Automatic Rule Refinement for Information Extraction

Bin Liu
University of Michigan
binliu@umich.edu

Laura Chiticariu
IBM Research - Almaden
chiti@us.ibm.com

Vivian Chu
IBM Research - Almaden
chuv@us.ibm.com

H.V. Jagadish
University of Michigan
jag@umich.edu

Frederick R. Reiss
IBM Research - Almaden
frreiss@us.ibm.com

ABSTRACT

Rule-based information extraction from text is increasingly being used to populate databases and to support structured queries on unstructured text. Specification of suitable information extraction rules requires considerable skill and standard practice is to refine rules iteratively, with substantial effort. In this paper, we show that techniques developed in the context of data provenance, to determine the lineage of a tuple in a database, can be leveraged to assist in rule refinement. Specifically, given a set of extraction rules and correct and incorrect extracted data, we have developed a technique to suggest a ranked list of rule modifications that an expert rule specifier can consider. We implemented our technique in the *SystemT* information extraction system developed at IBM Research – Almaden and experimentally demonstrate its effectiveness.

1. INTRODUCTION

Information extraction — the process of deriving structured information from unstructured text — is an important aspect of many enterprise applications, including semantic search, business intelligence over unstructured data, and data mashups. The structured data that information extraction systems produce often feed directly into important business processes. For example, an application that extracts person names from email messages might load this name information into a search index for electronic legal discovery; or it may use the name to retrieve employee data for help desk problem determination. Because the outputs of information extraction are so closely tied to these processes, it is essential that the extracted information have very high precision and recall; that is, the system must produce very few false positive or false negative results.

Most information extraction systems use rules to define important patterns in the text. For example, a system to identify person names in unstructured text would typically contain a number of rules like the rule in Figure 1. The example in the figure is written in English for clarity; an information extraction would typically use a rule language such as AQL [22], JAPE [12], or XLog [6, 30].

In some systems, the outputs of these rules may feed directly into applications [12, 14, 19, 24]. Other systems use rules as the feature extraction stage of various machine learning algorithms, as in [15, 23, 26]. In either case, it is important for the rules to produce very accurate output, as downstream processing tends to be highly sensitive to the quality of the results that the rules produce.

If a match of a dictionary of common first names occurs in the text, followed immediately by a capitalized word, mark the two words as a "candidate person name".

Figure 1: An example information extraction rule, in English.

Developing a highly accurate set of extraction rules is difficult. Standard practice is for the developer to go through a complex iterative process: First, build an initial set of rules; then run the rules over a set of test documents and identify incorrect results; then examine the rules and determine refinements that can be made to the rule sets to remove incorrect results; and finally repeat the process. Of these steps, the task of identifying rule refinements is by far the most time-consuming. An extractor can easily have hundreds of rules, and the interactions between these rules can be very complex. When changing rules to remove a given incorrect result, the developer must be careful to minimize the effects on existing correct results. In our experience building information extraction rules for multiple enterprise software products, we found that identifying possible changes for a single false positive result can take hours.

In the field of data provenance, techniques have been developed to trace the lineage of a tuple in a database through a sequence of operators. This lineage also encodes the relationships between source and intermediate result tuples and the final result. In this paper, we bring these techniques to bear on the problem of information extraction rule refinement. Intuitively, given a false positive result of information extraction, we can trace its lineage back to the source to understand exactly why it is in the result. Based on this information, we can determine what possible changes can be made to one or more operators along the way to eliminate the false positive, without eliminating true positives. Actually realizing this vision, the central contribution of this paper, requires addressing some challenges, as outlined in Section 4.

Most information extraction rules can be translated into relational algebra operations. Over such an operator graph, provenance-based analysis, developed in Section 5, produces a set of proposed rule changes in the form of "remove tuple t from the output of operator O". We refer to this class of rule changes as *high-level changes*. To remove a "problem" tuple from the output of a rule, the rule developer needs to know how to modify the extraction primitives that make up the rule. We call such changes *low-level changes*. (Extraction primitives include regular expressions and filtering predicates like "is followed by"). These modifications may

Proceedings of the VLDB Endowment, Vol. 3, No. 1

in turn result in the removal of additional tuples besides the "problem" tuple, and the developer needs to consider these side-effects in evaluating potential rule changes, while simultaneously keeping the rules as simple and easy to maintain as possible.

In Sections 5 and 6, we develop a framework for enumerating the low-level changes that correspond to a given set of high-level changes. We also develop efficient algorithms for computing the additional side-effects of each proposed low-level change. Using this information, we then rank low-level changes according to how well they remove false positives without affecting existing correct answers or complicating the rule set. This ranked list of low-level changes is then presented to the rule developer.

We have embodied these ideas in a software system that automates the rule refinement process and implemented it in the SystemT[1] information extraction system [11, 22, 27]. Given a set of rules, a set of false positive results that the rules produce, and a set of correct results, our system automatically identifies candidate rule changes that would eliminate the false positives. The system then computes the overall effects of these changes on result quality and produces a ranked list of suggested changes that are presented to the user. The system can be also used in fully automated mode, where the highest ranked change is automatically applied in each iteration. We have extensively evaluated the system, and present representative results to demonstrate its effectiveness in Section 7.

We begin with a discussion of related work in Section 2, and preliminaries in Section 3.

2. RELATED WORK

The field of **data provenance** studies the problem of explaining the existence of a tuple in the output of a query. A recent survey [10] overviews various provenance notions for explaining *why* a tuple is in the result, *where* it was copied from in the source database, and *how* it was generated by the query. It is the latter type of provenance, *how-provenance* [17], that is leveraged in our system to generate the set of high-level changes: place-holders in the rule set where a carefully crafted modification may result in eliminating one false positive from the output. However, this is only the first step of our approach. In a significant departure from previous work on data provenance, our system generates a ranked list of concrete rule modifications that remove false positives, while minimizing the effects on the rest of the results and the structure of the rule set.

Early work in **information extraction** produced a number of rule-based information extraction systems based on the formalism of cascading regular expression grammars. Examples include FRUMP [14], CIRCUS [24], and FASTUS [19]. The Common Pattern Specification Language [5] provided a standard way to express these grammars and served as the basis for other rule-based systems like JAPE [12] and AFsT [8]. In recent years the database community has developed other rule languages with syntaxes based on SQL [21, 22, 32] and Datalog [6, 30]. The techniques that we describe in this paper can be used to automate the rule refinement process across all these different classes of rule languages.

Other work has used machine learning to perform information extraction, and a variety of systems of different flavors have been developed, ranging from entity relation detection (e.g., [36]) to iterative IE (e.g., Snowball [4]) and open IE (e.g., TextRunner [35]). Researchers have employed a variety of techniques, including covering algorithms [31], conditional random fields [23, 26], support-vector machines [36], and mixtures of multiple learning models [15, 35]. The work that we describe in this paper is complementary

[1] Available for download at http://alphaworks.ibm.com/tech/systemt.

