = L*
9	⟨prefix⟩ → S	Prefix (I, S)	

Figure 5: A Program in our framework for processing names

1	Andrew	M
2	John	M
3	Mary	F
4	...	...

FNames

1	Alex	Alexander	M
2	Andy	Anderson	M
3	Andy	Andrew	M
4	Becky	Rebecca	F

NickNames

1	Smith
2	Johnson
3	Williams
4	...

LNames

1	Mr
2	Ms
3	Dr
4	...

Prefix

Figure 6: Example Tables used in Sample Program

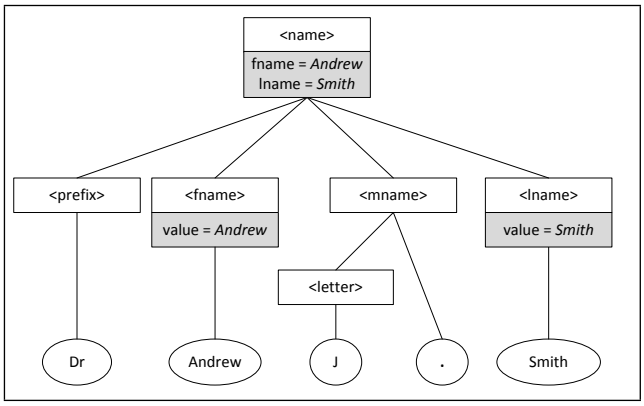

Figure 8: Parse Tree for "Dr Andrew J. Smith"

that, by definition, the grammar rules in G' do not contain variables, and we use standard CFG semantics to define a derivation. There could exist more than one parse for an input record, and we consider all such parses. Following standard convention, we represent a parse of r as a parse tree.

Example 4. The program in Figure 5 is designed to process input records with a single attribute (plain strings), and parsing of the strings starts with the nonterminal ⟨name⟩. Figure 8 shows the parse tree for the string `Dr Andrew J. Smith`. The nonterminals in the parse tree are shown using unshaded rectangular boxes and the terminals (leaf nodes) are shown using ovals. □

For each parse tree of r, the framework produces one output record. To construct the output record, we evaluate actions along the nodes of the parse tree in a bottom-up fashion. Each non-leaf node N in the parse tree is associated with an augmented rule $\langle R, true, A\rangle$ of G', and we evaluate action A at node N. The input records for A are the records produced by evaluating actions at the nonterminal child nodes of N. Recall that the arity of A is the number of nonterminals in the body of R, which is exactly the number of nonterminal child nodes of N. The overall output record for the parse tree is the record produced by evaluating the action corresponding to the root node of the parse tree. We note that actions are similar to *semantic items* of compiler grammars [1].

Example 5. Continuing Example 4, Figure 8 shows using shaded rectangles the records produced by evaluating actions along the nodes of the parse tree. (Nodes without shaded rectangles produce empty records.) For example, the derivation of ⟨fname⟩ uses the rule $\langle fname\rangle \rightarrow Andrew$ with action $value = Andrew$ (rule 1 in Figure 7). Evaluating this action produces the record $\langle value : Andrew\rangle$. We can verify that evaluating the action at the root node similarly produces the record $\langle fname : Andrew, lname : Smith\rangle$. □

To see the utility of our framework for managing entity representations, we note that the sample program of Figure 5 standardizes simple variations in the representation of people names. First, it handles variations in the order in which the first name and last name appear: For example, we can verify that the program produces the same (single) output record $\langle fname : Andrew, lname : Smith\rangle$ for both `Dr Andrew J. Smith` (see Example 5) and `Smith, Andrew J.`. Second, the program handles variations resulting from the use of a nickname for the first name: The program produces two output records for the string `Smith, Andy J.`. The first is $\langle fname : Andrew, lname : Smith\rangle$ (same as above), and the second, $\langle fname : Anderson, lname : Smith\rangle$. The parse tree for the first is shown in Figure 9. Both output records can be valid standardizations of the given name and our framework exposes this uncertainty by producing both in the output.

Weights: The framework associates a non-negative real number weight with every output record. To define weights,

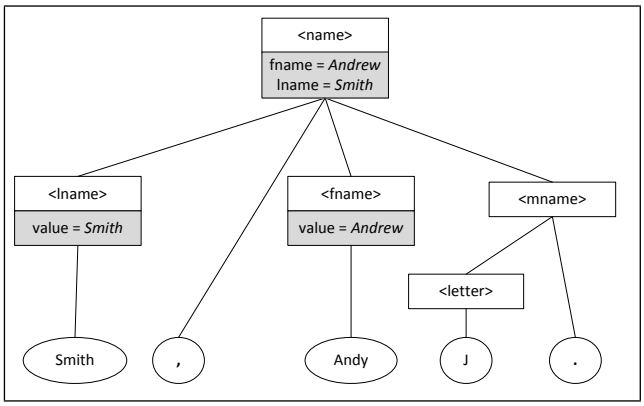

Figure 9: Parse Tree for "Smith, Andy J."

we assume as input a *weighting scheme* that assigns a non-negative real number weight to each augmented rule in the expanded program G'. The weight of an output record is defined as the sum of weights of augmented rules involved in the parse that produces the output record. Weights capture the confidence in a particular parsing of the input record, and, by design, lower weights indicate a higher confidence.

