Refining Information Extraction Rules using Data Provenance

Bin Liu[1,*] Laura Chiticariu[2] Vivian Chu[2] H.V. Jagadish[1] Frederick R. Reiss[2]

[1]University of Michigan [2]IBM Research – Almaden

Abstract

Developing high-quality information extraction (IE) rules, or extractors, is an iterative and primarily manual process, extremely time consuming, and error prone. In each iteration, the outputs of the extractor are examined, and the erroneous ones are used to drive the refinement of the extractor in the next iteration. Data provenance explains the origins of an output data, and how it has been transformed through a query. As such, one can expect data provenance to be valuable in understanding and debugging complex IE rules. In this paper we discuss how data provenance can be used beyond understanding and debugging, to automatically refine IE rules. In particular, we overview the main ideas behind a recent provenance-based solution for suggesting a ranked list of refinements to an extractor aimed at increasing its precision, and outline several related directions for future research.

1 Introduction

As vast amounts of data are made available in pure unstructured format on the web and within the enterprise, Information Extraction (IE) – the discipline concerned with extracting structured information from unstructured text – has become increasingly important in recent years. Today, information extraction is an integral part of many applications, including semantic search, business intelligence over unstructured data, and data mashups.

In order to identify regions of text that are of interest, most IE systems make use of *extraction rules*. Rule-based systems such as CIMPLE [10], GATE [8], or SystemT [3, 16], use rules throughout the entire extraction flow. An example rule in these systems might read, in plain English, *“identify a match of a dictionary of salutations followed by a match of a dictionary of last names and mark the entire region as a candidate person”*. The rule would be formally expressed using the system's rule language such as XLog [23] in CIMPLE, JAPE [7] in GATE, or AQL [21] in SystemT. Machine learning-based systems such as [12, 17] use rules in the first stage of an extraction flow, to identify basic features such as capitalized words, or phone numbers. These basic features are fed as input to various machine learning algorithms that implement the rest of the extraction process. We refer the reader to recent tutorials [5, 6, 11] and survey [22] for more details on information extraction systems. Regardless of the kind of system, machine learning-based or rule-based, the output of extraction rules must be extremely accurate in order to minimize the negative impact on subsequent data processing steps, i.e., the rest of the extraction flow, or the application consuming the extracted information.

Copyright 2010 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*] Work partially done while at IBM Research – Almaden.

Developing highly accurate information extraction rules (or extractors, in short) is a laborious process, extremely time consuming, and error prone. The developer starts by building an initial set of rules, and subsequently proceeds with an iterative process consisting of three steps: (1) execute the rules over a set of test documents, and identify mistakes in the form of wrong results (false positives), as well as missing results (false negatives); (2) analyze the rules to understand the causes of the mistakes; and (3) finally determine refinements that can be made to the extractor to correct the mistakes. This 3-step process is repeated until the developer is satisfied with the accuracy of the extractor.

Traditional research in the field of *data provenance* [2, 24] seeks to explain the presence of a piece of data in the result of a query: *why* it is in the result, *where* it originated in the input database, and *how* it has been transformed by the query. Such kind of provenance has been recently referred to as *provenance for answers*. In contrast, recent work on *provenance for non-answers* [1, 14, 15] seeks to explain the reasons for which an expected result is *missing* from the output. Together, techniques for computing provenance for answers and respectively, non-answers, can be used to explain false positives, and respectively false negatives, therefore facilitating step 2 of the rule development process described above.

While provenance helps the developer in identifying possible causes for incorrect results (step 2), it does not directly address the challenges associated with identifying rule refinements (step 3). In practice, an extractor may contain hundreds of rules, and the interactions between these rules can be very complex. Therefore, designing a rule refinement is a "trial and error" process, in which the developer implements multiple candidate refinements and evaluates them individually in order to understand their effects: *"Are any mistakes being removed ?"*, and side-effects: *"Are any correct results affected ?"* An "ideal" refinement would eliminate as many mistakes as possible, while minimizing effects on existing correct results. Based on our experience building information extraction rules for multiple enterprise software products, designing a single such refinement may take hours. In the same spirit, refining a complex extractor when switching from one application domain to another can take weeks or even months [4, 20].

