Extracting Structured Data from Web Pages

Arvind Arasu Hector Garcia-Molina

Stanford University

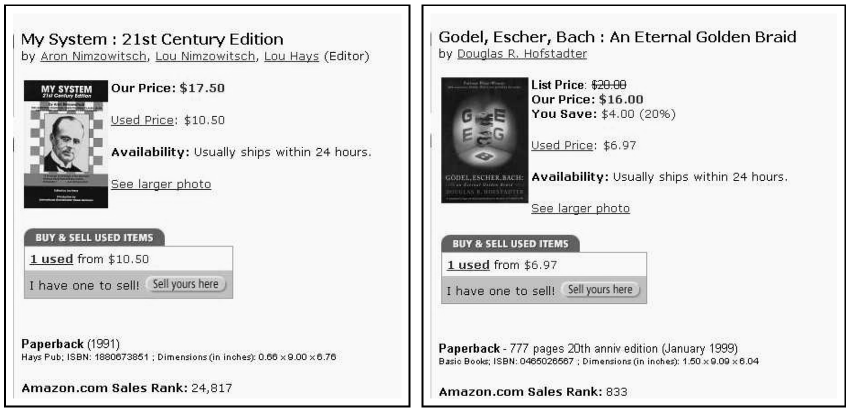
{arvinda,hector}@cs.stanford.edu

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 the web pages without any learning examples or other similar human input. We formally define the notion of a template, and propose a model that describes how values are encoded into pages using a template. We present an extraction algorithm that uses sets of words that have similar occurrence pattern in the input pages, to construct the template. The constructed template is then used to extract values from the pages. We show experimentally that the extracted values make semantic sense in most cases.

1 Introduction

The World Wide Web is a vast and rapidly growing source of information. Most of this information is in unstructured HTML pages that are targeted at a human audience. The unstructured nature of these pages makes it hard to do sophisticated querying over the information present in them. There are, however, many web sites that contain a large collection of pages that have more “structure.” These web pages encode data from an underlying structured source, like a relational database, and are typically generated dynamically. An example of such a collection is the set of book pages in Amazon [1]. Figure 1(a) shows two example book pages from Amazon. There are two important characteristics of such a collection of pages. First, all the pages in the collection encode the same kind of data. For instance, each page in Figure 1(a) contains the title, authors, and the price of a book. In other words, the “schema” of the data in each page is the same. Second, the data in each page is encoded in a similar fashion. In both pages of Figure 1(a), the title of the book appears in the beginning, followed by the word “`by`”, followed by the author(s). In other words, these pages can be generated from a common *template* by “plugging-in” values for the title, the list of authors and so on.


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

(a) Two book pages from Amazon [1]

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

(b) Extracted Data

This paper studies the problem of *automatically* extracting structured data from a collection of pages described above, without any human input like manually generated rules or training sets. By structured data, we mean "relations" — sets of tuples of the same kind that can be stored and processed in a database. For instance, from a collection of pages like those shown in Figure 1(a) 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 1(b)). Note that, as Figure 1(b) indicates, it is not our goal to find semantically meaningful attribute names for the extracted data.

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 is also useful in information integration systems [9, 17, 15, 11], which integrate the data present in different web-sites.

It is not surprising, therefore, that extracting structured data from web-pages is a well-studied problem in the database and AI communities. However, most of the existing work on this problem [14, 13, 16, 6, 7, 12] assumes significant human input, for example, in form of training examples of the data to be extracted. This is time consuming, especially if the template used to generate the pages changes often. In contrast, we aim for complete automation in the extraction process. To the best of our knowledge, ROADRUNNER project [8] is the only work that tries to solve the same problem as we do. However, ROADRUNNER makes several simplifying assumptions that limits its applicability. We defer a more technical comparison between our work and theirs, until Section 7.

The basic idea of our approach is as follows. First, we deduce the template of the input set of pages. As we mentioned earlier, template is the text of the pages that is "independent" of the actual data encoded in the pages, and is more or less "common" to all the input pages. For example, in Figure 1(a), the text `by`, `OurPrice`:, `ListPrice`:, `Availability` are all part of the template. Once the template is deduced, the remaining text in each page that is not part of the template is extracted as the data.

Although our basic approach looks intuitively simple there are several challenges that have to be addressed in order to make it feasible.

