Extracting Structured Data from Web Pages

Arvind Arasu
Stanford University
arvinda@cs.stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

ABSTRACT

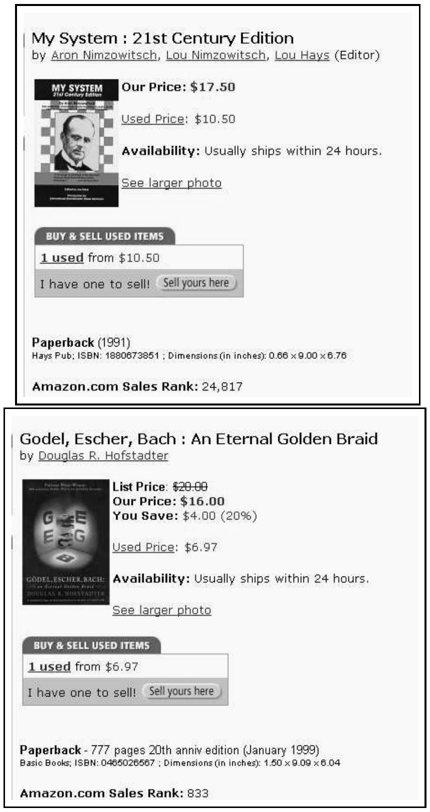
Many web sites contain large sets of pages generated using a common template or layout. For example, Amazon lays out the author, title, comments, etc. in the same way in all its book pages. The values used to generate the pages (e.g., the author, title,...) typically come from a database. In this paper, we study the problem of automatically extracting the database values from such template-generated web pages without any learning examples or other similar human input. We formally define a template, and propose a model that describes how values are encoded into pages using a template. We present an algorithm that takes, as input, a set of template-generated pages, deduces the unknown template used to generate the pages, and extracts, as output, the values encoded in the pages. Experimental evaluation on a large number of real input page collections indicates that our algorithm correctly extracts data in most cases.

1. INTRODUCTION

The World Wide Web is a vast and rapidly growing source of information. Most of this information is in the form of unstructured text, making the information hard to query. There are, however, many web sites that have large collections of pages containing structured data, *i.e.*, data having a structure or a *schema*. These pages are typically generated dynamically from an underlying structured source like a relational database. An example of such a collection is the set of book pages in Amazon [2] (Figure 1). The data in each book page has the same schema, *i.e.*, each page contains the title, list of authors, price of a book and so on.

This paper studies the problem of *automatically* extracting structured data encoded in a given collection of pages, without any human input like manually generated rules or training sets. For instance, from a collection of pages like those in Figure 1 we would like to extract book tuples, where each tuple consists of the title, the set of authors, the (optional) list-price, and other attributes (Figure 2).

Extracting structured data from the web pages is clearly very useful, since it enables us to pose complex queries over the data. Extracting structured data has also been recognized as an important sub-problem in information integration systems [7, 25, 17, 11], which integrate the data present in different web-sites. Therefore, there has been a lot of recent research in the database and AI communities on the problem of extracting data from web pages (sometimes called information extraction (IE) problem).



My System : 21st Century Edition
by Aron Nimzowitsch, Lou Nimzowitsch, Lou Hays (Editor)

Our Price: $17.50

Used Price: $10.50

Availability: Usually ships within 24 hours.

See larger photo

BUY & SELL USED ITEMS
1 used from $10.50
I have one to sell! Sell yours here

Paperback (1991)
Hays Pub; ISBN: 1880673851 ; Dimensions (in inches): 0.66 × 9.00 × 6.76

Amazon.com Sales Rank: 24,817

Godel, Escher, Bach : An Eternal Golden Braid
by Douglas R. Hofstadter

List Price: ~~$20.00~~
Our Price: $16.00
You Save: $4.00 (20%)

Used Price: $6.97

Availability: Usually ships within 24 hours.

See larger photo

BUY & SELL USED ITEMS
1 used from $6.97
I have one to sell! Sell yours here

Paperback - 777 pages 20th anniv edition (January 1999)
Basic Books; ISBN: 0465026567 ; Dimensions (in inches): 1.50 × 9.09 × 6.04

Amazon.com Sales Rank: 833

Figure 1: Two book pages from Amazon

An important characteristic of pages belonging to the same site and encoding data of the same schema, is that the data encoding is done in a consistent manner across all the pages. For example, in both the pages of Figure 1, the title of the book appears in the beginning followed by the word "`by`," followed by the author(s). In other words, the above pages are generated using a common "template" by "plugging-in" values for the title, list of authors and so on. Most of the information extraction techniques proposed so far, and the technique that we propose in this paper, exploit the template-based encoding for extracting data from the pages. Specifically, the techniques use either a partial or complete knowledge of the template used to generate the pages, to extract the data. For example, in Figure 1, the price of a book can be extracted by retrieving the text immediately following the template-text "`OurPrice` :."

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

Page	A	B	C	···
1	`MySystem` ...	`Aron` ...	(NULL)	···
2	`Godel`, ...	`Douglas` ...	`20.00`	···
⋮	⋮	⋮	⋮	⋮

Figure 2: Extracted Data

The primary difference between various information extraction techniques lies in how the knowledge of the template is acquired by the extraction system. The earliest information extraction techniques rely on a human to encode knowledge of the template into a program called *wrapper*, which then extracts data. In Hammer et al. [12] a human expresses some part of the template as declarative rules, and a "wrapper generator" converts these rules into a wrapper. More recent systems like XWRAP [18], Wien [15], Stalker [19], and SoftMealy [13] use human generated training examples that identify data in a small number of pages, to learn knowledge of the template.

Unlike the previous work described above, our goal is to deduce the template without any human input, and use the deduced template to extract data. There are at least two reasons why absence of human input is beneficial. First, human input is time consuming and error-prone. A single web page used in training could potentially contain a large number of data values of interest, and the human trainer has to identify each one of them; and the training could require many such human annotated pages. Second, many collections of pages are semi-structured, and contain optional attributes. If an optional attribute appears very rarely, it is possible for the human trainer to miss the optional attribute altogether. Both the above problems are further aggravated by the fact that, in practice, templates change very frequently, requiring repeated human intervention.

There are two fundamental challenges in automatically deducing the template. First, and foremost, there is no obvious way of differentiating between text that is part of template and text that is part of data. Any word could be part of template, or data or both. Note that it is not necessary for a word that is part of template to occur in every page (*e.g.*, "`ListPrice` :" in Figure 1). Conversely, a common English word like "is" could occur as part of data in every input page. Second, the schema of data in pages is usually not a "flat" set of attributes, but is more complex and semi-structured. The schema could contain non-atomic attributes that are sets of values (*e.g.*, set of reviews of a book), or attributes that are optional (*e.g.*, "list" price in Figure 1). The existence of complex schema makes both the tasks of definition and automatic recognition of template harder. In fact the existence of complex schema makes our problem very closely related to the problem of regular expression inference which is known to be very hard (see Section 2.4).

We know of only two previous work on automatic extraction, namely, RoadRunner [5] and IEPAD [4]. There are fundamental differences, both in problem formulation and solution approach, between our work and the above two. Both RoadRunner and IEPAD make the simplifying assumption that an HTML tag is always part of the template of the page. Although, statistically, HTML tags do tend to occur in template, there are a significant number of cases where they occur within data. The implications of the above assumption and the differences in our solution approaches are discussed in Section 7.

We make two clarifications regarding our assumptions and goals. First, our goal is not to try to semantically name the extracted data. We assume that renaming, for example, attribute A in Figure 2 as "TITLE", is done as a postprocessing step, possibly with human help. Second, we assume that our input pages conform to a common schema and template. We do not consider the problem of automatically obtaining such pages from web sites. It is reasonably easy for a human to identify web collections of interest that have a common schema and then run a crawler to gather the pages.

The rest of the paper is organized as follows. Section 2 provides the preliminary definitions, proposes a model for page creation and formally states the data extraction problem. Section 3 provides a brief overview of our algorithm, ExAlg, for solving the extraction problem. ExAlg described in greater detail in sections 4 and 5. Section 6 describes our experiments, while Section 7 describes related work.

2. MODEL, PROBLEM FORMULATION

In this section we formally define structured data, the kind of data that we are hoping to extract from the web pages. We also propose a model for page creation that describes how data is encoded using a template. Finally, we formulate the data extraction problem that we are trying to solve in this paper.

2.1 Structured Data

Structured Data is any set of data values conforming to a common *schema* or *type*. A type is defined recursively as follows [1]:

1. The *Basic Type*, denoted by $\mathcal{B}$, represents a string of *tokens*. A token is some basic unit of text. For the rest of the paper, we define a token to be a word or a HTML tag.

2. If $T_1, \ldots, T_n$ are types, then their ordered list $\langle T_1, \ldots, T_n\rangle$ is also a type. We say that the type $\langle T_1, \ldots, T_n\rangle$ is constructed from the types $T_1, \ldots, T_n$ using a *tuple constructor* of *order* n.

3. If T is a type, then $\{T\}$ is also a type. We say that the type $\{T\}$ is constructed from T using a *set constructor*.

We use the term *type constructor* to refer to either a tuple or set constructor. An *instance* of a schema is defined recursively as follows.

1. An instance of the basic type, $\mathcal{B}$, is any string of tokens.

2. An instance of type $\langle T_1, T_2, \ldots, T_n\rangle$ is a tuple of the form $\langle i_1, i_2, \ldots, i_n\rangle$ where $i_1, i_2, \ldots, i_n$ are instances of types $T_1, T_2, \ldots, T_n$, respectively. Instances $i_1, i_2, \ldots, i_n$ are called attributes of the tuple.

3. An instance of type $\{T\}$ is any set of elements $\{e_1, \ldots, e_m\}$, such that $e_i (1 \leq i \leq m)$ is an instance of type T.

We also use term *value* to denote an instance. Also, *string* denotes a string of tokens. Sometimes type constructor symbols, $\{\}$ and $\langle\rangle$, are subscripted to help us refer to the corresponding type constructors.