In this article we explore the question of whether provenance can be used beyond understanding the incorrect behavior of extraction rules, to automatically refine the rules, therefore facilitating step 3 of the rule development process. We start by illustrating the challenges associated with refining extraction rules using a simple motivating example (Section 2). In Section 3, we overview the main ideas behind a recently proposed provenance-based solution for suggesting a ranked list of refinements aimed at improving an extractor's precision that we have implemented in the SystemT[1] information extraction system [3, 16, 21] developed at IBM Research – Almaden. In doing so, we keep the discussion at a high-level, and refer the reader to [19] for technical details. We conclude by outlining several directions for future research in Section 4.

2 Challenges in Developing Information Extraction Rules

Figure 1(a) illustrates a simple extractor for identifying mentions of person names consisting of rules R_1 to R_5. The figure also illustrates the output of the extractor on an input document consisting of a (fictitious) news report as markup in the input text itself, as well as using a familiar tuple-based representation. At a high-level, the semantic of the extraction rules are as follows. Rules R_1 and R_2 look for mentions of first and last names by identifying matches of entries in dictionaries 'first_names.dict' and 'last_names.dict' in the input text. These mentions are used by rules R_3 and R_4 to identify candidate person names. In particular, R_3 looks for mentions of a first name followed immediately by a last name, therefore marking, for instance, `Morgan Stanley` and `John Stanley` as candidate persons. Rule R_4 identifies mentions of last names preceded by a salutation (e.g., `Mr.`, `Ms.`), therefore marking, for example, `Stanley` in "... *Mr. Stanley* ..." as candidate person. Finally, R_5 constructs the output of the extractor by unioning together the person mentions identified by R_3 and R_4.

[1] http://www.alphaworks.ibm.com/tech/systemt

Figure 1: Example Person extractor (a); an input document (b); the results of the extractor on the input document (c); provenance graph of output tuple p_2 (d).

As the reader might have gleaned from the example, the basic unit of data in the information extraction context is a region of text, called a *span*. An information extraction program takes as input a document text (represented as relation *Document* with a single span attribute *text*), and outputs a set tuples, each with one or more attributes of type span. A span is associated with begin and end offsets in the input text (see Figure 1(b)). For simplicity, however, we distinguish between spans with the same text value using unique identifiers, see for example p_2, and p_7. In specifying the extraction rules, we use SQL enriched with a few constructs for expressing text-specific operations, such as basic feature extraction (e.g., *Dictionary()* table function to specify dictionary matching), and span manipulation scalar functions (e.g., combine two spans into a larger span – *Merge()*, select the left context of a span of certain length – *LeftContext()*) and predicates (e.g., check if two spans follow one another within a certain distance – *Follows()*, check that a span satisfies a regular expression – *MatchesRegex()*). The syntax is self-explanatory and we do not insist on the details. We note that we use an SQL-like language for ease of exposition in this article. The language is similar to SystemT's AQL rule language used in our implementation, and can express most popular constructs in other IE rule languages such as JAPE or XLog.

It is easy to see that the extractor in Figure 1 suffers from low precision: while it identifies all correct person mentions, or true positives (e.g., `John Stanley`, Mr. `Stanley`), it also identifies incorrect mentions, or false negatives (e.g., `Smith Barney`, `Morgan Stanley`). Suppose a developer wishes to improve the precision of the extractor. To this end, her goal is to refine the rules in order to remove as many false positives from its output, while preserving true positives as much as possible. Table 1 lists a few of the many possible rule modifications (expressed in both English and our rule language), along with their effects and side-effects. For example, M_1 adds a predicate to R_5 that filters mentions containing the term `Stanley`. While this eliminates two of the three false positives, it also removes two correct results (i.e., `John Stanley`, and Mr. `Stanley`). Alternatively, M_2 modifies R_3 in a similar fashion with the side-effect of removing a single correct result (`John Stanley`). M_3 modifies R_2 so that `Stanley` is no longer identified as a last name, which affects both R_3 and R_4, which will fail to identify correct candidates `John Stanley` and Mr. `Stanley`, respectively. Similarly, M_4 modifies R_1 so that `Morgan` is no longer identified as first name, thus causing R_3 to fail to mark `Morgan Doe` as a candidate person. The rest of modifications do not have any side-effects: M_5 and M_5' have the same outcome as M_2, while M_6 removes `Smith Barney`, but only one of the mentions of `Morgan Stanley`. Finally, assuming the developer has available an extractor for identifying mentions of organizations, she may use it to add a subtraction on top of R_5 so that person mentions completely contained within an organization mention are subtracted from the final result. Assuming `Smith Barney` and `Morgan Stanley` are identified as organizations, M_7 results in removing all incorrect person mentions, while not affecting any correct ones.

