Building a Generic Debugger for Information Extraction Pipelines

Anish Das Sarma
Yahoo, CA, USA
anish.dassarma@gmail.com

Alpa Jain
Yahoo, CA, USA
alpa@yahoo-inc.com

Philip Bohannon
Yahoo, CA, USA
plb@yahoo-inc.com

ABSTRACT

Complex information extraction (IE) pipelines are becoming an integral component of most text processing frameworks. We introduce a first system to help IE users analyze extraction pipeline semantics and operator transformations interactively while debugging. This allows the effort to be proportional to the need, and to focus on the portions of the pipeline under the greatest suspicion. We present a generic debugger for running post-execution analysis of any IE pipeline consisting of arbitrary types of operators. For this, we propose an effective provenance model for IE pipelines which captures a variety of operator types, ranging from those for which full to no specifications are available. We have evaluated our proposed algorithms and provenance model on large-scale real-world extraction pipelines.

Categories and Subject Descriptors: H.4.0 Information Systems Applications: General

General Terms: Algorithms

Keywords: Information extraction, provenance

1. INTRODUCTION

Information extraction (IE) systems identify structured information and, not surprisingly, IE systems are becoming a critical first-class operator in a large number of text-processing frameworks. As a concrete example, search engines are moving beyond a "keyword in, document out" paradigm to providing structured information relevant to users' queries (e.g., providing contact information for businesses when user queries involve business names). For this, search engines typically rely on having available large repositories of structured information generated from web pages or query logs using IE systems. With the increasing complexity of IE pipelines, a critical exercise for IE developers and even users is to *debug*, i.e., perform a thorough post-mortem analysis of the output generated by running an entire or partial extraction pipeline. Despite the popularity of IE pipelines, very little attention has been given to building effective ways to trace the control or data flow through an extraction pipeline.

Prior work [14] recognizes the need for interactive debuggers and proposes methods that use complete knowledge of how each operator functions. However, prior information regarding the specifications of the operators may not be available (e.g., off-the-shelf black-box operators). In the absence of full function specifications of an operator, the only (straightforward) approach to debugging is exploring all data in the pipeline. However, such an approach is clearly infeasible due to the sheer volume of data. For instance, debugging a simple pipeline involving 10 operators with 10,000 input records per operator would require 100K records to be manually examined; typical data sizes are even larger.

This paper presents PROBER (for Provenance-Based Debugger), the first generic framework for debugging information extraction pipelines composed of arbitrary ("black-box") operators. A critical task towards building debuggers is that of tracing and linking output records from each operator and understanding their transformations across different operators in the pipeline. To trace the lineage of any arbitrary record in the output, we propose a novel provenance model for IE pipelines. With debugging in mind, our provenance model tries to minimize the amount of user effort necessary in resolving the fate of the records in the output. For example, provenance for (incorrect) output records *only* refer to input tuples that impacted this output record.

2. PROBLEM FORMULATION

While IE pipelines may vary in their implementation logic [1, 11, 12, 13] several underlying common components can be abstracted from the implementation details. We characterize information extraction pipelines for the task of performing post-mortem analysis.

Definition 2.1. **[Record]** *A* record *r is a basic unit of data (e.g., a tuple), consisting of a globally unique identifier $I(r)$, and value $V(r)$. We use R to denote the set of all records.* □

Definition 2.2. **[Operator]** *An* operator *is defined by a function $O : (I_1, I_2, \cdots, I_N) \rightarrow R$, where each $I_i \subseteq R$ is a set of records. In practice, the function O may be unknown to us.* □

Intuitively, an operator takes as input an N-tuple of sets of records and outputs one set of records. Specifications on how an operator generates an output record may be available in

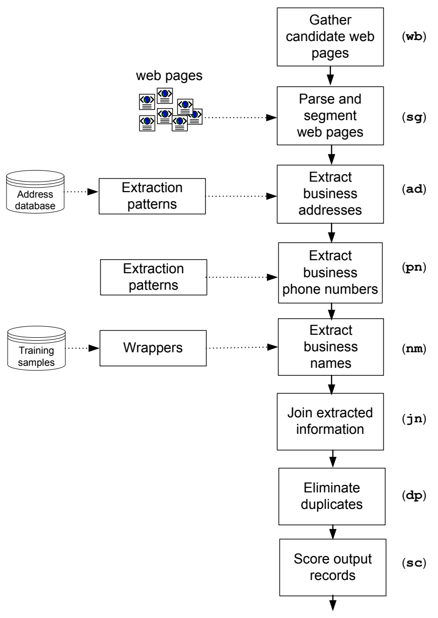


Figure 1: Example of an IE pipeline to generate business names and their contact information.

varying forms. Specifically, we consider the following four scenarios involving operator specifications.