1. **Complex Schema:** The "schema" of the information encoded in the web pages could be very complex with arbitrary levels nesting. For instance, each book page can contain a set of authors, with each author having a set of addresses and so on. Even the notion of a template is not very obvious in the presence of such complex schema.

2. **Template vs Data:** Syntactically, there is nothing that distinguishes the text that is part of the template and the text that is part of the data. This makes the task of identifying the template challenging.

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 EXTRACT problem, that we are trying to solve in this paper. Section 3 provides a brief overview of our algorithm, EXALG, for solving the EXTRACT problem. EXALG described in greater detail in sections 4 and 5. Section 6 describes our experiments. Section 7 describes related work, and Section 8 our concluding remarks.

2 Model and 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 EXTRACT 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 [5]:

1. The *Basic Type*, denoted by $\mathcal{B}$, represents a string of *tokens*. A token is some basic unit of text. We define a token to be a word or a HTML tag. However a token could have been defined as a bit or a character as well.

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 a set of elements $\{e_1, e_2, \ldots, e_m\}$, such that $e_i (1 \leq i \leq m)$ is an instance of type T.

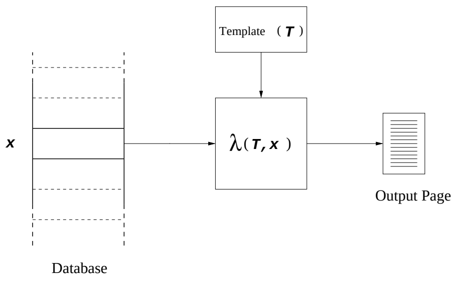
We also use term *value* to denote an instance. Also, *string* denotes a string of tokens. Sometimes type constructor symbols, $\{\}$ and $\langle\rangle$, are subscripted to help us refer to the corresponding type constructors.

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. $\square$

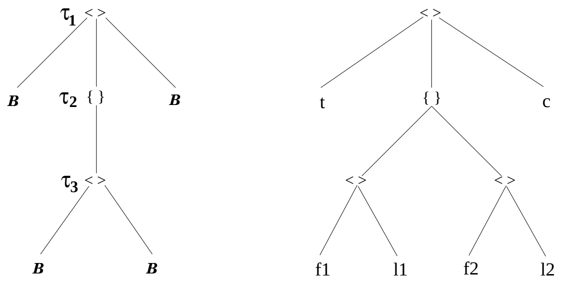
Schemas and values can be equivalently viewed as trees. Figure 1(d) shows the tree representation of schema S_1 and value x_1. A sub-tree of a schema tree is also a schema, and is called a *sub-schema* of the original schema. A sub-value of a value is similarly defined.

2.2 Model of Page Creation

We now describe a model for page creation. According to our model (Figure 1(c)), 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 $\lambda(T, x)$.



(c) Model for Page Creation



(d) Example Schema and Instance

Figure 1:

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). $\square$

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 $2(a)$ (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 $2(b)$. $\square$

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

(a)

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

(b)

Figure 2: 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 sub-values 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, the 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, $p_i = \lambda(T, x_i)$ $(1 \le i \le n)$, 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 $\langle a_1Bb_1, c_1 \rangle$. $\square$

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 3 as a running example. Each page in $\mathcal{P}_e$ contains the title and the set of reviews of a

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

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 4. 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 $\{x_{e1}, x_{e2}, x_{e3}, x_{e4}\}$. □

```
⟨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 3: Input pages of EXTRACT problem

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

```
⟨
⟨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 4: The correct solution to the EXTRACT problem

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 4 is subscripted to help us refer to it subsequently. The page-tokens of $\mathcal{P}_e$ in Figure 3 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 4.

3 Overview of our Approach

In this paper, we present an algorithm, EXALG to solve the EXTRACT problem. Figure 5 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 EXALG for our running example.