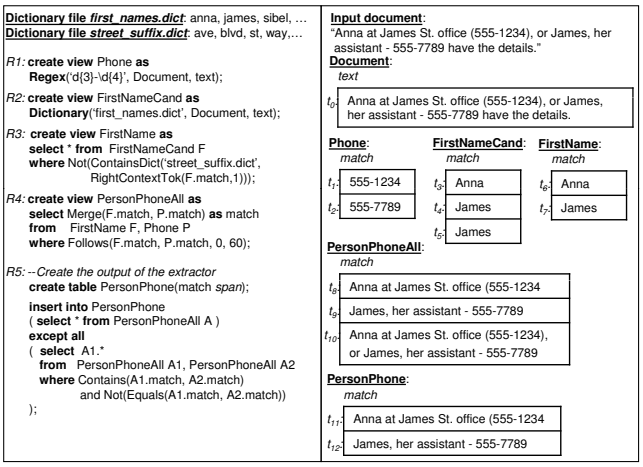

Figure 2: Example extraction program, input document *D*, and view instances created by the extraction program on *D*.

to this previous work. Our system employs a semi-automatic iterative process with a human in the loop, which represents a new area of the design space for information extraction systems. This design choice allows our system to handle highly complex rule structures and to leverage expert input. Whereas machine learning models are generally opaque to the user, the rules that our system produces can be understood and "debugged" by the rule developer.

Recently, [13] has shown how introducing transparency in a machine learning-based iterative IE system, by recording each step of the execution, enables the automatic refinement of the machine learning model via adjusting weights and removing problematic seed evidence. Our work differs from [13] in that we consider automatic refinement in the context of rule-based systems, and therefore our space of refinements is completely different.

In practice, information extraction systems that employ machine learning generally use rules to extract basic features that serve as the input, and our techniques can be used to assist in the process of developing these rules. Additional previous work has used machine learning for extraction subtasks like creating dictionaries [28] and character-level regular expressions [25]. These techniques are complementary to the work we describe in this paper. In particular, our work provides a mechanism for "plugging in" these algorithms as low-level change generation modules.

Finally, [29] describes an approach for refining an extraction program by posing a series of questions to the user. Each question asks for additional information about a specific feature of the desired extracted data. The features considered are pre-defined. For each question, the corresponding selection predicate is added to the extraction program. Our work differs fundamentally from the approach of [29] in that it automatically suggests fully-specified rule refinements based on labeled extracted data, as opposed to asking the user to fill in the blanks in template questions. Furthermore, we consider a much broader space of refinements ranging from adding/modifying selection/join predicates and dictionary extraction specifications, to adding subtraction sub-queries. To the best of our knowledge, ours is the first system for suggesting concrete rule refinements based on labeled extracted data.

3. PRELIMINARIES

Different information extraction systems have different rule languages [6, 12, 22, 30]. However, most rule languages in common

use share a large set of core functionality. In this paper, we use SQL for expressing information extraction rules in order to describe the theory behind our system in a way that is consistent with previous work on data provenance. Specifically, we use the SELECT - PROJECT - JOIN - UNION ALL - EXCEPT ALL subset of SQL. Note that UNION ALL and EXCEPT ALL are not duplicate-removing, as per the SQL standard [1].

Our use of SQL does not in any way preclude the application of our work to other rule languages. As discussed in Appendix A, the basic structure of different IE rule languages contains key similarities to the SQL representation used here. These languages define the extractor as a set of rules with dependency relationships that can be used to construct a provenance graph for computing high-level changes. Rules are made up of atomic operations that can be modified, added, or deleted to create low-level changes. As such, the high-level/low-level change framework that we define in this paper carries over easily to the rule languages in common use today.

Extensions to SQL. To make our examples easier to read, we augment SQL with some basic information extraction primitives.

We add a new atomic data type called $span$ for modeling data values extracted from the input document. A $span$ is an ordered pair $\langle begin, end\rangle$ that identifies the region of an input document between the *begin* and *end* offsets. For clarity, we may sometimes identify a $span$ using its string value in addition to the begin and end offsets, or we may simply drop the offsets when they are clear from the context. For example, to identify the region starting at offset 0 and ending at offset 5 in the document from Figure 2, we may use the notations $\langle 0, 5\rangle$, or $\langle 0, 5\rangle$: "Anna", or simply, "Anna".

We model the input document as a table called Document with a single attribute of type $span$ named *text*. We also add several predicates, scalar functions, and table functions to SQL's standard set of built-in functions. We define these functions as we use them, and also include a complete list in Appendix B.

Example Rules. Figure 2 shows an example rule program, expressed in SQL, which extracts occurrences of person names and their phone numbers. The program consists of individual rules, labeled R_1 through R_5. Rules R_1 through R_4 define logical views, while rule R_5 materializes a table of extraction results.

Rule R_1 illustrates one of the shorthands that we add to SQL: the *Regex* table function, which evaluates a regular expression over the text of one or more input spans, and returns a set of output spans that mark all matches of the expression. In the case of rule R_1, the regular expression finds phone numbers of the form $xxx - xxxx$.

Rule R_2 shows another addition to SQL: the *Dictionary* table function. Similar to the *Regex* table function, *Dictionary* identifies all occurrences of a given set of terms specified as entries in a dictionary file. For R_2, the dictionary file contains a list of common first names. The rule defines a single-column view *FirstNameDict* containing a span for each dictionary match in the document.

Rule R_3 uses a filtering dictionary that matches abbreviations for street names on the right context of names, to filter out first names that are street names, e.g., "James St.". The view definition uses two of the scalar functions that we add to SQL: *RightContextTok* and *ContainsDict*. *RightContextTok* takes a span and a positive integer n as input and returns the span consisting of the first n tokens to the right of the input span. The *ContainsDict* function, used here as a selection predicate, takes a dictionary file and a span and returns *true* if the span contains an entry from the dictionary file.

Rule R_4 identifies pairs of names and phone numbers that are between 0 and 60 characters apart in the input document. The view definition uses two of the scalar functions that we add to SQL: *Follows* and *Merge*. The *Follows* function, used here as a join predicate, takes two spans as arguments, along with a minimum and maximum character distance. This function returns *true* if the spans are within the specified distance of each other in the text. The *Merge* function takes a pair of spans as input and returns a span that exactly contains both input spans. The `select` clause of R_4 uses *Merge* to define a span that runs from the beginning of each name to the end of the corresponding phone number.

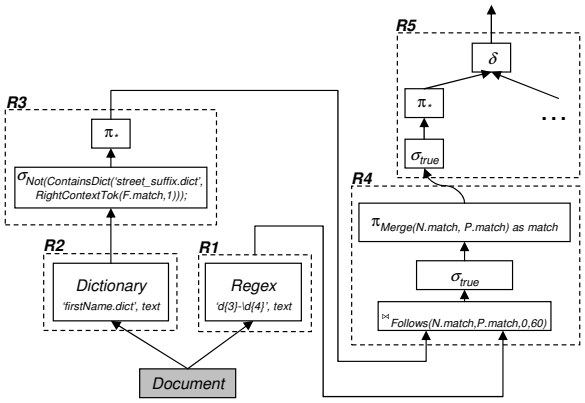

Figure 3: Canonical representation of rules in Figure 2.