An operator is said to be a *black-box* if we have no information about it. In this case, naturally, the only way to gain information about a black-box operator is by executing it on input sets of records. In contrast, we have *exact* information about an operator O if we know precisely *which* input records contributed to each output record, *and how*. We have *Input-Output (IO) specifications* when for each output record, we know which input records were used to construct it, however exactly how a record is generated is unknown to us. Finally, we may have *integrity constraints*, e.g., key-foreign key relationships, satisfied by the input and output records. For instance, an operator may support a 'debug' mode where each output record is assigned an id associated with the input records that generated it. Effectively, using key-foreign keys we have the same information as that in IO specifications, but this information is (indirectly) available using dependencies on the values of *fields* in input and output records.

Next, we define various (standard) properties of an operator, that help design specialized algorithms for building provenance and debugging effectively. As we will see later, these properties may be learned by sampling or the operator specifications (when available) described above.

Definition 2.3. **[Properties]**

- `monotonic`: *Operator O is* `monotonic` *iff* $\forall I_1, I_2 \subset R : (I_1 \subseteq I_2) \Rightarrow (O(I_1) \subseteq O(I_2))$.
- `one-to-one`: *Operator O is* `one-to-one` *iff: (a)* $\forall I \subset R : O(I) = \bigcup_{r \in I} O(\{r\})$*; (b)* $\forall r \in R$: $|O(\{r\})| \leq 1$.
- `one-to-many`: *Operator O is* `one-to-many` *iff* $\forall I \subset R : O(I) = \bigcup_{r \in I} O(\{r\})$.
- `many-to-one`: *Operator O is* `many-to-one` *iff* $\forall I \subset R$, $\exists$ *a partition* $P_I = \{I_1, \ldots, I_n\}$ *of* I^1 *such that: (a)* $O(I) = \bigcup_{i=1}^{n} O(I_i)$*; and (b)* $\forall i : |O(I_i)| \leq 1$.

□

[1](a) $I = \bigcup_{i=1}^{n} I_i$ (b) $\forall i \neq j : (I_i \cap I_j) = \emptyset$.

Definition 2.4. **[Extraction Pipeline]** *An extraction pipeline P is defined by a DAG $G(V, E)$ consisting of a set V of nodes and a set E of edges where each node $v \in V$ corresponds to an operator O in the pipeline. An edge $a \to b$ between nodes a and b indicates that the output from the operator represented by a is input to operator represented by b. We have a single special node $s \in V$ with no incoming edges representing the operator that takes input to the pipeline, and one special node $t \in V$ with no outgoing edges representing the operator that outputs the final set of records.* □