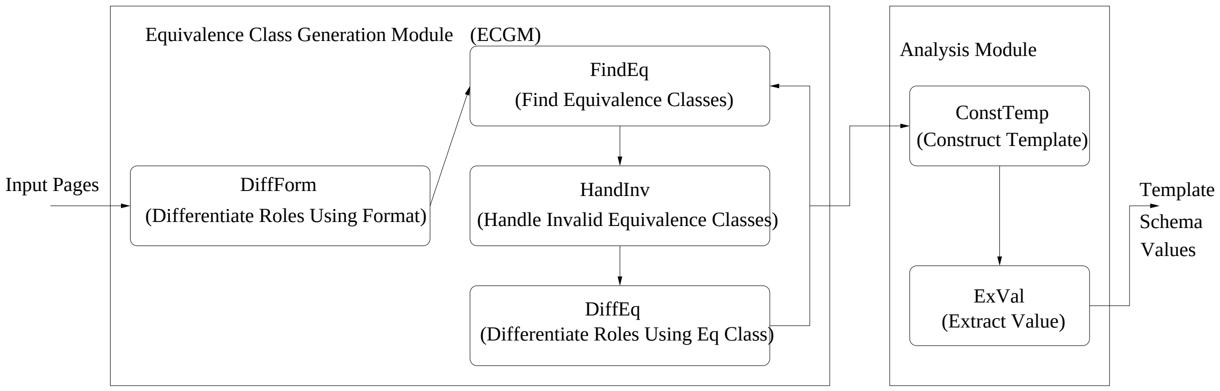

Equivalence Class Generation Module (ECGM)
Input Pages
DiffForm
(Differentiate Roles Using Format)
FindEq
(Find Equivalence Classes)
HandInv
(Handle Invalid Equivalence Classes)
DiffEq
(Differentiate Roles Using Eq Class)
Analysis Module
ConstTemp
(Construct Template)
ExVal
(Extract Value)
Template
Schema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

Figure 5: Modules of EXALG

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 $\{\langle\texttt{html}\rangle, \langle\texttt{body}\rangle, \texttt{Book}, \ldots, \langle/\texttt{html}\rangle\}$, 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 tokens: $\{\langle\texttt{li}\rangle, \texttt{Reviewer}, \texttt{Rating}, \texttt{Text}, \langle/\texttt{li}\rangle\}$. 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 6) 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

[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 5.

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

Figure 6: $\mathcal{E}_{e1}$ at end of ECGM module

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 mapping 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 6). 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.

Also, we have not described Sub-module HandInv in this section. HandInv detects and removes

[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

"invalid" LFEQs formed in FINDEQ. It does not play any role for our simple running example since there were no invalid LFEQs formed.

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 $\mathcal{P} = \{p_1, \dots, p_n\}$, where $p_i = \lambda(T^{\mathcal{S}}, x_i)(1 \le i \le n)$. Schema $\mathcal{S}$ consists of type constructors $\{\tau_1, \dots, \tau_k\}$. The pages $\{p_1, \dots, p_n\}$ form the input to EXALG. Note, however, that EXALG does not have knowledge of T, $\mathcal{S}$ and $\{x_1, \dots, x_n\}$.

Definition 4.1 (**Occurrence Vector**) The *occurrence-vector* of a token t, is defined as the vector $\langle f_1, \dots, f_n \rangle$, 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 $\langle$**html**$\rangle$ are generated by by template-token $\langle$**html**$\rangle_1$. One the other hand, a token like **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, **Name**$_5$ and **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. Note that the tokens in this equivalence class occur very infrequently — in just a single page. This observation is valid in general for real web pages. 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.*, `Databases` in our running example) usually occur infrequently and therefore do not occur in an LFEQ.

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 $(1 \leq j < k \leq m)$,

[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).

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

Figure 7: Occurrence and span of an occurrence of equivalence class

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, $\text{Pos}(1), \ldots, \text{Pos}(m-1)$. $\text{Pos}(k)$ $(1 \leq k < m)$ 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 7 illustrates the span and and $\text{Pos}(i)$ $(1 \leq i \leq 2)$ 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 $\text{Pos}(p)$ 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 $\text{Pos}(5)$ 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 in Appendix A.

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 $\langle/\texttt{b}\rangle$ 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 follow-

[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.

ing 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, 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 5). 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 $\mathcal{B}$.

$\text{Pos}(p)$ 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: $\text{Pos}(6)$ and $\text{Pos}(10)$ of $\mathcal{E}_{e1}^+$ are non-empty; $\text{Pos}(5)$, $\text{Pos}(8)$ and $\text{Pos}(11)$ of $\mathcal{E}_{e3}^+$ are non-empty.

