Building Structured Web Community Portals: A Top-Down, Compositional, and Incremental Approach

Pedro DeRose[1], Warren Shen[1], Fei Chen[1], AnHai Doan[1], Raghu Ramakrishnan[2]
[1]University of Wisconsin-Madison, [2]Yahoo! Research

ABSTRACT

Structured community portals extract and integrate information from raw Web pages to present a unified view of entities and relationships in the community. In this paper we argue that to build such portals, a top-down, compositional, and incremental approach is a good way to proceed. Compared to current approaches that employ complex monolithic techniques, this approach is easier to develop, understand, debug, and optimize. In this approach, we first select a small set of important community sources. Next, we compose plans that extract and integrate data from these sources, using a set of extraction/integration operators. Executing these plans yields an initial structured portal. We then incrementally expand this portal by monitoring the evolution of current data sources, to detect and add new data sources. We describe our initial solutions to the above steps, and a case study of employing these solutions to build DBLife, a portal for the database community. We found that DBLife could be built quickly and achieve high accuracy using simple extraction/integration operators, and that it can be maintained and expanded with little human effort. The initial solutions together with the case study demonstrate the feasibility and potential of our approach.

1. INTRODUCTION

The World-Wide Web hosts numerous communities, each focusing on a particular topic, such as database research, movies, digital cameras, and bioinformatics. As such communities proliferate, so do efforts to build *community portals*. Such a portal collects and integrates relevant community data, so that its members can better discover, search, query, and track interesting community activities.

Most current portals display community data according to topic taxonomies. Recently, however, there has been a growing effort to build *structured data portals*. Such portals *extract* and *integrate* information from raw Web data, to present a unified view of *entities* and *relationships* in the community. For example, the Citeseer portal extracts and integrates publications to create an entity-relationship (ER) graph that captures citations. As another example, the Internet Movie Database (IMDB) extracts and integrates actor and movie information to create an ER graph that captures relationships among these entities.

In general, structured portals are appealing because they can provide users with powerful capabilities for searching, querying, aggregating, browsing, and monitoring community information. They can be valuable for communities in a wide variety of domains, ranging from scientific data management, government agencies, and enterprise intranets, to business and end-user communities on the Web.

Unfortunately, despite the growing importance of such structured portals, today there is still no general consensus on how best to build them. In commercial domains, many structured portals have been developed, such as Yahoo! Finance, Froogle, EngineSpec, Zoominfo, and IMDB. Little, however, has been published on their development. Our private communications suggest that they often employ a combination of manual and automatic techniques, which is developed specifically for the target domain and hence difficult to port.

In the research community, several semi-automatic solutions have been developed to build structured portals. Notable examples include Citeseer, Cora [29], Deadliner [26], Rexa, and Libra [30]. Most of these works take a "shotgun" approach. They start by discovering *all* Web sources deemed relevant, in an attempt to maximize coverage. Next, they apply *sophisticated* extraction and integration techniques (e.g., HMM [29], CRF [31]), often *uniformly* to all data sources. Finally, they expand the portal by periodically re-running the source discovery step. These solutions achieve good coverage and incur relatively little human effort. However, they are often difficult to develop, understand, and debug (e.g., they require builders to be well-versed in complex learning technologies). Given the monolithic nature of the employed techniques, it is also often difficult to optimize the run-time and accuracy of these solutions.

Given the limitations of current approaches, in this paper we argue that building structured portals using a top-down, compositional, and incremental approach is a good way to proceed. In this approach, we first select a small set of important community sources. Next, we create plans that extract and integrate data from these sources to generate entities and relationships. These plans, however, are not monolithic "blackboxes" that generate all entities and relationships from all data sources. Instead, each plan focuses

VLDB '07, September 23-28, 2007, Vienna, Austria.

on particular entities and relationships using a composition of extraction and integration operators. In creating each plan, we decide which operators to use, in which order, and on which sources. Executing these plans yields an initial structured portal. We then incrementally expand this portal by monitoring certain current sources for mentions of new sources. The following example illustrates our approach.

EXAMPLE 1. *Consider building a structured portal for the database research community. Toward this goal, current approaches often start by finding as many relevant data sources as possible. This can include those that are well-known (e.g., DBLP, DBworld), those that are linked to by well-known sources, and those that are suggested by search engines (e.g., in response to database related keyword queries). In contrast, our approach starts with a small set of relevant data sources, such as DBworld, the homepages of the 300 most productive database researchers, seminar pages at the top 200 CS departments, etc.*

Next, in the extraction and integration step, prior approaches often apply some monolithic solution to extract all structures from all data pages, then integrate them to form entities and relations in a bottom-up fashion. Our approach, however, is top-down and compositional. It starts with an entity or relation type X (e.g., serve-on-PC), then considers where to get most instances of X most reliably. Suppose it decides DBworld. Then it creates a plan to find instances of X from only DBworld pages (not from any other pages, even if they contain serve-on-PC information). In creating this plan, our approach employs a set of extraction and integration operators. For instance, an operator may extract all person mentions from the raw data, then another operator matches these mentions to create person entities. Similarly, another set of operators create conference entities. Then a final set of operators discover serve-on-PC relations between these person and conference entities.

Finally, to expand, current approaches often probe the structure of the Web (e.g., using focused crawling [6, 8]) to find and add new data sources. In contrast, we observe that most new relevant data sources will eventually be mentioned within community sources. Hence, instead of focused crawling the Web, we monitor certain dynamic community sources (e.g., DBworld, researcher blogs, conference pages) to find mentions of new data sources (e.g., the URL of an upcoming workshop).

We argue that the above approach to building structured portals is a good way to proceed for several important reasons. First, since we start with a small set of high-quality sources, and incrementally expand (rather than gathering all potentially relevant data sources at once, to build the initial portal, as in current approaches), we can control the quality of the sources admitted to the system, thereby maintaining system accuracy. Second, since we compose plans from modular extraction/integration operators, such plans are often easier to develop, understand, debug, and modify than monolithic and complex plans in current approaches. Third, the compositional nature of our approach also makes it highly amenable to optimization efforts, as we will briefly discuss. Finally, and somewhat surprisingly, we found that in our approach, even plans with relatively simple extraction/integration operators can already achieve high accuracy. The following example illustrates this point.

EXAMPLE 2. *Continuing with the above example, consider the problem of matching researcher mentions (i.e., deciding if "David DeWitt" and "D. DeWitt" refer to the same real-world entity). Numerous complex and powerful solutions have been developed for this problem [25]. In our approach, however, we found that the simple plan of matching mentions if they share similar names already achieves a high accuracy of 98% F_1 (see Section 6). This is because in many Web communities, people intentionally use as distinct names as possible to minimize confusion. As another example, consider finding the relation give-talk(person,dept). Prior approaches may write a powerful plan for this relation, then apply it to all data pages. In our approach, we start by deciding that we can obtain most "giving talk" instances from seminar pages. We then monitor the 200 seminar pages in the system. If a person name A occurs in the seminar page of a department B and is not preceded by the word "host", then we conclude that A gives a talk at B. This plan is simple to write, and can already achieve a high accuracy of 88% F_1 (see Section 6).*

In the rest of the paper we elaborate on the above reasons. We start by considering the problem of selecting an initial set of data sources. Given the large number of data sources potentially relevant to a community, selecting a good initial set is often difficult. To assist the builder in this task, we develop a solution that automatically ranks a large set of data sources in decreasing order of relevance to the community. To compute source relevance, our solution exploits the content of sources, as well as "virtual links" across sources as implied by the presence of domain entities (Section 2).