As illustrated by our example, the developer faces several challenges when designing a rule refinement for removing false positives from the result to improve the precision of the extractor.

Challenge 1: Which rule should be modified ? In general, there are multiple rules that may be refined in order to remove a false positive. For example, as illustrated by M_1 to M_4, a modification to any of R_1, R_2, R_3 and R_5 may potentially result in removing `Morgan Stanley` from the output. However, the overall effects and side-effects of such potential modifications are very different.

Challenge 2: What kind of modification should be made ? Once a rule has been chosen for refinement, there are multiple possible classes of modifications to choose from. For example, should we add a selection predicate as in M_6, or a more complex subtraction that involves adding a new rule as in M_7 ?

Challenge 3: How should the modification be parametrized ? Within each class or rule modifications there are multiple parameters to configure. Consider for example the addition of a selection predicate. Should the

ID	*Rule refinement*	*Effect*	*Side-effect*
M_1	Filter Person if it contains the term `Stanley` Add selection predicate *Not(ContainsDict(’invalidPersonFragment.dict’, match))* to R_5	p_2, p_7	p_3, p_5
M_2	Filter Candidate$_1$ if it contains the term `Stanley` Add selection predicate *Not(ContainsDict(’invalidPersonFragment.dict’, match))* to R_3	p_2, p_7	p_3
M_3	Remove entry `Stanley` from dictionary ‘last_names.dict’	p_2, p_7	p_3, p_5
M_4	Remove entry `Morgan` from dictionary ‘first_names.dict’	p_2, p_7	p_4
$M_5(M'_5)$	Filter Person (Candidate$_1$) if it contains the term `Morgan Stanley` Add selection predicate *Not(ContainsDict(’invalidPerson.dict’, match))* to R_5 (R_3)	p_2, p_7	–
M_6	Filter (Candidate$_1$) if right context contains organization clue, e.g., `Co` Add selection predicate *MatchesDict(’orgClue.dict’,RightContextTok(L.match)))* to R_3	p_1, p_7	–
M_7	Identify person mentions contained within organization mentions; Subtract them from Person	p_1, p_2, p_7	–

Table 1: Potential rule refinements for the extractor in Figure 1, along with their effects and side effects.

predicate be applied to the match itself as in M_5, or the context as in M_6, and if so, what is the length of the context? Furthermore, should the predicate be based on a regular expression (such as in rule R_4), or a dictionary (e.g., M_5, M_6)? Which regular expression or dictionary is most appropriate?

Although not illustrated by our example, the developer faces similar challenges when designing refinements to address false negatives, with the goal of improving recall. Furthermore, the space of refinements increases dramatically as we move from a simple example to a real-world extractor consisting of hundreds of rules.

3 A Provenance Approach to Automatic Rule Refinement

Clearly, it is not feasible for a rule developer to consider all possible refinements. In general, in each iteration of the development process, an experienced developer focuses on a few mistakes that exhibit some commonality. She designs several refinements by considering a few alternatives of rule, type of refinement and parameters, evaluates each refinement individually, and choose one that she finds most appropriate in addressing the target mistakes. Finally, she repeats the process, usually targeting a different class of mistakes. As the amount of human effort required in this “trial and error” process is considerable, an interesting question is whether it is possible to automate some (if not all) of the above steps. In the rest of this article we discuss how provenance enables an approach for systematically and efficiently exploring a large space of possible refinements, and computing a list of “most effective” refinements to be presented to the rule developer.