Finally, R_5 materializes the table PersonPhone, which constitutes the output of our extractor. It uses an EXCEPT ALL clause to filter out candidate name–phone spans strictly containing another candidate name–phone span. The join predicate of the second operand of the EXCEPT ALL clause illustrates two other text-based scalar functions: *Equals*, which checks if two spans are equal, and *Contains*, which tests span containment. Note that the false positive t_{10} in PersonPhoneAll that associates Anna with James' phone number is filtered out by R_5, since its span strictly contains other candidate name-phone spans (i.e., from t_8 and t_9).

Canonical rule representation. To simplify our subsequent discussions, we shall assume a canonical algebraic representation of extraction rules as trees of operators, such that for each rule, there is a direct one-to-one translation to this canonical representation and back. The canonical representation is very similar, if not identical for the SELECT - FROM - WHERE - UNION ALL - EXCEPT ALL subset of the language, to the representation of SQL statements in terms of relational operators. A rule in the form "SELECT $attributes$ FROM $R_1, \ldots, R_n$ WHERE $join\_predicates$ AND $selection\_predicates$" is represented in the usual way as the sequence of project – select – join operators shown below:

$$\pi_{attributes}(\sigma_{selection\_preds}(\bowtie_{join\_preds}(R_1, \ldots, R_n)))$$

When table functions like *Dictionary* and *Regex* appear in the FROM clause of a SELECT statement, we translate these table functions to operators by the same names.

Figure 3 illustrates the canonical representation of our example extractor from Figure 2, where the dashed rectangles indicate the correspondence between parts of the operator tree and rule statements. (The part corresponding to the second operand of the EXCEPT ALL clause in rule R_5 is omitted.) Note that when the WHERE clause of a rule does not contain any selection predicates (e.g., R_4), the condition in the select operator of the corresponding canonical representation is simply $true$.

4. OVERALL FRAMEWORK

Given a set of examples in the output of an extractor, each labeled correct or incorrect by the user, our goal is to generate a ranked list

of possible changes to the rules that result in eliminating the incorrect examples from the output, while minimizing the effects on the rest of the results, as well as the rules themselves. Our solution operates in two stages: High-level change generation (Section 5) and low-level change generation (Section 6).

The high-level change generation step generates a set of *high-level changes* of the form "*remove tuple* t *from the output of operator* Op *in the canonical representation of the extractor*". Intuitively, removing a tuple t from the output of rule R translates to removing certain tuples involved in the *provenance of* t *according to the canonical operator tree of* R. Our solution leverages previous work in the *data provenance* [10] in generating the list of high-level changes. These high-level changes have the potential to remove all incorrect examples from the output. For example, high-level changes for removing the tuple t_{10} from the output of rule R_4 would be "*remove tuple* $t_{10} : (Anna, 555 - 7789)$ *from the output of the join operator in rule* R_4", or "*remove tuple* $t_3 : (Anna)$ *from the output of the* Dictionary *operator in rule* R_2".

A high-level change indicates *what* operator to modify to remove a given tuple from the final output. However, a high-level change does not tell *how* to modify the operator in order to remove the offending tuple. High-level changes are only the first step towards automating the rule refinement process.

If a rule developer were presented with a set of high-level changes, he or she would need to overcome two major problems in order to translate these high-level changes into usable modifications of the information extraction rule set.

The first problem is one of *feasibility*: The rule writer cannot directly remove tuples in the middle of an operator graph; she is restricted to modifying the rules themselves. It may not be possible to implement a given high-level change through rule modifications, or there may be multiple possible ways to implement the change. Suppose that the Dictionary operator in our example has two parameters: The set of dictionary entries and a flag that controls case-sensitive dictionary matching. There are at least two possible implementations of the second high-level change described above: Either remove the entry `anna` from the dictionary, or enable case-sensitive matching. It is not immediately obvious which of these possibilities is preferable.

The second problem is one of *side-effects*. A single change to a rule can remove multiple tuples from the output of the rule. If the rule developer chooses to remove the dictionary entry for `anna`, then every false positive that matches that entry will disappear from the output of the Dictionary operator. Likewise, if he or she enables case-sensitive matching, then every false positive match that is not in the proper case will disappear. In order to determine the dependencies among different high-level changes, the rule developer needs to determine how each high-level change could be implemented and what are the effects of each possible implementation on other high-level changes.

Just as modifying a rule to remove one false positive result can simultaneously remove another false positive result, this action can also remove one or more *correct* results. There may be instances in the document set where the the current set of rules correctly identifies the string "Anna" as a name. In that case, removing the entry `anna` from the dictionary would eliminate these correct results. A given implementation of a high-level change may actually make the results of the rules worse than before.

In the second step of our solution, we go beyond the work done in data provenance and show how to address the issues of feasibility and side-effects. We introduce the concept of a *low-level change*, a specific change to a rule that implements one or more high-level changes. Example low-level changes implementing the two high-level changes above are "*Modify the maximum character distance of the* Follows *join predicate in the join operator of rule* R_4 *from* 60 *to* 50", and "*Modify the* Dictionary *operator of rule* R_2 *by removing entry* `anna` *from dictionary file first_names.dict*", respectively.

Rather than presenting the user with a large and rather unhelpful list of high-level changes, our system produces a ranked list of low-level changes, along with detailed information about the effects and side-effects of each one. Logically speaking, our approach works by generating all low-level changes that implement at least one high-level change; then computing, for each low-level change, the corresponding set of high-level changes. This high-level change information is then used to rank the low-level changes.

A naive implementation of this approach would be prohibitively expensive, generating huge numbers of possible changes and making a complete pass over the corpus for each one. We keep the computation tractable with a combination of two techniques: pruning individual low-level changes using information available at the operator level and computing side-effects efficiently using cached provenance information.

Since low-level changes are expressed in terms of our internal representation as canonical operator trees, we translate them back to the level of rule statements (there is a direct one-to-one translation), prior to showing them to the user. For instance, our two example low-level changes would be presented to the user as '*Modify the maximum character distance of the* Follows *join predicate in the WHERE clause of rule* R_4 *from* 60 *to* 50", and respectively, "*Modify the input of the* Dictionary *table function of rule* R_2 *by removing entry* `anna` *from input dictionary file first_names.dict*." The user chooses one change to apply, and the entire process is then repeated until the user is satisfied with the resulting rule set.

5. GENERATING HIGH-LEVEL CHANGES

DEFINITION 5.1 (HIGH-LEVEL CHANGE). *Let* t *be a tuple in an output table* V. *A* high-level change for t *is a pair* (t', Op), *where* Op *is an operator in the canonical operator graph of* V *and* t' *is a tuple in the output of* Op *such that eliminating* t' *from the output of* Op *by modifying* Op *results in eliminating* t *from* V.

Intuitively, for the removal of t' from the output of Op to result in eliminating t from the final output, it must be that t' *contributes* to generating t. In other words, t' is involved in the *provenance of* t according to the rule set. Hence, to generate all possible high-level changes for t, we first need to compute the provenance of t. Next, we shall first discuss how provenance is computed in our system, and then describe our algorithm for generating high-level changes.