Given an initial set of sources, we then consider the problem of creating extraction/integration plans that, when applied to these sources, produce a desired ER graph. We describe in detail a methodology to create such plans incrementally, piece by piece, using a set of extraction/integration operators. Further, we show how relatively simple plans can be constructed using simple implementations of these operators (Section 3). Then, to wrap up the description of our approach, we show how to maintain the initial portal, and expand it over time by monitoring selected current sources to detect new data sources (Section 4).

Finally, we describe Cimple, a workbench that we have been developing at Wisconsin and Yahoo! Research (Section 5). By "workbench", we mean a set of tools that developers can use to quickly build structured community portals in our approach. As a case study, we have employed this workbench to build DBLife, a structured portal for the database community. We found that DBLife could be built quickly with high accuracy using simple extraction/integration operators, and that it can be maintained and expanded with little human effort (Section 6).

To summarize, in this paper we make the following contributions:

- We show that a top-down, compositional, and incremental approach is a good way to build structured community portals, as it is relatively easy to develop, understand, debug, and optimize.
- We describe an end-to-end methodology for selecting data sources, creating plans that generate an initial structured portal (in form of an ER graph) from these sources, and evolving the portal over time. We show how these plans can be composed from modular extraction and integration operators.

- We describe an implementation of the above approach in the Cimple workbench, and a case study with DBLife, a portal built using this workbench. The case study demonstrates that our approach can already achieve high accuracy with relatively simple extraction/integration operators, while incurring little human effort in portal maintenance and expansion.

For space reasons, in what follows we will focus only on the key ideas of our approach, and refer the readers to the full paper [13] for a complete description.

2. SELECTING INITIAL DATA SOURCES

To build a community portal in our approach, the builder starts by collecting an initial set of data sources that are highly relevant to the community. We now describe how to collect such data sources. The next section describes how to create an initial portal in the form of an ER data graph from these sources. Section 4 then describes how to maintain and evolve this initial portal.

To select initial sources, we observe that the 80-20 phenomenon often applies to Web community data sources, i.e., 20% of sources cover 80% of interesting community activities. Thus, a builder B can start by selecting as much of this core 20% as possible, subject to his or her time constraints.

To illustrate, for the database research community, B can select home pages of top conferences (e.g., SIGMOD, PODS, VLDB, ICDE) and the most active researchers (e.g., PC members of top conferences, or those with many citations), the DBworld mailing list, and so on. As another example, for the digital camera community, B can select home pages of top manufacturers (e.g., Sony, Nikon, Olympus), top review websites (e.g., *dpreview.com*, *cnet.com*), and camera sections of top retailers (e.g., *bestbuy.com*, *amazon.com*).

In many cases, B is a domain expert and already knows many of these prominent data sources. Still, it can be difficult even for a domain expert to select more relevant sources beyond this set of prominent ones. Hence, we have developed RankSource, a tool that helps B select data sources. To use RankSource, B first collects as many community sources as possible, using methods commonly employed by current portal building approaches (e.g., focused crawling, querying search engines, etc. [6]). Next, B applies RankSource to rank these sources in decreasing order of relevance to the community. Finally, B examines the ranked list, starting from the top, and selects the truly relevant data sources. We now describe three versions of RankSource that we have developed, in increasing degree of effectiveness.

PageRank-only: Let S be the set of sources collected by B. To rank sources in S, this RankSource version exploits the intuition that (a) community sources often link to highly relevant sources, and (b) sources linked to by relevant sources are more likely to be relevant. This is the same intuition used in algorithms such as PageRank [5] to evaluate the relevance of Web pages. Hence, we apply PageRank to rank our sources. To do so, we first crawl Web pages in each source $s \in S$ to a pre-specified depth. Then, we build a graph G over S by creating a node for each $s \in S$, and an edge $u \rightarrow v$ if a page in source u links to a page in source v.

Next, we calculate the PageRank for each node $u \in G$. Assume that nodes $v_1, \ldots, v_n$ link to u and u itself links to $c(u)$ nodes. Then we compute the PageRank of u as

$$P(u) = (1-d) + d\sum_{i=1}^{n} P(v_i)/c(v_i)$$

where d is a pre-specified damping factor. Finally, we return the sources (i.e., nodes of G) in decreasing order of their PageRank to the builder B.

We found that this RankSource version achieves limited accuracy because some highly relevant sources are not linked to often. For example, in the database domain, few sources link to individual conference pages, even though these pages are highly relevant. Thus, we extend PageRank-only to consider both link structure and source content.

PageRank + Virtual Links: Much community information involves entities. Thus, if a highly relevant source discusses an entity, it intuitively follows that other sources that discuss this entity may also be highly relevant. To leverage this idea, we extend PageRank-only as follows. First, we collect as many entity names as we can within the community. Such collections of names are often readily available with just a little manual scraping of certain Web sources. Next, we find occurrences of these entity names in the sources of S (by applying the mention extraction operator ExtractMbyName described in Section 3.1.1). We say a source s mentions an entity e if e's name (modulo a name-specific transformation) occurs in a page crawled from s. Next, we create graph G, as in PageRank-only, then add *virtual links* between sources that mention overlapping entities. Specifically, if sources s and t mention entity e, we add links $s \rightarrow t$ and $t \rightarrow s$. Finally, we compute PageRank as before.

Somewhat surprisingly, our experiments showed that these virtual links actually reduced accuracy. We discovered that this is because we link sources even if they only share entities that are not their main focus. Thus, most virtual links are noise, reducing accuracy. We address this problem by considering only entities highly relevant to each source, as the next RankSource version discusses.

PageRank + Virtual Links + TF-IDF: To avoid noise, we want to add a virtual link between sources only if they share an entity that is highly relevant to both. Intuitively, an entity e is relevant to source s if it appears often in s, but in few other sources. We quantify this using TF-IDF [33] (term frequency-inverse document frequency), a popular metric for measuring the relevance of terms in documents.

Let E be the collection of entities we have found so far. We first calculate the TF-IDF score for each entity $e \in E$ mentioned in each source $s \in S$. This score is the product of the *term frequency* (i.e., how often e appears in s) and the *inverse document frequency* (i.e., how rarely e appears in S). Formally, let E_s be the entities mentioned in s. For each entity $e \in E_s$, let $c(e)$ be the number of times e appears in s. Then, we compute the term frequency of e as

$$\mathrm{TF}(e,s) = \frac{c(e)}{\sum_{f \in E_s} c(f)},$$

the inverse document frequency of e in S as

$$\mathrm{IDF}(e,S) = \log \frac{|S|}{|\{t \in S : e \in E_t\}|},$$

and the TF-IDF score of e in s given S as

$$\text{TF-IDF}(e,s,S) = \mathrm{TF}(e,s) \cdot \mathrm{IDF}(e,S).$$

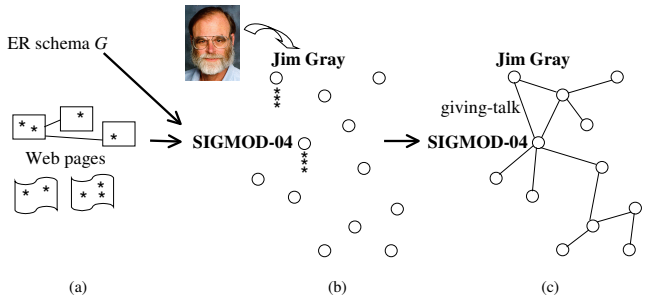

Figure 1: Workflow of P_{day} for the database domain

Next, for each source $s \in S$, we filter out any entity $e \in E_s$ where TF-IDF$(e, s, S) < \theta$ for some threshold θ. Finally, we apply PageRank + Virtual Links as before. Section 6 show that, compared to the above two RankSource versions, this version dramatically improves accuracy. Hence, it is the version we currently use in our approach.

