Computer Sciences Department

SLIC: On-The-Fly Extraction and Querying of Web Data

Robert McCann
Pedro DeRose
AnHai Doan
Raghu Ramakrishnan

Technical Report #1558

September 2006

UNIVERSITY OF WISCONSIN MADISON

SLIC: On-the-Fly Extraction and Querying of Web Data

Robert McCann[1], Pedro DeRose[1], AnHai Doan[1], Raghu Ramakrishnan[2]
[1] University of Illinois [2]Yahoo! Research

Abstract

Increasingly, Web data is displayed in pages generated according to a template (e.g., product listings at amazon.com, faculty directories, class schedules). This trend makes structured querying of such Web data a valuable capability for a growing number of applications, including many ad hoc, exploratory, and short-lived tasks. Unfortunately, current methods for answering such queries require writing complex Perl scripts or creating customized wrappers and storing the extracted data in a DBMS, which is often overkill for these types of on-the-fly tasks. In this paper we propose SLIC, a solution to this problem. Given a set of Web pages generated according to some templates, SLIC allows the user to quickly pose SQL queries, obtain initial results, and then iterate with the system to get increasingly better results. At each step, SLIC asks relatively simple questions to solicit minimal structural information from the user in order to extract data and refine the answers. Extensive experiments on real-world domains show that for many practical queries (1) SLIC is significantly faster than current methods, and (2) the user needs to answer only a few relatively simple questions before obtaining useful answers. SLIC thus provides a promising first step toward a principled solution for on-the-fly extraction and querying of Web data.

1. Introduction

Much of the data on the World-Wide Web is exposed in HTML pages that conform to some *structure template* [8, 20, 19, 4]. Figure 1 shows examples of pages that export data on houses and high schools. Other examples include product descriptions at *amazon.com*, publications at DBLP, and and protein data at bioinformatic databases.

To process such data, the typical solution today is to write customized programs called *wrappers*, which specify how to *extract* structured data from the HTML pages (e.g., [19, 4]). The extracted data is then load into a database (e.g., a RDBMS) or fed into additional programs (e.g., Perl scripts), for further processing and querying.

This extract-then-query approach works well for *long-running* information needs of a *large user community*, where *many queries* will be asked over an extended period of time by many people, thereby amortizing the effort invested upfront on developing the wrappers. Examples of such information needs include warehousing of Web data for mining purposes, comparison shopping systems, and integration systems over Deep Web sources that serve multiple users over long periods of time [16, 21, 6, 17].

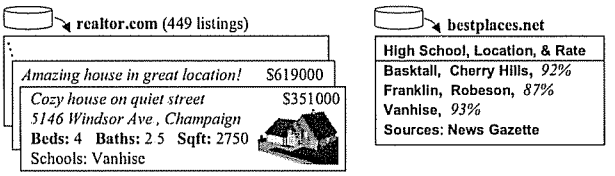


Figure 1. Examples of HTML pages that export data according to some structure template.

Increasingly, however, many information needs are *ad hoc* and *on-the-fly*, involving just a few queries with *short-lived* value, often for a single user [12]. For example, a researcher may want to obtain the list of all SIGMOD papers in the past ten years whose titles contain "XML" or "semi-structured." This list can be gleaned from the corresponding SIGMOD conference pages at DBLP, and is clearly an ad-hoc information need in that the user is unlikely to want to issue the query repeatedly, and that there is not a large community that is likely to be interested in it.

Other examples arise naturally in many settings. Domain scientists often want to execute complex queries that join data from multiple Web pages on the fly, for exploratory analysis [11]. Many Web users increasingly want lightweight tools to assist in querying Web data (e.g., to compare statistics on product rating) [14, 13]. As yet another example, corporate users at Thomson Legal Inc. (*tlrg.com*) often want to extract and query data from competing law firms' websites on the fly, to quickly compile statistics regarding these law firms (e.g., cases won, lost, and areas covered), then leverage those statistics in presentations to potential clients [2].

For such on-the-fly information needs, the extract-then-query approach is not well suited. The fundamental reason is that today it still takes too much time to write wrappers. Writing them by hand is well known to be difficult and labor intensive. Many wrapper construction solutions have been proposed (e.g., [20, 19, 4]). However, they are still difficult to set up, often taking days even for expert users (e.g., to tune various system parameters). They are also notoriously brittle, often requiring much manual intervention to achieve high accuracy.

To address the above problem, we propose SLIC (Structured Lazy Information ExtraCtion), a solution that enables fast, on-the-fly extraction and querying of template-based Web data. The key ideas underlying SLIC are as follows:

1. Extract only necessary attributes: To extract and query Web data, current solutions typically extract values for *all or almost all* attributes present in HTML pages, on the ground that there may be many queries in the future that can touch any of those attributes. In our on-the-fly settings, however, the user is likely to ask only *a few* queries. Hence, to minimize user efforts, we only extract values for the attributes that are mentioned in the query at hand. For example, given the set of HTML pages in Figure 1.a, each of which describes a house for sale, suppose the user wants to execute a query Q that lists all houses whose price exceeds $500,000. Then we only extract values for price. We therefore construct *query-centric wrappers*.

2. Ask users only relatively simple questions: Current solutions typically ask users to manually write extraction rules or to deploy a semi-automatic wrapper construction system. Either choice is laborious, as we have argued earlier. In contrast, we only ask users to answer relatively simple questions regarding the appearance of the attributes, such as "is price listed in bold font on these Web pages?" and "is price hyperlinked?". We then leverage user answers to these questions to extract attribute values.

3. Extract approximate attribute values: Current solutions extract *exact* attribute values. In contrast, we observe that *approximate* values (which often are much easier to extract) are often sufficient for answering a user query satisfactorily. To illustrate, suppose the user has answered "yes" to the question "is price listed in bold font?". Then we still do not know the exact values of price, because a bold-font number can still be, e.g., a discount rate. However, suppose that the given HTML pages contain no bold-font number that exceeds 500,000. Then we know immediately that the answer to query Q above is empty: no house for sale can have price above $500,000. As another example, suppose that only five HTML pages contain bold-font numbers above 500,000. Then we can simply return all five pages as an "approximate superset" result to Q. Since the size of the approximate superset result is small, the user can easily shift through to obtain the true result set (i.e., houses with price exceeding $500,000 in this case). Note that in both examples, there was no need to construct complex rules that extract *exact* values for price. Asking the user a simple question was sufficient to extract *approximate* price values, which in turn was sufficient to answer the user query to his or her satisfactory.