5.1 Computing Provenance

Various definitions have been proposed for describing the *provenance of a tuple* t *in the result of a query* Q: *why-provenance*: the set of source tuples that contribute to the existence of t in the result, *where-provenance*: the locations in the source database where each field of t has been copied from, and *how-provenance*: the source tuples, and how they were combined by operators of Q to produce t. Among these, how-provenance is the more complete version, since it generalizes why-provenance, and "contains" where-provenance in a certain sense [10]. It is also the most suitable in our context, since knowing which source tuples and how they have been combined by Q to generate an undesirable output tuple t is a pre-requisite to modifying Q in order to remove t from the result. Therefore, in this paper we shall rely on how-provenance extended to handle text-specific operators (e.g., *Regex*, *Dictionary*).

Given a set of rules Q and input document collection D, a conceptual procedure for computing how-provenance at the level of

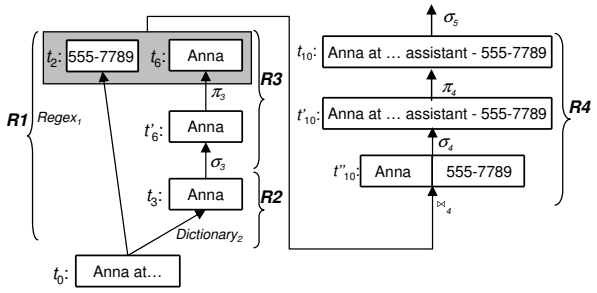

Figure 4: Provenance of tuple t_{10} from Figure 2.

the operator graph of Q is as follows. Each tuple passing through the operator graph (i.e., source , intermediate, or output tuple) is assigned a unique identifier. Furthermore, each operator "remembers", for each of its output tuples t, precisely those tuples in its input responsible for producing t. This procedure can be thought of as constructing a *provenance graph for Q on D* that contains an edge $\{t_1, \ldots, t_n\} \xrightarrow{Op} t$ for each combination $\{t_1, \ldots, t_n\}$ of input tuples to operator Op, and their corresponding output tuple t. This provenance graph essentially embeds the provenance of each tuple t in the output of Q on D. As an example, Figure 4 shows the portion of the provenance graph for our example in Figure 2 that embedds the provenance of tuple t_{10}. A procedural definition for the notion of provenance graph is given in Appendix C.

In computing the provenance graph, we use a query rewrite approach similar to [16]. The approach of [16] is to rewrite an SQL query Q into a *provenance query* Q^p by recursively rewriting each operator Op in the relational algebra representation of Q. The rewritten version preserves the result of the original operator Op, but adds additional *provenance attributes* through which information about the input tuples to Op that contributed to the creation of an output tuple is propagated. Given Op and a tuple t in its output, the additional information is sufficient to reconstruct exactly those tuples in the input of Op that generated t. Conceptually, the provenance query Q^p records the flow of data from input to output of Q, thus essentially computing the provenance graph of Q for the input document collection. The implementation of our system extends the rewrite approach of [16] to handle text-specific operators. The extensions are straightforward and details are omitted.

5.2 Generating High-Level Changes

Given a set of rules Q, an input document collection D and a set of false positives in the output of Q on D, our algorithm GenerateHLCs for generating high-level changes proceeds as follows. (The pseudocode appears in Appendix D.) First, the provenance graph of Q and D is recorded using the rewrite approach outlined in Section 5.1. Second, for each false positive t, the algorithm traverses the provenance graph starting from the node corresponding to t in depth-first order, following edges in reverse direction. For every edge $\{\ldots\} \xrightarrow{Op} t'$ encountered during the traversal, one high-level change "remove t' from the output of Op" is being generated.

Suppose the algorithm is invoked on rules R_1 to R_4, with negative output tuple t_{10} and input document from Figure 2. Referring to Figure 4, the algorithm traverses the provenance graph starting from t_{10} (thus visiting each node in the provenance of t_{10}) and outputs the following high-level changes: (t_{10}, π_4), (t'_{10}, σ_4), $(t''_{10}, \bowtie_4)$, $(t_2, Regex_1)$, (t_6, π_3), (t'_6, σ_3), $(t_3, Dict_2)$.

6. GENERATING LOW-LEVEL CHANGES

In terms of the relational algebra, a *low-level change* is defined as the change to the configuration of a single operator (e.g., changing the numerical values used in a join condition), or insertion of a new operator subtree between two existing operators. Since the space of all low level changes is unlimited, we limit the discussion in this paper to low-level changes that *restrict* the set of results returned by the query, to make the problem tractable. This is in the same philosophy as [25] – users generally start with a query with high recall and progressively refine it to improve the precision.

6.1 Producing Low-Level Changes

Given a set of high-level changes, our goal is to produce a corresponding set of low-level changes, along with enough information about the effects of these changes to rank them. One semi-naive way to compute these low-level changes is to iterate over the operators in the canonical relational algebra representation of the annotator, performing the following three steps:

1. For each operator, consider all the high-level changes that could be applied at that operator.
2. For each such high-level change, enumerate all low-level changes that cause the high-level change.
3. For each such low-level change, compute the set of tuples that the change removes from the operator's output.
4. Propagate these removals up through the provenance graph to compute the end-to-end effects of each change.

This approach computes the correct answer, but it would be extremely slow. This intractability stems directly from the two challenges discussed in Section 4: *feasibility* and *side-effects*.

First, the feasibility problem makes step 2 intractable. Just as there could be no feasible low-level change that implements a given high-level change, there could easily be a nearly infinite number of them. For example, consider a high-level change to remove an output tuple of a dictionary operator. Suppose the dictionary has 1000 entries, one of which produces the tuple. By choosing different subsets of the other 999 entries, one can generate $2^{999} - 1$ distinct low-level changes, any of which removes the desired tuple!

We address this aspect of feasibility by limiting the changes our system considers to a set that is of tractable size, while still considering all feasible combinations of high-level changes at a given operator. In particular, we generate, for each operator, a single low-level change for each of the k best possible combinations of high-level changes; where k is the total number of changes that the system will present to the user. We enforce these constraints through careful design of the algorithms for generating individual types of low-level changes, as we describe in Section 6.2.

The side-effects problem causes problems at step 4 of the above approach. Traversing the provenance graph is clearly better than re-running the annotator to compute the effects of each change. However, even if it generates only one low-level change per operator, the overall cost of this approach is still $O(n^2)$, where n is the size of the operator tree. Such a computation rapidly becomes intractable, as moderately complex annotators can have thousands of operators.

We can reduce this complexity from quadratic to linear time by leveraging our algorithm for enumerating high-level changes. The algorithm in Section 5.2 starts with a set of undesirable output tuples and produces, for each input tuple, a set of high-level changes that would remove the tuple. We can easily modify this algorithm to remember the mapping from each high-level change back to the specific output tuple that the change removes.

By running this modified algorithm over every output tuple, in-

cluding the correct outputs, we can precompute the end-to-end effects of any possible side-effect of a low-level change. With a hash table of precomputed dependencies, we can compute the end-to-end effects of a given low-level change in time proportional to the number of tuples the change removes from the local operator.

Applying the optimizations described above to the semi-naive algorithm yields the following steps for generating low-level changes.

1. Precompute the mapping from intermediate tuples to the final output tuples they generate.
2. For each operator and each category of low-level change, compute a top-k set of low-level changes.
3. Compute the local effects of each low-level change.
4. Use the table from step 1 to propagate these local effects to the outputs of the annotator.