3. CONSTRUCTING THE ER GRAPH

Let T be the initial set of data sources selected by builder B. We now describe how to extract and integrate data to construct an ER graph over T. Toward this goal, builder B begins by defining an ER schema G that captures entities and relations of interest to the community. G consists of multiple types of *entities* (e.g., person, paper) and *relations* (e.g., write-paper, co-author), each of which is associated with a set of *attributes* (e.g., name, email).

Next, builder B applies a crawling program to the sources in T at regular intervals to retrieve Web pages. The crawling interval is domain-specific, but for simplicity, in this paper we will assume *daily* crawling. For any source $s \in T$, if s comprises standard Web pages without forms, then we crawl s and retrieve all pages up to a pre-specified depth. Otherwise s is accessible only via a form interface, in which case we execute a set of queries (pre-specified by the builder) over s, to retrieve a set of Web pages.

Let W_1 be the first-day *snapshot*, i.e., all Web pages retrieved in the first day. Then, B creates and applies a plan P_{day} to W_1 to generate a daily ER graph D_1. On the second day, B crawls the data sources to retrieve W_2, then applies P_{day} to W_2 to generate the daily ER graph D_2, and so on.

Let $D_1, \ldots, D_n$ be the daily ER graphs generated from snapshots $W_1, \ldots, W_n$, respectively. Builder B then creates and applies a plan P_{global} to these daily ER graphs to create a global ER graph D_{global} that spans all the days that the portal has been in use. B can then offer a variety of user services over D_{global} (e.g., browsing, keyword search, querying, etc., not discussed in this paper).

In the rest of this section we will describe how B creates the daily plan P_{day} and the global plan P_{global}.

3.1 Create the Daily Plan P_{day}

The daily plan P_{day} takes as input a daily snapshot W and the ER schema G, then produces as output a daily ER graph D. Builder B creates P_{day} as follows. First, for each entity type $e \in G$, B creates a plan P_e that discovers all entities of type e from W. For example, given the ER schema G and the set of Web pages in Figure 1.a, a plan P_e for entity type "person" may find person instances such as Jim Gray, and a plan for entity type "conference" may find conference instances such as SIGMOD-04, as shown in Figure 1.b.

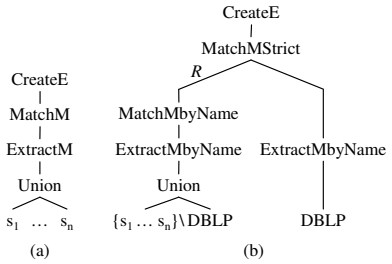

Figure 2: Plans for extracting and matching mentions

Next, for each relation type r, B creates a plan P_r that discovers all instances of relation r that "connect" the entities discovered in the first step. For example, a plan P_r for relation "giving-talk" may discover an instance of this relation between entities Jim Gray and SIGMOD-04, as shown in Figure 1.c (with an edge between the two entities).

Finally, B merges plans P_e and P_r to form the daily plan P_{day}. In what follows, we describe how to create plans P_e and P_r (merging them to form P_{day} is straightforward).

3.1.1 Create Plans to Discover Entities

A Default Plan: Figure 2.a shows $P_{default}$, perhaps the simplest plan that builder B can apply to find entities of type e. This plan first unions all Web pages retrieved from all data sources $s_1, \ldots, s_n$ to create the snapshot W.

Next, it applies an operator ExtractM to find all mentions of type e in Web pages of W (e.g., person mentions like "Jim Gray", "J. Gray", "Dr. Gray", etc.). Then, the plan applies an operator MatchM to find matching mentions, i.e., those referring to the same real-world entity. This in effect partitions the mentions into groups $g_1, \ldots, g_k$, where all mentions in a group g_i refer to the same real-world entity.

Finally, plan $P_{default}$ applies an operator CreateE to create an entity e_i for each group of mentions g_i. Creating e_i means assigning values to its attributes using the mentions in g_i. For example, if $g_i = \{$"Jim Gray", "J. N. Gray", "Dr. Gray"$\}$, then CreateE may create an entity e_i with name "Jim N. Gray" and title "Dr.".

The problem of extracting and matching mentions, as encapsulated in operators ExtractM and MatchM, is well-known to be difficult. Variations of matching mentions, for example, are also known as record linkage, entity matching, reference reconciliation, deduplication, and fuzzy matching. Numerous solutions have been developed for these problems, ranging from relatively simple rule-based methods to sophisticated learning-based techniques (e.g., [25]). Virtually any of these solutions can be implemented for the above operators.

In the context of Web community portals, however, we found that relatively simple solutions can already work quite well. Specifically:

- For extracting mentions, we have implemented ExtractMbyName, a simple dictionary-based solution that "matches" a collection of entity names N against the pages in W to find mentions. Specifically, ExtractMbyName first creates variations for each entity name $n \in N$ by applying type-specific perturbation rules to n (e.g., given the person name "John Kyle Smith", it creates variations "John Smith", "Smith, J. K.", etc.). Then, it

finds all occurrences of these variations in W, applies some simple filters to exclude overlapping occurrences, and returns the remaining occurrences as mentions.

- For matching mentions, we have implemented MatchMbyName, which matches mentions if they are similar modulo a type-specific transformation (e.g., it may declare that "John Kyle Smith", "John Smith", and "J. Smith" all match, because they are similar modulo a name-specific transformation).

Now let P_{name} be the $P_{default}$ plan that employs ExtractMbyName and MatchMbyName. We found that P_{name} already works quite well for DBLife, achieving an average F_1 of 98% for persons, papers, and organizations (see Section 6). This is because in the database domain, as well as in many Web communities, entity names are often intentionally designed to be as distinct as possible, to minimize confusion for community members. Hence, P_{name} may already work well in such cases. P_{name} is also broadly applicable, because builder B often can quickly compile relatively comprehensive collections of entity names with minimal manual effort (e.g., by screen scraping from well-known data sources).

Hence, in our approach B will apply plan P_{name}, whenever applicable, to find entities.

A Source-Aware Plan: P_{name} is not perfect, however. It sometimes matches mentions poorly in *ambiguous* sources, e.g., those that also contain data from external communities. For instance, in the database domain, DBLP is ambiguous because it contains data from non-database communities. Hence, it may contain a mention "Chen Li" of a database researcher as well as another mention "Chen Li" of an HCI researcher. MatchMbyName cannot tell such mentions apart, thus reducing the accuracy of P_{name}.

In such cases, we propose that builder B examine the output of P_{name} to identify the set of ambiguous sources $A \subseteq \{s_1, \ldots, s_n\}$ that cause low accuracy. Then, instead of P_{name}, B should employ a "stricter" plan P_{strict}. This plan first applies ExtractMbyName and MatchMbyName to the set of unambiguous sources S A to produce a result R, then applies a stricter matching operator MatchMStrict to the union of R and the set of ambiguous sources A. The following example illustrates this plan.