4. Interleaving extraction and execution: Finally, current solutions execute user queries *only after* the data has been extracted. We propose to interleave data extraction and query execution. For example, if we have known only that price is in bold font, we can already perform an *approximate* extraction, by extracting all bold-font numbers, knowing that prices must be among those. Then we execute query Q over the approximate extracted data, to produce an approximate result (e.g., the five HTML pages that contain bold-font numbers above 500,000, as mentioned above). If the user wants "better" results, we then can ask more questions (e.g., "is price hyperlinked?"), then repeat the extraction and execution process. This way, we can stop the moment the user is satisfied with the results, thereby minimizing user efforts, an important requirement for on-the-fly scenarios.

Building on the above ideas, SLIC operates as follows. Given a structured query Q posed by a user U over a set of HTML pages P, SLIC analyzes Q and P, and provides an initial approximate result. It then iterates with user U to provide increasingly better results. In each iteration SLIC asks U 1-2 relatively simple questions, designed to solicit additional information about the attributes mentioned in Q. The key observation is that all pages in the set P are generated from the same template, and the user's feedback about a given example page allows us to refine the extraction of information from *pages* in P. (In practice, P may include subsets of pages, with a different template for each subset, but the basic idea still applies.) SLIC then leverages the answers to re-extract the data and re-execute query Q. SLIC stops when the user is satisfied or when it determines that the query result has converged to the correct one.

SLIC thus enables fast, user-friendly extraction and querying of Web data on the fly, while minimizing user efforts. Developing SLIC, however, raises significant challenges. An important contribution of this paper is to identify the key challenges that must be resolved to develop an effective SLIC-like solution for on-the-fly extraction and querying of Web data. A second contribution is a solution to a key technical problem facing SLIC: how to efficiently execute approximate string joins over approximately extracted data? Approximate joins allow matching of syntactically different data values, such as "David Smith" matching "D. Smith" [10]. The need for such join is acute in processing Web data, where a name often appear in myriad formats (e.g., "D. Smith", "Smith, David", "Mr. Smith", etc.). Consequently, approximate join has attracted significant attention (e.g., [10, 18]). The SLIC setting however raises novel challenges. Here, each extracted data value can be a large set of string, due to the approximate nature of the extraction process. Consequently, instead of matching *string pairs* (e.g., "David Smith" vs. "D. Smith"), as is currently the case [10, 18], in our settings the approximate join must match *sets of strings*, such as {"David Smith", "Smith", "Dave S"} versus {"D. Smith", "Prof. Smithson"}. In this paper we show why current approximate join solutions (e.g., [10, 18]) do not work for this case, then develop an effective solution. Overall, we make the following contributions:

- Introduce SLIC, a novel approach to extracting and

querying template-based Web data on the fly. SLIC constructs query-directed, approximate wrappers by asking the user only relatively simple questions. It also interleaves the process of extraction and execution, to converge to a satisfactory query result quickly, with minimal user efforts.

- Identify the main challenges in developing SLIC and develop an effective solution to approximate string join over approximately extracted data, a key technical challenge in SLIC setting.
- Describe extensive experiments with a SLIC's prototype over several real-world Web data sets. The experiments show that for many practical queries SLIC is significantly faster than current methods, with minimal user efforts, thus demonstrating the promise of SLIC.

In the rest of the paper we define the problem, describe our SLIC solution, then experiments with SLIC in details.

2. Problem Definition

We now describe the on-the-fly extraction and querying problem considered in this paper.

Data: Suppose the user has collected a set of HTML pages $\mathcal{W} = \{P_1, \ldots, P_n\}$ that describe real-world entities, such as products, houses, proteins, etc. Figure 1 show examples of such pages, which describe houses and high schools, and are collected from two real-estate Web sources. We assume that the data in the pages conform to a set of structure and format templates (the same assumption made by current wrapper work [20, 19, 4]). House descriptions in Figure 1.a for example always display an optional comment, followed by price, then address, number of beds, and so on. The comment, when displayed, is always in italics, the price is always prefixed with "$", etc. We consider the common scenarios where the description of each item occupies a whole page (e.g., houses in Figure 1.a), or they are listed sequentially on a page (e.g., schools in Figure 1.b). Considering more complex scenarios (e.g., where item descriptions can be interleaved) is a subject of future research.

Users and Queries: We assume that the user wants to pose just one or a few queries over the data $\mathcal{W}$. The user starts with a query, then may make up subsequent queries on the fly, in an exploratory fashion. We consider structured queries that can be expressed in a SQL-like language. Examples of users who are capable and often want to write such queries include domain scientists, intelligence analysts, data mining experts, and skilled users on the Web and at enterprises. For lay users, it may be possible to provide a GUI that translates user queries into structured queries (akin to GUIs provided by *amazon*, *bestbuy*, etc.), but this is beyond the scope of the current paper.

Superset Semantics for Query Results: Recall from Section 1 that we provide approximate query results (in each iteration of SLIC). Since the user often wants to know *all* correct result tuples (e.g., all houses whose price exceed $500,000), currently we always return a *superset* of the true query result. The user can then further clean the result to obtain only the correct ones. Our goal is then to quickly converge to the true result set or to the smallest possible superset, so that the user can minimize the time spent post-processing the result produced by SLIC. For future work we will explore more general semantics based on combinations of precision and recall.

We can now define our problem as follows: *design SLIC such that given a set of HTML pages $\mathcal{W}$ and a query Q, described as above, interact with the user to produce the smallest possible superset of the true query result, while minimizing the amount of user interaction.*

3. The SLIC Approach

In this section we first describe a usage scenario. We then provide a high-level overview of the working of SLIC, and highlight the main technical challenges. Section 4 then develops a solution for one such challenge: effective execution of approximate string joins over extracted data.