Weights allow a programmer to use "loose" rules, rules that the programmer is not very confident about. The programmer can assign a higher weight to such rules, and any parse involving these rules would get a higher weight. As we illustrate in Section 5, such loose rules are useful to handle incomplete domain knowledge.

In all of our experiments, we use the following simple weighting scheme called the *uniform weighting scheme*: Consider an augmented rule $\mathcal{R}'$ in the expanded program G' and let $\mathcal{R}$ denote the rule in G that was used to generate $\mathcal{R}'$. The weight of $\mathcal{R}'$ is defined as the log of number of rules in G' that were generated from rule $\mathcal{R}$ in G. Intuitively, the weighting scheme prefers parses involving more specific rules compared to more general rules. We illustrate the intuition behind the uniform weighting scheme using our case studies in Section 5.

4. IMPLEMENTATION

We now briefly describe a simple implementation of our framework. Our implementation uses a combination of string matching using the Aho-Corasick algorithm [10] and grammar parsing. Our study of the implementation techniques for the framework is fairly preliminary, and a more comprehensive exploration of the space of implementation techniques is future work. We note however that even the simple techniques of this paper achieve a data processing rate of a few thousand records per second for the programs and dataset we use in our case studies.

We first outline how our implementation proceeds in the absence of any actions. At first glance, it appears that traditional parsing methods would suffice since the variables act simply as a shorthand. Specifically, given a program G, we can construct the expanded program G' in a preprocessing step; given an input record r, we can use traditional parsing techniques to parse r using the grammar rules of G'. The main problem with this approach is that the scale of the expanded grammar G' can be very large, of the order of the database size. Our implementation technique, instead of constructing the full expanded program G', constructs at query time a *partially* expanded program G'_r that contains a subset of the rules in G' that are relevant to r, and uses G'_r to parse the record r.

To construct G'_r, we consider each augmented rule $\mathcal{R} = \langle R, P, A\rangle$ in G, and enumerate as before all possible assignments of constants to variables in $\mathcal{R}$ such that predicate P evaluates to true; we now enforce the additional constraint that if a variable X occurs in the body of R, then the constant c assigned to variable X should be a substring of the record r. As before, for each assignment, we add one augmented rule to G'_r obtained by substituting variables in $\mathcal{R}$ with the corresponding constant values. If c is not a substring of r, we can show that any rule obtained by substituting c for X cannot participate in parsing r, therefore both G' and G'_r have the same set of parses for the record r.

We use the Aho-Corasick algorithm [10] to efficiently construct G'_r at query time. Consider an augmented rule $\mathcal{R} = \langle R, P, A\rangle$ and let X be a variable in the body of R, i.e., R is of the form $\langle N\rangle \rightarrow \alpha X \beta$. We compute in a preprocessing step a set of possible assignments to variable X by evaluating the query:

$$\mathit{Dictionary}(X)\text{:-}P(X, \ldots)$$

Recall from Section 3 that the variable X is guaranteed to occur in an EDB predicate of P. We build an Aho-Corasick index over the set of possible values returned by the above query. At query time, given an input record r, we use the Aho-Corasick indexes over variables in R to efficiently identify all assignments of constant values to variables in $\mathcal{R}$ such that the predicate P evaluates to true and which satisfy the constraint that constants assigned to variables such as X are substrings of r.

Example 6. Consider the sample program shown in Figure 5 and the input record that consists of the single string *Smith, Andy J.*. The table NickNames is likely to consist of lots of nick names. The only ones relevant to this input record are those that are its substrings, in this case the string *Andy*. Before we begin parsing input records, we build a dictionary of values for each variable in the grammar. The dictionary for the variable N would contain the output of the datalog query:

$$\mathit{Dictionary}(N)\text{:-}\mathit{NickNames}(I, N, F, G)$$

This set of strings is indexed using an Aho-Corasick automaton for substring matching. We note here that if we wish to be error-tolerant in this step, we can use other techniques proposed in prior work [5] instead of the Aho-Corasick algorithm. □

5. CASE STUDIES

We now present case studies of using our framework on different data sets. The purpose of these case studies is two fold: First, we use the case studies to illustrate the utility of our framework for data normalization, and for simplifying and improving the quality of record matching. Second, we use the case studies to illustrate the actual use of the framework by describing the rules, actions, and external domain knowledge that we use for each case study.