EXAMPLE 3. *Figure 2.b shows a plan P_{strict} that finds person entities in the database domain. At first, P_{strict} operates exactly like P_{name} over all non-DBLP data sources. It produces a result R specifying all mention groups $g_1, \ldots, g_k$ over these sources. Next, P_{strict} unions R with all mentions extracted from only DBLP using* ExtractMbyName *obtaining a result U. Then, it applies an operator* MatchMStrict *to U.*

MatchMStrict *is stricter than* MatchMbyName *in that it matches mentions in U using not just their names, but also their* contexts *in the form of related persons. Specifically,* MatchMStrict *first creates for each person mention $p \in U$ a set of related persons $r(p)$, e.g., by adding to $r(p)$ all person mentions found within a k-word distance of p. Next, it "enriches" the mentions: for any two mentions p and q matched in R (i.e., belonging to the same group g_i in result R), it adds $r(p)$ to $r(q)$ and vice-versa. Finally,* MatchMStrict *declares two mentions in U matched only if they have similar names and share at least one related person.*

If necessary, B can repeat the above steps, using progressively stricter matchers over increasingly ambiguous sources, until P_{strict} is sufficiently accurate (see [34] for an initial solution on how to semi-automatically find a good P_{strict}, given a set of matchers and a set of data sources).

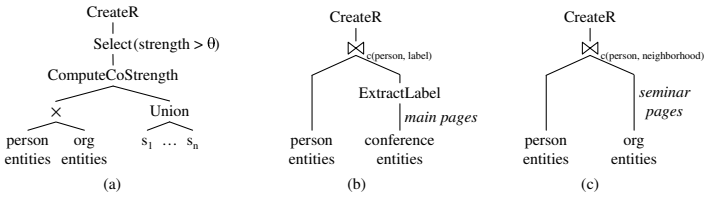

Figure 3: Plans for finding relations

3.1.2 Create Plans to Find Relations

In the next step, B creates plans to find relations among the discovered entities. Since relations can differ significantly in their natures, a single plan template is unlikely to work well for all relation types. Hence, our strategy is to identify types of relations that commonly occur in Web communities, then create plan templates for each of these types. In this paper, we focus on three such types: co-occurrence, label, and neighborhood relations.

Co-occurrence Relations: Such a relation $r(e, f)$ causes entities e and f to frequently co-occur in close proximity on data pages. Examples include affiliation(person,org), co-author(person,person), and write-paper(person,paper).

To find such relations, we have implemented an operator ComputeCoStrength, which inputs an entity pair (e, f), then outputs a number that quantifies how often and closely mentions of e and f co-occur. Specifically, let P be the set of all pages containing mentions of both e and f. We quantify the co-occurrence strength as $s(e, f) = \sum_{p \in P} 1/d(e, f, p)^{\alpha}$, where $d(e, f, p)$ is the shortest distance (in words) between a mention of e and a mention of f in page $p \in P$, and α is prespecified. While more sophisticated ways to measure $s(e, f)$ exist, we found that this simple measure already works well in our experimental domains (see Section 6).

Builder B can now create a plan P_r that establishes relation r between any entity pair (e, f) where $s(e, f)$, as computed by ComputeCoStrength over a set of data sources F, exceeds a threshold θ. B may have to experiment with different F and θ until reaching a desired accuracy.

EXAMPLE 4. *Figure 3.a shows* DBLife*'s P_{affil} plan. This self-explanatory plan discovers all pairs (e, f) where person e is affiliated with organization f. Note that it operates over the data of all sources in T. Here, operator* CreateR *creates a relation instance in a fashion similar to that of* CreateE.

Label Relations: The notion of a label is inherently subjective, but, briefly, it refers to titles, headlines, subject lines, section names, certain bold text, and so on within a data page. For instance, in a conference call for paper, example labels might include header text "Objectives of the Conference", and bold text "Important Dates", "PC Members", etc. Such labels can help us discover a variety of relations. For example, if we find a mention m of a person e in a page for a conference c, and if a label immediately preceding m contains the phrase "PC Members", then it is very likely that e is serving on the PC of c.

To find such relations, we have implemented an operator ExtractLabel, which inputs a set of pages P and outputs all labels found in P (see [13] for a detailed description). B can

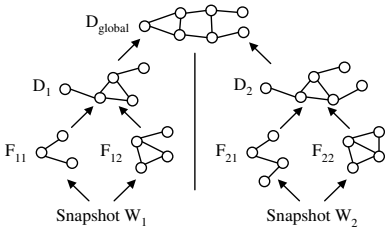

Figure 4: Merging graph fragments into daily ER graphs, and then into a global ER graph

then employ this operator to construct plans that find label relations, as the following example illustrates.

EXAMPLE 5. *Figure 3.b shows* DBLife*'s P_{served} plan. This plan inputs a set of persons and conferences. Then, for each conference c, it applies* ExtractLabel *to find all labels in the main pages of c (these are pages that have been specified to belong to c, e.g., c's homepage, using an attribute* main-pages*). Next, it finds all pairs (p, c) such that (a) a mention m of person p occurs in a main page of c, (b) there exists a label l immediately before m (i.e., there is no other label in between), and (c) l contains the text "PC" or "Program Committee". Finally, for each found pair (p, c), the plan creates a relation indicating that p serves on the PC of c.*

Neighborhood Relations: Such relations are similar in spirit to label relations, but require that, instead of labels, the immediate *neighborhood* (e.g., a window of k words on both sides) of a mention should satisfy some condition. B can easily construct plans for such relations, as the following example illustrates.

EXAMPLE 6. *Figure 3.c shows* DBLife*'s P_{talk} plan. The plan finds all pairs (p, o) such that (a) a mention m of person p appears in a seminar page of organization o, and (b) the word "host" does not appear within a 3-word window preceding m. For each found pair (p, o), the plan creates a relation indicating that p gives a talk at o.*

Discussion: When B judges a relation to be of one of the above co-occurrence, label, or neighborhood relation types, B can use the above plan templates. Otherwise, B creates a domain-specific plan. We found that the above three relation types are pervasive (e.g., constituting 75% of relations in the current DBLife).

3.2 Decompose the Creation of Daily Plan

So far we have assumed that B will create a single unified daily plan P_{day}, covering all entity and relation types in the community ER schema G, "in one shot". In reality, it is difficult to create such a plan, for the following reasons. (1) The ER schema G can be large (tens or hundreds of entity and relation types), and creating a large P_{day} for such schemas can be overwhelming. (2) B often want to spread the workload of plan creation over several persons (e.g., volunteers) by asking each of them to build a plan for a *part* of G. (3) Schema G often *evolves* over time, as community needs or B's idea of what the portal should contain change. Thus, if we add an extension G' to the current schema G, we may not want to re-create a plan for $G \cup G'$ from scratch. Rather, we may want to create a plan for G', then merge this plan with the plan for G.

For these reasons, we allow builder B to decompose schema G into smaller pieces $G_1, \ldots, G_n$, then create a plan P_i for each piece G_i using the methods described above in Section 3.1. Given a snapshot W_k of day k, B then applies plans $P_1, \ldots, P_n$ to W_k to create ER fragments $F_{k1}, \ldots, F_{kn}$. Finally, B merges these ER fragments to obtain the daily ER graph D_k. Figure 4 illustrates how two ER fragments F_{11} and F_{12} are merged to create the ER graph D_1 for day 1. It shows a similar process that creates D_2.

To merge ER fragments, B first uses one of these fragments as the initial daily graph. He or she then merges the remaining ER fragments into the daily graph using two operators: MatchE and EnrichE. Briefly, MatchE(D_i, F_{ij}) finds all matching nodes (i.e., entities) between D_i and F_{ij}. EnrichE(D_i, F_{ij}) then "enriches" D_i, by transferring certain information from entities in F_{ij} to matching entities in D_i (see [13] for more details on MatchE and EnrichE). For example, suppose that a paper entity p in F_{ij} has attribute num-citations $= 50$, and that p matches a paper entity q in D_i. Then EnrichE may set num-citations of q to 50, or create this attribute for q if it does not yet exist. Thus, in a sense, the daily ER graph D_i functions as a "daily data warehouse" that "siphons" data from the individual ER fragments F_{ij}.