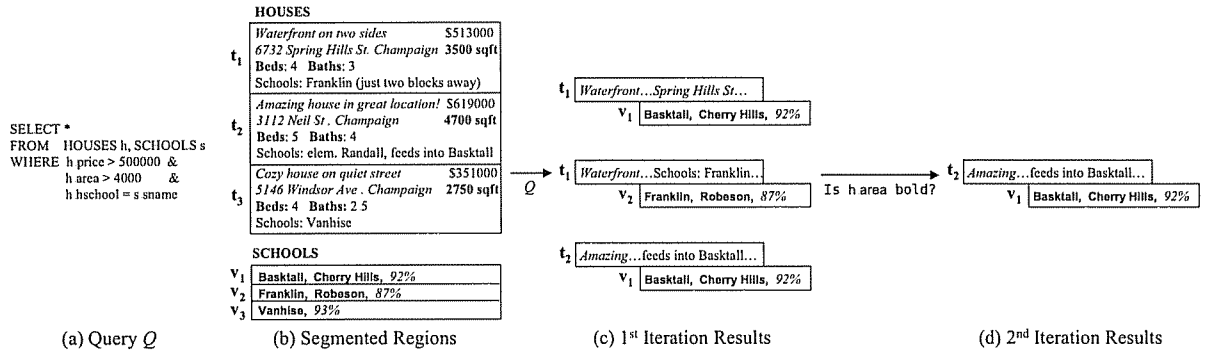
3.1 Usage Scenario

Consider a user X who wants to find all houses for sale at *realtor.com* of more than 4,000 square feet and price above $500,000. For each house, X also wants to find all information on the associated high school from *bestplaces.net*. X proceeds as follows.

1. Providing Web pages to SLIC: X first retrieves from *realtor.com* a set $\mathcal{H}$ of Web pages that describe houses and feeds them into SLIC, specifying that they conform to a structure template that X will refer to as HOUSES. Similarly, X retrieves from *bestplaces.net* a single Web page S that describes high schools, feeds it to SLIC, and names the structure template SCHOOLS.

2. Posing a Query Q to SLIC: Next, X writes a SQL query Q to express the desired information, as shown in Figure 2.a. Observe that query Q refers to the templates HOUSES and SCHOOLS, in effect behaving as if the data on houses and schools has been extracted into two relational tables with the same names. Observe also that X "makes up" attribute names (e.g., price, area) and refer to them in Q as if they have been extracted. (In reality, of course, these tables do not exist physically.)

3. Interacting with SLIC to segment Web pages: Given query Q, SLIC then starts by segmenting Web pages that belong to the templates touched by Q into individual records, each of which describes an item referred to in Q (e.g., h and s, see Figure 2.a). For template HOUSES, each individual Web page is already a record (for h), as indicated by X, so no segmentation is necessary. For template SCHOOLS, the retrieved Web page S must be segmented into individual records, each of which describes a school. To do so, SLIC interacts with X via a GUI interface



(a) Query Q (b) Segmented Regions (c) 1st Iteration Results (d) 2nd Iteration Results

Figure 2. A usage scenario of SLIC.

and asks X to highlight several regions that contain record boundaries (see Section 3.2). Within a few minutes (1-2 in our experiments) SLIC has segmented page S into individual school records. The segmented records are shown in Figure 2.b. To keep the scenario manageable, we assume that the user has retrieved only three Web pages from *realtor.com*; these pages are shown as records in "table" HOUSES in Figure 2.b. Page S has been segmented by SLIC and its three records are shown in table SCHOOLS.

4. Inspecting query results and answering questions: SLIC then iterates. In the first iteration it executes query Q and produces the result in Figure 2.c. This result shows three pairs of records, each describing a house that SLIC thinks may satisfy the user's request (i.e., price above $500,000 and area above 4000), together with the associated school information. Observe that this result is a superset of the true result (which is the record pair $t_2 - v_1$).

X quickly scans the produced superset result and, seeing that houses in t_1 and t_3 are admitted despite having insufficient square footage, tells SLIC to narrow the superset. SLIC responds with a question about the appearance of data in HOUSES: "is h.area bold?". X answers "yes". SLIC then enters the second iteration. It leverages the new information regarding h.area to re-extract the data and re-executes query Q, then shows X the new result in Figure 2.d. This time, the superset result contains only the correct pair of records. X is satisfied, and indicates to SLIC to terminate execution.

It is important to emphasize that Steps 1, 2, and 3 above are relatively simple and are shared by virtually all current solutions that extract and query Web data. After data collection in Step 1, segmenting Web pages into records (Step 3) is typically executed (e.g., by a Perl script or a GUI-based system, like what SLIC does here) before creating the wrapper, or becomes a part of the wrapper. Posing a structured query Q to the extracted data (Step 2) is carried out over an RDBMS or as a part of yet another Perl script.

The above steps take negligible time compared to Step 4. As discussed earlier, to execute this step, current solutions build full-fledged wrappers that extract *exact* data values, a time-consuming and tedious process during which the user must either write wrappers or deploying and tuning a wrapper construction system. In contrast, as seen above, SLIC "transparently" handles this entire process. It asks the user only relatively simple questions, and only as many as necessary to satisfactorily answer the query at hand. Our experiments (Section 5) show that SLIC executes this step in far less time than that incurred by current solutions.

3.2 The Working of SLIC

We now provide a high-level overview of SLIC's approach, and discuss the key challenges facing SLIC,

User queries: As a first step, we currently consider select-project-join SQL queries. Besides *theta*-join commonly found in RDBMs, we consider also approximate string joins, because such joins are crucial for processing heterogeneous Web data, as argued in the Introduction. Section 4 formally defines approximate string joins and provide effective implementations for such joins in our contexts.

Segmenting Web pages into records: Given a user query Q, SLIC starts by segmenting certain Web pages into individual records (as discussed in Section 3.1). Since the pages that we consider display the records sequentially (Section 2), SLIC needs to find only an HTML string to serve as the record separator. It finds this string via a short interactive session in which the user highlights several Web regions that contain record boundaries. SLIC then applies the inferred HTML string to segment pages. This method is simple but appears very effective (e.g., finding separators in 1-2 minutes, and separating records correctly in all of our experiments).

Features, questions, and answers: In the next step, SLIC can *optionally* ask the user a question regarding the appearance of some attribute in Q. To do so, SLIC makes use of a set of *appearance features*: numeric, string, boldfont, etc. SLIC can then ask questions of the form "what is the value for feature f of attribute a?". Possible answers include "yes" and "no". Each feature f is also associated with two procedures: Verify and Refine. The approximate wrapper employs these procedures to extract data from Web pages. Roughly speaking, Verify(R, f, v) returns

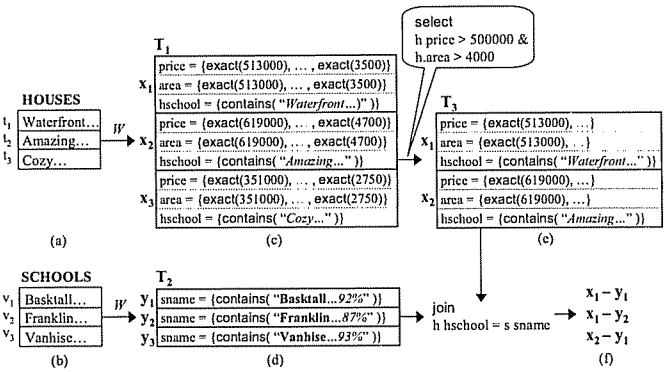


Figure 3. An example of SLIC's executing a query over Web data.

true if $f = v$ holds for text region (i.e., a contiguous span of text) R, and false otherwise, and Refine(R, f, v) returns all subregions R' of R for which Verify(R', f, v) is true. The appearance features and the associated procedures are provided *in advanced* by the SLIC designer, when SLIC is built, *not* by the user. If SLIC does ask a question, then the user answer is encoded an approximate wrapper, which we describe next.