Id	*Name*
1	Professor Michael J. Bannon, Head of Department
2	Bannon, Michael
3	Dr. Carol Barrett (also Biotech. Office F.13, EXT. 2809)
4	Ms. Orla Benson, Alumni and Special Events Manager
5	O Ceallaigh, Sile

Figure 10: Example records in UCD data

5.1 UCD People Data

This is a publicly available dataset that is part of the Riddle repository for record matching [18]. The dataset consists of 5332 single attribute people records, where each record contains a person name along with other optional details such as the person's affiliation and office address. Figure 10 shows five sample records from this dataset. The overall goal is to match records based on people names, while ignoring the optional details. Two names match if they agree on the first and last names, or if they agree on the last name and one of the first names is an initial of the other.

Program

In order to perform the above matching task, our goal is to transform input record to normalized output records with first- and last-name attributes. Figure 11 shows part of the program that we use to accomplish this goal. The root non-terminal for parsing the input records is ⟨person⟩. Rule 1 specifies that a person consists of a name (⟨name⟩) followed by an optional description (⟨desc⟩).

The generative grammar for parsing names is the one shown in Figure 5 with simple additions such as making the middle name optional. In order to recognize names, we use compiled tables of first and last names. Our first source of names is the US Census website [23] which publishes lists of first and lastnames (from Census 2000). These lists, while very clean, are not comprehensive. Our second source of names is the list of authors from DBLP [12]. DBLP has more than 650,000 authors and has a much better coverage of names than the US Census data in terms in distinct first names and last names. However, DBLP only contains full author names, not separate first and last names. In order to compile first and last names, we simply run the *same program* over the DBLP author data. To enable identifying new names, we allow ⟨fname⟩ and ⟨lname⟩ to be unknown words (rules 5 and 8). This produces a relatively clean list of 79000 first names and 183000 last names. This step illustrates how our framework itself can be used to collect useful domain knowledge. We note that the UCD dataset does not specifically contain authors and is unrelated to DBLP. We used DBLP simply as a source of names: any other dataset containing large number of clean names could have been similarly used.

It is harder to precisely model the other details such as a person's affiliation and address. We notice that such details are often plain English text—see, for example, records 1 and 4 in Figure 10. We therefore model the details as a sequence of known English words (rule 9). We compiled a list of English words from Wiktionary [25]. We finally use a catch-all rule (rule 10) that models ⟨desc⟩ as an arbitrary sequence of tokens that matches any text.

The generative grammar specified above is fairly "loose" in how it models input records. Any word can qualify as a first

Id	*Rule*	*Predicate*
1	⟨person⟩ → ⟨name⟩ ⟨desc⟩?	
2	⟨name⟩ → ...	
3	⟨fname⟩ → F	USCensusFName(F)
4	⟨fname⟩ → F	DBLPFName(F)
5	⟨fname⟩ → ⟨Word⟩	
6	⟨lname⟩ → L	USCensusLName(L)
7	⟨lname⟩ → L	DBLPLName(L)
8	⟨lname⟩ → ⟨Word⟩	
9	⟨desc⟩ → W+	Wiktionary(W)
10	⟨desc⟩ → ⟨token⟩+	

Figure 11: (Part of) Program for UCD dataset

name or a last name, and any sequence of tokens qualifies as a valid description associated with a name. The grammar parses almost any piece of text, and many parses can be incorrect. For example, record 1 in Figure 10 can be parsed with `Professor` as first name and `Michael` as last name and the rest as description. However, these generic rules do not cause a serious drop in overall quality due to the use of weights. All of our case studies use the uniform weighting scheme described in Section 3. This weighting scheme assigns a much lower weight to the parse where `Michael` (rule 3) is parsed as first name and `Bannon` (rule 6) as last name, compared to the incorrect parse above. Generic rules such as rule 5 are still important to be able to handle unknown first names. For example, if in a given input record, the last name is known and the first name unknown, the generic rule above will correctly identify the (unknown) first name.

Quality of Data Transformation

The overall data transformation quality of the above rules is extremely high. We measure quality throughout by manually evaluating a random sample of output records. For input records that admit multiple parses and therefore produce multiple output records, we pick the output record with the best parse, i.e., minimum weight. Also, we use throughout the uniform weighting scheme, so no weights were provided as explicit input. For this particular dataset and the above set of rules, a random sample of 100 output records contained no incorrect incorrect ones, indicating a very high quality of the output.

Figure 12 quantifies the utility of various rules and external tables. We use *coverage* to denote the number of input records that admit at least one parse and therefore produce at least one output record. We report on the impact on coverage of the rules 3-5 and 6-8, which are the main rules for parsing first and last names, respectively. The rightmost bar (*All*) in Figure 12 shows the coverage using all the rules, the middle bar (*DBLP+USCensus*), the coverage after dropping the generic rules 5 and 8, and the left bar (*USCensus*), the coverage using rules 3 and 6 alone. We observe that the overall coverage using all the rules is fairly high—over 99%. Using just the US Census table for parsing brings down the coverage to 64%. DBLP tables have a significant impact on coverage; as noted earlier DBLP is a fairly comprehensive source of names—even non-western names are well represented. Finally, about 700 records rely on generic rules 5