For an occurrence of equivalence class $\mathcal{E}$ and a non-empty $\text{Pos}(p)$ of $\mathcal{E}$, $\text{PosString}(\mathcal{E}, p)$ is the string formed by concatenating tokens and equivalence classes[7] that occur in $\text{Pos}(p)$ 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 $\text{Pos}(p)$ of the above occurrence of $\mathcal{E}$. As an example, $\text{PosString}(\mathcal{E}_{e1}^+, 10)$ of the only occurrence of $\mathcal{E}_{e1}^+$ in $\mathcal{P}_2$ is the string

[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.

"$\mathcal{E}^+_{e3}\mathcal{E}^+_{e3}$". 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 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 $t_i (1 \le i \le l)$ be the token corresponding to dtoken d_i. Let $s_i (1 \le i \le q)$ 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: $C_{i_1} = t_1 \ldots t_{s_1}$, $C_{i_j} = t_{s_{(j-1)}+1} \cdots t_{s_j}$ $(1 < j \le q)$, and $C_{q+1} = t_{s_q} \ldots t_l$. The strings $C_{i_j} (1 \le j \le q+1)$ 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: $T_{\mathcal{E}_i} = \langle T_{\mathcal{E}_{i,s_1}}, \ldots, T_{\mathcal{E}_{i,s_q}} \rangle_{\langle C_1, C_2, \ldots, C_{q+1} \rangle}$.

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

	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 equivalence classes	$T_{\mathcal{B}}$
6	Unknown	$T_{\mathcal{B}}$

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

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.

[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

6 Experiments

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

A1: LFEQs are always formed due to "deterministic" causes. By this we mean that LFEQs are either valid, or if they are not, they are formed due to one of the causes anticipated by HANDINV, so that the output LFEQs of HANDINV are valid.

A2: There is a sufficient number of dtokens with unique roles after differentiation using Observation 4.4 to bootstrap the process of forming LFEQs and differentiating using the LFEQs to discover additional dtokens. Also, as a result of the iterative differentiation, eventually, all dtokens generated by template-tokens have unique roles and become part of the valid equivalence classes.

A3: For each tuple constructor τ_j in $\mathcal{S}$, a large number of template-tokens is associated with τ_j in T, and τ_j is instantiated a non-zero number of times in a large number of pages. Therefore, if Assumption A2 holds, a valid LFEQ derived from τ_j is formed.

A4: Strings associated with tuple constructors are non-empty.

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 above are not satisfied. For the purpose of experimentation we have built a data extraction system based on EXALG.

We describe the results of our experiments for 9 representative collections of input pages. Of these, 6 collections ($1-6$ of Table 2) were obtained from the ROADRUNNER site [8]. The remaining 3 collections were crawled from well-known data rich sites like E-bay [2] and Netflix [4]. The crawled web-pages were usually the first few search results for some search query. The schema of these collections is more complex (larger number of type constructors), and "less-structured", *i.e.*, has a large number of disjunctions and optionals. In Section 7 we argue that the techniques used in [8] do not work well for collections with such complex schema.

Recall that EXALG uses two parameters — SIZETHRES and SUPTHRES. In our experiments we set SIZETHRES to 3. We wanted the SIZETHRES to be as small as possible, since any type constructor with less than SIZETHRES template-tokens associated with it fails to be discovered by EXALG(Assumption A3). We did not use the value 2 since this leads to the formation of a lot of invalid equivalence classes involving start-tags and their matching end-tags. For SUPTHRES we used a value 0.25 times the number of input pages. This value was empirically determined to be good.

For each collection C above, we *manually* generated the schema S_m of the values encoded in each page of the collection, using the semantics of the application. We ignored non-text values like images and and values occurring within tag attributes (*e.g.*, urls) when generating S_m. Let S_e denote the output schema of our automatic system. 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 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 the extracted schema such that for each page in C, each value of A_m occurred as part of a value of A_e in that page.
- *Incorrect:* A_m was classified as incorrect if it was neither correct nor partially correct.

As an illustration of how an attribute in S_m would be classified as incorrect, consider a hypothetical collection of book pages, containing an attribute, book-title. Also assume that there is a token (word) w that occurs exactly once in the title of every page in the collection. In this case, EXALG will push w to the template, and will extract two attributes corresponding to the book title, (corresponding to the text of the book title before and after the word w), making the attribute book-title in our manual schema incorrect.