Constructing and applying approximate wrappers: We define such a wrapper to consist of a set of *domain constraints*. Each domain constraint (a, f, v) specifies that for any text region that is a value for attribute a, its feature f takes value v. An example constraint is (price, numeric, yes). For the entire execution of a query Q, SLIC employs a single approximate wrapper W. Every time the user answers a question, a domain constraint is constructed and added to W. At the very start, when no question has been asked, W includes only domain constraints that SLIC can infer from the description of query Q. For example, from query Q in Figure 2.a, SLIC creates constraints (price, numeric, yes) and (area, numeric, yes) based on the selection condition (price *geq* 500000) & (area $\geq$ 4000). For any other attribute f in Q (e.g., hschool, sname), SLIC creates the default constraint (f, string, yes). It then initializes wrapper W with these constraints. It then applies wrapper W to extract approximate data from the Web pages. In what follows we describe how the extracted approximate data is represented in SLIC.

Representing approximate data: Consider again wrapper W mentioned above. Suppose we want to apply W to record t_1 (the first one under HOUSES, see Figure 2.b) to extract price. Then knowing only that price is numeric, we will extract the set {513000, 6732, 4, 3, 3500}. We can then assign this set as the value of price in record t_1. In general, we can model the extracted approximate data as a set of records, such that each attribute in each record are assigned *a set of values*.

The above naive model however is not scalable, because as query execution progresses, the number of values per attribute often grows very quickly, posing both storage and processing problems. Hence in SLIC we employ a more efficient though less expressive model, called *compact table*. A compact table consists of multiple records, such that each attribute A in each record is associated with a set of *assignments*: $A = \{m_1(R_1), m_2(R_2), \ldots, m_k(R_k)\}$, where each m_i is an *exact* or *contain* assignment, and each R_i is a text region (i.e., a continuous text span).

An *exact* assignment $exact(R)$ means that A's value is exactly region R (modulo an optional cast from string to numeric). For example, $exact$("92") means that A's value is 92. A *contain* assignment $contain(R)$ means that A maps exactly into a region *inside* region R. For example, assignment $contain$("Cherry Hills,92") means that A's value is a region inside "Cherry Hills,92", but we do not know exactly which region. Thus, a contain assignment conceptually specifies a potentially large set of exact assignments.

To illustrate compact tables, suppose record v_1 (under SCHOOLS in Figure 2.b) is the HTML fragment "Basktall,Cherry Hills, 92%". Then applying the wrapper W mentioned earlier to extract sname will produce sname = $contain$("Basktall... 92%"). This states that sname is a string inside the above long HTML string, but we do not know which. Now suppose the user has replied "yes" to the question "is sname bold?". Then wrapper W can leverage this answer to "refine" the value of sname to $contain$("Basktall,Cherry Hills"). Figure 3.a-d shows how wrapper W extracts data from HOUSES and SCHOOLS into two compact tables T_1 and T_2.

Executing SQL queries over approximate data: We have described how SLIC applies wrapper W to extract approximate data into compact tables. Once this is done, SLIC executes user query Q over the tables, by (1) compiling Q into an execution plan P, and (2) executing P over the tables. SLIC currently constructs a simple execution plan P, which applies all applicable selections, then joins, then projections.

Executing P then reduces to executing its operators. Consider selection. This operator traditionally operates over relational tables. Now we must modify it to work over compact ones. In particular, to preserve the superset semantics (Section 2), this operator must be modified to return *all* tuples that *may* satisfy the selection condition. To illustrate, consider an execution plan P for query Q mentioned earlier. Suppose P starts by applying selection to compact table T_1 with condition c =(price $>$ 500000) & (area $>$ 4000) (see Figure 3.c). The first tuple x_1 of table T_1 has price = {exact(513000), ..., exact(3500)} and area = {exact(513000), ..., exact(3500)}. Since price contains at least one value above 500000, and area contains at least one value above 4000, operator selection concludes that x_1 may

satisfy c, and admits x_1. Similarly it admits x_2. The operator drops x_3 because x_3 does not contain any value above 500000 and hence cannot possibly satisfy c. Selection operator then produces as output compact table T_3 that consists of two tuples x_1 and x_2 (see Figure 3.e. Operators projection and join can be modified in a similar fashion to work over compact tables. In this case, the join operator produces as output the three tuple pairs shown in Figure 3.f, which is output as the result of executing plan P.

Interacting with the user and re-executing the query: Once query Q has been executed, SLIC displays the results to the user (as discussed earlier in Section 3.1). If the user is not yet satisfied, SLIC asks a new question to solicit more information about the attributes in query Q. Next, SLIC (a) adds the user answer to wrapper W in form of a domain constraint, as discussed earlier, (b) applies W to the Web pages again to extract new compact tables, and (c) re-executes plan P on the new compact tables. SLIC also detects and notifies the user when the result for a query appears to have "converged" to the correct result.

Key challenges in developing SLIC: In the current SLIC version, we have implemented relatively simple, "placeholder" solutions to the following problems: (1) applying approximate wrappers to extract data, (2) finding an optimized execution plan P for user query Q, (3) executing P, and hence the modified SQL operators, over the extracted approximate data. (4) selecting a question to ask the user, (5) re-extracting data, but incorporating the knowledge gleaned from the user answer, and (6) re-executing plan P over the newly extracted data. Instead, we have focused our effort on developing an efficient solution for approximate string join (see the next section), as we found that such a solution is critical to making SLIC usable. We found that the above SLIC version already works quite well with a broad variety of practical Web data processing scenarios (see Section 5).