3.3 Create the Global Plan P_{global}

Let $D_1, \ldots, D_n$ be the daily ER graphs generated from snapshots $W_1, \ldots, W_n$, respectively. Builder B then merges them to create a global ER graph D_{global} that spans all the days that the portal has been in use. Figure 4 illustrates the above process for Day 1 and Day 2.

As can be seen, this process is similar to merging ER fragments to create a daily ER graph, and thus employs the same operators MatchE and EnrichE. However, in this context, we can design a very simple MatchE by exploiting the observation that an entity's mentions usually do not change much from one day to the other. Let e be an entity in D_k, and $M_k(e)$ be the set of mentions of e found in Day k. Likewise, let f be an entity in D_{global}, and $M_{k-1}(f)$ be the set of mentions of f found in Day $k-1$. Then we compute the Jaccard similarity measure $J(e, f) = |M_k(e) \cap M_{k-1}(f)|/|M_k(e) \cup M_{k-1}(f)|$, and declares e and f matched if $J(e, f)$ exceeds a threshold θ. As far as we can tell, this simple MatchE achieves perfect accuracy in the current DBLife system.

We note that, in a sense, D_{global} functions as a "global data warehouse" that "siphons" data from the individual daily ER graphs D_i (just like D_i functions as a "daily data warehouse" siphoning data from individual ER fragments).

4. MAINTAINING & EXPANDING

We now describe how to maintain the initial portal and expand it over time. Maintaining an initial portal means ensuring that, as time passes, its data sources stay up-to-date. For example, if the initial DBLife operates over the 400 most productive researchers, the top 10 database conferences, and all their home pages, then we must maintain this coverage over time. Our current assumption is that this maintenance can be executed periodically, with relatively little manual work, because in most Web communities, the relevancy of data sources changes slowly over time. Section 6, for example, shows that we have maintained DBLife with only about 1 hour per month for the past 2 years.

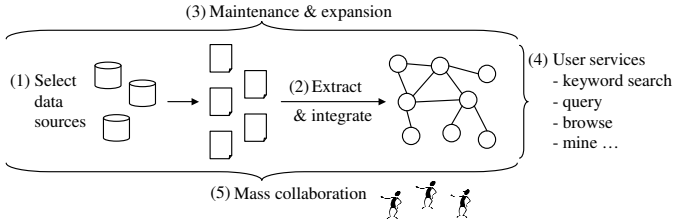

Figure 5: The five stages of building a portal with Cimple

Expanding an initial portal then means adding new relevant data sources, e.g., the homepages of new conferences and workshops beyond the top 10 conferences. (We can also add new entities, e.g., to the collection of entity names of ExtractMbyName, but we do not consider such scenarios in this paper.) A popular solution to expansion has been *focused crawling*: crawl the Web starting from a set of seed sources (e.g., those of the initial portal) in a focused fashion (e.g., follow only "highly promising" links) to find new relevant data sources [6, 8].

Focused crawling is certainly applicable to our context. However it is relatively complex to deploy and difficult to tune for a desired accuracy. Hence, for now we explore a complementary and simple strategy that exploits an important characteristic of Web communities. Specifically, we observe that most new important data sources will *eventually* be mentioned *within* the community in certain data sources U (as "advertisements" from some community members to the others). Hence, we simply monitor sources U, and write plans to extract mentions of new data sources. Since these plans are specific to U, they often are simple to write.

For example, to detect new database conferences and workshops, we monitor DBworld. If a DBworld message is of the type "conference announcement", then we parse the first few lines of the messages to extract a conference abbreviation X and a URL Y (if any). Y then becomes a new data source associated with conference X. If applied to all data sources, this simple method achieves only 50% precision in recognizing conference abbreviations. However, since we apply it only to DBworld messages, it achieves 97% precision.

Depending on the particular setting, builder B may decide to add newly found data sources automatically into the extraction/integration plans discussed in Section 3, or to "vet" them before adding. To "vet" them, B can employ the algorithm RankSource (Section 2) to evaluate their relevance.

5. THE CIMPLE WORKBENCH

We are now in a position to describe the Cimple workbench currently under development. Below we first describe how a builder B would use the workbench components, then how common characteristics of Web communities have influenced our development of these components.

5.1 Components of the Workbench

The workbench consists of an initial portal "shell" (with many built-in administrative supports), a methodology on populating the shell, a set of operator implementations, and a set of plan optimizers.

Portal Shell with Built-in Supports: To develop a community portal, a builder B starts with the "empty" portal shell provided by the workbench. This shell already has built-in supports for many "administrative" tasks, such as generic crawling, archiving and retrieving crawled data over time, identity generation and management, configuration file management, and result visualization, among others. These tasks, if implemented from the scratch, can easily consume a few man months of development effort.

Methodologies: Builder B then follows a specification in the workbench, which shows how decompose the problem of building a community portal into stages. Figure 5 shows this decomposition. To construct a portal, (1) builder B starts by selecting a set of initial relevant data sources. (2) Next, B constructs plans that crawl, extract, and integrates data from the sources, to form a global ER graph. (3) Over time, B must maintain and further evolve the initial portal, and (4) provide a variety of user services over the extracted and integrated data. (5) Finally, B can leverage the multitude of users in the community - in a mass collaboration fashion - to assist in all of the above tasks.

In this paper we have focused on Stages 1-3 (see [12] for our ongoing work for Stages 4-5). For Stages 1-3, the specification has identified a set of operators (e.g., ExtractM, MatchM, CreateE, ComputeCoStrength, ExtractLabel, MatchE, EnrichE, etc.), and show how to employ these operators to construct extraction/integration plans, as described in Section 3. A builder B can then either follow these guidelines, or modify them to suit a particular context.

Operator Implementations: To construct plans using the above generic operators, the current workbench already contains a set of operator implementations, e.g., ExtractMbyName for ExtractM. These implementations are relatively simple, but can already be highly effective in building portals (see Section 6).

Optimizers: Since the space of possible extraction and integration plans is often large, finding a good plan can be difficult. Hence, we have also been working on developing optimizers that builder B can use to (semi)automatically find good plans. For example, in [34] we describe an initial solution that semi-automatically finds a plan that discovers entities with high accuracy, given a set of mention matchers and data sources. In [35] we describe a solution that automatically finds a plan that minimizes the time to discover certain relations. We are currently "packaging" such optimizer solutions to be a part of the Cimple workbench.

5.2 Exploiting Community Characteristics

Throughout the paper, we have briefly touched on how common Web community characteristics have influenced our design decisions. We now summarize and expand upon these points. We note that builder B can also leverage these characteristics to build simple yet effective plans.

(1) Web communities often exhibit 80-20 phenomena. Hence, we adopted an incremental approach, in which we start with just the 20% most prominent data sources, because these sources often already cover 80% of interesting community activities. As another example, when collecting entity names for an entity type e (e.g., for dictionary-based methods), we can stop after collecting just the 20% most prominent names, because these names often are already involved in 80% of interesting activities related to e.