We use an instance from our experiments to illustrate partially correct attributes. Each movie page in our Netflix collection had an attribute, movie-title. There was also an optional attribute, local-name, for some foreign language movies. However, since the local-name appeared in a very small number of input pages (Assumption A3 fails), EXALG did not recognize the optional attribute. Instead, it combined the optional attribute with the local attribute whenever the former occurred in a page. For this case, we classified both the attributes, movie-title and local-name, as partially correct.

The above classification was done with a view on assumptions A1-A4. It can be shown that if Assumption A1 is satisfied then none of the attributes would be incorrect, irrespective of whether the other assumptions are satisfied or not. Partially correct attributes result if assumptions A2-A4 are not satisfied.

We have placed the detailed results of our experiments at the URL [3]. 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.

Index	Collection	No. pages	No. of leaf attributes	Correct	Partially Correct	Incorrect
1	Amazon Cars	21	13	13	0	0
2	Amazon Pop Artist Lists	19	5	5	0	0
3	Baseball players	10	7	7	0	0
4	rpmpackages	20	6	6	0	0
5	UEFA national teams info	20	9	9	0	0
6	UEFA team players	20	2	2	0	0
7	E-bay	50	22	18	4	0
8	Netflix	50	29	23	6	0
9	ATP Tennis Player Profiles	32	35	33	2	0

Table 2: Experimental Results

Table 2 summarizes the experimental results. Specifically, it shows for each collection C, the size of the collection, the total number of leaf attributes in S_m, and the distribution of these attributes into one if the 3 categories described earlier.

The results in Table 2 clearly demonstrate that ExAlg is very effective in correctly extracting data from web page collections. ExAlg correctly extracted the values of most of the attributes encoded in the input set of pages. For the more complex collections there were a few partially correct attributes. As we mentioned earlier, partially correct attributes result when ExAlg extracts data at a "granularity" less than the best possible. This happens, for example, when ExAlg combines together adjacent attributes, or includes some words that are part of the template within the extracted data. Although partially correct attributes are less desirable than correct attributes, they are better than incorrect attributes. This is because one can potentially convert partially correct attributes to correct attributes by developing more sophisticated techniques for (post)processing each leaf attribute of the schema extracted by ExAlg. We plan to develop such techniques as part of our future work. Finally, the absence of incorrect attributes indicates that all the input collections satisfied Assumption A1.

Our experimental results also illustrate another desirable property of ExAlg — the impact of the failed assumptions is localized. For example, if Assumption A3 is not valid for some type constructor τ_j, then ExAlg fails to extract the attributes of τ_j, and sometimes attributes of "surrounding" type constructors (see our example of partially correct attributes involving Netflix movie-title, local-title above). In the full paper, we provide more detailed description why the impact of failed assumptions is localized.

The running time of ExAlg depends on the number of iterations of the loop involving the sub-modules FindEq, HandInv and DiffEq. The running time of each sub-module is linear in the input size. For all the input collections ExAlg the number of iterations was less than ten. Hence, for all practical purposes, ExAlg is linear in the input size. So far, ExAlg that we have described, works on the entire input collection of pages. When the input collection is large, we can modify ExAlg to work in a "wrapper-mode." In this mode, instead of using the entire collection to generate the template, ExAlg uses a small sample of pages to generate the template, and then uses the generated template to extract the data from the full collection.

7 Related Work

Most of the related work use a "wrapper-based" system for extracting data. In a wrapper based system, extraction is a two step process. In the first step, a *wrapper* for the given set of pages is generated. In the second step, the wrapper is used to extract the data from the web pages. A wrapper is just a program that extracts the data from the set of pages. Note that a wrapper is specific to the set of pages — the wrapper for the pages of one web site will be different from the wrapper for the pages of a second web site. In Hammer et al. [12] a human expresses the location of the data to be extracted as declarative rules. A program converts these rules into a wrapper. In [14, 13, 16, 6] training examples consisting of the pages and the data that occur in those pages are used to "learn" the wrapper using various machine learning techniques. All the above wrapper-based system clearly require more human input either in the form of rules or learning examples. If the template changes, as happens frequently in practice, new rules of examples may be needed.

DIPRE [7] is an instance of a non wrapper-based system that uses learning examples. But the interesting aspect of DIPRE is that it tries to extract relations from the entire web and not just a particular web site as in our case. But their techniques apply only to simple "relations" while we aim to extract data with more complex schema.