4. Efficient Approximate Joins

Approximate string joins are critical in processing Web data, as we have argued in the Introduction. We now describe such joins. We then show why current solutions (e.g., [10, 18]) do not work for our context: approximate joins over approximately extracted data. Finally, we describe an effective solution that we currently employ in SLIC.

Given relational tables with a single value per tuple attribute, approximate joins are typically defined as follows [10]:

Definition 1 (Single-value Approximate Join) *Consider joining attributes a and b, which can take values $u_1, \ldots, u_n$ and $v_1, \ldots, v_m$, respectively. A single-value approximate join matches u_i and v_j, joining the corresponding tuples, if and only if $sim(u_i, v_j) \geq \theta$, where sim returns a similarity score between two values, and θ is some threshold.*

Figure 4.a shows a single-value approximate join on school names. Given a reasonable sim, we would match u_2 = "Basktall" to v_1 = "Basktall HS". In SLIC's context, attributes a and b often have multiple possible values. For example, in Figure 4.b, the value u_1 is the set of possible values {"Franklin", "two blocks"}. The join definition must therefore be modified as follows:

Definition 2 (Multi-value Approximate Join) *Consider joining attributes a and b, which can take values $U_1, \ldots, U_n$ and $V_1, \ldots, V_m$, respectively, where the U_i and V_j are sets of possible values. A multi-value approximate join matches U_i and V_j, joining the corresponding tuples, if and only if there exist $u \in U_i$ and $v \in V_j$ such that $sim(u, v) \geq \theta$, where sim returns a similarity score between two values, and θ is some threshold.*

This extension applies the intuition that when an attribute's value is one of set of possible values, to preserve superset semantics, the attribute passes a condition if and only if any of its possible values passes.

Two popular similarity functions are edit distance and TF/IDF [18, 10]. Implementing edit-distance joins in the SLIC context is relatively straightforward, so the rest of this section focuses on TF/IDF joins. Such joins are especially common [18], and implementing them in SLIC's context raises difficult challenges.

4.1. Challenges in Implementing TF/IDF Joins

The intuition used in the TF/IDF similarity function is that similar strings share rare tokens. Of course, a token's rarity is relative to the corpus from which the strings are drawn. When joining tables, the corpus is typically the union of values of the joining attributes. For example, the join in Figure 4.b has the corpus $\{u_1 \ldots u_n\} \cup \{v_1 \ldots v_m\}$. However, when attributes have multiple possible values, the true corpus is uncertain. To illustrate, consider the multi-value join in Figure 4.b. The first value for attribute a is either "Franklin" or "two blocks", the second is "Basktall", the third either "Vanhise" or "Vanhise Champaign", etc. The Cartesian product of all choices for a and b creates 8 possible single-value join scenarios. For instance, Figure 4.a and Figure 4.c are both possible scenarios, each with a different corpus.

Now consider joining U_1 = {"Franklin", "two blocks"} and V_1 = {"Basktall HS", "Urbana"}. Since we do not know the correct scenario, we must consider the corpora of all eight. For each scenario, knowing the corpus, we can compute similarity scores. To maintain the superset semantics, if U_1 joins V_1 in even one scenario, we must declare U_1 joins V_1 for our multi-value approximate join.

Considering all possible corpora is clearly impractical. Thus, we develop efficient approximate algorithms for multi-value approximate join, which we elaborate below.

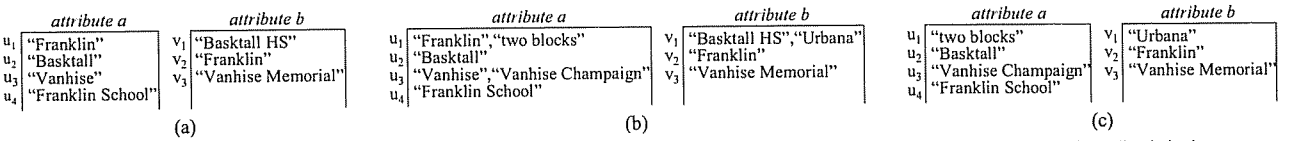


Figure 4. Examples of single-value approximate joins (cases a and c), and multi-value approximate join (case b). The join in case b conceptually consists of eight single-value joins, two of which are shown in cases a and c.

4.2. Efficient TF/IDF Join Algorithms for SLIC

Rather than considering all possible corpora, our key idea is to compute a bounding range for each token's rarity, then extend the TF/IDF join to operate over these ranges and produce an upper bound on similarity values. This section briefly describes the standard TF/IDF join algorithm and our proposed extension.

The Single-Value TF/IDF Join Algorithm: Given two values u and v, the single-value TF/IDF join computes $sim(u,v) = \vec{u} \cdot \vec{v}$, where $\vec{u}$ and $\vec{v}$ are TF/IDF vectors representing u and v.

To construct $\vec{u}$ and $\vec{v}$, we first construct the corpus C. Recall that attribute a has values $u_1, \ldots, u_n$. We tokenize each value u_i, then add the tokens to C as a *document*. We do the same for values of attribute b. C is thus a set of documents, each of which contains a set of tokens.

Let T be the set of tokens. Vector $\vec{u}$ has $|T|$ elements, one for each token $k \in T$. Let $\vec{u}_k$ be the element that corresponds to token k. We compute

$$\vec{u}_k = \alpha \cdot tf(k,u) \cdot log(idf(k)), \tag{1}$$

where α is a normalization constant computed as:

$$\alpha = 1/\sqrt{\sum_{k \in T} [tf(k,u) \cdot log(idf(k))]^2} \tag{2}$$

The *term frequency tf*(k,u) is the number of times k appears in u. The *inverse document frequency idf*(k) is $|C|$ divided by the number of documents containing k. Intuitively, if k occurs often in u yet rarely across all tuples, $\vec{u}_k$ is large, indicating that k is a distinguishing characteristic of u.

Note that many TF/IDF variants exist. We adopt the above variant because it has been shown to work well [18], is relatively simple to understand, and was found empirically to work well for our context (see Section 5).

The BJoin Algorithm: Now consider joining U and V in the multi-value context. Recall that they join if there exists $u \in U$ and $v \in V$ such that $sim(u,v) \geq \theta$. Our key idea is to compute an upper bound for $sim(u,v)$, denoted as $\overline{sim(u,v)}$, then join u and v if $\overline{sim(u,v)} \geq \theta$.