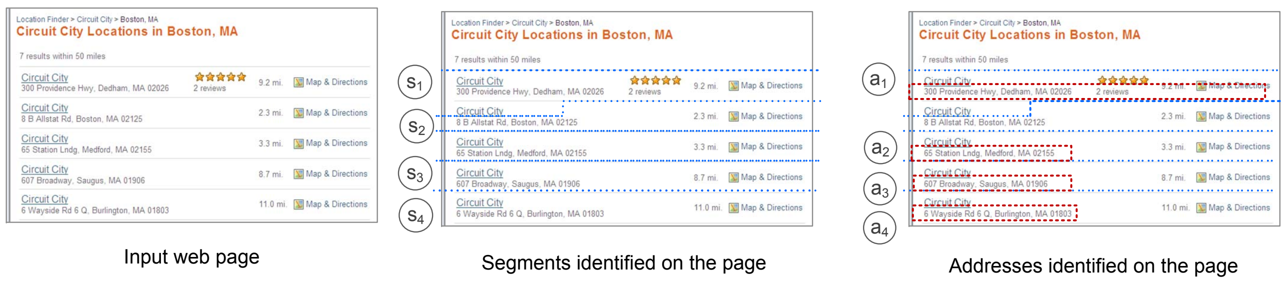
3. MOTIVATING EXAMPLE

Figure 1 shows a real-world extraction pipeline, `Business`, for building a large collection of businesses by extracting records of the form $\langle n, a, p \rangle$, where business n is located at address a with contact number p. The first step is to build a set of web pages likely to contain information regarding businesses which is done using a variety of document retrieval strategies. Specifically, we issue manually generated domain-dependent queries (e.g., "Toyota car dealership locations") as well as use form filling methods where entries such as model, make, and zipcode may be filled in order to fetch a list of car dealerships. This operator, denoted by `wb` is an example of a black-box operator with arbitrary properties.

Given a collection of web pages, operator `sg` parses the html page and identifies appropriate segments of text in this page, where ideally, each segment contains a complete target record (see Figure 2 for a real-world example). These segments are then processed by operators, `ad` and `pn`, which respectively identify an occurrence of an address and a phone number. The annotation from one operator is used by the subsequent operator to identify regions of text that should not be processed. `ad` and `pn` are implemented using handcrafted patterns based on a dictionary of address formats. The `nm` operator on the other hand needs to identify names of business which may be arbitrary strings and for this, we follow a wrapper-induction approach. In particular, using some training examples we learn a wrapper rule to identify candidate business names; these rules are based on the document structure of the html content. Of course, several other implementations for each of these operators are possible and the implementation details are orthogonal to our discussion since our goal is to build debuggers for pipelines with black-box operators where no implementation information may be available. The `jn` operator joins outupt from `ad`, `pn`, and `nm` to build candidate output records which are, in turn, processed by `dp` to eliminate duplicates. The final operator, assignes a confidence score `sc` to each output record.

We note that all our implementations of the above operators are monotonic. (Obviously, there may be non-monotonic implementations in other pipelines, but we primarily consider monotonic operators in this paper.) Although monotonic, the operators from the pipeline span a variety of properties, e.g., segmentation is a `one-to-many` operation, and by design one address is extracted from each segment, so address extraction is `one-to-one`, while deduplication is `many-to-one`. Candidate webpage generation and wrapper training, on the other hand are arbitrary, i.e. "`many-to-many`".

Given unexpected output records, an IE developer may want to answer some natural questions about the output [14]. (Figure 2 shows an example where `sg` generates an incorrect segment that leads to missing one address and


Input web page
s1
s2
s3
s4
Segments identified on the page
a1
a2
a3
a4
Addresses identified on the page

Location Finder > Circuit City > Boston, MA

Circuit City Locations in Boston, MA

7 results within 50 miles

Circuit City
300 Providence Hwy, Dedham, MA 02026 ★★★★★ 2 reviews 9.2 mi. Map & Directions

Circuit City
8 B Allstat Rd, Boston, MA 02125 2.3 mi. Map & Directions

Circuit City
65 Station Lndg, Medford, MA 02155 3.3 mi. Map & Directions

Circuit City
607 Broadway, Saugus, MA 01906 8.7 mi. Map & Directions

Circuit City
6 Wayside Rd 6 Q, Burlington, MA 01803 11.0 mi. Map & Directions

Figure 2: Sample input output for three steps, namely, `sg`, `ad`, and `pn`, in our extraction pipeline.

extracting one incorrect address.) Specifically, a developer may be interested in tracing all or part of the input records that contributed to a particular output record. For instance, given an incorrectly extracted record, we would like to know only the relevant subset of webpages and training data that impacted it, i.e., the *minimal* amount of input data necessary to identify the error. Motivated by the above observations, we focus on the following problem in the PROBER system.

PROBLEM 3.1. *Given a pipeline P, input I, and partial information about operators in P, we would like to (1) build* provenance *for the set of (intermediate and final) records in the pipeline; (2) expose provenance to developers through a query language and guide them in debugging the pipeline.*

4. PROVENANCE FOR IE PIPELINES

The notion of provenance is relatively well-understood for traditional relational databases (refer [6, 16]). A commonly advocated model [2] is to use a boolean-formula provenance, e.g., $S_1 \wedge (S_2 \vee \neg S_3)$. For the purpose of debugging extraction pipelines, such provenance models are not appropriate for two main reasons. First, unlike relational queries where the exact specifications of each operator are known, we may have black-box operators in our extraction pipeline. Second, for debugging, ideally we would like to limit the number of records (and simplify their interdependencies typically represented as boolean formulas for relational operators) a human has to assess in order to understand the issue at hand. With these in mind, we define a provenance model based on *minimal subsets* of operator inputs that capture necessary information (Section 4.1) and extend this basic model to operators where multiple minimal subsets may exist (Section 4.2) .