Our work is most closely related to the RoadRunner project [10, 8]. 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 [1] 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 to the assumption that there are no disjunctions in the input schema. The authors of RoadRunner themselves have pointed in [8] that this assumption does not hold for many collections of pages. Moreover, as the experimental results in [8] 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 [7]. The second direction of work is to develop techniques for automatically annotating the extracted data, possibly using the words that appear in the template.

References

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

[2] ebay.com. http://www.ebay.com.

[3] Experimental results. `http://www-db.stanford.edu/˜arvind/extract/index.html`.

[4] Netflix. http://www.netflix.com.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, Reading, Massachussetts, 1995.

[6] G. Barish, Y. S. Chen, D. DiPasquo, and C. A. Knoblock. Theaterloc: Using information integration technology to rapidly build virtual applications. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 681–682, 2000.

[7] Sergey Brin. Extracting patterns and relations from the world wide web. In *WebDB Workshop at 6th International Conference on Extending Database Technology, EDBT'98*, 1998.

[8] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, 2001.

[9] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeff Ullman, and Jennifer Widom. The tsimmis project: Integration of heterogenous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[10] St'ephane Grumbach and Giansalvatore Mecca. In search of the lost schema. In *Proceedings of the Intl. Conference of Database Theory (ICDT)*, 1999.

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

[12] Joachim Hammer, Hector Garcia-Molina, Junghoo Cho, Arturo Crespo, and Rohan 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), 1998.

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

[15] Alon Levy, Anand Rajaraman, and Joann 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.

[16] I. Muslea, S. Minton, and C. A. Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of Third International Conference on Autonomous Agents*, Seattle, WA, 1999.

[17] Jeffrey D. Ullman. Information integration using logical views. In *Proc. of the Internation Conference on Database Theory (ICDT)*, pages 19–40, 1997.

A Handling Invalid Equivalence classes

We now present a procedure HANDINV, which is a sub-module of EXALG, whose goal is to eliminate invalid equivalence classes. HANDINV uses the ordered and nesting properties of equivalence classes to detect the existence of invalid equivalence classes. It subsequently "processes" the invalid LFEQs found by sometimes discarding them, and sometimes breaking them into smaller LFEQs.

It follows from Observation 4.3 that any equivalence class that is not ordered is not valid, and for any pair of equivalence classes that are not nested, at least one of them is not valid. HANDINV works under the hypothesis that invalid equivalence classes expose their presence by not satisfying the orderedness property or the nesting property or both (which is the converse of Observation 4.3).

The input to HANDINV is a set of equivalence classes, possibly containing equivalence classes that are not ordered, or pairs of equivalence classes that are not nested. The output is an nested set of ordered equivalence classes. Before presenting HANDINV we examine some common causes for formation of invalid equivalence classes. These provide the insight for the procedure that follows.

A.0.1 Causes for formation of invalid equivalence classes

In practice there are three primary causes for the occurrence of invalid equivalence classes.

1 *Correlation of instantation of two or more type constructors.* The tokens having unique roles and associated with type constructors (two or more) having the same frequency of instantiation in every input page form an invalid equivalence classes. In practice, this happens because related information is sometimes physically distributed in non-contiguous parts of an output page. For example, an Amazon book page has an optional attribute for the rating of the book, and an optional set of customer reviews of the book. Any book that has one or more customer reviews has a rating, and the rating attribute is absent if the set of customer reviews does not exist.

2 *Frequently Occuring tuples.* In some cases, the same (sub-value) tuple repeatedly occurs in many different pages. The tokens occuring in the tuple tend to form an equivalence class since all the tokens occur whenever the tuple occurs and none otherwise. For example, a Netflix movie page has information about related movies. Each related movie is a tuple consisting of the movie name, rating, and other similar information. A movie could be related to several other movies, and therefore the tuple corresponding to the movie appears in the pages corresponding to all the movies that it is related to.

3 *Overlap in the tokens associated with different type constructors.* Consider two type constructors τ_1 and τ_2 of S. Let $t_1, t_2, \ldots$ be the set of tokens that are associated with both the type constructors. The set of tokens $t_1, t_2, \ldots$ form an equivalence class in a set of pages, if they are not associated with any other type constructor in the schema, or generated by any value-token. Consider our running example, assume that we use a size threshold of 2 to determine if an equivalence class is an LFEQ. In this case, in addition to the equivalence classes $\mathcal{E}_{c1}$ and $\mathcal{E}_{c3}$, there is an additional equivalence class $\mathcal{E}'$: $\{\langle \texttt{b} \rangle, \langle /\texttt{b} \rangle\}$ which has an occurrence vector $\langle 5, 8, 5, 2 \rangle$. $\mathcal{E}'$ is an example of an equivalence class formed due to the overlap of the tokens, $\langle \texttt{b} \rangle$ and $\langle /\texttt{b} \rangle$, that are associated with both τ_{c1} and τ_{c3} of S_c.