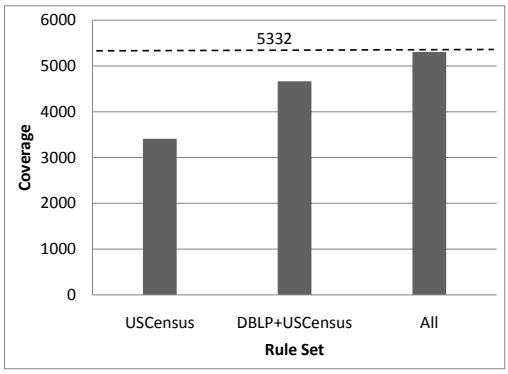

Figure 12: Coverage for UCD People

and 8, and we note that the generic rules do not seriously reduce quality.

Record Matching

We now comment on the utility of our framework for record matching, noting that the UCD People dataset was originally intended as a dataset to study record matching. Figure 13 shows the quality of record matching for a representative similarity function, *jaccard similarity*. Informally, for a given pair of records, jaccard similarity measures the weighted overlap between the tokens of the pairs. Jaccard similarity is known to be reasonably competitive with other similarity measures [6], and this is especially so for our dataset since there are few "pure" edit errors. For a given similarity threshold, we considered all pairs of records with jaccard similarity greater than the threshold as matches and evaluated the quality of the match. We note that for our single attribute dataset more sophisticated record matching techniques that combine similarity scores across multiple attributes do not apply.

Figure 13 shows, at various similarity thresholds, the estimated number of correct matching pairs and the estimated number of incorrect matching pairs: the dark region represents the correct matching pairs and the light region, the incorrect ones. We estimated these numbers by manually evaluating a random sample (of size 100) of pairs with similarity greater than the given threshold. Using a jaccard similarity threshold of around 0.5, we can identify about 2500 matching pairs at fairly high accuracy. Using a lower threshold reduces the match quality significantly. (The slight dip in the estimated number of correct pairs as we reduce similarity is probably due to a sampling error.)

Using a simple equi-self join over the records output by our framework, we can identify 3150 matches, indicated by the dotted line in Figure 13, which includes about 600 pairs not found by the similarity function approach. Presumably, these 600 pairs have very low similarity due to the presence of details not relevant to matching in one of the records.

Our overall point in this experiment is that we can leverage external domain knowledge, not accessible to a black-box similarity function, to "peek" inside strings and perform more sophisticated record matching.

5.2 Author Affiliations Dataset

This is a dataset (not publicly available) containing author affiliations from a well-known publications database. The dataset consists of 107,000 single attribute records containing affiliation strings. Our overall goal for this dataset

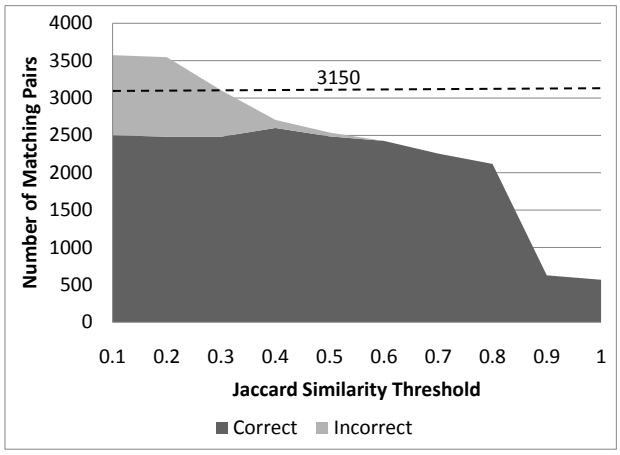

Figure 13: UCD People: Record Matching Quality

Id	*Affil*
1	Dalhousie University, School of human communication disorders, 5599 Fenwick Street, Halifax, Nova Scotia, Canada B3H 1R2
2	Dalhousie Univ., Halifax
3	Department of Informatics, Technische Universitat Munchen, Boltzmannstr. 3, D-85748 Garching, Germany
4	IIT Bombay
5	Department of Mathematics, University of Wisconsin, 480 Lincoln Drive, Madison, WI 53706

Figure 14: Example records in Affiliations dataset

is to transform each affiliation string to its root organization (e.g., university), ensuring that a given organization has a unique representation. This is exactly the functionality illustrated in Figure 3, but without the author names. The related record matching task is to identify all pairs of affiliations corresponding to the same organization. We restrict ourselves to academic affiliations from six representative countries: USA, Canada, UK, Germany, India, and Taiwan.

Challenges

The above data cleaning and transformation task involves several challenges: First, as in UCD data, there is a lot of extraneous information that needs to be ignored. Second, unlike people names, university names (names of academic institutions) admit a greater degree of variations, including abbreviations, when compared to people names. Often, what exactly constitutes the name of a university is less clear: e.g., `City University` vs. `City University of London`. Third, location information is sometimes important in order to disambiguate a university name (e.g., the location `Madison` in record 5). Finally, universities in non-English speaking countries often use both a local name and an English name: e.g., `Technical University of Munich` and `Technische Universitat Munchen`.