4.1 MISet: Basic Unit of Provenance

To define the provenance of an IE pipeline P, we begin by defining the provenance for each operator in P; we omit details on how to construct the provenance for each operator in P due to space constraints. We confine ourselves to extraction pipelines consisting of only monotonic operators (see Definition 2.3), which are a common case in practice (as in our motivating example from Section 2). We define the provenance of a monotonic extraction operator O based on the provenance for each output record $r \in R$ for O as in the definition below:

Ideally, we would like the provenance of r to represent precisely the set of records that contributed to r, however, as we will see, in practice it may not be possible to always determine the precise set of contributing records (e.g., in the absence of exact information about O), and even if possible it may be computationally intractable. For our goal of building a debugger, we observe that one of the main operations we expect users to perform is look at an (erroneous) output record r, and explore its provenance to determine the cause of this error. Therefore, a suitable provenance model is one that enables users to examine the fewest records required to decide the fate of an output record r. Formally, we define a basic unit of provenance, called MISet as follows:

DEFINITION 4.1. **[MISet]** *Given an operator O, its input I and output R, we say that $I_s \subseteq I$ is a* Minimal Subset *(MISet) of $r \in R$ if and only if: (1) $r \in O(I_s)$; and (2) $\forall I' \subset I_s : r \notin O(I')$. We use $M_{all}(O, I, r \in R)$ to denote the set of all MISets of O for input I and output record $r \in R$.* $\square$

Intuitively, an MISet gives the fewest input records required for a particular output record r to be present. Therefore, an MISet provides users with one possible reason for the occurrence of r. This, in turn, reduces the burden of manual annotation on the users; in the absence of MISets, a user may have to explore the entire input to understand what caused an error in the output. The notion of MISets primarily focuses on debugging the *presence* of records in the output.

We note that the notion of MISets has been proposed in the past [4] in the context of relational operations (called *minimal witnesses*). In practice, we may have more than one MISet possible for an output record as shown by the following example. Therefore, in the context of debugging extraction operators, we study the handling of non-uniqueness of MISets, and propose provenance definitions and construction algorithms based on combinations of MISets to facilitate debugging.

In practice, we may have more than one MISet possible for an output record as shown by the following example.

EXAMPLE 4.1. *Consider a (simple) record validation operator (e.g., `sc` in Figure 1) that computes the "support" of each record and outputs only records with sufficient support. Suppose `sc` outputs a record r if there are atleast 50 input records supporting it. Given an input of 100 records that could support r, the MISet for r is* any *subset of the input records of exactly 50 records.* $\square$

When multiple MISets are available, several ways of building provenance are possible, each differing in the extent to which they impact information and execution speed.

4.2 Handling Multiple MISets

Several formalisms for provenance model are possible when multiple MISets are available. We rigorously examine compositions of MISets, while capturing the spectrum of complete (and potentially intractable) provenance, to more tractable (but approximate) provenance.

Consider an operator O which consumes input I and generates output R; for a record $r \in R$, we denote the provenance of r as $P(O, I, r)$. We use subscripts P_* to capture various types of provenance and when clear from the context, we simply use $P_*(r)$ to denote $P_*(O, I, r)$.

All- and Any-provenance: Ideally for any output record, we would like to provide all possible information using MISets, i.e., capture all possible "causes" of an output record.

Definition 4.2. **[All-provenance]** *Given an operator O, input I, output R, and $r \in R$, we define* all-provenance *as $P_{all}(r) = M_{all}(O, I, r \in R)$.* □

In many cases P_{all} may be intractable to compute or store, and we may need to resort to "approximations" of it, presented shortly. Alternatively, we may want to find any one (or k) MISets.