We call invalid equivalence classes formed by the first two causes invalid equivalence classes of type A, and invalid equivalence classes formed by causes 3 and others invalid equivalence classes of type B.

The invalid equivalence classes of type A are just collections of valid equivalence classes[9]. They have easily verifiable characteristic occurrence properties — they are “almost” ordered and “almost” nested. Procedure HANDINV tries to partition these kind of invalid equivalence classes into their constituent valid equivalence classes. The Type B invalid equivalence classes cannot be partitioned to valid equivalence classes. Procedure HANDINV discards Type B equivalence classes whenever it detects one.

A.0.2 HANDINV

The classification of invalid equivalence classes into Type A and Type B equivalence classes requires a knowledge of the schema S and template T, which is not available to HANDINV. Procedure HANDINV just uses the ordered and nesting properties of the equivalence classes to predict if an invalid equivalence class is of Type A or Type B.

Definition A.1 (**Almost-Ordered**) An equivalence class $\mathcal{E}$ is *almost-ordered* if the tokens in the equivalence class can be partitioned into ordered equivalence classes that do not overlap. □

Example A.1 Consider an equivalence class $\mathcal{E} = \{A, B, C, D\}$ defined for an input of three pages. If the relative order of occurrence of the tokens in the three pages is $ABABCDCD$, $ABCD$, $ABABABCDCDCD$, respectively, then $\mathcal{E}$ is almost-ordered. $\mathcal{E}$ can be split into equivalence classes $\{A, B\}$ and $\{C, D\}$ which are ordered and do not overlap with each other. □

[9]This is not entirely true for equivalence classes formed by cause 2. However, it should become clear to the reader after reading Section 5 that no harm is done by ignoring this fact

Definition A.2 (**Nearly-Nested**) Two equivalence classes, $\mathcal{E}_i$ and $\mathcal{E}_j$ are said to be *nearly-nested* if for every token $t \in \mathcal{E}_i$ (and vice versa) there exists a position p, such that each occurrence of t is either outside the span of any occurrence of $\mathcal{E}_j$, or within $\mathrm{Pos}(p)$ of some occurrence of $\mathcal{E}_j$. $\square$

Example A.2 Consider two equivalence classes $\mathcal{E}_1 = \{A, B, C, D\}$ and $\mathcal{E}_2 = \{E, F, G, H\}$ for an input of three pages p_1, p_2, p_3. If the relative order of occurrences of the tokens $A - H$ in p_1, p_2, p_3 is $ABCDA(EF)BC(GH)D$, $A(EF)BC(GH)D$ and $EFGH$, respectively, the classes $\mathcal{E}_1$ and $\mathcal{E}_2$ are almost-nested with respect to each other. As an example, token E of $\mathcal{E}_2$ always either occurs in $\mathrm{Pos}(1)$ of an occurrence of $\mathcal{E}_1$ or does not occur within span of any occurrence of $\mathcal{E}_1$. $\square$

The following lemma is anologue of lemma 4.3 for Type A equivalence classes.

Lemma A.1 An unordered Type A equivalence class is almost-ordered. Two ordered equivalence classes, each of which is either a valid equivalence class or an invalid Type A equivalence class are almost-nested with respect to each other. $\square$

We are now ready to describe HANDINV.

First, HANDINV checks if each equivalence class is ordered, or almost-ordered. Any equivalence class that is not ordered or almost-ordered is not a valid equivalence class or a Type A equivalence class. Such equivalence classes are removed from consideration. Any almost-ordered equivalence class is partitioned into ordered non-overlapping equivalence classes (Definition A.1). This results in a set of ordered equivalence classes.