Provenance Graph. While examining the result of an extractor on a collection of documents, the rule developer records information on the results’ accuracy. Let us assume that she labels the extracted results as correct (i.e., true positives) or incorrect (i.e., false positives). With this information, we can leverage provenance directly in order to systematically address Challenge 1, i.e., *which rule to modify?* In particular, we are interested in a provenance model that captures *how* an output tuple has been generated by the extraction program from the input document. Such a provenance representation can be obtained by rewriting the extraction program to record, for each tuple generated by the system, the source or intermediate tuples *directly* responsible for generating that tuple. The additional information obtained via the rewritten program can be used to construct a *provenance graph*[2] that illustrates how source tuples and intermediate tuples have been combined by operators

[2] The provenance graph can be seen as a graphical representation of an instantiation of *provenance semirings* [13] in which each source (document) tuple is tagged with a unique id, and each derived tuple is tagged with an expression over a semiring having the set

of the extraction program in order to generate each of the output tuples. Similar to the representation of SQL using relational algebra operators, we assume a canonical representation of an extraction program as a DAG of operators (e.g., the SPJUD relational operators and text-specific operators such as *Dictionary*). However, in the context of information extraction, none of these operators are duplicate removing, as deduplication is explicitly applied at certain stages of the pipeline using a special consolidation operator. We do not consider this operator in this article, and therefore each output tuple has a single derivation. For example, the portion of the provenance graph corresponding to the derivation of the false positive p_2: `Morgan Stanley` is shown in Figure 1(d).

High-Level Changes. In order to remove a false positive from the final result, it is sufficient to remove any edge in its provenance graph, since each tuple has a single derivation. Therefore, by examining the provenance of each false positive, we can compile a list consisting of triplets of the form (Op, $intermediate\_result$, $false\_positive$), representing the fact that modifying operator Op in order to remove $intermediate\_result$ from its output causes $false\_positive$ to disappear from the result of the extractor. We call such triplets *high-level changes*. For example, the following high-level changes would be among those generated for p_2: ($Dictionary_1$, f_2, p_2), ($Dictionary_2$, l_2, p_2), ($\bowtie_3$, c_2'', p_2), (σ_3, c_2', p_2), $\ldots$, ($\cup_5$, p_2, p_2). Therefore, the choice of which rule to modify in order to remove p_2 from the output is narrowed down to R_1 - R_3 and R_5.

Low-Level Changes. A high-level change indicates which rule may be refined, but it does not tell us *how* to refine it (i.e., Challenges 2 and 3). We refer to a specific refinement that implements one or more high-level changes as a *low-level change*. For example, M_1 is a possible low-level change for the high-level change ($Dictionary_1$, f_2, p_2). It is easy to see that any high-level change can be trivially translated into a low-level change that removes exactly one false positive (e.g., M_5 and M_5'). However, such refinements are not useful in practice, as they do not generalize well to unseen documents. Intuitively, we would prefer non-trivial refinements (e.g., M_6 and M_7) that remove multiple false positives at a time, and thus have the potential to be effective in general. It turns out that high-level changes are useful towards addressing this problem. Specifically, grouping high-level changes by operator Op produces a maximal set of false positives that can be affected by any refinement of Op, along with the specific outputs that should be removed from the output of Op in order to achieve that maximal effect. For example, the three high-level changes for operator π_3 of R_3: (π_3, c_1, p_1), (π_3, c_2, p_2) and (π_3, c_5, p_7) indicate that modifying π_3 to remove c_1, c_2 and c_5 from its output results in eliminating all false positives p_1, p_2 and p_7. With this information, it is now possible to devise an algorithm for efficiently exploring a large space of types of refinements, as well as algorithms for parametrizing specific types of refinements, while not overfitting to the training document collection.