Domain Knowledge

We use a large amount of domain knowledge obtained from Wikipedia for this data cleaning task. The first type of domain knowledge that we use is location information, which consists of lists of countries, states (or provinces), and cities, and containment relationships between them. We obtain

	Table
1	City (pageId, Title, State)
2	State (pageId, Title, Country)
3	Country (pageId, Title)
4	Univ (pageId, Title, City, RootName)
5	WRed (pageId, Title, RTitle)

Figure 15: Wikipedia Tables for Affiliations

	Title	*City*	*RootName*
1	University of California, Berkeley	Berkeley	California
2	University of California, San Diego	San Diego	California
3	Stanford University	Stanford	Stanford

Figure 16: Example entries for Univ(pageId, Title, City, RootName)

city and state information only for the subset of countries we are interested in. We can get this information relatively easily from Wikipedia using one of two techniques: First, we can use Wikipedia *infoboxes* [26] which are structured information embedded within Wikipedia pages. For example, we can get a list of Indian cities and states by extracting all Wikipedia infoboxes named "Indian Jurisdiction". Second, we can use special *list* pages on Wikipedia to compile such information. For example, there exists a page on Wikipedia that contains a link to every town and city in England.

The second type of domain knowledge that we use is a compiled list of universities and other academic institutions. Conveniently, there is a Wikipedia infobox "University" that provides this information. The infobox for a university also includes other relevant information such as the city in which the university is present.

Figure 15 lists the main Wikipedia tables described above. For each entity (location or university), we identify the main Wikipedia page (which by design of Wikipedia exists and is unique); the `pageId` attribute in Figure 15 is the identifier for this main page. By design of Wikipedia, the title of a page is also a good representation for the entity represented in the page. Therefore, the values of the attribute Univ.Title contains the names of our compiled list of universities. For each university record, we store another attribute called *RootName*—this attribute is not obtained from Wikipedia, but instead generated automatically using our framework; we provide more details about this attribute shortly.

Redirects and Disambiguation Pages: Wikipedia has special *redirection* and *disambiguation* pages that contain rich information about alternate representations for entities. For example, the Wikipedia page for *Calif.* automatically redirects to the main page for the state of California. Similarly, the Wikipedia page for *CA* is a disambiguation page that contains a list of possible entities that `CA` can refer to, which includes California. We use redirects for our data cleaning tasks: the table *WRed* contains the redirection relationship (page with *pageId* redirects to page with title *RTitle*).

We note that the lists that we compile are exhaustive and not specific to the affiliation dataset, meaning that they can be used without any additional effort for other data cleaning tasks such as address cleaning. Also, there are initiatives

	Title	*RTitle*
1	UC Berkeley	University of California, Berkeley
2	Calif.	California
3	Köln	Cologne

Figure 17: Example Wikipedia Redirects

such as Yago [21] and Freebase [14] aimed at compiling large amounts of structured data, which can obviate the domain knowledge compilation step.

Program

Figure 18 shows part of the rules that we use for this dataset. Rule 1 specifies that an affiliation consists of an optional department, a university name (⟨univ⟩), and a location information (⟨loc-info⟩). Both ⟨univ⟩ and ⟨loc-info⟩ contain an attribute *City*, explained in detail later; the predicate of Rule 1 checks if these attribute values are equal. The nonterminal ⟨univ⟩ also contains an attribute *Name* which is the normalized university name. The action for Rule 1 simply copies the value of this attribute to the *Name* attribute for ⟨affil⟩.

Rule 2 specifies that a string is a university name if it is main Wiki page title of a university. This rule would, for example, recognize `University of California, Berkeley` as a university name. Rule 3 specifies that a string is a university name if it is the title of a Wiki page that redirects to a university page; the action for the rule sets the normalized name of this university to be the title of redirected page. This rule would recognize that `UC Berkeley` as a university, and set the *Name* attribute to expanded form above. However, Wikipedia redirects are not comprehensive: we cannot use these rules to recognize that `Univ of Calif., Berkeley` as a representation of the same university name.

To recognize variations of university names not covered by Wikipedia redirects, we further parse university names (using rules 4-14) and identify a *root-name* (*RootName*) for a university name. Informally, root-name represents the "core" of a university name; see Table 16 for examples. Rule 5 specifies that a university name can be a word followed by a `University` (or one of its variants, see Rule 14). This would parse university names such as `Brown University`, and identify `Brown` as the root-name. Rule 6 specifies that a university name can be of the form `University of` followed by a location (a city or a state). This would parse `University of California` and identify `California` as the root-name. Rule 7 and 8 are variants of Rule 6. Using these rules, in a preprocessing step, we identify the root-name for each university name in our compiled list and store it in the attribute *Univ.RootName*.