(2) Within a Web community, significant collections of entity/relation names are often available with just a little manual scraping of certain Web sources, hence our emphasis

Initial DBLife (May 31, 2005)		Time
Data Sources (846):	researcher homepages (365), department/organization homepages (94), conference homepages (30), faculty hubs (63), group pages (48), project pages (187), colloquia pages (50), event pages (8), DBWorld (1), DBLP (1)	2 days, 2 persons
Core Entities (489):	researchers (365), department/organizations (94), conferences (30)	2 days, 2 persons
Operators:	DBLife-specific implementation of MatchMStrict	1 day, 1 person
Relation Plans (8):	authored, co-author, affiliated with, gave talk, gave tutorial, in panel, served in, related topic	2 days, 2 persons

Maintenance and Expansion		Time
Data Source Maintenance:	adding new sources, updating relocated pages, updating source metadata	1 hour/month, 1 person

Current DBLife (Mar 21, 2007)	
Data Sources (1,075):	researcher homepages (463), department/organization homepages (103), conference homepages (54), faculty hubs (99), group pages (56), project pages (203), colloquia pages (85), event pages (11), DBWorld (1), DBLP (1)
Mentions (324,188):	researchers (125,013), departments/organizations (30,742), conferences (723), publication: (55,242), topics (112,468)
Entities (16,674):	researchers (5,767), departments/organizations (162), conferences (232), publications (9,837), topics (676)
Relation Instances (63,923):	authored (18,776), co-author (24,709), affiliated with (1,359), served in (5,922), gave talk (1,178), gave tutorial (119), in panel (135), related topic (11,725)

Figure 6: Activities and time to build, maintain, and expand DBLife

on starting with dictionary-based extraction operators such as ExtractMbyName. Even when this is not so, such collections can often be extracted relatively easily from some data sources (e.g., database conferences from DBworld), hence our emphasis on looking for such sources, then writing simple extraction plans tailored specifically to those sources.

(3) Entity names are often intentionally designed to be distinct, to avoid confusion by community members. Even when names clash, most such clashes are often limited to just a small percentage of names (again, the 80-20 phenomenon). Hence, we advocate using very simple name disambiguation methods (e.g., MatchMbyName) to correctly match most names, then applying "patches" (e.g., MatchMStrict) to handle the remaining hard-to-match names.

(4) Most new and interesting sources/entities/relations will eventually be mentioned within the community as a form of "advertisement" from one member to the others, hence our decision to expand via monitoring current sources.

(5) We can often solicit help from community members, in a mass collaboration fashion [17]. Hence, we can focus on acquiring the "short but prominent head", i.e., the relatively small number of prominent data sources, entities, relations, etc. We then design the portal so that community members can easily add the "long tail" (e.g., data sources that are related but not heavily used by most members). See [12] for our initial work on this topic.

6. CASE STUDY: DBLIFE

As a proof of concept, we conducted a preliminary case study by applying our Cimple workbench to build DBLife, a structured portal for the database community. Our goal is to illustrate how portals in our approach can be built quickly, maintained and expanded with relatively little effort, and already reach high accuracy with the current set of relatively simple operators.

6.1 Develop & Maintain DBLife

Manual Effort: Two developers (graduate students) spent roughly 4 months in 2005 to develop the Cimple workbench described in Section 5. Then, to build the initial DBLife portal, we first manually compiled a list of 846 prominent data sources and the names of 489 prominent entities (see "Initial DBLife" in Figure 6). Next, we implemented a strict matcher (MatchMStrict described in Section 3.1.1), specifically for DBLife, to match mentions from DBLP. Finally, we implemented 8 DBLife-specific plans to find relations. With this, we were able to launch the initial DBLife in May 2005 after a week of manual work by 2 developers.

Method	Precision At 10	50	100	200
PageRank-only	50.0	44.0	37.0	30.0
PageRank + Virtual Links	40.0	26.0	32.0	38.0
PageRank + Virtual Links +TF-IDF	80.0	80.0	77.0	64.0

Figure 7: Evaluation of RankSource variations

Since deployment, DBLife has required very little manual maintenance: adding new data sources, updating relocated pages, and updating certain metadata requires only about an hour per month (see "Maintenance and Expansion" in Figure 6).

By March 2007, DBLife has added 229 more data sources via expansion. It now processes 1,075 data sources daily, extracts roughly 324,000 mentions of nearly 16,700 entities, and has discovered nearly 64,000 relation instances between these entities (see "Current DBLife" in Figure 6).

Ease of Development: By March 2007, the code of DBLife has been deployed and extended by at least 13 individual developers in four different locations (CS departments at Illinois and Wisconsin, Biochemistry department at Wisconsin, and Yahoo Research). In all cases, the code (comprising 40,000 lines) was successfully deployed and development efforts started in a few days, using a simple user manual and a few hours of Q&A with us. We found that developers could quickly grasp the compositional nature of the system, and zoom in on target components. Since most such components (extractors, matchers, relation finders, etc.) are of "plug-and-play" nature and relatively simple, developers could quickly replace or tune them.

6.2 Accuracy of DBLife

We now evaluate the accuracy of the current DBLife. Our goal is to demonstrate that using the current set of relatively simple operators, DBLife already achieves high accuracy in selecting data sources, constructing ER graphs, and expanding to new data sources.

Selecting Data Sources: To evaluate the accuracy of RankSource (see Section 2), we first collected a large set of database-related data sources. Due to its 2 years of expansion, DBLife now crawls a relatively complete set of community sources. Hence, we took these 1,075 sources, then

Researchers	ExtractMbyName		
	R	P	F1
Daniel S. Weld	0.99	1.00	0.99
Richard T. Snodgrass	0.99	1.00	1.00
Roger King	1.00	0.88	0.93
Klaus R. Dittrich	1.00	1.00	1.00
Alon Y. Halevy	1.00	1.00	1.00
Mehul A. Shah	0.98	0.96	0.97
Panos K. Chrysanthis	1.00	1.00	1.00
Raghu Ramakrishnan	1.00	1.00	1.00
Jian Pei	1.00	1.00	1.00
Kevin Chen-Chuan Chang	0.99	0.95	0.97
Susan B. Davidson	1.00	0.99	0.99
Feifei Li	0.97	0.94	0.95
Feng Tian	1.00	1.00	1.00
Walid G. Aref	0.99	1.00	0.99
Torsten Grust	1.00	1.00	1.00
Jayavel Shanmugasundaram	1.00	1.00	1.00
Kyuseok Shim	0.97	1.00	0.99
Wolfgang Lindner	1.00	1.00	1.00
Nuwee Wiwatwattana	1.00	1.00	1.00
Ming-Syan Chen	1.00	0.84	0.91
Average	0.99	0.98	0.98

Figure 8: Accuracy of extracting mentions

added sources linked to from these sources, to obtain a set S of almost 10,000 data sources.

Next, we used RankSource to rank sources in S in decreasing order of relevance. (Recall from Section 2 that virtual-links versions of RankSource utilize a list of entity names. This experiment used the list of roughly 6,000 names of researchers and organizations in the current DBLife, see the second line under "Current DBLife" in Figure 6.)

To evaluate such rankings, in the next step we compiled a set of highly relevant data sources, called the *gold set*. This set contains (a) DBLP and DBworld main pages, (b) homepages of database researchers with 200+ citations, and (c) main pages, database group pages, event pages, the faculty hubs, and database project pages of 25 top database research organizations. In addition, since recent conferences often receive significant attention and hence can be considered highly relevant, we also added to the gold set the main pages of the three top database conferences in 2003–2006, resulting in a set of 371 data sources.