Given the set of high-level changes grouped by operator Op, our algorithm for generating a set of top-k "promising" refinements to be presented to the rule developer proceeds in three steps. First, for each group of high-level changes, we consider different types of low-level changes that can be applied to Op. For example, in the case of the $Dictionary$ operator we may consider removing some entries from the dictionary (e.g., as in M_3), or changing the matching mode from case insensitive to case sensitive. For the select operator, we may consider adding a selection predicate (e.g., as in M_6). In order to be able to evaluate a refinement's quality, the types of refinements considered are carefully selected to guarantee restricting the final result set. Indeed, it is impossible to evaluate a refinement that introduces new tuples in the extraction result, as we have no knowledge of whether these new tuples are correct or not. Second, for each Op and each type of low-level change, we generate k most promising sets of parameters for that low-level change. At the same time, we also compute the effects and side-effects of each concrete low-level change. Finally, we rank the low-level changes generated based on the quality improvement they achieve and present the top-k to the user.

In generating candidate sets of parameters, we apply algorithms specifically designed for each type of low-level change. Before discussing an example of such an algorithm, we briefly overview the main ideas underlying low-level change evaluation, which are shared by all our specific algorithms. A naive evaluation approach would

of all source tuple ids as domain, and equipped with operations $\cdot$ and $+$ for combining the provenance of tuples involved in a join, and respectively, when a tuple can be derived in multiple ways, and a unary function for representing the operator involved in the derivation.

be to simply implement each low-level change and execute the refined extractor. However, this is wasteful and would significantly affect the response time of the system, as the number of refinements considered in a single iteration might be in the order of thousands. A better approach is to traverse the provenance graph and record a mapping between each output tuple and each of the intermediate tuples involved in its provenance. By reverting this mapping we can obtain, for each intermediate tuple t in the result of some operator Op, the set of output tuples depending on it. Since each tuple has a single derivation tree, modifying Op to remove a tuple t from its local output guarantees the removal of all tuples depending on t from the final result. Furthermore, since each refinement can only restrict the final result, we also know the labels of all these tuples. Therefore, we can determine the effects and side-effects of the refinement on overall result quality by unioning this information for all tuples removed by the refinement from the local output of Op.

A detailed discussion of specific algorithms for parametrizing low-level changes is outside the scope of this article. To give the reader a flavor, however, we briefly describe the algorithm for parametrizing filter selection predicates (as in M_6) next. The algorithm systematically considers the tokens to the left or right of each span in a tuple affected by a high-level change. For each choice of left/right context of a given length up to a preset maximum value, the union of the contextual tokens forms a set of potential dictionary entries. For example, in the case of operator σ_3 of R_3, the tokens `Co` and `executives` would be considered as potential dictionary entries for a filtering predicate on the right context of length 1. We now need to evaluate the effects of using various subsets of these dictionary entries to filter the local output of σ_3. However, the number of all subsets can be prohibitively large. To avoid evaluating all of them, we adopt a greedy approach: we group together tuples according to which dictionary entries occur in the vicinity of their spans, evaluate the effect of each entry on result quality using the approach described above, and generate k refinements, each using top-k most effective entries (k is the maximum number of refinements that will be presented to the developer). Two refinements would be generated for our example, one using `Co`, while the other using both `Co` and `executives` as filter.

4 Conclusion and Directions for Future Research

We discussed some of the challenges in designing high-quality information extraction rules, and showed that provenance plays an important role in facilitating rule development that goes far beyond the general consensus that *"provenance is useful in understanding the program and its output"*. In particular, we discussed the main ideas behind a framework for automatically generating rule refinements aimed at improving the precision of an extractor based on user labeled examples. Very recently, the idea of extractor refinement based on provenance and labeled examples has been also approached in [9] in the context of iterative IE systems, however, refinements are considered only for the last step of an extraction flow, and not an entire extraction program as we do here.

Our approach is only a first step towards a general framework for automatic refinement of information extraction rules. We conclude by outlining several possible extensions.

Precision-driven Refinement. Besides the *provenance* $\rightarrow$ *high-level change* $\rightarrow$ *low-level change* paradigm, the usefulness of the framework lies in efficient algorithms for generating concrete refinements. While the framework implements several such algorithms, more could be added. In particular, incorporating existing approaches for learning basic features such as regular expressions [18] would be extremely useful. Furthermore, the assumption made in Section 3 is that an output tuple has a single derivation. Therefore, removing any single intermediate result involved in its provenance is sufficient to eliminate that output tuple. Formalizing the notion of high-level change in the case of multiple derivations, and designing efficient algorithms for generating concrete refinements in this context are interesting directions for future work.