Example 2.1. Consider a set of pages, each containing information about a book. Each page contains the title, the set of authors, and the cost of a book. Further, each author has a first name and a last name. Then the schema of the data encoded in the pages is $S_1 = \langle \mathcal{B}, \{\langle \mathcal{B}, \mathcal{B}\rangle_{\tau_3}\}_{\tau_2}, \mathcal{B}\rangle_{\tau_1}$. Schema S_1 has two tuple constructors, τ_1 and τ_3, and one set constructor, τ_2. An instance of S_1 is the value $x_1 = \langle t, \{\langle f_1, l_1\rangle, \langle f_2, l_2\rangle\}, c\rangle$ where, for example, t denotes the title of the book, f_1 denotes the first name of an author and c the cost. □

Schemas and values can be equivalently viewed as trees. Figure 3 shows the tree representation of schema S_1 and value x_1. A sub-tree of a schema tree is also a schema, and is called a *subschema* of the original schema. A sub-value of a value is similarly defined.

2.2 Model of Page Creation

We now describe a model for page creation. According to our model (Figure 4), a value x (taken from a database shown on the left) is encoded into a page using a *template* T. We denote the page resulting from encoding of x using T by $\lambda(T, x)$.

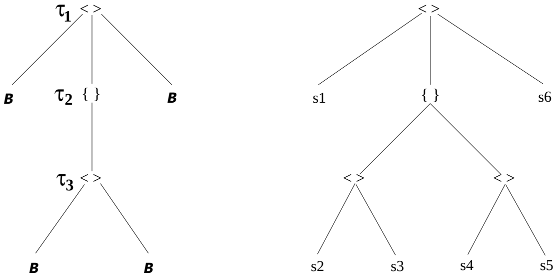


Figure 3: Example Schema and Instance

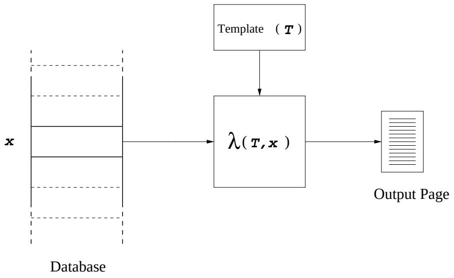


Figure 4: Model for Page Creation

Definition 2.1. (**Template**) A template T for a schema $\mathcal{S}$, is defined as a function that maps each type constructor τ of $\mathcal{S}$ into an ordered set of strings $T(\tau)$, such that,

1. If τ is a tuple constructor of order n, $T(\tau)$ is an ordered set of $n+1$ strings $\langle C_{\tau 1}, \ldots, C_{\tau(n+1)} \rangle$.

2. If τ is a set constructor, $T(\tau)$ is a string S_τ (trivially an ordered set of unit size). □

Optionally, we represent template T as $T^{\mathcal{S}}$ to denote that T is defined for schema $\mathcal{S}$. For case 1 (resp. case 2) of Definition 2.1, we say string $C_{\tau i}$ $(1 \leq i \leq n+1)$ (resp. string S_τ) is *associated* with type constructor τ. If a string is associated with a type constructor in a template, any token that occurs within the string is also said to be *associated* with the type constructor.

Example 2.2. A template ${T_1}^{S_1}$ for Schema S_1 of Example 2.1 is given by the mapping, $T_1(\tau_1) = \langle A, B, C, D \rangle$, $T_1(\tau_2) = H$, $T_1(\tau_3) = \langle E, F, G \rangle$ (each letter $A - H$ is a string). Template ${T_1}^{S_1}$ tells us how to encode a page from a value. For example, the encoding $\lambda(T_1, x_1)$ is the string $AtBEf_1Fl_1GHEf_2Fl_2GCcD$.

For concreteness, let strings $(A - H)$ be as shown in Figure $5(a)$ (ϵ represents an empty string and ␣ represents white space). The web page corresponding to the book tuple ⟨ C Programming Language, { ⟨ Brian, Kernighan⟩, ⟨ Dennis,Ritchie ⟩ }, \$30.00 ⟩ is shown in Figure $6(b)$. □

A	⟨html⟩⟨body⟩⟨b⟩Book :⟨/b⟩
B	by
C	⟨b⟩Cost :⟨/b⟩
D	⟨/body⟩⟨/html⟩
E, G	ϵ
F	␣
H	and

Figure 5: Template of Example 2.2

```
⟨html⟩
    ⟨body⟩
        ⟨b⟩Book :⟨/b⟩C Programming Language
        by Brian Kernighan and Dennis Ritchie
        ⟨b⟩Cost :⟨/b⟩$30.00
    ⟨/body⟩
⟨/html⟩
```

Figure 6: Template and Page of Example 2.2

Formally, given a template, $T^{\mathcal{S}}$, the encoding $\lambda(T, x)$ of an instance x of $\mathcal{S}$ is defined recursively in terms of encoding of subvalues of x. Since it causes no ambiguity, we use the $\lambda(T, x)$ notation for values x that are instances of sub-schema of $\mathcal{S}$.

1. If x is of basic type, $\mathcal{B}$, $\lambda(T, x)$ is defined to be x itself.

2. If x is a tuple of form $\langle x_1, \ldots, x_n \rangle_{\tau_t}$, $\lambda(T, x)$ is the string $C_1\ \lambda(T, x_1)\ C_2\ \lambda(T, x_2)\ \ldots\ \lambda(T, x_n)\ C_{n+1}$. Here, x is an instance of sub-schema that is rooted at type constructor τ_t in $\mathcal{S}$, and $T(\tau_t) = \langle C_1, \ldots, C_{n+1} \rangle$.

3. If x is a set of the form $\{e_1, \ldots, e_m\}_{\tau_s}$, $\lambda(T, x)$ is given by the string $\lambda(T, e_1)\ S\ \lambda(T, e_2)\ S\ \ldots\ S\ \lambda(T, e_m)$. Here x is an instance of sub-schema that is rooted at type constructor τ_s in $\mathcal{S}$, and $T(\tau_s) = S$.

We represent a template using an *infix* notation. For example, the template of Example 2.2 is represented as $\langle A * B\{\langle E * F * G\rangle\}_H C * D \rangle$. The "$*$" symbol is similar to UNIX wild-card, and indicates positions where values of basic type appear in an encoding using the template. Note that string H, associated with τ_2 of Example 2.2 is placed as subscript of $\{\}$.

Our model captures the requirement that the web pages be generated in a consistent manner. In particular, it ensures that values for the same attribute in a tuple occur in the same relative position with respect to the values of other attributes, in all the pages. In Example 2.2 above, the book name always occurs before the list of authors and the price. The encoding of the set captures the intuition that elements of a set are usually listed contiguously, and that the elements of the set are formatted in a similar manner.

2.3 Optionals and Disjunctions

As we saw in Section 2.1, a schema is built from two kinds of type constructors, tuple and set, and the basic type $\mathcal{B}$. There are two other kinds of type constructors that occur commonly in the schema of web pages, namely, optionals and disjunctions. For example the list-price of a book in Amazon book pages is optional since only pages for books sold at a discount price have list-price information. As an example of a disjunction, an address information in a web page could be in one of two formats, based on whether the address is a US address or not, in which case the schema of the address is a disjunction of the schema for US addresses and the schema for non-US addresses.

We view optionals and disjunctions as special type constructors built from set and tuple constructors. If T is a type, then $(T)?$ represents the optional type T, and is equivalent to $\{T\}_\tau$ with the constraint that in any instantiation τ has a cardinality of 0 or 1. Similarly, if T_1 and T_2 are types, $(T_1 \mid T_2)$ represents a type which is disjunction of T_1 and T_2, and is equivalent to $\langle \{T_1\}_{\tau_1}, \{T_2\}_{\tau_2} \rangle_\tau$, where for every instantiation of τ exactly one of τ_1, τ_2 has cardinality one and the other, cardinality zero.

The above view of optionals and disjunctions enables us to use our model of page creation for schema involving optionals and disjunctions without any modification.

2.4 Problem Statement

Extract Problem: Given a set of n pages, $p_i = \lambda(T, x_i)$ $(1 \leq i \leq n)$, created from some unknown template T and values $\{x_1, \ldots, x_n\}$, deduce the template T and values $\{x_1, \ldots, x_n\}$ from the set of pages alone.

In its general form, EXTRACT problem is ill-defined since there are several templates and values that could have created a given set of pages, as the following example illustrates.

Example 2.3. Consider three input pages $p_1 = Aa_1Bb_1Cc_1D$, $p_2 = Aa_2Bb_2Cc_2D$, $p_3 = Aa_3Bb_3Cc_3D$. These pages can be created from the template $\langle A * B * C * D \rangle$ and a corresponding set of values[1]. For instance, the value used to create p_1 is $\langle a_1, b_1, c_1 \rangle$. These pages can also be created from the template $\langle A * C * D \rangle$ and a corresponding set of values. For this template, the value used to create p_1 is $\langle a_1Bb_1, c_1 \rangle$. □

However, given a set of real web pages from a site like Amazon, a human rarely has any ambiguity in picking the right template and values encoded in the pages. Our goal is to solve the EXTRACT problem for real web pages, *i.e.*, produce the template and values that would be considered correct by a human.

Example 2.4. We use the instance of EXTRACT problem with the set of 4 pages $\mathcal{P}_e = \{p_{e1}, p_{e2}, p_{e3}, p_{e4}\}$ shown in Figure 7 as a running example. Each page in $\mathcal{P}_e$ contains the title and the set of reviews of a book. Each review contains the name of the reviewer, the rating given by the reviewer and the text of her comments. The entire text of comments is not shown due to space limitations. Arguably, the pages were created from template T_e, and values $\{x_{e1}, x_{e2}, x_{e3}, x_{e4}\}$ shown in Figure 9 and Figure 8, respectively. The schema of the values is $\mathcal{S}_e = \langle \mathcal{B}, \{\langle \mathcal{B}, \mathcal{B}, \mathcal{B} \rangle_{\tau_{e1}}\}_{\tau_{e2}} \rangle_{\tau_{e3}}$. The correct solution of the EXTRACT problem for the input $\mathcal{P}_e$ is the template T_e and values $\{x_{e1}, x_{e2}, x_{e3}, x_{e4}\}$. □

x_{e1} :	Databases	{ ⟨ John, 7, ... ⟩ }
x_{e2} :	Data Mining	{ ⟨ Jeff, 2, ... ⟩, ⟨ Jane, 6, ... ⟩ }
x_{e3} :	Query Opt.	{ ⟨ John, 8, ... ⟩ }
x_{e4} :	Transactions	ϕ