To compute $\overline{sim(u,v)}$, we use Equation 1 to write

$$\begin{aligned} sim(u,v) &= \vec{u} \cdot \vec{v} \\ &= \sum_{k \in u \cap v} \alpha_1\, tf(k,u)\, \log(idf(k)) \cdot \alpha_2\, tf(k,v)\, \log(idf(k)) \\ &= \alpha_1 \alpha_2 \sum_{k \in u \cap v} tf(k,u) tf(k,v) \log^2(idf(k)) \end{aligned} \tag{3}$$

The terms $tf(k,u)$ and $tf(k,v)$ count the number of times token k appears in u and v, and are trivial to compute. Consequently, to compute $\overline{sim(u,v)}$, we compute upper bounds for α_1, α_2, and $idf(k)$.

To compute an upper bound for $idf(k)$, we make a single pass over the union of values of attributes a and b to compute two quantities: $all(k)$ and $some(k)$. $all(k)$ is the number of tuples where all possible values of a or b contain k, and $some(k)$ is the number of tuples where at least one possible value of a or b contains k. It is easy to show that

$$idf(k) \in \left[\frac{N}{some(k)}, \frac{N}{all(k)}\right], \tag{4}$$

where N is the total number of values of a and b. Thus $N/all(k)$ is an upper bound for $idf(k)$.

To compute an upper bound for α_1, the normalization constant for $\vec{u}$, observe from Equation 2 that minimizing $\sum_{k \in T}[tf(k,u) \cdot log(idf(k))]^2$ yields this upper bound. For each term $[tf(k,u) \cdot log(idf(k))]$, again observe that $tf(k,u)$ is computed by trivial counting. So to minimize the term, we minimize $log(idf(k))$. This is done by setting $idf(k)$ to $N/some(k)$, as Formula 4 suggests. In effect, this is equivalent to upper bounding α_1 using the shortest possible unnormalized TF/IDF vector for u. We proceed similarly to compute an upper bound for α_2.

The CJoin Algorithm: BJoin treats U and V as sets of possible values. For instance, in Figure 4.b, the first value U_1 of attribute a is the set {"Franklin", "two blocks"}, meaning the true value is an element in the set. In practice, however, SLIC operates over compact tables, where each attribute value is a set of region assignments (see Section 3.2). Recall that we have two types of assignments: an *exact* assignment exact(R) means the true value is exactly R; a *contains* assignment contains(R) means the true value is a region inside R. If all assignments are exact, we can simply use algorithm BJoin. However, since contains assignments are common, we design the CJoin algorithm to handle them.

As a simple solution, CJoin can simply convert each assignment contains(R) into a set of exact assignments by enumerating all regions of R. For example, contains("$near\ Vanhise\ HS$") would become to {exact("$near$"), exact("$Vanhise$"), ... , exact("$nearVanhiseHS$")}. Once contains assignments have been converted, CJoin reduces to BJoin. However, since the conversion often produces an explosion of exact assignments, this is inefficient.

Domain	Data	Tables	Num Attrs	Num Pages	Num Tuples
Movies	3 pages, 3 tables, 1009 tuples	Roger Ebert's Greatest Movies List	1 / 1	1	242
		IMDB Top 250 Movies	1 / 4	1	250
		Prasanna Top Movies	2 / 1	1	517
DBLP	85 pages, 4 tables, 6033 tuples	Hector Garcia-Molina Pubs List	4 / 4	1	312
		SIGMOD Papers '75-'05	2 / 2	31	1787
		ICDE Papers '84-'05	2 / 2	22	1798
		VLDB Papers '75-'05	2 / 2	31	2136
Books	1078 pages, 3 tables, 10776 tuples	Amazon query on 'Database'	6 / 5	249	2490
		AddAll query on 'Database'	3 / 2	329	3286
		Barnes & Noble query on 'Database'	6 / 4	500	5000

Table 1. Real-world domains for our experiments

We therefore design CJoin to operate directly on contains assignments. The intuition is as follows. Suppose U is a single assignment contains(R), which can be converted to {exact(r_1), ..., exact(r_n)}. Suppose V is a single assignment exact(s). Then semantically, U and V match if there exists an r_i such that $sim(r_i, s) \geq \theta$. Consequently, if we compute an upper bound $\overline{R}$ over $sim(r_i, s), i \in [1, n]$, then U and V match if $\overline{R} \geq \theta$. This is clearly an approximation, but is also far more efficient than exhaustively considering each r_i. We compute $\overline{R}$ similarly to our treatment of BJoin, starting with Equation 3.

In general, consider two assignments $m_1(R_1)$ and $m_2(R_2)$. Similarly to our treatment of BJoin, we compute an upper bound on their similarity value by first computing the range of possible $idf(k)$ values for each token k. Specifically, we compute $all(k)$ as the number of tuples where both attributes a and b must contain k, and $some(k)$ as the number of tuples where at least one assignment contains k. Note that if R strictly subsumes k, assignment $contain(R)$ does not have to contain k.

Given $all(k)$ and $some(k)$, we compute an upper bound of $sim(m_1(R_1), m_2(R_2))$ as follows. We again upper bound $idf(k)$ with $N/all(k)$. We also upper bound α_1 and α_2 using the shortest possible unnormalized TF/IDF vectors for $m_1(R_1)$ and $m_1(R_1)$. However, for an assignment contains(R), any substring of R may be the true value. Considering this, we can show that if $m_1(R_1)$ is a contains assignment, the upper bound for α_1 normalizes a TF/IDF vector where every element is zero except the one corresponding to the token k in R_1, such that $k = argmin_t\ idf(t)$ where t ranges over all tokens in R_1.

Finally, we note that we have implemented various inverted indexes to speed up BJoin and CJoin but omit their description for space reasons. Algorithm CJoin is what we use in the current SLIC version for approximate string join.

5. Empirical Evaluation

We have evaluated SLIC on 27 query scenarios over 10 tables taken from 1166 HTML pages across 3 domains. Our goals were compare SLIC with current methods, and to evaluate its effectiveness for on-the-fly extraction and querying of Web data.

Domains: Table 1 summarizes the three real-world domains in our experiments. "Movies" contains 3 movie tables downloaded from 3 Web pages, "DBLP" 4 publication tables from 85 pages, and "Books" 3 book tables from 1078 pages. For each table we first downloaded a set of 1-500 Web pages. We then defined a schema for each table, consisting of 2-11 numeric and string attributes. For example, Table 1 contains "1 / 4" for "Num Attrs" for the "IMDB Top 250 Movies" table, indicating one string and four numeric attributes. Finally, we segmented each page into tuple regions, resulting in 242-5000 tuples per table.