Recall-driven Refinement. Our current framework handles refinements geared towards improving precision. However, equally important is improving the recall, i.e., generating refinements that remove false negatives, while not affecting existing correct results. Towards this end, it would be interesting to extend our framework to incorporate recent techniques for computing provenance of missing answers [1, 14, 15]. The notion of high-level change, and algorithms for efficiently generating low-level changes need to be revisited in this context.

Active Learning of Labeled Examples. The ability to automatically generate useful refinements depends on the number, variety and accuracy of the labeled examples provided to the system. While labeled data for certain types of popular extractors on general-purpose domains is publicly available [26], in most cases the labeled data must be provided by the rule developer. Unfortunately, labeling data is itself a tedious, time-consuming and error prone process. It would be interesting to investigate whether active learning techniques [25] can be used to present to the developer only those most informative examples, therefore facilitating the labeling process.

References

[1] A. Chapman and H. V. Jagadish. Why Not ? In *SIGMOD*, pages 523–534, 2009.

[2] J. Cheney, L. Chiticariu, and W. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[3] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An Algebraic Approach to Declarative Information Extraction. In *ACL*, 2010.

[4] L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, and S. Vaithyanathan. Domain Adaptation of Rule-based Annotators for Named-Entity Recognition Tasks. In *EMNLP (To appear)*, 2010.

[5] L. Chiticariu, Y. Li, S. Raghavan, and F. R. Reiss. Enterprise Information Extraction: Recent Developments and Open Challenges. In *SIGMOD (Tutorial)*, pages 1257–1258, 2010.

[6] W. Cohen and A. McCallum. Information Extraction and Integration: An Overview. *KDD(Tutorial)*, 2003.

[7] H. Cunningham. JAPE: a Java Annotation Patterns Engine. Research Memorandum CS – 99 – 06, University of Sheffield, May 1999.

[8] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *ACL*, 2002.

[9] A. Das Sarma, A. Jain, and D. Srivastava. I4E: Interactive Investigation of Iterative Information Extraction. In *SIGMOD*, pages 795–806, 2010.

[10] A. Doan, J. F. Naughton, R. Ramakrishnan, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong. Information Extraction Challenges in Managing Unstructured Data. *SIGMOD Record*, 37(4):14–20, 2008.

[11] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing Information Extraction: State of the Art and Research Directions. In *SIGMOD (Tutorial)*, pages 799–800, 2006.

[12] D. Freitag. Multistrategy learning for information extraction. In *ICML*, 1998.

[13] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.

[14] M. Herschel and M. Hernandez. Explaining Missing Answers to SPJUA Queries. *PVLDB (to appear)*, 2010.

[15] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[16] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: a System for Declarative Information Extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[17] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[18] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *EMNLP*, pages 21–30, 2008.

[19] B. Liu, L. Chiticariu, V. Chu, H. V. Jagadish, and F. R. Reiss. Automatic Rule Refinement for Information Extraction. *PVLDB (to appear)*, 2010.

[20] D. Maynard, K. Bontcheva, and H. Cunningham. Towards a semantic extraction of Named Entities. In *RANLP*, 2003.

[21] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An Algebraic Approach to Rule-Based Information Extraction. In *ICDE*, pages 933–942, 2008.

[22] S. Sarawagi. Information Extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[23] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.

[24] W. C. Tan. Provenance in Databases: Past, Current, and Future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.

[25] C. Thompson, M. Califf, and R. Mooney. Active Learning for Natural Language Parsing and Information Extraction. In *ICML*, pages 406–414, 1999.

[26] E. F. Tjong Kim Sang and F. De Meulder. Introduction to the CoNLL-2003 Shared Task: Language-independent Named Entity Recognition. In *CoNLL at HLT-NAACL*, pages 142–147, 2003.