Figure 8: The correct values

EXTRACT problem is an instance of regular grammar inference problem from positive examples (see [20] for a survey). For example, the pages generated by Template $T = \langle A * B \{\langle C * D \rangle\}\rangle$ belong to language represented by regular expression $L = A\Sigma^*B(C\Sigma^*D)^*$ (note slightly different semantics of "*" in T and L), where Σ denotes the alphabet. Conversely, any string of L is also generated by T. Hence, a solution for regular grammar inference problem can be used to solve EXTRACT. This is currently not feasible for several reasons. The theoretical notion of inference "in the limit" with positive examples alone is undecidable [9]. Hence, known applications of regular grammar inference problem use application specific heuristics for both problem formulation and solution. We have found most such heuristics inapplicable to our problem. An example of such an application is XTRACT [8], a system for learning DTDs from XML documents. XTRACT is designed to learn regular expressions of length of magnitude of 10s of symbols. It uses heuristics "inspired by real-life DTDs" to enumerate a small set of potential regular expressions, and uses MDL [22] principle to pick the "best" one. The size of regular expressions (templates) involved in our problem can often be greater than 10000. Further, the heuristics used by XTRACT does not generate any regular expression involving Σ^*, while it is important for us to consider such regular expressions.

```
⟨
⟨html⟩_1⟨body⟩_2
      ⟨b⟩_3 Book_4 Name_5 ⟨/b⟩_6 *
      ⟨b⟩_7 Reviews_8 ⟨/b⟩_9
      ⟨ol⟩_10
      {⟨
          ⟨li⟩_11
              ⟨b⟩_12 Reviewer_13 Name_14 ⟨/b⟩_15 *
              ⟨b⟩_16 Rating_17 ⟨/b⟩_18 *
              ⟨b⟩_19 Text_20 ⟨/b⟩_21 *
          ⟨/li⟩_22
      ⟩}
      ⟨/ol⟩_23
⟨/body⟩_24 ⟨/html⟩_25
⟩
```

Figure 9: The correct template

2.5 Miscellaneous Terminology, Definitions

An occurrence of a token in a template (resp. value, page) is called a *template-token* (resp. value-token, page-token). Note the distinction between a token and its occurrence. According to our model, each page-token is created from either a template-token or a page-token. Each template-token of T_e in Figure 9 is subscripted to help us refer to it subsequently. The page-tokens of $\mathcal{P}_e$ in Figure 7 that are created from a template-token have the same subscript as the template-token. Two page-tokens are said to have the same *role* if they have been generated by the same template-token. Therefore, two page-tokens in $\mathcal{P}_e$ have the same role *iff* they have the same subscript in Figure 9.

3. OVERVIEW OF OUR APPROACH

In this paper, we present an algorithm, EXALG to solve the EXTRACT problem. Figure 10 shows the different sub-modules of EXALG. Broadly, EXALG works in two stages. In the first stage (ECGM), it discovers sets of tokens associated with the same type constructor in the (unknown) template used to create the input pages. In the second stage (Analysis), it uses the above sets to deduce the template. The deduced template is then used to extract the values encoded in the pages. This section outlines the execution of EXALG for our running example.

In the first stage, EXALG (within Sub-module FINDEQ) computes "equivalence classes" — sets of tokens having the same frequency of occurrence in every page in $\mathcal{P}_e$. An example of an equivalence class (call $\mathcal{E}_{e1}$) is the set of 8 tokens $\{\langle$html$\rangle, \langle$body$\rangle,$ Book, $\ldots, \langle$/html$\rangle\}$, where each token occurs exactly once in every input page. There are 8 other equivalence classes. EXALG retains only the equivalence classes that are large and whose tokens occur in a large number of input pages. We call such equivalence classes LFEQs (for Large and Frequently occurring EQuivalence classes). For the running example there are two LFEQs. The first is $\mathcal{E}_{e1}$ shown above. The second, which we call $\mathcal{E}_{e3}$, consists of the 5 to-

[1] In many, but not all, cases, a template and a page created from the template uniquely identifies the value encoded in the page

```
⟨html⟩_1⟨body⟩_2
   ⟨b⟩_3 Book_4 Name_5 ⟨/b⟩_6 Databases
   ⟨b⟩_7 Reviews_8 ⟨/b⟩_9
   ⟨ol⟩_10
      ⟨li⟩_11
         ⟨b⟩_12 Reviewer_13 Name_14 ⟨/b⟩_15 John
         ⟨b⟩_16 Rating_17 ⟨/b⟩_18 7
         ⟨b⟩_19 Text_20 ⟨/b⟩_21 ...
      ⟨/li⟩_22
   ⟨/ol⟩_23
⟨/body⟩_24 ⟨/html⟩_25
```

(a:p_{e1})

```
⟨html⟩_1⟨body⟩_2
   ⟨b⟩_3 Book_4 Name_5 ⟨/b⟩_6 Data Mining
   ⟨b⟩_7 Reviews_8 ⟨/b⟩_9
   ⟨ol⟩_10
      ⟨li⟩_11
         ⟨b⟩_12 Reviewer_13 Name_14 ⟨/b⟩_15 Jeff
         ⟨b⟩_16 Rating_17 ⟨/b⟩_18 2
         ⟨b⟩_19 Text_20 ⟨/b⟩_21 ...
      ⟨/li⟩_22
      ⟨li⟩_11
         ⟨b⟩_12 Reviewer_13 Name_14 ⟨/b⟩_15 Jane
         ⟨b⟩_16 Rating_17 ⟨/b⟩_18 6
         ⟨b⟩_19 Text_20 ⟨/b⟩_21 ...
      ⟨/li⟩_22
   ⟨/ol⟩_23
⟨/body⟩_24 ⟨/html⟩_25
```

(b:p_{e2})

```
⟨html⟩_1⟨body⟩_2
   ⟨b⟩_3 Book_4 Name_5 ⟨/b⟩_6 Query Opt.
   ⟨b⟩_7 Reviews_8 ⟨/b⟩_9
   ⟨ol⟩_10
      ⟨li⟩_11
         ⟨b⟩_12 Reviewer_13 Name_14 ⟨/b⟩_15 John
         ⟨b⟩_16 Rating_17 ⟨/b⟩_18 8
         ⟨b⟩_19 Text_20 ⟨/b⟩_21 ...
      ⟨/li⟩_22
   ⟨/ol⟩_23
⟨/body⟩_24 ⟨/html⟩_25
```

(c:p_{e3})

```
⟨html⟩_1⟨body⟩_2
   ⟨b⟩_3 Book_4 Name_5 ⟨/b⟩_6 Transactions
   ⟨b⟩_7 Reviews_8 ⟨/b⟩_9
   ⟨ol⟩_10
   ⟨/ol⟩_23
⟨/body⟩_24 ⟨/html⟩_25
```

(d:p_{e4})

Figure 7: Input pages of EXTRACT problem

kens: $\{\langle\texttt{li}\rangle, \texttt{Reviewer}, \texttt{Rating}, \texttt{Text}, \langle/\texttt{li}\rangle\}$. Each token of $\mathcal{E}_{e3}$ occurs once in p_{e1}, twice in p_{e2} and so on. *The basic intuition behind* LFEQ*s is that it is very unlikely for* LFEQ*s to be formed by "chance". Almost always,* LFEQ*s are formed by tokens associated with the same type constructor in the (unknown) template used to create the input pages.* This intuition is easily verified for the running example where all tokens of $\mathcal{E}_{e1}$ (resp. $\mathcal{E}_{e3}$) are associated with τ_{e1} (resp. τ_{e3}) of $\mathcal{S}_e$ in T_e[2].

For this simple example, Sub-module HANDINV does not play any role, but for real pages HANDINV detects and removes "invalid" LFEQs — those that are not formed by tokens associated with a type constructor.

However, not all the tokens associated with τ_{e1} are in $\mathcal{E}_{e1}$. For example, the token Name does not occur in $\mathcal{E}_{e1}$ although it is associated with τ_{e1} in T_e. This happens because Name has multiple "roles" — it is associated with two type constructors, namely, τ_{e1} and τ_{e3}. EXALG tries to add more tokens to LFEQs by "differentiating" roles of tokens using the context in which they occur. For example, EXALG, infers (within Sub-module DIFFFORM)[3] that the "role" of Name when it occurs in Book Name is different from the "role" when it occurs in Reviewer Name, using the fact that these two occurrences always have different paths from the root in the html parse trees of the pages. EXALG also infers (within Sub-module DIFFEQ) that the role of $\langle\texttt{b}\rangle$ when it occurs in $\langle\texttt{b}\rangle$ BookName is different from the role, when it occurs in $\langle\texttt{b}\rangle$ Review, using the fact that these two occur in different "positions" with respect to the LFEQ $\mathcal{E}_{e1}$. The former always occurs between tokens $\langle\texttt{body}\rangle$ and Book of $\mathcal{E}_{e1}$, and the latter between tokens Book and Reviews. Returning to token Name, let us refer to Name as $\texttt{Name}^A$ when it occurs in Book Name and $\texttt{Name}^B$ when it occurs in Reviewer Name. We call $\texttt{Name}^A$ and $\texttt{Name}^B$ *dtokens* (for differentiated tokens). Now, EXALG computes the occurrence frequencies of the dtokens (again within FINDEQ) and checks if they belong to any of the existing LFEQs or form new ones. In this case, $\texttt{Name}^A$ occurs exactly once in every page and is, therefore, added to $\mathcal{E}_{e1}$. Similarly, $\texttt{Name}^B$ is added to $\mathcal{E}_{e3}$. Similarly, the dtokens formed from $\langle\texttt{b}\rangle$ and $\langle/\texttt{b}\rangle$ are added to one of $\mathcal{E}_{e1}$ and $\mathcal{E}_{e3}$. The reader can verify that the above step of differentiating tokens and adding them to existing LFEQs increases the size of $\mathcal{E}_{e1}$ (see Figure 11) from 8 to 13 and the size of $\mathcal{E}_{e3}$ from 5 to 12.