In the next section, we explain in detail how we perform step 2 efficiently for several different types of low-level changes.

6.2 Specific Classes of Low-Level Changes

We now introduce the specific types of low-level changes that our system currently implements, along with the techniques we use to generate these low-level changes efficiently. We measure result quality using the classical $F1$-*measure* – the harmonic mean of precision (percentage of true positives among all extracted answers) and recall (percentage of true positives among all actual answers).

Modify numerical join parameters. This type of change targets the join operator. We use the predicate function *Follows* as an example for all joins based on numerical values. Recall that $Follows(span_1, span_2, n_1, n_2)$ returns true if $span_1$ is followed by $span_2$ by a distance value in the range of $[n_1, n_2]$. Low-level changes to a *Follows* predicate involve shrinking the range of character distances by moving one or both of the endpoints.

Our approach to generate low-level changes for numerical join predicates involves interleaving the computation of side-effects with the process of iterating over possible numerical values. Recall that the end goal of our system is to produce a ranked list of low-level changes, where the higher-ranked changes produce a greater improvement in result quality according to an error metric. We use this ranking function to compute a *utility* value for each value in the range and remove those with low utility. In particular, we compute utility by probing each value in the range: remove it, propagate the change to the output, and compute the change in result quality as the utility of the value in consideration.

We now need to find the top-k sub-sequences in $[n_1, n_2]$ that corresponds to maximum summation of utility values. This problem can be solved with Kadane's algorithm [7] in $O(nk)$ time, where n is the number of values, and k is the number of ranges to find.

Remove dictionary entries. Another important class of low-level change involves removing entries from a dictionary file so as to remove the corresponding dictionary matches from the annotator's input features. Our approach to this type of change takes advantage of the fact that each dictionary entry produces a disjoint set of tuples at the output of the *Dictionary* operator.

As with numerical join parameters, we interleave the computation of low-level changes with the process of computing the effects of each change and the resulting improvement in utility. We start by grouping the outputs of the *Dictionary* operator by dictionary entry. For each dictionary entry that matches at least one high-level change, we compute the tuples that would disappear from the final query result if the entry was removed. We then rank the entries according to the effect that removing that entry would have on result quality. We then generate a low-level change for the top 1 entry, the top 2 entries, and so on, up to k entries. In addition to the dictionary operator, this class of changes also applies, analoguously, to select operators having a dictionary predicate such as *MatchesDict()*.

Add filtering dictionary. This class of changes targets the select operator. In addition to modifying, our system also generates new dictionaries and uses them to filter spans based on the presence of dictionary matches in close proximity. We produce filtering predicates by composing a span operation like *LeftContextTok* with a dictionary predicate like *Not(ContainsDict())* as in rule R_3 (Fig. 2).

To generate filtering predicates our system considers the tokens to the left or right of each span in a tuple affected by a high-level change. The union of these token values forms a set of potential dictionary entries. We rank the effects of filtering with these dictionary entries the same way that we rank changes involving removal of dictionary entries: we group together tuples according to which dictionary entries occur in the vicinity of their spans, and compute the effect of each potential entry on end-to-end result quality.

Add filtering view. Unlike all low-level changes discussed above, which apply to an individual operator, this last type of changes applies to an entire view. Specifically, it involves using subtraction to add a filter view on top of an existing view V. It removes spans from V that overlap with, contain, or are contained in some span of the filtering view. As an example, rule R_5 in Figure 2 implements a filtering view on top of *PersonPhoneAll*. To generate filtering views, our algorithm considers every pair of views V_1 and V_2 such that V_1 and V_2 are not descendants of one another in the canonical representation of the ruleset. For each filter policy (OVERLAP, CONTAINS, or CONTAINED) the algorithm identifies the tuples of V_1 that are in relationships with at least one V_2 span according to the policy, and ranks the resulting filters according to their effects on the overall end-to-end result quality.

7. EXPERIMENTS

We developed our refinement approach on top of the *SystemT* [11, 22, 27] information extraction system enhanced with a provenance rewrite engine as described in Section 5.1. In this section we present an experimental study of our system in terms of performance, and quality of generated refinements.

Extraction Tasks and Rule Sets. We use two extraction tasks in our evaluation: Person (person entity extraction) and PersonPhone (extraction of relationships between persons and their phone numbers). We chose Person because it is a classic named-entity extraction task and there are standard evaluation datasets available. We chose PersonPhone as an example of a relationship extraction task.

The Person extraction rule set consists of 14 complex rules for identifying person names by combining basic features such as capitalized words and dictionaries of first and last names. Example rules include "*CapitalizedWord* followed by *FirstName*", or "*LastName* followed by a comma, followed by *CapitalizedWord*". We have also included rules for identifying other named-entities such as *Organization*, *Address*, *EmailAddress*, that can be only used as filtering views, in order to enable refinements commonly needed in practice, where person, organizations and locations interact with each other in various ways (e.g., "Morgan Stanley" may be an organization, or a person, "Georgia" may be a person, or a U.S. state).

The PersonPhone extraction rule set consists of 11 complex rules for identifying phone/extension numbers, and a single rule "*Person* followed within 0 to 60 chars by *Phone*" for identifying candidate person–phone relationships (as in rule R_4 from Figure 2). To evaluate the system on the relationship task, we use a high-quality Person extractor to identify person names in the PersonPhone task. Note that the system is evaluated separately on the Person task, and we focus on the relationship extractor for the PersonPhone task.

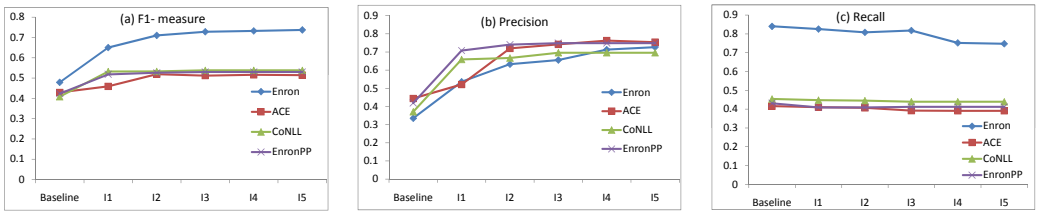

Figure 5: Result Quality After Each Iteration of Refinement: $F1$-measure (a), Precision (b), and Recall (c).

Datasets. Appendix E lists the characteristics of the following datasets used in our evaluation.

- *ACE*: collection of newswire reports, broadcast news and conversations with Person labeled data from the ACE05 Dataset [3].
- *CoNLL*: collection of news articles with Person labeled data from the CoNLL 2003 Shared Task [34].
- *Enron*, *EnronPP*: collections of emails from the Enron corpus [2] annotated with Person and respectively PersonPhone labels.

Detailed experimental settings are introduced in Appendix F.

7.1 Quality Evaluation

The goal of the quality evaluation is to validate that our system generates high quality refinements in that: 1) they improve the precision of the original rules, while keeping the recall fairly stable, and 2) they are comparable to refinements that a human expert would identify. To this end, we evaluate the quality of refinements produced by our system on various datasets, and perform a user study where a rule refinement task is presented to human experts whose actions are compared with those suggested by our system.