Queries: Table 2 lists the nine queries used to evaluate SLIC, ranging from single attribute selections to three-table and multi-column joins. In these queries, operator "SIMILAR" refers to the approximate string join (see Section 4). We evaluate SLIC and alternative methods on 27 scenarios involving these queries, as described below.

Methods: We consider three alternative methods that can be used to answer SQL queries over Web data. First, we can employ wrapper-based methods to uncover the structured data. However, current wrapper systems are difficult to set up, and rather brittle. We experimented with several publicly available systems on our data sets, and found that they either failed to find any structure, or required significant manual correcting efforts. In addition, setting system parameters, such as which DOM subtrees should be ignored during parsing, or whether to enable backtracking during search, require a deep understanding of both the data and the systems. Wrapper methods thus are rather "heavy weight" for our on-the-fly query contexts.

We then opted to compare SLIC with two methods that are commonly used today, by casual users as well as domain scientists. The first is to *manually* collect the answers from the data. The second method employs Perl scripts and also RDBMSs if necessary. Here, we first wrote a script to extract the relevant attributes from each tuple, then computed the answer either within the script or by loading the structured data into a RDBMS and posing the query, whichever we deemed fastest.

Runtime and accuracy: Figure 5 shows the runtime and accuracy achieved using SLIC vs. the "Manual" and "Perl+DB" methods described above. Consider the very first row which shows "Movies" domain, "IMDB few votes" query, and 10 under "Num Tuples per Table". This row describes the following scenario. First, reduce the sizes of the tables involved in the query to just 10 tuples each, by random sampling. Next, execute the query using these tables.

The rest of the row shows the run time of "Manual", "Perl+DB", and SLIC methods, then the accuracy of SLIC. Each time point was averaged over the times of 3 volunteers (graduate students in our case), who employed the particular method. The row shows that "Manual" takes 1 minute and "Perl+DB" takes 28 minutes. It also shows two time amounts for SLIC, under "total time" (1 minute) and "interactive time" (1 minute). For all three methods, "total time"

Domain	Description	FROM	WHERE
Movies	IMDB top movies with few votes	IMDB	votes < 25,000
	Ebert top movies from 50's and 60's	Ebert	1950 ≤ year AND year < 1970
	Unanimous top movies	IMDB, Ebert, Prasanna	IMDB.title SIMILAR Ebert.title AND Ebert.title SIMILAR Prasanna.title
DBLP	Garcia-Molina journal pubs	Garcia-Molina	journal year ≠ NULL
	VLDB short pubs	VLDB	last page < first page + 5
	SIGMOD/ICDE pubs sharing authors	SIGMOD, ICDE	SIGMOD.authors SIMILAR ICDE.authors
Books	B&N books over $100	Barnes	b&n price > 100
	Amazon books to buy used	Amazon	list price = new price AND used price < new price
	Cheaper at Amazon than B&N	Amazon, Barnes	Amazon.title SIMILAR Barnes.title AND Amazon.new price < Barnes.b&n price

Table 2. SQL queries for our experiments

Domain	Query	Num Tuples per Table	Manual Total Time (min)	Perl + DB Total Time (min)	SLIC			
					Total Time (min)	Interactive Time (min)	Num Correct Tuples	Superset Size
Movies	IMDB few votes	10	1	28	1	1	0	100%
		100	1	29	1	1	31	100%
		250	3	29	1	1	128	100%
	Ebert top in 50's and 60's	10	1	31	1	1	4	100%
		100	1	31	1	1	31	100%
		242	3	31	1	1	80	100%
	Unanimous top movies	10	1	58	1	1	0	100%
		100	14	58	2	2	13	170%
		242-517	80	58	4	4	61	161%
DBLP	Garcia-Molina journal pubs	10	1	34	1	1	5	100%
		100	2	34	1	1	21	100%
		312	5	34	1	1	95	100%
	VLDB short pubs	100	4	37	1	1	30	100%
		500	19	37	1	1	119	100%
		2136	—	37	3	3	318	100%
	SIGMOD/ICDE shared authors	100	76	55	6	5	15	100%
		500	—	56	8	5	318	100%
		1787-1798	—	57	15	6	1756	114%
Books	B&N books over $100	100	4	33	1	1	11	100%
		500	20	33	1	1	52	100%
		5000	—	33	8	7	397	100%
	Amazon buy used	100	4	42	3	3	20	100%
		500	19	43	4	3	155	100%
		2490	—	43	5	4	537	100%
	Cheaper at Amazon than B&N	100	137	57	31	30	45	100%
		500	—	57	34	31	238	100%
		2490-5000	—	97	95	35	906	102%

Figure 5. Accuracy and run time of Manual vs. Perl+DB vs. SLIC.

is computed from when the user is given the data and the query, until when the results are obtained (or in SLIC case, until the time SLIC reports convergence). For SLIC, "interactive time" is the total number of minutes the user is required to remain at the console (i.e., until SLIC notifies the user that it will ask no further questions).

The last two columns of the first row show the number of correct tuples in the result of the query (0 in this case) and the superset size (100%). The value x% means the superset returned is at x% of the size of the set of correct tuples. (This set is found using the "Perl+DB" method.) The closer x is to 100, the better result for SLIC. "Manual" and "Perl+DB" always obtained 100%.

Figure 5 shows that, as expected, "Manual" does not scale to large data sets (we attempted "Manual" on some of these, but stopped after it became clear that the method was not scalable). It however outperformed "Perl+DB" on small data sets. "Perl+DB" spent most of its time in writing and debugging Perl scripts.

Compared to "Manual", SLIC achieved comparable or lower total time on small data sets (e.g., 1-4 minutes). It clearly can also handle large data sets for which "Manual" is not scalable. Compared to "Perl+DB", in all cases SLIC achieved an order-of-magnitude reduction, or comparable in total time. For example, on "Unanimous top movies" queries, "Perl+DB" incurred 58 minutes, whereas SLIC incurred only four minutes. The most comparable case was "Cheaper at Amazon than B&N" in the last row of the figure. Here "Perl + DB" finished in 97 minutes and SLIC in 95. In this case SLIC spent most time in extraction (e.g., B&N has complex HTML). We believe SLIC's time can be significantly reduced by a reimplementation in C (the current SLIC version is in Perl). Besides run time advantages, it is important to note that SLIC only requires the user to be able to answer intuitive questions (e.g., "*Is price in bold font?*"). In contrast, "Perl + DB" requires that the user is competent in a scripting language (e.g., Perl), building HTML wrappers, and interacting with RDBMSs.