EXALG enters the second stage when it cannot grow LFEQs, or find new ones. In this stage, it builds an output template $T'^{S'}$ using the LFEQs constructed in the previous stage. In order to construct S', EXALG first considers the root LFEQ — the LFEQ whose tokens occur exactly once in every input page. In our running example $\mathcal{E}_{e1}$ is the root LFEQ. EXALG determines the positions between consecutive tokens of $\mathcal{E}_{e1}$ that are *non-empty*[4]. A position between two consecutive tokens is empty if the two tokens always occur contiguously, and non-empty, otherwise. There are two *non-empty positions* in $\mathcal{E}_{e1}$: the position between tokens 6 ($\langle/\texttt{b}\rangle$) and 7 ($\langle\texttt{b}\rangle$), and between tokens 11 ($\langle\texttt{ol}\rangle$) and 12 ($\langle/\texttt{ol}\rangle$). The position between the first ($\langle\texttt{html}\rangle$) and the second ($\langle\texttt{body}\rangle$) token of $\mathcal{E}_{e1}$ is empty since $\langle\texttt{body}\rangle$ always occurs immediately after $\langle\texttt{html}\rangle$. EXALG generates a tuple constructor τ'_{e1} of order 2 (one attribute for each non-empty position of $\mathcal{E}_{e1}$) corresponding to $\mathcal{E}_{e1}$. The first non-empty position does not have any equivalence classes occurring within it. EXALG uses this information to deduce that the type of the first attribute of τ'_{e1} is $\mathcal{B}$. The second non-empty position (between $\langle\texttt{ol}\rangle$ and $\langle/\texttt{ol}\rangle$) always has zero or more occurrences of $\mathcal{E}_{e3}$. For this case, EXALG recursively constructs the type T_{e3} corresponding to $\mathcal{E}_{e3}$, and deduces the type of the second attribute of τ'_{e1} to be $\{T_{e3}\}_{\tau'_{e2}}$. It can be verified that T_{e3} constructed by EXALG is $\langle\mathcal{B},\mathcal{B},\mathcal{B}\rangle_{\tau'_{e3}}$. The output schema, S', produced by EXALG is the type corresponding to root equivalence class, $\mathcal{E}_{e1}$, which is $\langle\mathcal{B},\{\langle\mathcal{B},\mathcal{B},\mathcal{B}\rangle_{\tau'_{e3}}\}_{\tau'_{e2}}\rangle_{\tau'_{e1}}$.

EXALG constructs the output template T' by generating a map-

[2] The subscript $e1$ (resp. $e3$) of $\mathcal{E}_{e1}$ (resp. $\mathcal{E}_{e3}$) has been chosen to correspond to the subscript of τ_{e1} (resp. τ_{e3}). This also explains why there is no $\mathcal{E}_{e2}$ — there are no tokens associated with τ_{e2} in T_e.

[3] For exposition, the sequence of execution of EXALG described here is slightly different from the actual sequence described in Section 4. Actually, DIFFFORM executes before FINDEQ as suggested by Figure 10.

[4] The discussion of this stage of EXALG uses the fact that $\mathcal{E}_{e1}$ and $\mathcal{E}_{e3}$ are *ordered*. We will discuss this in Section 4.

$$\underbrace{\boxed{\langle\texttt{html}\rangle\ \langle\texttt{body}\rangle\ \langle\texttt{b}\rangle\ \texttt{Book Name}\ \langle/\texttt{b}\rangle}}_{C_{11}}\qquad\underbrace{\boxed{\langle\texttt{b}\rangle\ \texttt{Reviews}\ \langle/\texttt{b}\rangle\ \langle\texttt{ol}\rangle}}_{C_{12}}\qquad\underbrace{\boxed{\langle/\texttt{ol}\rangle\ \langle/\texttt{body}\rangle\ \langle/\texttt{html}\rangle}}_{C_{13}}$$

Figure 11: $\mathcal{E}_{e1}$ at end of ECGM module

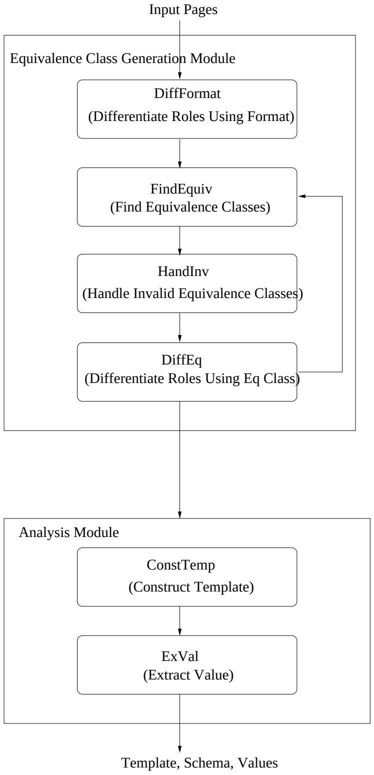

Input Pages
Equivalence Class Generation Module
DiffFormat
(Differentiate Roles Using Format)
FindEquiv
(Find Equivalence Classes)
HandInv
(Handle Invalid Equivalence Classes)
DiffEq
(Differentiate Roles Using Eq Class)
Analysis Module
ConstTemp
(Construct Template)
ExVal
(Extract Value)
Template, Schema, Values

Figure 10: Modules of EXALG

ping from each type constructor in S' to ordered set of strings. By definition, since τ'_{e1} is a tuple constructor of order 2, $T'(\tau'_{e1})$ is an ordered set of 3 strings, $\langle C_{11}, C_{12}, C_{13}\rangle$. EXALG constructs the above 3 strings from tokens of $\mathcal{E}_{e1}$. The string C_{11} is the ordered set of tokens of $\mathcal{E}_{e1}$, that occur before the first non-empty position: $\langle\texttt{html}\rangle\langle\texttt{body}\rangle\ldots\langle/\texttt{b}\rangle$. The string C_{12} is the ordered set of tokens between the first non-empty position and the second. The strings C_{13} is similarly constructed (see Figure 11). EXALG infers that the mapping $T'(\tau'_{e2})$ is the empty string, since there is no "separator" between consecutive occurrences of $\mathcal{E}_{e3}$. The mapping $T'(\tau'_{e3})$ is constructed similar to the mapping $T'(\tau'_{e1})$ described earlier.

The reader can verify that $T' = T_e$ and $S' = \mathcal{S}_e$. We have not described how EXALG extracts the data values. But for this case, the values are uniquely defined given T' and $\mathcal{P}_e$, and can be verified to be equal to $\{x_{e1}, x_{e2}, x_{e3}, x_{e4}\}$. Therefore, EXALG produces the correct output on our running example.

4. EQUIVALENCE CLASSES

This section defines an equivalence class, and describes how equivalence classes are used in EXALG. Except when we refer to our running example, the discussion of sections 4, 5, and 6 is in the context of an arbitrary set of pages $\mathcal{P} = \{p_1, \ldots, p_n\}$, where $p_i = \lambda(T^{\mathcal{S}}, x_i)(1 \leq i \leq n)$. Schema $\mathcal{S}$ consists of type constructors $\{\tau_1, \ldots, \tau_k\}$. The pages $\{p_1, \ldots, p_n\}$ form the input to EXALG. Note, however, that EXALG does not have knowledge of T, $\mathcal{S}$ and $\{x_1, \ldots, x_n\}$.

Definition 4.1. (**Occurrence Vector**) The *occurrence-vector* of a token t, is defined as the vector $\langle f_1, \ldots, f_n\rangle$, where f_i is the number of occurrences of t in p_i. □

Definition 4.2. (**Equivalence Class**) An *equivalence class* is a maximal set of tokens having the same occurrence-vector. □

The set of equivalence classes define a partition over the set of tokens that occur in $\mathcal{P}$. As we saw in Section 3, there are 9 equivalence classes (including $\mathcal{E}_{e1}$ and $\mathcal{E}_{e3}$) for pages $\mathcal{P}_e$ of our running example. The occurrence vector of tokens in $\mathcal{E}_{e1}$ is $\langle 1, 1, 1, 1\rangle$ and the occurrence vector of tokens in $\mathcal{E}_{e3}$ is $\langle 1, 2, 1, 0\rangle$.

We are interested in equivalence classes because, in practice, tokens associated with the same type constructor in T, tend to occur in the same equivalence class. In our running example, 8 of the 13 tokens associated with τ_{e1} in T_e, occur in $\mathcal{E}_{e1}$. Observe that all occurrences of these 8 tokens are generated by unique template-tokens. For example, all occurrences of token $\langle\texttt{html}\rangle$ are generated by template-token $\langle\texttt{html}\rangle_1$. One the other hand, a token like $\texttt{Name}$ that does not occur in $\mathcal{E}_{e1}$ (in spite of being associated with τ_{e1} in T_e) is generated by more than one template-token, namely, $\texttt{Name}_5$ and $\texttt{Name}_{14}$. A token is said to have *unique role*, if all the occurrences of the token in the pages, is generated by a single template-token.

Observation 4.1. Tokens associated with the same type constructor τ_j in T that have unique-roles occur in the same equivalence class. □

If, as in Observation 4.1, all the tokens of an equivalence class, $\mathcal{E}$, have unique roles and are associated with the same type constructor τ_j of S, we say that $\mathcal{E}$ is *derived* from τ_j. We call an equivalence class *valid* if it is derived from some τ_j, and *invalid*, otherwise. For instance, in our running example, the equivalence class $\{\texttt{Data}, \texttt{Mining}, \texttt{Jeff}, \texttt{2}, \texttt{Jane}, \texttt{6}\}$ (with occurrence vector $\langle 0, 1, 0, 0\rangle$) is invalid. However, observe that the tokens in this equivalence class occur very infrequently—in just a single page. The above observation is valid in general for real web pages and is formalized below. Define *support* of a token, to be the number of pages in which the token occurs. The support of an equivalence class is the common support of the tokens in it. The *size* of an equivalence class is the number of tokens in the equivalence class.

Observation 4.2. For real pages, an equivalence class of large size and support is usually valid. □

We call such equivalence classes LFEQs (for Large and Frequent EQuivalence class). Observation 4.2 is true because LFEQs are rarely formed by "chance". Two tokens rarely have the same occurrence frequency in a large number of pages unless they occur in the pages due to the same "reason". Typically, the number of times two type constructors are instantiated is not the same in every input page[5]. Therefore, tokens associated with different type constructors usually do not occur in the same equivalence class. Tokens generated by value-tokens (*e.g.*, $\texttt{Databases}$ in our running example) usually occur infrequently and therefore do not occur in an LFEQ.