At the time of actual processing, we recognize a string to be a university name if it can be parsed using rules 4-14 and the root-name obtained from the parsing is equal to the root-name of some university in the compiled list. The main benefit of parsing a university name comes from the fact that we can now leverage alternate representations for locations and identify alternate representations for university names. For example, using these rules, we can use the redirection from `Calif.` to `California`, to correctly parse and normalize `Univ of Calif., Berkeley` to its full form. (A different parse would also normalize this string to `University of California, San Diego`, which shares the same root-name; we comment on location-based disambiguation which elimi-

Id	*Rule*			*Predicate*	*Action*
1	⟨affil⟩	→	⟨dept⟩? ⟨univ⟩$_1$ ⟨loc-info⟩$_2$	*1.City = 2.City*	*Name = 1.Name*
2	⟨univ⟩	→	U	Univ(P,U,C,R)	*Name = U, City = C*
3	⟨univ⟩	→	U'	WRed(P1,U',U), Univ(P2,U,C,R)	*Name = U, City = C*
4	⟨univ⟩	→	⟨univ-base⟩$_1$ (',' ⟨loc⟩)?	Univ(P,U,C,R), R=*1.RootName*	*Name = U, City = C*
5	⟨univ-base⟩	→	⟨word⟩$_1$ ⟨univ-spec⟩		*RootName = 1.Value*
6	⟨univ-base⟩	→	⟨univ-spec⟩ 'OF' ⟨loc⟩$_1$		*RootName = 1.Name*
7	⟨univ-base⟩	→	⟨univ-spec⟩ ⟨loc⟩$_1$		*RootName = 1.Name*
8	⟨univ-base⟩	→	⟨loc⟩$_1$ ⟨univ-spec⟩		*RootName = 1.Name*
9	⟨loc⟩	→	⟨city⟩		*Name = 1.Name*
10	⟨loc⟩	→	⟨state⟩		*Name = 1.Name*
11	⟨city⟩	→	C	City(P,C,S)	*Name = C*
12	⟨state⟩	→	S	State(P,S,C)	*Name = S*
13	⟨state⟩	→	S	WRed(P1,S',S), State(P2,S,C)	*Name = S*
14	⟨univ-spec⟩	→	'University' | 'Univ' | 'Universität'		
15	⟨loc-info⟩	→	⟨token⟩+⟨city⟩$_1$⟨token⟩+⟨state⟩?⟨token⟩+ ⟨country⟩?		*City = 1.Name*

Figure 18: (Part of) Program for Affiliation dataset

nates such parses below.) This kind of parsing is also very useful for handling English and non-English names of universities. For example, the university name `Universität Köln` is correctly normalized to `University of Cologne` using the Wiki redirection from `Köln` to `Cologne`.

We emphasize that the rules for parsing university names need not be comprehensive. For university names that we cannot parse using these rules, we simply rely on the basic rules 2 and 3.

The non-terminal ⟨loc-info⟩ parses the location component of an affiliation that follows a university name and extracts location-specific attributes. Figure 18 illustrates the attribute *City*, but our full program also extracts attributes such as *State* and *Country* if present. Rule 1 uses this information to disambiguate universities based on location: The predicate associated with Rule 1 checks that the *City* attribute of ⟨univ⟩ (which comes from the *Univ* table) is identical to the *City* attribute of ⟨loc-info⟩. We note that this location-based disambiguation works even if the city does not immediately follow the university name (e.g., record 5 in Figure 14).

As in the case of UCD people data, the rules that we use for processing affiliations are fairly generic, suggesting that they can be generated fairly easily. For example, we parse a department (⟨dept⟩) simply as a sequence of tokens and do not attempt to model the exact structure of a department string. The rules for ⟨loc-info⟩ do not understand addresses in detail, just the fact that they possibly contain a city, state, and country; all other details are modeled as simple token sequences. As we will report shortly, we get output with acceptable quality even with these generic rules, and this is mainly due to large amounts of external domain knowledge combined with weighting.

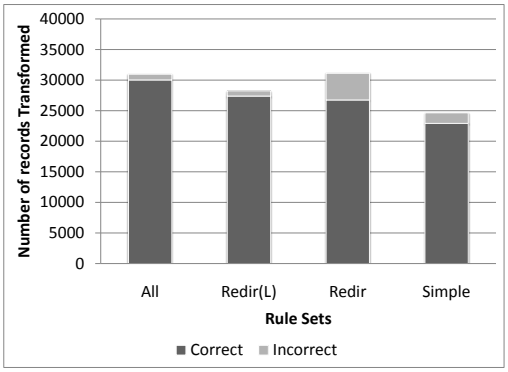

Figure 19: Coverage and Quality for Affiliations

Quality of Data Transformation

We now comment on the quality of data transformation. Recall that the dataset consists of 107000 affiliations. Based on a sample, we estimate that around 37000 of these records correspond to academic affiliations in the six countries of interest. For the purposes of evaluations we ran our framework over all the records, but disregarded those that did not belong to the subset of interest while manually evaluating quality. Notice (Figure 18) that we normalize each university to the title of the Wikipedia main page for the university. Most universities have a unique Wikipedia main pages, so this normalization is unambiguous. This also means that universities that do not have a Wikipedia page cannot be handled by the program of Figure 18, however such universities only comprise a small fraction.