Definition 4.3. **[Any-provenance]** *Given integer $k > 0$, operator O, input I, output R, and $r \in R$, we define* any-provenance *as any $P_{any}(r) \subseteq P_{all}(r)$ of size $\min(k, |P_{all}(r)|)$.* □

Impact-provenance: Given restricted amount of editorial resources, we may want to explore the most *impactful*, i.e., top-l input records sorted by their impact instead of any-k MISets. Our next definition of provenance ranks tuples based on their expected impact on the output record, measured by the number of MISets in which a tuple is present.

Definition 4.4. **[Impact-provenance]** *Given an operator O, input I, output R, and $r \in R$, we define* impact-provenance *as $P_{imp}(r) = \{(i, c_i) | \exists M \in P_{all}, i \in M, c_i = \sum_{M \in P_{all}, i \in M} 1\}$.* □

Union- and Intersection-provenance: Our next goal is to summarize P_{all} using two approximations: (1) We obtain an "upper bound" provenance that captures the set of all inputs *possible* for r, instead of exact combinations of inputs. Therefore, we define the union-provenance of r to be the union of all its MISets. (2) We obtain a "lower bound" provenance that captures the set of all possible inputs *necessary* for r; we define the intersection-provenance of r to be the common input records among all MISets.

Definition 4.5. **[Union-Provenance]** *Given an operator O, input I, output R, and $r \in R$, we define* union-provenance *as $P_{uni}(o) = \bigcup_{I_s \in M_{all}(O, I, r \in R)} I_s$.* □

Definition 4.6. **[Intersection-Provenance]** *Given an operator O, input I, output R, and $r \in R$, we define* intersection-provenance *as $P_{int}(o) = \bigcap_{I_s \in M_{all}(O, I, r \in R)} I_s$.* □

It can be seen easily that for operators with unique MISets, P_{uni} and P_{int} coincide.

5. RELATED WORK

Recent work [7, 8, 9, 10, 15] has looked at providing *exploration* phases to determine if a text database is appropriate for an IE task, but little or no insight is provided into why unexpected results are produced, and how to debug them. There is a large body of work on provenance for relational data (refer [6, 16]), and more recently [3] on understanding provenance information, which not address the problem of building provenance for black-box operators to facilitate IE debugging with minimal editorial effort, the primary goal of our work. Our recent work [14] considered debugging for iterative IE, and [5] looked at provenance for *non-answers* in results of extracted data. However, these papers assumed complete knowledge of each operator in some form. Also, in [14], we considered a specific type of extraction, namely iterative information extraction and studied how a simple 'provenance structure' helped debugging. In this paper, we provide a system for constructing provenance for *ad-hoc debugging tasks* for *any* extraction pipeline with black-box operators.

6. REFERENCES

[1] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *DL*, 2000.

[2] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[3] A. Chapman and H. V. Jagadish. Understanding provenance black boxes. *Distributed and Parallel Databases*, 27(2), Apr. 2010.

[4] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in databases*, 1, 2009.

[5] 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.

[6] R. Ikeda and J. Widom. Data lineage: A survey. Technical report, Stanford University, 2009.

[7] A. Jain, A. Doan, and L. Gravano. Optimizing SQL queries over text databases. In *ICDE*, 2008.

[8] A. Jain and P. G. Ipeirotis. A quality-aware optimizer for information extraction. *ACM Transactions on Database Systems*, 2009.

[9] A. Jain, P. G. Ipeirotis, A. Doan, and L. Gravano. Join optimization of information extraction output: Quality matters! Technical Report CeDER-08-04, New York University, 2008.

[10] A. Jain and D. Srivastava. Exploring a few good tuples from text databases. In *ICDE*, 2009.

[11] G. Kasneci, S. Elbassuoni, and G. Weikum. Ming: mining informative entity relationship subgraphs. In *CIKM*, 2009.

[12] M. Paşca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and searching the world wide web of facts - step one: The one-million fact extraction challenge. In *Proceedings of AAAI-06*, 2006.

[13] P. Pantel and M. Pennacchiotti. Espresso: leveraging generic patterns for automatically harvesting semantic relations. In *Proc. of ACL*, 2006.

[14] A. D. Sarma, A. Jain, and D. Srivastava. I4E: Interactive investigation of iterative information extraction. In *SIGMOD*, 2010.

[15] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Towards best-effort information extraction. In *SIGMOD*, 2008.

[16] W.-C. Tan. Provenance in Databases: Past, Current, and Future. *IEEE Data Engineering Bulletin*, 2008.