[5] This statement is not valid if Schema $\mathcal{S}$ is not in "canonical" form (*e.g.*, $\langle\langle\mathcal{B}\rangle_{\tau_1}, \langle\mathcal{B}\rangle_{\tau_2}\rangle$). However, for any schema there always exists a "structurally equivalent" schema in canonical form (*e.g.*, $\langle\mathcal{B}, \mathcal{B}\rangle$ for the example schema above).

$$\boxed{p : \overbrace{t_1 \underbrace{\ldots}_{\mathrm{Pos}(1)} t_2 \underbrace{\ldots}_{\mathrm{Pos}(2)} t_3}^{\text{Span of first occurrence}} \ldots \overbrace{t_1 \underbrace{\ldots}_{\mathrm{Pos}(1)} t_2 \underbrace{\ldots}_{\mathrm{Pos}(2)} t_3}^{\text{Span of second occurrence}}}$$

Figure 12: Occurrence and span of an occurrence of equivalence class

Observation 4.2 forms the crux of our extraction technique, which can be loosely summarized as follows: *since typically* LFEQ*s consist only of tokens associated with the same type constructor in the (unknown) input template, use* LFEQ*s to deduce the template and schema*.

There are two main obstacles that we must overcome in order to make the above idea feasible. First, note that Observation 4.2 is heuristic. There is no guarantee that all the LFEQs for a set of pages satisfy Observation 4.2. In practice, we have observed that there are always some invalid LFEQs. Second, an LFEQ, even if it is valid, only contains tokens that have unique roles, and therefore, only contains partial information about the template used to generate the pages. We address both these obstacles in this section. But, in order to do so, we observe a few properties that valid equivalence classes satisfy.

4.1 Properties of Equivalence classes

Definition 4.3. (**Ordered Equivalence Classes**) An equivalence class is *ordered*, if its tokens can be ordered $\langle t_1, \ldots, t_m \rangle$, such that, for every page p_i $(1 \leq i \leq n)$, and every pair of tokens t_j, t_k $(1 \leq j < k \leq m)$,

1. If t_j occurs at least l times in p_i, the l^{th} occurrence of t_j in p_i occurs before the l^{th} occurrence of t_k in p_i, and
2. If t_j occurs at least $(l{+}1)$ times in p_i, the $(l{+}1)^{st}$ occurrence of t_j in p_i is after the l^{th} occurrence of t_k in p_i.

We denote the above ordered equivalence class by $\langle t_1, \ldots, t_m \rangle$. □

Let $\mathcal{E} = \langle t_1, \ldots, t_m \rangle$ be an ordered equivalence class, and let the tokens of $\mathcal{E}$ occur f times in page p_j. Then, we say that $\mathcal{E}$ *occurs* f times in p_j. The i^{th} occurrence of $\mathcal{E}$ refers collectively to the i^{th} occurrence of tokens $t_1, \ldots, t_m$ in p_j. The *span* of the i^{th} occurrence of $\mathcal{E}$ in p_j is the text starting at (and including) i^{th} occurrence of t_1 and ending at (and including) i^{th} occurrence of t_m in p_j. The span of each occurrence of $\mathcal{E}$ is sub-divided into $(m-1)$ *positions*, namely, $\mathrm{Pos}(1), \ldots, \mathrm{Pos}(m-1)$. $\mathrm{Pos}(k)$ $(1 \leq k < m)$ of i^{th} occurrence of $\mathcal{E}$ in p_j denotes the text starting at (but not including) i^{th} occurrence of t_k and ending at (but not including) i^{th} occurrence of t_{k+1}. Figure 12 illustrates the span and and $\mathrm{Pos}(i)$ $(1 \leq i \leq 2)$ for two occurrences of an equivalence class $\langle t_1, t_2, t_3 \rangle$ in a page.

Definition 4.4. (**Nesting of Equivalence classes**) A pair of equivalence classes, $\mathcal{E}_i$ and $\mathcal{E}_j$ is *nested* if,

1. The span of any occurrence of $\mathcal{E}_i$ does not overlap with the span of any occurrence of $\mathcal{E}_j$, or
2. The span of all occurrences of $\mathcal{E}_j$ is within $\mathrm{Pos}(p)$ of some occurrence of $\mathcal{E}_i$ for some fixed p; or vice-versa.

A set of equivalence classes $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ is nested if every pair of equivalence classes of the set is nested. □

Observation 4.3. A valid equivalence class is ordered and a pair of two valid equivalence classes is nested. □

It can be verified that $\mathcal{E}_{e1}$ and $\mathcal{E}_{e3}$ are ordered. The set $\{\mathcal{E}_{e1}, \mathcal{E}_{e3}\}$ is nested since the span of each occurrence of $\mathcal{E}_{e3}$ is always within $\mathrm{Pos}(5)$ of an occurrence of $\mathcal{E}_{e1}$.

4.2 Handling Invalid Equivalence classes

As we mentioned earlier, there are always some invalid LFEQs that are formed, for most input sets of web pages. However, typically invalid LFEQs are either not ordered or not nested with respect to other LFEQs. Module HANDINV takes as input a set of LFEQs (determined by FINDEQ), detects the existence of invalid LFEQs using violations of ordered and nesting properties, and "processes" the invalid LFEQs found — it discards some of the LFEQs completely, and breaks others into smaller LFEQs. The output of HANDINV is an ordered set of nested (with high probability valid, see Section 6) LFEQs. A detailed description of HANDINV is given in the full version of the paper.

4.3 Differentiating roles of tokens

Recall that the fundamental idea of EXALG is to use LFEQs to discover the template tokens. However, typically an LFEQ only contains tokens that have unique roles. Therefore, not all template-tokens can be discovered using LFEQs. This section presents a powerful technique, called *differentiating roles of tokens*, that is used in EXALG to discover a greater number (in practice, almost all) of template-tokens. Briefly, when we differentiate roles of tokens, we identify "contexts" such that the occurrences of a token in different contexts above necessarily have different roles. The notion of a context should be clear when we present the two techniques for differentiating roles used in EXALG.

The first technique for differentiating roles uses the html formatting information of input pages. An html page can be equivalently viewed as a parse tree. An *occurrence-path* of a page-token is the path from the root to the page-token in the parse tree. For instance, the occurrence-path of the first $\langle$/b$\rangle$ in p_{e1} is $\langle\texttt{html}\rangle\langle\texttt{body}\rangle\langle/\texttt{b}\rangle$[6].

Observation 4.4. In practice, two page-tokens with different occurrence paths have different roles. □

Equivalently, Observation 4.4 asserts that all page-tokens generated by a template-token have the same occurrence-path. It can be verified that Observation 4.4 is valid for our running example. In the full version of the paper, we use well-formed properties of html pages to argue that Observation 4.4 is true for real-world pages and templates.

The second technique for differentiating roles uses valid equivalence classes, and is based on the following observation.

Observation 4.5. Let $\mathcal{E}$ be a valid equivalence class derived from τ_i. The role of an occurrence of a token t, which is outside the span of any occurrence of $\mathcal{E}$, is different from the role of an occurrence which is within the span of some occurrence of $\mathcal{E}$. Further, the role of an occurrence of t, which is within $\mathrm{Pos}(l)$ of some occurrence of $\mathcal{E}$, is different from the role of an occurrence of t, which is within $\mathrm{Pos}(m)$ $(m \neq l)$ of some occurrence of $\mathcal{E}$. □

Equivalently, Observation 4.5 asserts that all page-tokens generated by a template-token occur within a fixed $\mathrm{Pos}(p)$ of $\mathcal{E}$, or outside the span of any occurrence of $\mathcal{E}$. Observation 4.5 can be proved in a straight-forward way based on the definition of our model. In our running example, all page-tokens generated by template-token $\langle\texttt{b}\rangle_3$ occur in $\mathrm{Pos}(2)$ of some occurrence of $\mathcal{E}_{e1}$, and outside the span of any occurrence of $\mathcal{E}_{e3}$.

We differentiate roles of a token by identifying a set of contexts for the token using Observation 4.4 or 4.5, such that, each occurrence of the token is within some unique context of the set; and,

[6] There is a bit of abuse of notation here. The $\langle\texttt{html}\rangle$ in the occurrence-path above does not refer to start-tag, but to the html "element" in the parse tree.

occurrences of the token in different contexts has different roles. The set of contexts is the set of occurrence-paths of the token, if we use Observation 4.4, and the set of positions of $\mathcal{E}$, if we use Observation 4.5 with a valid equivalence class $\mathcal{E}$. We use the term *dtoken* (for differentiated token) to jointly refer to a token and a context, identified by differentiation. For example, if we differentiate token $\langle\texttt{b}\rangle$ in our running example using Observation 4.4, 2 dtokens are formed: one corresponding to the occurrence-path (context) $\langle\texttt{html}\rangle\langle\texttt{body}\rangle\langle\texttt{b}\rangle$, and the other to $\langle\texttt{html}\rangle\langle\texttt{body}\rangle\langle\texttt{ol}\rangle\langle\texttt{li}\rangle\langle\texttt{b}\rangle$. Instead, if we differentiate using Observation 4.5 with $\mathcal{E}_{e1}$, 3 dtokens are formed: the first corresponds to context defined by $\mathrm{Pos}(2)$ of occurrences of $\mathcal{E}_{e1}$, the second to context defined by $\mathrm{Pos}(3)$, and the third to context defined by $\mathrm{Pos}(5)$.

A dtoken is almost like a token (a token is a dtoken with no context). We extend the notation defined for tokens to dtokens. The following is a collection of statements and notation related to dtokens: Each occurrence of a dtoken is generated by a template-token or a value-token; by definition, each template-token generates a unique dtoken; a dtoken is said to have a *unique role* if all occurrences of the dtoken is generated by a single template token; a page can be viewed as a string of dtokens.

4.3.1 Equivalence Classes and dtokens

For exposition, we have defined equivalence classes as sets of tokens. In fact, ExAlg works with equivalence classes defined using dtokens. Most of the discussion in this section admits a straightforward generalization from tokens to dtokens. We re-state the main ideas in terms of dtokens.

An occurrence vector of a dtoken is the vector of occurrence frequencies of the dtoken in the input pages. An equivalence class is a maximal set of dtokens having the same occurrence vector. The dtokens generated by tokens associated with the same type constructor τ_j in T and having unique roles occur in the same equivalence class (generalization of Observation 4.1). Observation 4.2 and Observation 4.3 are also valid for equivalence classes defined using dtokens. Section 4.2 can also be generalized to dtokens. Finally, the roles of dtokens itself could be further differentiated using one of the two techniques described earlier. As an illustration of the last statement, consider the three dtokens formed by differentiating roles of token $\langle\texttt{b}\rangle$ using Observation 4.5 with $\mathcal{E}_{e1}$. The third dtoken (one with context $\mathrm{Pos}(5)$ of $\mathcal{E}_{e1}$) can be further differentiated into 3 new dtokens using Observation 4.5 with a different equivalence class $\mathcal{E}_{e3}$. For instance, the first of the 3 new dtokens corresponds to context defined by $\mathrm{Pos}(5)$ of $\mathcal{E}_{e1}$, and $\mathrm{Pos}(1)$ of $\mathcal{E}_{e3}$.