Figure 19 shows the overall quality and coverage for different rule sets. The rightmost bar (*Simple*) presents results for the case where only Rule 2 is used for parsing universities. The next bar (*Redir*) considers the case where only Rules 2 and 3 are used for parsing. For these two rule sets we do not use any location details. The next bar (*Redir(L)*)

is identical to *Redir* except that it uses location information to disambiguate universities using the predicate in Rule 1. Finally, the leftmost bar (*All*) presents results for the case where all rules described earlier are used.

Using all the rules, we get a coverage of about 31000 records, which is about 84% of the estimated number of relevant records. The estimated accuracy of the output records is fairly high, around 97%. On the other hand, using just rule 2 that uses exact match on *Univ.Title* to identify a university name brings down the coverage to around 24000 records. Using Wikipedia redirects significantly improves coverage, however the quality drops significantly to about 84%. Most of the errors are due to location based ambiguity, which happens due to records such as 5 in Figure 14 where the university name is far from the city name. By extracting location details and using it for disambiguation improves the quality to 97%, with the coverage of about 28000 records.

This result indicates that Wikipedia redirects are a good source of alternate representations for university names. We observed that they are not a clean source of representations, however: For example, there exists a inexplicable redirection from `Cambridge tool and die` to the MIT main page. However, these do not seem to affect quality significantly since, for example, irrelevant strings such as the one above are unlikely to occur within affiliations.

Finally, by using more sophisticated rules that parse a university name, we improve the coverage by almost 10% to around 31000 records, while still maintaining high data transformation quality.

Record Matching

Figure 20 shows the quality of record matching using the same jaccard similarity function we used for the UCD dataset. Again, for various similarity thresholds, we compute all pairs of records that have similarity greater than the threshold and estimate the correct matches by manually evaluating a random sample. We evaluated the result over all 107000 records not just the 37000 university affiliations in the six countries. The number of matching pairs over the entire subset is an upperbound on the number of matching pairs within just the subset of interest. Figure 20 plots the estimated number of correct and incorrect matches at various thresholds. Even at a very low jaccard similarity threshold, the number of correct matching pairs identified by the similarity function approach is less than 1M. In contrast, the number of matches on the output records, obtained by a simple equi-self-join is over 2M (shown by the dotted line in Figure 20. This is more than double the number of pairs obtained using the similarity function method, although the program that we use with the framework works with just 37000 of the 107000 records.

6. DISCUSSION

We now discuss the relationship of our framework to some of the important threads of prior work in data cleaning and information extraction, while deferring a more detailed discussion of specific related work to Section 7.

Record Matching

As noted in Section 1, prior work on record matching has largely focused on similarity function design. Our case studies indicate that with the right pre-processing, the need for approximate equality when performing record matching is minimized and indeed, often eliminated. However, we do recognize that in general, there will be a need for string similarity joins to capture specific variations such as typos and misspellings. Our framework is not intended to replace this body of work. Rather, it can be used as a pre-processing step before performing vanilla record matching.

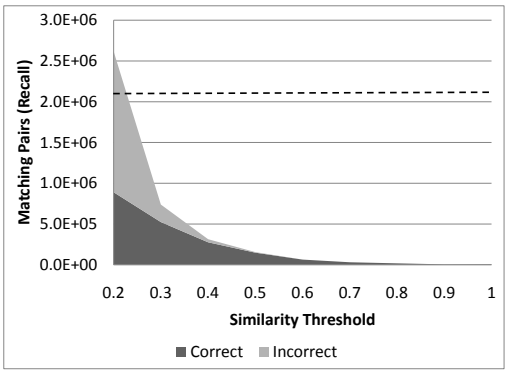

Figure 20: Affiliations: Record Matching Quality

Pay-As-You-Go

There has been substantial amount of recent work on the area of pay-as-you-go data quality management and information extraction [17, 15]. The key observation here is that our goal must not necessarily be to clean an entire data set since doing so is difficult. Rather, we "pay as we go" where we use for example reference tables that cover only a part of the data to clean a subset of the data and gradually expand our coverage as we get more reference data.

We note that our framework is designed to accommodate this viewpoint — we have fallback options such as using Σ^* for various non-terminals as illustrated in our case studies. These fallback options are useful primarily when our reference data is not comprehensive. As and when we can write more precise rules to capture the data representation, the need for such fallback options decreases.

Lineage

Our parse trees constitute a natural notion of lineage that can be used to program on top of our module. For instance, a data cleaning developer using our framework can choose to not use the rule weighting options, instead writing if-then-else logic on top to capture her parse tree preferences.