Experiment 1. We use 4 workloads in this experiment: the Person task on *ACE*, *CoNLL* and *Enron* datasets, and the PersonPhone task on the *EnronPP* dataset. For each workload, we run the system for k iterations starting from the baseline rule set. After each iteration, the refinement with the highest improvement in $F1$-measure on the training set is automatically applied. Figure 5 shows the quality of k refined rule sets on the test set of each workload, when k is varied from 1 to 5. Note that the quality of the baseline rule sets is as expected in practice, where developers usually start with a query with reasonable recall and progressively refine it to improve precision. As can be seen, our system achieves significant improvements in $F1$-measure between 6% and 26% after only a few iterations. This improvement in $F1$-measure does not arise at the expense of recall. Indeed, as shown in Figures 5(b-c), the precision after 5 iterations improves greatly when compared to the baseline rule set, while the recall decreases only marginally. The $F1$-measure and precision plateau after a few refinements for two reasons. First, many false positives are removed by the first few high ranked refinements, therefore substantially decreasing the number of examples available in subsequent iterations. Second, removing some of the other false positives requires low-level changes that are not yet implemented in our system (e.g., modifying a regular expression).

Experiment 2. In this experiment we compare the top refinements generated by our system with those devised by human experts. To this end, we conducted a user study in which two experts A and B were given one hour to improve the rule set for the Person task using the *Enron* train set. Both experts are IBM researchers (not involved in this project) who have written over 20 information extraction rule sets for important IBM products over the past 3 years. To ensure a fair comparison, the experts were restricted to types of refinements supported in our current implementation (Section 6.2).

ID	Description	P	R	$F1$	I_1	I_2
	Baseline	35.2	85.0	49.8		
A_1,B_1	Filter Person by Person (CONTAINED)	57.3	83.7	68.0	1	n/a
A_2	Dictionary filter on CapsPerson	70.3	83.9	76.5	4	4
A_3,B_4	Dictionary filter on Person	71.8	83.8	77.4		
A_4	Filter PersonFirstLast by DblNewLine (OVERLAP)	72.6	84.0	77.9	9	5
A_5	Filter PersonLastFirst by DblNewLine (OVERLAP)	72.7	84.1	78.0	9	5
A_6,B_2	Filter PersonLastFirst by PersFirstLast (OVERLAP)	73.5	84.1	78.4	5	3
A_7,B_3	Filter Person by Org (OVERLAP)	74.1	82.5	78.0	3	1
A_8	Filter Person by Address (OVERLAP)	74.3	82.4	78.1	11	9
A_9	Filter Person by EmailAddress (OVERLAP)	77.3	81.7	79.4	12	6

Table 1: Expert refinements and their ranks in the list of generated refinements after iterations 1 and 2 (I_1, I_2).

Table 1 shows the refinements of both experts and corresponding improvements in $F1$-measure achieved on the test set for expert A. (Expert B's refinements are a subset of A's.) The table also shows the rank of each expert refinement in the list automatically generated by our system in iteration 1, and iteration 2 (after applying the top-most refinement). We observed that the top refinement suggested by the system (remove person candidates strictly contained within other person candidates) coincides with the first refinement applied by both experts (i.e., A_1 and B_1). Furthermore, with a single exception, all expert refinements appear among the top 12 results generated by our system in the first iteration. The dictionary filter generated in iteration 1 consisted of 12 high-quality entries incorrectly identified as part of a person name (e.g., "Thanks", "Subject"). It contains 27% of all entries in corresponding refinement A_2, and all entries in the filter dictionary on person candidates of B_4. Furthermore, in both iterations, the system generated a slightly better refinement compared to A_4 and A_5 that filters all person candidates overlapping with a double new line. This achieves the combined effect of A_4 and A_5, while producing a refined rule set with a slightly simpler structure (a single filter, instead of two).

The system also suggested refinements not useful at first glance, for example, a dictionary filter on one token to the right of person candidates containing initials. This was due to the baseline rule set not identifying names with middle initial. While not helpful in improving precision, this refinement is helpful in improving recall, by signaling to the developer additional person candidate rules based on contextual clues, However, this is subject of our future work. Based on the observations above, we believe it is reasonable to conclude that our system is capable of generating rule refinements that are comparable in quality to those generated by human experts.

7.2 Performance Evaluation

The goal of our performance evaluation is two-fold: to validate that our algorithm for generating low-level changes is tractable, since it should be clear that without the optimizations in section 6, CPU cost would be prohibitive, and to show that the system can automatically generate refinements faster than human experts.

The table below shows the running time of our system in the first

3 iterations with the Person rule set on the *Enron* dataset, when the size of the training data is varied between 100 and 400 documents.

Train set #docs	I_1 (sec)	I_2 (sec)	I_3 (sec)	$F1$ after I_3 (%)
100	35.3	1.8	1.1	74.9
200	44.5	6.0	4.2	70.2
300	72.9	9.9	6.3	72.1
400	116.4	21.3	13.6	70.0

As shown above, the system takes between 0.5 and 2 minutes for the first iteration, which includes the initialization time required for loading the rule operators in memory, running the extractor on the training set, and computing the provenance graph, operations performed exactly once. Once initialized, the system takes under 20 seconds for subsequent iterations. As expected, the running time in each iteration decreases, since less data is being processed by the system after each refinement. Also note that the $F1$-measure of the refined rule set after iteration 3 (refer to last column of the table) varies only slightly with the size of the training set.

We note that in each iteration the system sifts through hundreds of documents, identifies and evaluates thousands of low-level changes, and finally presents to the user a ranked list of possible refinements, along with a summary of their effects and side-effects. When done manually, these tasks require a large amount of human effort. Recall from Experiment 2 that the experts took one hour to devise, implement and test their refinements, and reported taking between 3 and 15 minutes per refinement. In contrast, our system generates almost *all* expert's refinements in iteration 1, in about 2 minutes!

8. CONCLUSIONS AND FUTURE WORK

As we seek to leverage database technology to manage the growing tide of poorly structured information in the world, information extraction has gained growing importance. Most information extraction is based on painstakingly defined extraction rules that are error-prone, often brittle, and subject to continuous refinement. This paper takes a significant step towards simplifying IE rule development through the use of database provenance techniques.

Specifically, this paper showed how to modify extraction rules to eliminate false positives in the extraction result. Standard provenance techniques only consider the provenance of tuples in the result set, and hence are not useful for addressing false negatives. However, recent provenance work [9, 18, 20] has begun to develop tools to reason about expected tuples not present in the result set. We believe these techniques can be adapted to our framework to address false negatives. However, this is the subject of future work.

Acknowledgements. We thank Rajasekar Krishnamurthy and Yunyao Li for participating in our expert user study, and the anonymous reviewers for their insightful comments.

9. REFERENCES

[1] Database languages – SQL – Part 1: Framework (SQL/Framework). Technical report. ISO/IEC 9075-1:2003.

[2] The Enron corpus. www.cs.cmu.edu/enron/.

[3] Automatic Content Extraction 2005 Evaluation Dataset, 2005.

[4] E. Agichtein and L. Gravano. *Snowball*: Extracting Relations from Large Plain-Text Collections. In *ACM DL*, 2000.