Accuracy: Figure 5 shows that in 23 out of 27 scenarios SLIC returned the exact results (e.g., superset size 100%). The largest superset sizes (170% and 161%) occurred when the number of results returned was relatively small (22 and 98 tuples), which could be easily corrected with manual

postprocessing. The results thus suggest that SLIC can achieve high accuracy in a relatively short amount of time.

User Interaction: We have also randomly selected nine query scenarios, and examined them to evaluate the effectiveness of SLIC in soliciting domain knowledge to quickly converge on the correct result (not shown due to space limitation). The results show that in all nine scenarios SLIC converged in only a few iterations (2-10) to the exact result sets, suggesting that SLIC is effective in soliciting domain knowledge from the user to converge quickly to highly accurate query results.

6. Related Work

Our work is most related to the dual problems of extracting structured data and managing uncertain data. The problem of extracting structured data from text, HTML pages, email, etc. has received significant attention in the database, AI, Web, and KDD communities [1, 26, 27, 7]. Many recent works (e.g., [26, 27]) develop powerful learning-based extraction methods (e.g., HMM, CRF). For template-based Web pages, numerous works have developed wrapper solutions (e.g., [3, 19, 4]), or data description languages [7]. Other works address the issue by directly transforming HTML pages into more structured models, such as XML [25, 9]. The key commonality underlying these methods is that they tend to require substantial start-up efforts (e.g., manual labeling of training data or rule construction), and hence are not well suited for our ad hoc query contexts. Furthermore, they are often geared toward extracting a broad variety of structured data for long-term or repeated querying. In contrast, SLIC is strictly *query-centric*: it extracts only as much structure as necessary to answer a user query, a principle that appears well suited for ad hoc, one-time query contexts.

The extraction process often involves some uncertainty (as SLIC can attest). Managing uncertain data has recently received much attention [24, 28, 5, 15]. Our work here manages uncertainty in the context of integrating Web data, a specific and practical problem. Our work also examines how domain knowledge can be interactively solicited and exploited to reduce the uncertainty, thereby contributing a potentially novel aspect to research on managing uncertain data.

Finally, prior work such as [23] has also discussed set-valued representations, but there the focus was more on efficient *storage* solutions, which can potentially be valuable in our contexts. The work [22] also discusses set-based joins, but does not consider TF/IDF semantics, which raised challenging issues (e.g., unknown corpora) in our settings.

7. Conclusion and Future Work

We have introduced the problem of on-the-fly extraction and querying of template-based Web data. We developed the SLIC system, which to the best of our knowledge is the first end-to-end solution proposed for this important problem. SLIC constructs query-directed, approximate wrappers by asking the user only relatively simple questions. It also interleaves the process of extraction and execution, to converge to a satisfactory query result quickly, with minimal user efforts. We evaluated SLIC extensively on real-world domains, demonstrating that for many interesting and practical queries the user can obtain useful results after only a few simple questions, and that SLIC is significantly faster than current solutions. SLIC thus provides a promising first step toward a principled solution for on-the-fly processing of Web data. In the future we plan to improve upon several technical components of SLIC, such as extending relational query optimization techniques for our framework and investigating the challenges highlighted in Section 3.2.

References

[1] E. Agichtein and V. Ganti. Mining reference tables for automatic text segmentation. In *Proc. of KDD-04*, 2004.

[2] K. Al-Kofahi and J. Conrad. Large-scale legal and business information management: R&d at thomson legal & regulatory. In *DAIS Seminar Series, UIUC*, 2006.

[3] A. Arasu and H. Garcia-Molina. Extracting structured data from Web pages. In *Proc. of SIGMOD-03*, 2003.

[4] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *The VLDB Journal*, pages 109–118, 2001.

[5] N. Dalvi and D. Suciu. Answering queries from statistics and probabilistic views. In *VLDB*, 2005.

[6] A. Deutsch, Y. Katsis, and Y. Papakonstantinou. Determining source contribution in information integration systems. In *PODS*, 2005.

[7] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *POPL*, 2006.

[8] D. Gibson, K. Punera, and A. Tomkins. The volume and evolution of web page templates. In *WWW (Special interest tracks and posters)*, 2005.

[9] J. Graupmann, R. Schenkel, and G. Weikum. The spheresearch engine for unifi ed ranked retrieval of heterogeneous xml and web documents. In *VLDB*, 2005.

[10] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *WWW*, 2003.

[11] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt, and G. Heber. Scientifi c data management in the coming decade. In *SIGMOD Record 34(4)*, Dec. 2005.

[12] A. Halevy and C. Li. http://www2.cs.washington.edu/nsf2003/fi nal-reports/idm2003-report.pdf. 2004.

[13] http://2dpancakes.typepad.com/ernie/2004/12/where_cs_facult.html.

[14] http://randomratio.blagspot.com/2005_05_01_randomratio_archive.html.

[15] E. Hung, L. Getoor, and V. Subrahmanian. Pxml: A probabilistic semistructured data model and algebra. In *ICDE*, 2003.

[16] P. Ipeirotis, A. Ntoulas, J. Cho, and L. Gravano. Modeling and managing content changes in text databases. In *ICDE*, 2005.

[17] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa. Supporting exploratory queries in databases. In *DASFAA*, 2004.

[18] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, 2004.

[19] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proc. of IJCAI-97*, 1997.

[20] K. Lerman, L. Getoor, S. Minton, and C. Knoblock. Using the structure of web sites for automatic segmentation of tables. In *SIGMOD*, 2004.

[21] E. Rahm, A. Thor, D. Aumueller, H. Do, N. Golovin, and T. Kirsten. ifuice - information fusion utilizing instance correspondences and peer mappings. In *WebDB*, 2005.

[22] K. Ramasamy, J. Patel, J. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, 2000.

[23] S. Sarawagi and A. Kirpal. Effi cient set joins on similarity predicates. In *SIGMOD*, 2004.

[24] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.

[25] R. Torlone and P. Atzeni. Chameleon: an extensible and customizable tool for web data translation. In *VLDB*, 2003.

[26] V. Vydiswaran and S. Sarawagi. Learning to extract information from large websites using sequential models. In *COMAD*, 2005.

[27] B. Wellner, A. McCallum, F. Peng, and M. Hay. An integrated, conditional model of information extraction and coreference with application to citation matching. In *Proc. of UAI-04*, 2004.

[28] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.