Another application of this lineage is to use it for downstream data cleaning tasks such as record matching where we can condition the matches not only on the schema of the records output by our framework, but also on the parse tree that produced them.

Uncertainty

It is widely recognized that there is a need to manage uncertainty in tasks such as data cleaning and information extraction. While there are several sources of such uncertainty, one important source is the differences in representation. We can thus view our framework as providing a tool to manage the uncertainty in the data.

Further, our framework incorporates the idea of "possible worlds". Thus, our notion of variation allows for multiple possible variations for the same entity. We also return multiple parse trees for the same input record with an accom-

panying score. This is consistent with the vast body of work on probabilistic databases. Indeed, the weights we return lend themselves to be interpreted as probabilities that can be used by a probabilistic database management system.

7. RELATED WORK

The huge body of work on data cleaning and record matching is closely related to this paper [13, 16]. As mentioned earlier, much of the work on record matching has focused on developing and using sophisticated similarity functions such as Cosine [8], FMS [7], HMM [6], Jaccard, and others. Our framework is related to the class of techniques that fall under the ETL process [19], but none of these techniques support sophisticated reasoning with entity representations like our framework.

The well-known problem of data segmentation is closely related to our work. The goal of segmentation is to partition a string into its basic components or attributes, and previous work has proposed a variety of techniques and algorithms [4, 20]. This includes work that exploits external dictionaries for more effective segmentation [9]. The segmentation technique proposed in [24] is particularly related to our work since it uses a generative grammar-based approach for segmentation. Almost all techniques for segmentation are learning-based. In contrast, we have explored our framework mostly using manually programmable rules. Exploring learning alternatives is an interesting avenue for future work.

There has been lot of recent work on exploiting Wikipedia as a source of clean domain knowledge. Yago [21] is a system for automatically extracting structured facts and relations from Wikipedia using Wordnet, and it currently consists of over 5 million clean facts. [11] exploits Wikipedia for the problem of named entity disambiguation.

8. CONCLUSIONS

This paper presented a declarative, programmatic framework for reasoning with and manipulating entity representations. The framework is programmed using context free grammar rules; the framework combines elements of database style querying to enable a programmer to easily exploit large amounts of external domain knowledge. The framework has a variety of applications in data normalization, parsing, and record matching. In particular, the framework can be used to preprocess records to simplify subsequent record matching logic.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley.

[2] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, Apr. 2008.

[3] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, Aug. 2003.

[4] V. R. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. In *SIGMOD*, pages 175–186, May 2001.

[5] K. Chakrabarti, S. Chaudhuri, V. Ganti, et al. An efficient filter for approximate membership checking. In *SIGMOD*, pages 805–818, June 2008.

[6] A. Chandel, O. Hassanzadeh, N. Koudas, et al. Benchmarking declarative approximate selection predicates. In *SIGMOD*, pages 353–364, June 2007.

[7] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, June 2003.

[8] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Trans. on Information Systems*, 18(3):288–321, July 2000.

[9] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *SIGKDD*, pages 89–98, Aug. 2004.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[11] S. Cucerzan. Large scale named entity disambiguation based on wikipedia data. In *EMNLP-CoNLL Joint Conf.*, pages 708–716, June 2007.

[12] DBLP. `http://www.informatik.uni-trier.de/~ley/db/index.html`.

[13] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowledge and Data Engg.*, 19(1):1–16, Jan. 2007.

[14] Freebase. `http://www.freebase.com/`.

[15] A. Y. Halevy, M. J. Franklin, and D. Maier. Dataspaces: A new abstraction for information management. In *DASFAA*, pages 1–2, Apr. 2006.

[16] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, pages 802–803, June 2006.

[17] J. Madhavan, S. Cohen, X. L. Dong, et al. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350, Jan. 2007.

[18] RIDDLE: Repository of information on duplicate detection, record linkage, and identity uncertainty. `http://www.cs.utexas.edu/users/ml/riddle/`.

[19] E. A. Rundensteiner. Special issue editor. *IEEE Data Engineering Bulletin*, 22(1), 1999.

[20] S. Sarawagi and W. W. Cohen. Semi-markov conditional random fields for information extraction. In *NIPS*, Dec. 2004.

[21] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from wikipedia and wordnet. *J. Web Semantics*, 6(3):203–217, Sept. 2008.

[22] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, chapter 3. Computer Science Press, Rockville, MD, 1988.

[23] U.S. census bureau. `http://www.census.gov/genealogy/names/`.

[24] P. A. Viola and M. Narasimhan. Learning to extract information from semi-structured text using a discriminative context free grammar. In *SIGIR*, pages 330–337, Aug. 2005.

[25] Wiktionary. `http://www.wiktionary.org/`.

[26] F. Wu and D. S. Weld. Automatically refining the Wikipedia Infobox ontology. In *WWW*, pages 635–644, Apr. 2008.