[5] D. E. Appelt and B. Onyshkevych. The Common Pattern Specification Language. In *TIPSTER workshop*, 1998.

[6] N. Ashish, S. Mehrotra, and P. Pirzadeh. XAR: An Integrated Framework for Information Extraction. In *WRI Wold Congress on Computer Science and Information Engineering*, 2009.

[7] J. L. Bentley. Programming Pearls: Algorithm Design Techniques. *Commun. ACM*, 27(9):865–873, 1984.

[8] B. Boguraev. Annotation-based Finite State Processing in a Large-Scale NLP Architecture. In *RANLP*, 2003.

[9] A. Chapman and H. V. Jagadish. Why Not? In *SIGMOD*, 2009.

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

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

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

[13] A. Das Sarma, A. Jain, and D. Srivastava. I4E: Interactive Investigation of Iterative Information Extraction. In *SIGMOD*, 2010.

[14] D. DeJong. An Overview of the FRUMP System. In *Strategies for Natural language Processing*. 1982.

[15] D. Freitag. Multistrategy Learning for Information Extraction. In *ICML*, 1998.

[16] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *ICDE*, 2009.

[17] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, 2007.

[18] M. Herschel and M. Hernandez. Explaining Missing Answers to SPJUA Queries. *PVLDB*, 2010.

[19] J. R. Hobbs, D. Appelt, J. Bear, D. Israel, M. Kameyama, and M. Tyson. FASTUS: a System for Extracting Information from Text. In *HLT*, 1993.

[20] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the Provenance of Non-Answers to Queries over Extracted Data. *PVLDB*, 1(1), 2008.

[21] A. Jain, P. Ipeirotis, A. Doan, and L. Gravano. Join Optimization of Information Extraction Output: Quality Matters! In *ICDE*, 2009.

[22] 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.

[23] J. Lafferty, A. McCallum, and F. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *ICML*, 2001.

[24] W. Lehnert, J. McCarthy, S. Soderland, E. Riloff, C. Cardie, J. Peterson, F. Feng, C. Dolan, and S. Goldman. UMass/Hughes: Description of the CIRCUS System Used for MUC-5. In *MUC*, 1993.

[25] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular Expression Learning for Information Extraction. In *EMNLP*, 2008.

[26] F. Peng and A. McCallum. Accurate Information Extraction from Research Papers Using Conditional Random Fields. In *HLT-NAACL*, 2004.

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

[28] E. Riloff. Automatically Constructing a Dictionary for Information Extraction Tasks. In *KDD*, 1993.

[29] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan. Toward Best-Effort Information Extraction. In *SIGMOD*, 2008.

[30] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *VLDB*, 2007.

[31] S. G. Soderland. Learning Text Analysis Rules for Domain-specific Natural Language Processing. Technical report, U. Mass., 1996.

[32] S. Tata, J. M. Patel, J. S. Friedman, and A. Swaroop. Declarative Querying for Biological Sequences. In *ICDE*, 2006.

[33] C. Thompson, M. Califf, and R. Mooney. Active Learning for Natural Language Parsing and Information Extraction. In *ICML*, 1999.

[34] 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*, 2003.

[35] A. Yates, M. Banko, M.Broadhead, M. J. Cafarella, O. Etzioni, and S. Soderland. TextRunner: Open Information Extraction on the Web. In *HLT-NAACL (Demonstration)*, 2007.

[36] S. Zhao and R. Grishman. Extracting Relations with Integrated Information Using Kernel Methods. In *ACL*, 2005.

APPENDIX

A. RULE LANGUAGES

Figure 6 shows examples of rule languages referenced in recent work. The figure shows three different implementations of the rule that we had described earlier in Figure 1. Each implementation uses a different rule language, but all three generate the same output, except in certain corner cases.

In general, information extraction rule languages often differ in syntax and overall expressive power [27]. However, most rule languages in common use share a large set of core functionality. Furthermore, the common core functionality of most information extraction rule languages can be expressed as standard SQL, with a few text-specific extensions described next.

B. ADDITIONS TO SQL

In the examples in this paper, we augment the standard set of SQL functions with the following text-specific functions:

1. Predicates and scalar functions for manipulating spans, used for expressing *join* and *selection predicates*, and creating new values in the SELECT clause of a rule; and

2. Table functions for performing three crucial IE tasks: *regular expression matching*, *dictionary matching*.

Figure 7 lists these text-specific additions, along with a brief description of each.

The ability to perform character-level *regular expression matching* is fundamental in any IE system, as many basic extraction tasks such as identifying phone numbers or IP addresses can be achieved using regular expressions. For our example rule in Figure 6, regular expression matching is appropiate for identifying capitalized words in the document, and is expressed, for instance, in AQL lines 5 – 6, and xLog line 5 in Figure 6.

For this purpose, we have added to our language the *Regex* table function (refer to Figure 7), which takes as input a regular expression, a relation name R, and an attribute of type span A of R, and computes an instance with a single span-typed attribute called *match* containing all matches of the given regular expression on the A values of all tuples in R.

A second fundamental IE functionality is *dictionary matching*: the ability to identify in an input document all occurrences of a given set of terms specified as entries in a dictionary file. Dictionary matching is useful in performing many basic extraction tasks such as identifying person salutations (e.g., "Mr", "Ms", "Dr"), or identifying occurrences of known first names (e.g., refer to Figure 6, line 4 of JAPE, lines 3–4 of AQL, and line 3 of xLog). The *Dictionary* table function serves this purpose in our language: it takes as input the name of a dictionary file, a relation name R, and an attribute of type span A of R, and computes an instance with a single span-typed field called *match* containing all occurrences of dictionary entries on the A values of all tuples in R.

A third component of information extraction rules is a toolkit of span operations. Table 7 lists the text-based scalar functions that our system uses to implement various operations over the $span$ type. Note the distinction between scalar functions that return a boolean value (e.g., *Follows*) and can be used as join predicates, and scalar functions that return non-boolean values (e.g., Merge), used as selection predicates, and to create new values in the SELECT clause of rules.

C. PROVENANCE ASSOCIATED WITH OPERATORS

Definition C.1 formalizes the notion of *provenance graph* used in this paper. Note that the intention of the formalism below is not to propose yet another definition for provenance. In fact, when restricted to the SPJU fragment of SQL, Definition C.1 corresponds to the original definition of how-provenance of [17][2]. Rather, our goal is to provide a pictorial representation of provenance that we can use in discussing the algorithm for computing high-level changes.

DEFINITION C.1. *[Provenance graph] Let Q be a set of rules and D be a document collection. The* data flow graph of Q and D, *or in short,* the data flow graph of Q *when D is understood from the context, is a hypergraph $G(V, E)$, where V is a set of hypervertices, and E is a set of hyperedges, constructed as follows. For every operator Op in the canonical representation of Q:*

- *If $Op = Regex_{(regex,A)}(R)$, or $Op = Dictionary_{(dict\_file,A)}(R)$, then for every $t \in R$ and every output tuple $t' \in Op(t)$, V contains vertices $v_t, v_{t'}$ and E contains edge $v_t \xrightarrow{Op} v_{t'}$. We say that* the provenance of t' according to Op is t.