Since a builder B would typically examine the top k sources in such a ranking to find highly relevant sources, we report *precision at k*, i.e., the fraction of sources in the top k that are in the gold set (and thus considered highly relevant). Figure 7 shows precision at 10, 50, 100, and 200 for all three RankSource methods. (Note that since the initial sources builder B assembles need not contain all highly relevant data sources, we do not report the recall of RankSource.)

The results show that a direct application of PageRank (the first line) achieves low precisions (e.g., 30% at top-200). Interestingly, adding virtual links (the second line) actually reduced precisions in many cases. Adding a TF-IDF evaluation of the virtual links (the 3rd line) helps remove noisy virtual links, and dramatically increases precision. Overall, the complete RankSource (in the third line) achieves high precisions, ranging from 80% at top-10 to 64% at top-200. This suggests that RankSource can effectively help the builder select highly relevant sources.

Constructing the ER Graph: Next, we evaluate DBLife's accuracy in constructing the ER graph. Ideally, we would like to compute this accuracy over *all* decisions that DBLife has made in extracting, matching, relation finding, etc. However, this is clearly impossible. Hence, we decided instead to evaluate DBLife's accuracy over a random sample of 20

Researchers	P_{name}			P_{strict}		
	R	P	F1	R	P	F1
Daniel S. Weld	1.00	1.00	1.00	0.99	1.00	0.99
Richard T. Snodgrass	1.00	1.00	1.00	0.99	1.00	1.00
Roger King	**1.00**	0.97	**0.99**	**0.79**	0.97	**0.87**
Klaus R. Dittrich	**1.00**	1.00	**1.00**	**0.84**	1.00	**0.91**
Alon Y. Halevy	1.00	1.00	1.00	1.00	1.00	1.00
Mehul A. Shah	1.00	0.94	0.97	1.00	0.94	0.97
Panos K. Chrysanthis	1.00	1.00	1.00	1.00	1.00	1.00
Raghu Ramakrishnan	1.00	1.00	1.00	0.99	1.00	1.00
Jian Pei	1.00	1.00	1.00	1.00	1.00	1.00
Kevin Chen-Chuan Chang	1.00	1.00	1.00	1.00	1.00	1.00
Susan B. Davidson	1.00	1.00	1.00	0.94	1.00	0.97
Feifei Li	1.00	0.97	0.99	1.00	0.97	0.99
Feng Tian	1.00	**0.54**	**0.70**	1.00	**0.90**	**0.95**
Walid G. Aref	1.00	1.00	1.00	0.99	1.00	1.00
Torsten Grust	1.00	1.00	1.00	1.00	1.00	1.00
Jayavel Shanmugasundaram	1.00	1.00	1.00	1.00	1.00	1.00
Kyuseok Shim	1.00	1.00	1.00	1.00	1.00	1.00
Wolfgang Lindner	1.00	0.88	0.94	0.97	1.00	0.99
Nuwee Wiwatwattana	1.00	1.00	1.00	1.00	1.00	1.00
Ming-Syan Chen	1.00	0.97	0.99	0.90	0.97	0.93
Average	1.00	0.96	0.98	0.97	0.99	0.98

Figure 9: Accuracy of discovering entities

researchers (see the first column of Figure 8). We chose researchers because DBLife infers more information about researchers than any other entity type, and we selected only 20 researchers because even computing accuracies on this sample already took several days of manual effort. By accuracy, we mean the standard measures of precision P, recall R, and $F_1 = 2PR/(P+R)$.

Extracting mentions: To find researcher mentions, the current DBLife applies operator MatchMbyName (with a collection of names extracted from DBLP using a simple wrapper). Figure 8 shows the accuracies of this operator over the 20-researcher sample. To compute these accuracies, for each researcher X, we first found the set A of all mentions of X in all pages (of a daily snapshot). Suppose then that MatchMbyName found the set B of mentions of X. Then we computed $P = |A \cap B|/|B|$ and $R = |A \cap B|/|A|$.

The results show that this simple operator already achieves very high accuracy: precision of 84–88% in two cases and 94–100% in the rest, recall of 97–100%, and F_1 of 91–100%.

Discovering entities: Next, we evaluated DBLife's accuracy in discovering entities. Since the key step in this process is matching mentions (once this has been done, creating entities is straightforward), we evaluated the accuracy of this key step, with respect to each researcher. Specifically, for each researcher X in the sample, let F_X be the set of mentions of X found by DBLife (using MatchMbyName, as discussed earlier). Now suppose after matching mentions, DBLife assigned a set A_X of mentions to X. Then we compute $P = |F_X \cap A_X|/|A_X|$ and $R = |F_X \cap A_X|/|F_X|$. Figure 9 shows the results over the 20-researcher sample.

The figure shows that when matching mentions by just name similarity (using plan P_{name} described in Section 3.1.1), DBLife already achieves high accuracy of 94–100% F_1 in 19 out of 20 cases. In the case of Feng Tian, however, it achieves F_1 of only 70%, due to a precision of 54%. In this case, DBLP contains many names that share the same first initial "F" and last name "Tian", but belong to different researchers. Hence, the name-based matcher MatchMbyName performed poorly. And a low precision of 54% makes information collected about Feng Tian practically unusable.

Plan P_{strict} (described in Section 3.1.1) then "patches" up this case by employing a stricter matcher MatchMStrict over

Researchers	authored			affiliated with			served in			gave talk			gave tutorial			in panel		
	R	P	F1	R	P	F1	R	P	F1	R	P	F1	R	P	F1	R	P	F1
Daniel S. Weld	0.70	1.00	0.82	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
Richard T. Snodgrass	0.84	1.00	0.91	1.00	1.00	1.00	0.83	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Roger King	0.78	1.00	0.87	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Klaus R. Dittrich	0.81	1.00	0.89	1.00	1.00	1.00	0.75	1.00	0.86	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Alon Y. Halevy	0.49	1.00	0.66	0.00	0.00	0.00	0.70	0.88	0.78	0.75	1.00	0.86	1.00	1.00	1.00	0.40	1.00	0.57
Mehul A. Shah	0.91	1.00	0.95	0.00	0.00	0.00	0.67	1.00	0.80	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Panos K. Chrysanthis	0.59	1.00	0.74	1.00	1.00	1.00	0.86	0.86	0.86	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Raghu Ramakrishnan	0.98	1.00	0.99	1.00	0.50	0.67	0.90	0.95	0.92	0.63	1.00	0.77	1.00	1.00	1.00	0.50	1.00	0.67
Jian Pei	0.92	1.00	0.96	1.00	0.50	0.67	0.71	0.92	0.80	1.00	1.00	1.00	0.00	1.00	0.00	1.00	0.00	0.00
Kevin Chen-Chuan Chang	0.81	1.00	0.89	1.00	1.00	1.00	0.93	0.87	0.90	0.00	1.00	0.00	0.50	1.00	0.67	1.00	0.33	0.50
Susan B. Davidson	0.54	1.00	0.70	1.00	1.00	1.00	1.00	0.67	0.80	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Feifei Li	0.80	1.00	0.89	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Feng Tian	0.67	0.50	0.57	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Walid G. Aref	0.99	1.00	0.99	1.00	1.00	1.00	0.80	0.92	0.86	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Torsten Grust	0.65	1.00	0.79	1.00	1.00	1.00	0.50	0.75	0.60	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Jayavel Shanmugasundaram	0.84	1.00	0.91	1.00	0.50	0.67	0.75	0.67	0.71	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Kyuseok Shim	0.96	1.00	0.98	1.00	1.00	1.00	0.70	0.93	0.80	1.00	1.00	1.00	0.50	1.00	0.67	1.00	1.00	1.00
Wolfgang Lindner	0.64	1.00	0.78	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Nuwee Wiwatwattana	0.67	1.00	0.80	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Ming-Syan Chen	0.58	1.00	0.73	0.00	1.00	0.00	0.73	0.89	0.80	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Average	0.76	0.98	0.84	0.85	0.83	0.80	0.84	0.81	0.77	0.87	1.00	0.88	0.90	1.00	0.92	0.95	0.92	0.89

Figure 10: Accuracy of finding relations

DBLP (see Figure 2.b). It dramatically improves precision from 54% to 90% for Feng Tian, while keeping recall at 100%. P_{strict} reduces the accuracy of some of the remaining cases, e.g., Roger King and Klaus R. Dittrich. These names are not ambiguous in DBLP and MatchMStrict proves to be "too strict". Thus, F_1 scores for these cases drop from 99–100% to 87–91%, but still remain at a usable level.