4.4 Equivalence Class Generation Module

The input to ECGM is the set of input pages $\mathcal{P}$. The output of ECGM is a set of LFEQs of dtokens and pages $\mathcal{P}$ represented as strings of dtokens.

First, Sub-module DiffForm differentiates roles of tokens in $\mathcal{P}$ using Observation 4.4, and represents the input pages $\mathcal{P}$ as strings of dtokens formed as a result of the differentiation. The sub-modules FindEq, HandInv and DiffEq iterate in a loop. In each iteration, the input pages are represented as strings of dtokens. This representation changes from one iteration to other because new dtokens are formed in each iteration. FindEq computes occurrence vectors of the dtokens in the input pages and determines LFEQs. FindEq needs two parameters, SizeThres and SupThres, to determine if an equivalence class is an LFEQ. Equivalence classes with size and support greater than SizeThres and SupThres, respectively, are considered LFEQs. HandInv processes LFEQs determined by FindEq, as described in Section 4.2 and produces a nested set of ordered LFEQs. DiffEq optimistically assumes that each LFEQ produced by HandInv is valid, and uses Observation 4.5 to differentiate dtokens. If any new dtokens are formed as a result, it modifies the input pages to reflect the occurrence of the new dtokens, and the control passes back to FindEq for another iteration. Otherwise, ECGM terminates with the set of LFEQs output by HandInv, and the current representation of input pages as the output.

On our running example, with SizeThres and SupThres both set to 3, ECGM runs for two iterations, and produces two equivalence classes, $\mathcal{E}^+_{e1}$ and $\mathcal{E}^+_{e3}$, of sizes 13 and 12, respectively. The ordered set of tokens corresponding to dtokens in $\mathcal{E}^+_{e1}$ is $\langle\langle\texttt{html}\rangle, \langle\texttt{body}\rangle, \langle\texttt{b}\rangle, \texttt{Book}, \ldots, \langle/\texttt{body}\rangle, \langle/\texttt{html}\rangle\rangle$, and that of $\mathcal{E}^+_{e3}$ is $\langle\langle\texttt{li}\rangle, \langle\texttt{b}\rangle, \texttt{Reviewer}, \ldots, \langle/\texttt{li}\rangle\rangle$.

We conclude this section with a remark on representation of dtokens. It might seem extremely complex to store context information of a dtoken. In fact, it is not necessary to explicitly store any context information of a dtoken. Context information of a dtoken is implicitly stored in its occurrences in the pages. In our prototype implementation we used integers to represent dtokens, and maintained a mapping from each dtoken integer to the token (a character string) corresponding to the dtoken.

5. BUILDING TEMPLATE AND EXTRACTING VALUES

This section describes Analysis module of ExAlg. The input of Analysis module is a set of LFEQs and a set of pages represented as strings of dtokens, and the output a template and a set of values. Analysis module consists of two sub-modules: ConstTemp and ExVal (Figure 10). We do not describe ExVal in this paper since it is reasonably straightforward to derive it.

5.1 Notation

We need the following algebra of templates to describe the recursive construction of templates in ConstTemp: (a) If ${T_1}^{S_1}, {T_2}^{S_2}$, $\ldots {T_m}^{S_m}$ are templates, and $C_1, C_2, \ldots, C_{m+1}$ are strings, $T^S = \langle T_1, T_2, \ldots, T_m\rangle_{\langle C_1, C_2, \ldots, C_{m+1}\rangle}$ denotes a template, where $S = \langle S_1, S_2, \ldots S_n\rangle_\tau$. T is defined by mappings $T(\tau) = \langle C_1, C_2, \ldots, C_{m+1}\rangle$, and $T(\tau_k) = T_i(\tau_k)$, for all τ_k in $S_i (1 \leq i \leq m)$; (b) If $T_i^{S_i}$ is a template, and H a string, $T^S = \{{T_i}^{S_i}\}_H$ denotes a template, where $S = \{S_i\}_\tau$. T is defined by mappings $T(\tau) = \langle H\rangle$ and $T(\tau_k) = T_i(\tau_k)$, for all τ_k in S_i; (c) $T_i^{S_i}$ (for schema $(S_i)?$) and $(T_i^{S_i} \mid T_j^{S_j})$ (for schema $(S_i \mid S_j)$) are similarly defined; (d) $T_{\mathcal{B}}$ denotes the trivial template for the basic type $\mathcal{B}$.

$\mathrm{Pos}(p)$ of an ordered equivalence class $\mathcal{E} = \langle d_1, d_2, \ldots, d_n\rangle$ is defined to be *empty* if dtokens d_p and d_{p+1} always occur contiguously. An equivalence class is defined to be *empty* if all its positions are empty. In our running example, both $\mathcal{E}^+_{e1}$ and $\mathcal{E}^+_{e3}$ are non-empty: $\mathrm{Pos}(6)$ and $\mathrm{Pos}(10)$ of $\mathcal{E}^+_{e1}$ are non-empty; $\mathrm{Pos}(5)$, $\mathrm{Pos}(8)$ and $\mathrm{Pos}(11)$ of $\mathcal{E}^+_{e3}$ are non-empty.

For an occurrence of equivalence class $\mathcal{E}$ and a non-empty $\mathrm{Pos}(p)$ of $\mathcal{E}$, $\mathrm{PosString}(\mathcal{E}, p)$ is the string formed by concatenating tokens and equivalence classes[7] that occur in $\mathrm{Pos}(p)$ of that occurrence of $\mathcal{E}$, but do not occur within the span of some other equivalence class $\mathcal{E}'$ whose span is also within $\mathrm{Pos}(p)$ of the above occurrence of $\mathcal{E}$. As an example, $\mathrm{PosString}(\mathcal{E}^+_{e1}, 10)$ of the only occurrence of $\mathcal{E}^+_{e1}$ in p_{e2} is the string "$\mathcal{E}^+_{e3}\mathcal{E}^+_{e3}$". Although a dtoken formed from token `Rating` occurs in $\mathrm{Pos}(10)$ of the above occurrence $\mathcal{E}^+_{e1}$, it is not present in $\mathrm{PosString}(\mathcal{E}^+_{e1}, 10)$ since it is within the span of an

[7] More formally, some unique symbol corresponding to each equivalence class. We use the name of the equivalence class (*e.g.*, $\mathcal{E}^+_{e1}$, $\mathcal{E}^+_{e3}$) as its symbol.

occurrence of $\mathcal{E}^+_{e3}$.

5.2 ConstTemp

Let $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_m\}$ be the input set of LFEQs of ANALYSIS module. For every non-empty equivalence class $\mathcal{E}_i$, CONSTTEMP recursively constructs a template, $\mathcal{T}_{\mathcal{E}_i}$, corresponding to $\mathcal{E}_i$, and a template, $T_{\mathcal{E}_{i,p}}$, corresponding to each non-empty position p of $\mathcal{E}_i$. The output template of CONSTTEMP is the template corresponding to the root equivalence class — the equivalence class with occurrence vector $\langle 1, 1, \cdots \rangle$[8].

The template $T_{\mathcal{E}_i}$ is defined in terms of $T_{\mathcal{E}_{i,p}}$. Let $\mathcal{E}_i = \langle d_1, d_2, \ldots, d_l \rangle$, and let $t_i (1 \leq i \leq l)$ be the token corresponding to dtoken d_i. Let $s_i (1 \leq i \leq q)$ denote the non-empty positions of $\mathcal{E}_i$. Define $q+1$ strings $C_{i_1}, C_{i_2}, \ldots, C_{i_{q+1}}$ as follows: $C_{i_1} = t_1 \ldots t_{s_1}$, $C_{i_j} = t_{s_{(j-1)}+1} \ldots t_{s_j}$ $(1 < j \leq q)$, and $C_{q+1} = t_{s_q} \ldots t_l$. The strings $C_{i_j} (1 \leq j \leq q+1)$ just partition the tokens $t_1, \ldots, t_l$ using the non-empty positions of $\mathcal{E}_i$. The template $T_{\mathcal{E}_i}$ is defined as: $T_{\mathcal{E}_i} = \langle T_{\mathcal{E}_{i,s_1}}, \ldots, T_{\mathcal{E}_{i,s_q}} \rangle_{\langle C_{i_1}, C_{i_2}, \ldots, C_{i_{q+1}} \rangle}$.

To construct template $T_{\mathcal{E}_{i,p}}$, CONSTTEMP checks if the set of strings, $\mathrm{PosString}(\mathcal{E}_i, p)$, corresponding to every occurrence of $\mathcal{E}_i$, has some recognizable pattern. Table 1 lists some patterns that our prototype implementation of CONSTTEMP used, and the definition of $T_{\mathcal{E}_{i,p}}$ for each pattern, if the set of strings, $\mathrm{PosString}(\mathcal{E}_i, p)$, has that pattern. In our running example, $\mathrm{PosString}(\mathcal{E}^+_{e1}, 6)$ is a string of dtokens for every occurrence of $\mathcal{E}^+_{e1}$, which matches Pattern 5 of Table 1. Therefore, $T_{\mathcal{E}^+_{e1},6}$ is defined to be $T_{\mathcal{B}}$. $\mathrm{PosString}(\mathcal{E}^+_{e1}, 10)$ is always a string of 0 or more occurrences of "$\mathcal{E}^+_{e3}$", which matches Pattern 1, and hence $T_{\mathcal{E}^+_{e1},10}$ is defined to be $\{T_{\mathcal{E}^+_{e3}}\}_\epsilon$. The reader can recursively construct $T_{\mathcal{E}^+_{e3}}$ and verify that the output template, $T_{\mathcal{E}^+_{e1}}$ produced by CONSTTEMP is the same as the correct template T_e.