- *If $Op = \pi_A(R)$, where A is a set of attributes, then for every $t \in R$ and corresponding output tuple $t' = \pi_A(t)$, V contains vertices $v_t, v_{t'}$ and E contains edge $v_t \xrightarrow{\pi_A} v_{t'}$. We say that* the provenance of t' according to π_A is t.

- *If $Op = \sigma_C(R)$, where C is a conjunction of selection predicates, then for every $t \in R$ and corresponding output tuple $t' = \sigma_C(t)$ (if any), V contains vertices $v_t, v_{t'}$ and E contains edge $v_t \xrightarrow{\sigma_C} v_{t'}$. We say that* the provenance of t' according to σ_C is t.

- *If $Op = \bowtie_C (R_1, \ldots, R_n)$, where C is a conjunction of join predicates, then for every $t_1 \in R_1, \ldots, t_n \in R_n$ and corresponding output tuple $t' = \bowtie (t_1, \ldots, t_n)$ (if any), V contains vertices $v_{t_1}, \ldots, v_{t_n}$ and hypervertex $\{v_{t_1}, \ldots, v_{t_n}\}$, and E contains hyperedge $\{v_{t_1}, \ldots, v_{t_n}\} \xrightarrow{\bowtie_C} v_{t'}$. We say that* the provenance of t' according to $\bowtie_C$ is $\{t_1, \ldots, t_n\}$.

- *If $Op = \cup(R_1, R_2)$, then for every $t_1 \in R_1$ (or $t_2 \in R_2$) and corresponding output tuple $t' \in \cup(\{t_1\}, \emptyset)$ (or respectively, $t' \in \cup(\emptyset, \{t_2\})$), V contains vertices v_{t_1} (or v_{t_2}) and $v_{t'}$, and E contains edge $v_{t_1} \xrightarrow{\cup} v_{t'}$ (respectively, $v_{t_2} \xrightarrow{\cup} v_{t'}$). We say that* the provenance of t' according to $\cup$ is t_1 (or respectively, t_2).

- *If $Op = \delta(R_1, R_2)$, then for every $t \in R_1$ such that $t \notin R_2$ and corresponding output tuple $t' \in \{t\} - R_2$, V contains vertices $v_t, v_{t'}$ and E contains edge $v_t \xrightarrow{\delta} v_{t'}$. We say that* the provenance of t' according to δ is t.

[2]Note that since each tuple is assigned a unique identifier, we are essentially in the realm of set semantics.

JAPE [12]

```
Rule: CandidatePersonName
 Priority: 1
   (
     { Lookup.kind == firstName }
     { Token.orthography == initialCaps }
   ):match
--> :match.kind = "CandidateName";
```

AQL [27]

```
create view CandidatePersonName as
select CombineSpans(F.name, L.name) as name
from (extract dictionary FirstNameDict
    on D.text as name from Document D) F,
  (extract regex /[A-Z][a-z]+/
    on D.text as name from Document D) L
where FollowsTok(F.name, L.name, 0, 0)
consolidate on name;
```

XLog [6, 30]

```
CandidatePersonName(d, f, l) :-
    docs(d),
    firstNamesDict(fn),
    match(d, fn, f),
    match(d, "[A-Z][a-z]+", l),
    immBefore(f, l);
```

Figure 6: The rule from Figure 1, expressed in three different information extraction rule languages

Type	Format	Description
Predicate functions	*Follows/FollowsTok*($span_1$,$span_2$,n_1,n_2)	Tests if $span_2$ follows $span_1$ within n_1 to n_2 characters, or tokens
	Contains/Contained/Equals($span_1$,$span_2$)	Tests if $span_1$ contains,is contained within, or is equal to $span_2$
	MatchesRegex/ContainsRegex(r, $span$)	Tests if $span$ matches (contains a match for, resp.) regular expression r
	MatchesDict/ContainsDict(*dict*, $span$)	Tests if $span$ matches (contains a match for, resp.) an entry of dictionary d
Scalar functions	*Merge*($span_1$, $span_2$)	Returns the shortest span that completely covers both input spans
	Between($span_1$, $span_2$)	Returns the span between $span_1$ and $span_2$
	LeftContext/LeftContextTok($span$, n)	Returns the span containing n chars/tokens immediately to the left of $span$
	RightContext/RightContextTok($span$, n)	Returns the span containing n chars/tokens immediately to the right of $span$
Table functions	*Regex*(r, R, A)	Returns all matches of regular expression r in all $R.A$ values.
	Dictionary(d, R, A)	Returns all matches of entries in dictionary d in all $R.A$ values.

Figure 7: Text-specific predicate, scalar, and table functions that we add to SQL for expressing the rules in this paper.

D. COMPUTING HIGH-LEVEL CHANGES

Our algorithm GenerateHLCs for computing a set of high-level changes, given a set of rules Q, an input document collection D and a set of false positives in the output of Q on D, is listed below.

GenerateHLCs(G, X, D)
Input: Operator graph G of a set of rules Q, set X of false positives in the output of G (i.e., Q) applied to input document collection D.
Output: Set H of high-level changes.
Let $H = \emptyset$.

1. Compute the provenance graph $G_p^{Q,D}$ of Q and D;
2. For every $t \in X$ do CollectHLCs($G_p^{Q,D}$, t, H);
3. Return H.

Procedure CollectHLCs(G_p, t', H)
Input: Provenance graph G_p, node t' in G_p, set of high-level changes H.
If t' is a tuple of the Document instance, return.

Otherwise, let e: $T \xrightarrow{Op} t'$ be the incoming edge of t' in G_p. Do:

1. Add (t', Op) to H;
2. If e is of type $t'' \xrightarrow{Op} t'$, where $Op \in \{\pi, \sigma, \cup, \delta, Regex, Dictionary\}$, do CollectHLCs($G_p$, t'', H).
   Otherwise, e is of type $\{t_1, \ldots, t_n\} \xrightarrow{\bowtie} t'$. Do CollectHLCs($G_p$, t_i, H), for all t_i, $1 \leqslant i \leqslant n$.

E. EVALUATION DATASETS

The characteristics of the datasets used in our experiments in terms of number of documents and labels in the train and test sets are listed below.

Dataset	Train set		Test set	
	#docs	#labels	#docs	#labels
ACE	273	5201	69	1220
CoNLL	946	6560	216	1842
Enron	434	4500	218	1969
EnronPP	322	157	161	46

These datasets are realistic in practical scenarios, and in fact, both *ACE* and *CoNLL* have been used in official Named Entity Recognition competitions [3, 34]. We note that in practice, rule developers are unlikely to examine a very large number of documents, and obtaining labeled data is known to be a labor intensive and time consuming task. (Machine learning techniques such as active learning [33] have been used to facilitate the latter task.)

F. EXPERIMENTAL SETTINGS

We developed our rule refinement approach in *SystemT* v0.3.6 [11, 22], the information extraction system developed at IBM Research – Almaden, enhanced with a provenance rewrite engine as described in Section 5.1. Our implementation uses SystemT's AQL rule language [27]. The experiments were run on a Ubuntu Linux version 9.10 with 2.26GHz Intel Xeon CPU and 8GB of RAM. All experiments, unless otherwise stated, are from a 10-fold cross-validation.