Next, HANDINV checks the nesting property of every pair of remaining equivalence classes. If two equivalence classes $\mathcal{E}_i$ and $\mathcal{E}_j$ are neither nested nor almost-nested with respect to each other at least one of them is an Type B invalid equivalence class. HANDINV greedily picks the one that is badly nested with a lot of other equivalence classes and deletes it from the set of equivalence classes. The intuition is that the non-Type A equivalence class in the pair is likely to be not nested properly with a lot of other equivalence classes, while the valid or Type A equivalence class in the pair is not likely to be. In other words the valid and Type A equivalence classes mutually "vote" each other as "good", and thus help in locating the "bad" equivalence classes. The following example illustrates this idea.

Example A.3 Consider our running example. Let the size threshold for determining whether an equivalence class is an LFEQ or not be 2. For this support threshold, there are three equivalence classes $\mathcal{E}_{c1}$, $\mathcal{E}_{c3}$ and

$\mathcal{E}' = \{\langle\texttt{b}\rangle, \langle/\texttt{b}\rangle\}$. With the knowledge of the schema S_c and template T_c we know that $\mathcal{E}'$ is an invalid Type B equivalence class.

Considering the nesting properties of the three pairs formed from the three equivalence classes. The equivalence classes $\mathcal{E}_{c1}$ and $\mathcal{E}_{c3}$ are nested with respect to each other. The other pairs $\mathcal{E}_{c1}, E'$ and $\mathcal{E}_{c3}, E'$ are not nested. Since E' participates in a larger number of non-nesting relationships E', HANDINV predicts that E' is a Type B equivalence class and is discarded. $\square$

After the previous step, equivalence classes of every pair of remaining equivalence classes are either nested or almost-nested with respect to each other. In the final step, HANDINV considers each pair of equivalence classes that is almost-nested, but not nested. Let $\mathcal{E}_i$ and $\mathcal{E}_j$ be a pair of almost-nested equivalence classes. At least one of $\mathcal{E}_i$ and $\mathcal{E}_j$ is an invalid Type A equivalence class. It is also possible that both are invalid Type A equivalence classes. HANDINV tries to predict the invalid equivalence classes and partition them into smaller equivalence classes, such that the resulting set of equivalence classes are nested.

It can be easily verified that if $\mathcal{E}_i$ is a valid equivalence class and $\mathcal{E}_j$ an invalid Type A equivalence class, then each occurrence of $\mathcal{E}_j$ overlaps with an occurrence of $\mathcal{E}_i$ but not vice versa (otherwise, $\mathcal{E}_i$ and $\mathcal{E}_j$ would have the same occurrence vector and would have belonged to the same equivalence class). In this case, HANDINV partitions $\mathcal{E}_j$ — each contiguous set of tokens of $\mathcal{E}_j$ that occur in the same position p of an occurrence of $\mathcal{E}_i$ belong to the same partition. Each partition of token is made into a separate equivalence class.

If there exists some occurrence of $\mathcal{E}_i$ that does not overlap with $\mathcal{E}_i$ and there exists some occurrence of $\mathcal{E}_j$ that does not overlap with $\mathcal{E}_i$, then neither $\mathcal{E}_i$ and $\mathcal{E}_j$ are valid, and both are partitioned — each contiguous set of tokens of $\mathcal{E}_j$ (resp. $\mathcal{E}_i$) that occur in the same position p of an occurrence of $\mathcal{E}_i$ (resp. $\mathcal{E}_j$) belong to the same partition.

Example A.4 Consider the two equivalence classes $\mathcal{E}_1$ and $\mathcal{E}_2$ from Example A.2. HANDINV would predict that both equivalence classes are invalid since there exist an occurrence (first occurrence in p_1) of $\mathcal{E}_1$ that does not overlap with an occurrence of $\mathcal{E}_2$, and an occurrence of $\mathcal{E}_2$ (page p_3) that does not overlap with an occurrence of $\mathcal{E}_1$. HANDINV would partition $\mathcal{E}_1$ into three equivalence classes $\{A\}, \{B, C\}, \{D\}$ and $\mathcal{E}_2$ into two equivalence classes $\{E, F\}$ and $\{G, H\}$.

On the other hand, if the relative occurrence of tokens $A - H$ in p_3 were to be $ABCD$, then HANDINV would predict that $\mathcal{E}_1$ is valid and $\mathcal{E}_2$ is invalid. In this case, it would partition $\mathcal{E}_2$ alone in to two equivalence classes $\{E, F\}$ and $\{G, H\}$. $\square$

The set of equivalence classes after the last step is the output of HANDINV. It can be verified that the output is a nested set of ordered equivalence classes.