	Pattern	$T_{\mathcal{E}_{i,p}}$
1	$\mathcal{E}_j \mathcal{E}_j \ldots$	$\{T_{\mathcal{E}_j}\}_\epsilon$
2	$\mathcal{E}_j S \mathcal{E}_j S \ldots \mathcal{E}_j$	$\{T_{\mathcal{E}_j}\}_S$
3	$\mathcal{E}_j$ or $\mathcal{E}_k$	$T_{\mathcal{E}_j} \mid T_{\mathcal{E}_k}$
4	ϵ or $\mathcal{E}_j$	$(T_{\mathcal{E}_j})?$
5	string of dtokens and empty eq. classes	$T_{\mathcal{B}}$
6	Unknown	$T_{\mathcal{B}}$

Table 1: Patterns used in definition of $T_{\mathcal{E}_{i,p}}$

6. EXPERIMENTS

EXALG makes several assumptions regarding the unknown template and values used to generate its input pages. We summarize the important assumptions:

A1: A large number of tokens occurring in template have unique roles, to bootstrap the formation of equivalence classes and subsequent differentiation.

A2: A large number of tokens is associated with each type constructor. Further, each type constructor is instantiated a large number of times in the input pages. This assumption is ensures that the equivalence class derived from a type constructor is recognized as an LFEQ.

A3: There is no "regularity" in encoded data that leads to the formation of invalid equivalence classes.

A4: There are "separators" around data values. In our model, this translates to the assumption that the strings associated with type constructors are non-empty. As we indicated in Section 1 this assumption is made in some related in most information extraction tasks.

All the assumptions above are heuristics. We study experimentally (a) to what extent the assumptions are satisfied, and (b) the impact on the output of EXALG when some of the assumptions are not satisfied. For the purpose of experimentation we have built a data extraction system based on EXALG.

6.1 Setup

We conducted experiments on 46 different input collections of pages. These collections were obtained as follows:

- RISE [21]: (6 collections) RISE is a repository of collections used for experimental evaluation of IE techniques. RISE includes collections used in evaluation of WIEN and STALKER. Only 6 of the 10 collections in RISE are relevant to our extraction problem. The rest of the collections are meant for extraction from more unstructured human generated data.
- ROADRUNNER [5]: (15 collections) These represent all the collections available at ROADRUNNER site [23] (except the repetitions from RISE).
- IEPAD [4]: (14 collections) These represent all the collections obtained from IEPAD site [14]. The pages in these collections are results of queries from various search-engines.
- *Misc*: (10 collections) These collections were crawled by us from various well-known sites like E-bay, DBLP, Google and Sigmod Anthology.

6.2 Evaluation

For comparison purposes, we *manually* generated the schema, S_m, of the values encoded in each page, of a collection C. The schema, S_m, was generated based on the semantics of the source. For example, if the input consisted of Amazon book pages (Fig. 1), the manually generated schema would have attributes like title, set of authors and optional "list" price. We ignored values encoded as tag attributes (*e.g.*, urls, images) while generating S_m. Further, we picked the granularity of the leaf attributes (attributes of basic type, $\mathcal{B}$) of S_m based on existence of separators. For example, we did not split a date (*e.g.*, `Nov. 8 2002`) into attributes for day, month and year. Let S_e denote the schema automatically extracted by our system for Collection C. For evaluation, we considered each leaf attribute A_m in S_m, and classified it into one of the following 3 categories to reflect how successful our system was in extracting values of A_m.

- *Correct:* A_m was classified as correct if there existed a leaf attribute A_e in S_e such that for each page in C, the set of values of A_m in the page is equal to the set of extracted values of A_e in the page.

- *Partially Correct:* A_m was classified as partially correct if it was not correct and there existed a leaf attribute A_e in S_e such that for each page in C, each value of A_m occurred as part of a value of A_e in that page. This happens when the granularity of extraction is coarser than the desired granularity. For example, for Amazon book pages (Fig. 1), if the contiguous attributes "List Price" and "Our Price" were extracted as a single attribute by our system (with the text "`OurPrice :`" occurring within each extracted value of the attribute), then both the former attributes would be classified as partially correct.

[8] We can always ensure that such an equivalence class exists be prepending and appending greater than SIZETHRES number of dummy tokens to beginning and end of each page respectively

- *Incorrect:* A_m was classified as incorrect if it was neither correct nor partially correct. This happens when our system mis-aligns values of A_m, *i.e.*, different values of A_m occur as part of values of different extracted attributes.

Our evaluation scheme serves two purposes. First, it gives us insight as to what extent the assumptions A1-A4 hold. Specifically, it can be shown that if Assumption A3 holds, and no invalid equivalence classes are formed, then, none of the attributes of S_m would be classified incorrect. In addition, if assumptions A1 and A2 are satisfied as well, then all the attributes of S_m would be classified as correct. Second, the evaluation scheme indicates how useful our system is in extracting data. Clearly, the output of our system is most useful when it extracts attributes in S_m correctly. However, the output of our system is useful to some extent even when some attributes of S_m are extracted only partially correctly. If a leaf attribute A_m of S_m was classified as partially correct (by identifying an extracted leaf attribute A_e), then the correct values for A_m can be extracted by just focusing on the extracted values of A_e. This task is likely to be easier than extracting values of A_m from the entire page.

Although, our evaluation scheme just checks the correctness of leaf attributes, and not the entire schema S_m, almost always the correctness of the leaf attributes closely reflects the correctness of the entire schema. It is extremely hard for our algorithm to extract the leaf attributes correctly while deducing the schema of the attributes wrongly.

6.3 Results

We have placed the detailed results of our experiments at the URL [6]. The above link contains, for each collection, the set of input pages in the collection, the template discovered by our system, and the values extracted for each input page. It also has a log of the execution of our system for each input collection. The log contains the information like the set of LFEQs formed, the set of strings $\mathrm{PosString}(\mathcal{E}, p)$ and the pattern that matches the set (Section 5), for every non-empty position p of an LFEQ $\mathcal{E}$, and so on. Finally, the above link contains details of our evaluation — the manual schema S_m that we constructed for each input collection, and for each attribute A_m in S_m, the category that we assigned A_m to, and the reason for doing so.

Table 2 summarizes our experimental results. Column S of Table 2 indicates the source of a collection (see beginning of this section): I (IEPAD), R (RISE), RR (ROADRUNNER) and M (Misc.). Columns N and a denote, respectively, the number of pages and the number of attributes in the manual schema S_m for a collection; Columns c, p and i denote the number of leaf attributes of S_m that were classified as correct, partially correct and incorrect, respectively.

The results in Table 2 clearly demonstrate that EXALG is very effective in extracting data. For 18 or 40% of the input collections our system correctly extracted all the attributes. For other collections there were a small number of attributes that were extracted only partially correctly. Specifically, on an average, around 80% of the attributes were extracted correctly. Column Acc of Table 3 shows the accuracy of our system for each of the 4 classes of collections. Since the number of attributes varies widely from one collection to other, we also computed the normalized average by scaling the number of leaf attributes in S_m to the same value for all collections, and this is shown for each class of collection in Column $Norm.\ Acc$. Finally, for all our inputs there were no attributes of S_m that were incorrectly extracted.

S	Site	N	a	c	p	i
$I1$	Altavista	10	11	9	2	0
$I2$	Cora	10	7	7	0	0
$I3$	Excite	10	11	11	0	0
$I4$	Galaxy	10	15	12	3	0
$I5$	Hotbot	10	6	5	1	0
$I6$	LA Weekly	10	6	4	2	0
$I7$	Lycos	10	21	19	2	0
$I8$	Magellan	10	5	5	0	0
$I9$	Metacrawler	10	8	3	5	0
$I10$	Northernlight	10	7	7	0	0
$I11$	Openfind	10	?	?	?	?
$I12$	SavvySearch	10	6	6	0	0
$I13$	Stptcom	10	6	6	0	0
$I14$	Webcrawler	10	10	10	0	0
$R1$	Bigbook	235	5	5	0	0
$R2$	IAF	200	7	1	6	0
$R3$	Okra	252	8	8	0	0
$R4$	Quote Server	200	16	16	0	0
$R5$	Zagat's Guide	91	4	1	3	0
$R6$	LA Weekly	28	6	4	2	0
$RR1$	Amazon(Pop)	19	5	5	0	0
$RR2$	Amazon(Cars)	21	13	13	0	0
$RR3$	buy.com(subcat)	10	9	9	0	0
$RR4$	buy.com(prod)	10	11	10	1	0
$RR5$	wine.com(acc)	10	5	5	0	0
$RR6$	wine.com(prod)	10	8	4	4	0
$RR7$	uefa.com(teams)	20	9	9	0	0
$RR8$	uefa.com(play)	20	2	2	0	0
$RR9$	MLB(players)	10	7	7	0	0
$RR10$	MLB(stat)	10	62	49	13	0
$RR11$	Barn.&Nob.	7	5	5	0	0
$RR12$	Barn.&Nob.(SW)	10	4	4	0	0
$RR13$	nba.com	10	98	66	32	0
$RR14$	rpmfind.net	20	6	6	0	0
$RR15$	rpmfind.net	20	4	4	0	0
$M1$	E-bay	50	22	18	4	0
$M2$	Netflix	50	29	23	6	0
$M3$	US Open	32	35	33	2	0
$M4$	DBLP	25	7	2	5	0
$M5$	Google	20	10	6	4	0
$M6$	Citeseer	50	14	6	8	0
$M7$	Sigmod(Anth)	100	12	10	2	0
$M8$	Patents	50	23	16	7	0
$M9$	CNET	25	10	9	1	0
$M10$	Slashdot	24	10	8	2	0
	Total		585	468	117	0

Table 2: Experimental Results

We highlight, from our experiments, some common cases when our system failed to extract data correctly. Since several attributes were extracted only partially correctly, clearly some assumptions A1, A2 or A4 failed to hold. We have observed empirically that the most common case is failure of Assumption A2, *i.e.*, there are type constructors that have a very few template tokens associated with them. For example, Collection $R2$, on which our system performed badly, contains a set of addresses in each page encoded with the template[9] $\{\langle \texttt{Name} :* \texttt{<br>}, (\texttt{Email} :* \texttt{<br>})?, (\texttt{Organization} :* \texttt{<br>})?, (\texttt{Update} : * \texttt{<br>})?\rangle\}$. For this template there are just two template tokens associated with each type constructor. Consequently, these template tokens were not discovered using LFEQs, and our system failed to correctly extract the 4 attributes above. Interestingly, ROADRUNNER [5] also reports failing on this collection. Assumption A2 also commonly failed when a set is plainly encoded using very simple html listing constructs like $\langle \cdots \{\langle \texttt{<li>} * \texttt{<li>}\rangle\} \cdots\rangle$ (only a part of the template is shown). This happens, for example, in Citeseer($M6$) where the set of citations and set of related documents are listed using html $\texttt{<li>}$. It seems possible to engineer our system to handle some of the above bad cases by incorporating more knowledge of html tags into it.

[9] We have simplified the template slightly for exposition

Source	Acc (%)	$Norm.\ Acc$ (%)
IEPAD	87.4	87.7
RISE	76.1	67.7
RoadRunner	79.8	92.5
Misc.	76.1	71.0
Total	80.0	82.7

Table 3: Accuracy grouped by source

There were also cases where Assumption A1 failed, and template tokens did not get completely differentiated. This usually happened for some very commonly occurring html tags, but tended to occur only when Assumption A2 failed to hold as well. Finally, Assumption A4 rarely failed except for semantically closely related attributes like date and month (*e.g.*, `Nov. 8 2002`) which we chose to ignore as mentioned earlier.

Sometimes, failure of assumptions A2 and A1 makes the problem *fundamentally* hard, *i.e.*, no automatic extraction algorithm can correctly extract data. For example, each page of Collection $R5$ contains data on cuisine-type and set of addresses for restaurants. This data is encoded using template $\langle \cdots \texttt{<br>} * \texttt{<br>} \{\texttt{<br>} *\} < \texttt{br} > \cdots \rangle$. This template is clearly indistinguishable from $\langle \cdots \{\texttt{<br>} *\} \texttt{<br>} \cdots \rangle$.[10]

The above example suggests that, although there is scope for improvement of our system, for example, by incorporating more html knowledge, automatic extraction can never be completely accurate. Therefore, human effort is necessary to identify attributes that have not been extracted correctly. Such attributes should either be extracted manually or by using other techniques. Note that even techniques based on training example do not currently achieve 100% accuracy.

We briefly discuss how errors could be handled when using our algorithm. As we pointed earlier, the extraction of partially correct attributes need not be on the entire page, but within a (usually) much smaller region of the page identified by our system. We are exploring alternatives whereby user can provide feedback after examining the extracted data and template, to help our system re-extract data correctly. For example, the user might identify tokens of template that are incorrectly extracted as data. We are currently building a GUI in order to aid post processing of extracted data. We expect that human input for our system during post processing step should be significantly less than human input required for training examples.

Our experimental results also illustrate another desirable property of ExAlg — the impact of the failed assumptions is localized. Even for the input collections that had some partially correct attributes, indicating that either Assumption A1 or A2 were violated, there were many other attributes that were extracted correctly. If assumptions A1 or A2 are violated for a tuple constructor τ_j, then only the attributes occurring within τ_j and the "surrounding" attributes are not correctly extracted. In the full paper, we provide more detailed description why the impact of failed assumptions is localized.

6.4 Input Size

ExAlg makes very few assumptions about the size of the input collection. As long as each type constructor occurs more than SupThres times in the input collection, it stands a chance of being discovered, subject to the other assumptions holding for the collection. The experiments indicate that our system works well for collections of very small size ($\leq$ 10 pages). A large number of pages in the collections should improve the quality of the extracted data since it is more likely that type constructors occur greater than SupThres times.

However, extraction time could become important for large collections. The running time of ExAlg is linear, in practice, since the number of iterations within ECGM is almost always less than 5, and each iteration is itself linear. For our experimental set of collections, on a 1000 MHz. CPU, the C++ implementation of our system was able to process input pages at an average rate of 110 KB/sec. Further, it is not necessary to run ExAlg on the entire collection. ExAlg can be modified in a straightforward manner to generate a template using a small sub-set of pages and use to generated template to extract data from a larger set, like most previous information extraction techniques do.

6.5 Parameters

Recall that ExAlg uses two parameters — SizeThres and SupThres. In our experiments we set SizeThres and SupThres to an extremely small value: 3. Decreasing these two parameters increases the likelihood of Assumption A2 being satisfied while decreasing the likelihood of that of Assumption A3. Increasing the value of these two parameters has the opposite effect. As our results indicate the value of 3 works well in practice. In addition, we experimentally verified that increasing the value of these two parameters gradually reduces the correctness of extraction.

7. RELATED WORK

There has been a lot of recent work related to Information Extraction. These can be classified along different dimensions: sources of information targeted (human vs. machine generated), degree of automation, complexity of data extracted (flat vs. nested). Section 1 briefly mentioned some of the closely related work. We refer the reader to a recent survey [16] and tutorial [24] for more related work. Here we focus on highlighting the differences between our work and, RoadRunner and IEPAD.

IEPAD uses repeating patterns of closely occurring HTML tags to identify and extract data. The above technique is applicable to extracting data of a limited type: set of flat tuples, from each page. For example, this technique is suitable to extract data from search results pages, as evidenced by their experimental collections (Table 2). Further, since not all repeating patterns contain useful data, IEPAD uses various heuristic techniques to characterize those that do.

Our work is most closely related to the RoadRunner [10, 5]. RoadRunner uses a model of page creation using a template that is very similar to ours. RoadRunner starts off with the entire first input page as its initial template. Then, for each subsequent page it checks if the page can be generated by the current template. If it cannot be, it modifies its current template so that the modified template can generate all the pages seen so far. There are several limitations to the RoadRunner approach:

1. RoadRunner assumes that every HTML tag in the input pages is generated by the template. This assumption is crucial in RoadRunner to check if an input page can be generated by the current template. This assumption is clearly invalid for pages in many web-sites since HTML tags can also occur within data values. For example, a book review in Amazon [2] could contain tags — the review could be in several paragraphs, in which case it contains $\langle \texttt{p} \rangle$ tags, or some words in the review could be highlighted using $\langle \texttt{i} \rangle$ tags. When the input pages contain such data values RoadRunner will either fail to discover any template, or produce a wrong template.
2. RoadRunner assumes that the "grammar" of the template used to generate the pages is union-free. This is equivalent

[10] Except if the extraction algorithm uses the fact that the set constructor in the latter template never has cardinality zero, but using this can "overfit" in most cases.

to the assumption that there are no disjunctions in the input schema. The authors of RoadRunner themselves have pointed in [5] that this assumption does not hold for many collections of pages. Moreover, as the experimental results in [5] suggest, RoadRunner might fail to produce any output if there are disjunctions in the input schema.

3. When RoadRunner discovers that the current template does not generate an input page, it performs a complicated heuristic search involving "backtracking" for a new template. This search is exponential in the size of the schema of the pages. It is, therefore, not clear how RoadRunner would scale to web page collections with a large and complex schema.

8. CONCLUSION

This paper presented an algorithm, ExAlg, for extracting structured data from a collection of web pages generated from a common template. ExAlg first discovers the unknown template that generated the pages and uses the discovered template to extract the data from the input pages. ExAlg uses two novel concepts, equivalence classes and differentiating roles, to discover the template. Our experiments on several collections of web pages, drawn from many well-known data rich sites, indicate that ExAlg is extremely good in extracting the data from the web pages. Another desirable feature of ExAlg is that it does not completely fail to extract any data even when some of the assumptions made by ExAlg are not met by the input collection. In other words the impact of the failed assumptions is limited to a few attributes.

There are several interesting directions for future work. The first direction is to develop techniques for crawling, indexing and providing querying support for the "structured" pages in the web. Clearly, a lot of information in these pages is lost when naive key word indexing, and searching is used. We indicate two specific problems in this direction. First, how do we automatically locate collections of pages that are structured? Second, is it feasible to generate some large "database" from these pages? Any technique for solving the latter problem has to be much less sophisticated than the one discussed here, possibly by sacrificing accuracy for efficiency. Also when we work at the scale of the entire web we might be able to leverage the redundancy of the data on the web as in Brin [3]. The second direction of work is to develop techniques for automatically annotating the extracted data, possibly using the words that appear in the template.

Acknowledgments

We thank Mayank Bawa and Chen Li for many stimulating discussions on the problem. This work was partially supported by NSF Grant EIA-0085-896.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, Reading, Massachussetts, 1995.

[2] Amazon.com. http://www.amazon.com.

[3] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB Workshop at 6th Intl. Conf. on Extending Database Technology*, 1998.

[4] C. Chang and S. Lui. IEPAD: Information extraction based on pattern discovery. In *Proc. of 2001 Intl. World Wide Web Conf.*, pages 681–688, 2001.

[5] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 109–118, 2001.

[6] Experimental results. `http://www-db.stanford.edu/~arvind/extract/`.

[7] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[8] M. Garofalokis, A. Gionis, R. Rastogi, S. Seshadr, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 165–176, 2000.

[9] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[10] S. Grumbach and G. Mecca. In search of the lost schema. In *Proc. of 1999 Intl. Conf. of Database Theory*, pages 314–331, 1999.

[11] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 276–285, 1997.

[12] J. Hammer, H. Garcia-Molina, J. Cho, A. Crespo, and R. Aranha. Extracting semi structure information from the web. In *Proceedings of the Workshop on Management of Semistructured Data*, 1997.

[13] C. N. Hsu and M. T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems Special Issue on Semistructured Data*, 23(8):521–538, 1998.

[14] IEPAD:. `http://www.csie/ncu.edu.tw/~chia`.

[15] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proc. of the 1997 Intl. Joint Conf. on Artificial Intelligence*, pages 729–737, 1997.

[16] A. Laender, B. Ribeiro-Neto, A. da Silva, and J. Teixeira. A brief survey of web data extraction tools. *Sigmod Record*, 31(2), 2002.

[17] A. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, pages 251–262, 1996.

[18] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 611–621, 2000.

[19] I. Muslea, S. Minton, and C. A. Knoblock. A hierarchical approach to wrapper induction. In *Proc. of Third Intl. Conf. on Autonomous Agents*, pages 190–197, 1999.

[20] L. Pitt. Inductive inference, DFAs, and computational complexity. *Analogical and Inductive Inference*, pages 18–44, 1989.

[21] RISE:. `http://www.isi.edu/~muslea/RISE/`.

[22] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[23] RoadRunner:. `http://www.dia.uniroma3.it/db/roadRunner/index.html`.

[24] S. Sarawagi. Automation in InformationExtraction and Data Integration (tutorial). VLDB, 2002.

[25] J. D. Ullman. Information integration using logical views. In *Proc. of 1997 Intl. Conf. on Database Theory*, pages 19–40, 1997.