These results therefore suggest that in Web communities, it may be a good idea to employ a relatively simple mention matching method to cover the vast majority of cases, then employ "patches" of stricter matching methods to cover the relatively fewer problematic cases. This would be in sharp contrast to current practice, where, typically, a single complex matching method is employed and tuned to maximize the overall accuracy [25].

Discovering relations: Next, we evaluate DBLife's accuracy in finding relations. Figure 10 shows the results for 6 researcher relations discovered by DBLife (results for the remaining relations are similar). Here DBLife discovered instances of authored using a plan tailored to DBLP, affiliated-with using ComputeCoStrength, and the remaining 4 relations using label-based plans (see Section 3.1.2). We compute precisions and recalls in a manner similar to the case of extracting mentions.

The results show that DBLife discovers relations with high accuracy: 77–92% F_1 on average (see last line of Figure 10). In affiliated-with, for example, DBLife finds the correct organizations (in the top-2) in 17 out of 20 cases. In only one case (Alon Y. Halevy) does DBLife not provide an organization. Furthermore, whenever DBLife provides an incorrect organization, the researcher has moved from that organization relatively recently. In served-in, the relatively low average F_1 of 77% is caused by just two cases, Feifei Li and Nuwee Wiwatwattana, where there are no true services and DBLife misidentifies a single service for each.

Expanding to New Data Sources: DBLife currently monitors DBworld to detect and add new conference data sources. If a DBworld message m is of type "conference announcement", then DBLife judges m to be a database conference announcement only if m mentions more than k researchers who are already in DBLife (where k is a prespecified threshold). DBLife then extracts any URLs in m as new data sources.

This method currently finds database conferences with 65% recall, 66% precision, and 66% F_1. By increasing k, we can achieve 83% precision, at a lower recall of 45%. Thus, the method has proven quite effective as a way to gather many database conferences quickly, with little effort.

Providing More Context using Current Portals: Even if DBLife achieves 91% F_1 in finding a relation, some may not consider this a high accuracy if current state-of-the-art portals already achieve, for example, 97%. Consequently, we examine the accuracy of several current portals to place DBLife's accuracies in a broader context.

We examine DBLife in the context of two academic community portals: Rexa (http://rexa.info) and Deadliner [26]. Rexa crawls the Web, then employs sophisticated learning techniques to extract and integrate information about researchers and papers [31, 40]. Figure 11 shows DBLife's accuracy in the context of Rexa's accuracy, again for the 20-researcher sample. Columns 2–3 of the figure show precisions in discovering entities, i.e., matching mentions correctly to each researcher. Columns 4–5 show the precision of finding authored instances, and Columns 6–7 show the precisions of finding affiliated-with instances. Note that we cannot report recall because we do not have access to Rexa's code, and hence do not know the corpus it processes.

The figure shows that DBLife does achieve high accuracies for these tasks when placed in the context of Rexa's accuracies. Note that these accuracies are not directly comparable because the two systems operate on different data corpora. A direct comparison is also not fair in that Rexa attempts to cover the entire CS community, not just the database one. Hence, Rexa must deal with a far larger amount of noise, leading to lower precisions. But this observation also raises an interesting future research question: to cover the entire CS community, can we just build many smaller communities, such as DBLife, AILife, SELife, etc.? If the total manual effort of building and maintaining these communities remains manageable, then this may be a good way to

Researchers	Rexa Match Mentions P	DBLife Match Mentions P	Rexa Find Pubs F1	DBLife Find Pubs F1	Rexa Find Orgs F1	DBLife Find Orgs F1
Daniel S. Weld	0.99	1.00	0.33	0.82	1.00	1.00
Richard T. Snodgrass	0.99	1.00	0.67	0.91	0.00	1.00
Roger King	0.53	0.97	0.84	0.87	0.00	1.00
Klaus R. Dittrich	0.95	1.00	0.77	0.89	1.00	1.00
Alon Y. Halevy	0.95	1.00	0.34	0.66	0.00	0.00
Mehul A. Shah	0.72	0.94	0.91	0.95	0.00	0.00
Panos K. Chrysanthis	1.00	1.00	0.55	0.74	1.00	1.00
Raghu Ramakrishnan	0.99	1.00	0.68	0.99	0.00	0.67
Jian Pei	0.93	1.00	0.57	0.96	0.00	0.67
Kevin Chen-Chuan Chang	0.54	1.00	0.50	0.89	0.00	1.00
Susan B. Davidson	1.00	1.00	0.49	0.70	1.00	1.00
Feifei Li	1.00	0.97	0.30	0.89	1.00	1.00
Feng Tian	0.20	0.90	0.67	0.57	1.00	1.00
Walid G. Aref	1.00	1.00	0.71	0.99	0.00	1.00
Torsten Grust	1.00	1.00	0.30	0.79	1.00	1.00
Jayavel Shanmugasundaram	1.00	1.00	0.57	0.91	0.00	0.67
Kyuseok Shim	0.94	1.00	0.75	0.98	0.00	1.00
Wolfgang Lindner	0.38	1.00	0.41	0.78	0.00	1.00
Nuwee Wiwatwattana	1.00	1.00	0.50	0.80	0.00	1.00
Ming-Syan Chen	0.99	0.97	0.45	0.73	0.00	0.00
Average	0.86	0.99	0.57	0.84	0.35	0.80

Figure 11: Accuracies of DBLife in the broader context provided by Rexa

cover the broader CS community with high accuracy.

Deadliner [26] is a niche portal that extracts conference information (e.g., titles, deadlines, PC members) from a variety of data sources. It employs a combination of handcrafted extractors and learning methods to combine the extractors. We evaluate Deadliner in terms of discovering instances of served-in (this is the only kind of data that both Deadliner and DBLife find). Deadliner discovers served-in instances from DBworld messages with 86% recall, 86% precision, and 86% F_1. In this context, DBLife achieves high accuracies of 98% recall, 97% precision, and 98% F_1, also for served-in over DBworld messages.

7. RELATED WORK

Information Extraction & Integration: Information extraction (IE) dates back to the early 80's, and has received much attention in the AI, KDD, Web, and database communities (see [1, 9, 18] for recent tutorials). The vast majority of works improve IE accuracy (e.g., with novel techniques such as HMM and CRF [9]), while some recent pioneering works improve IE time (e.g. [7, 24]). Most works develop basic IE solutions to extract a single type of entity (e.g., person names) or relation (e.g., advising). IE developers then commonly combine these solutions — often as off-the-shelf IE "blackboxes" — with additional procedural code into larger IE programs. Since such programs are rather difficult to develop, understand, and debug, recent works have developed compositional or declarative IE frameworks, such as UIMA, GATE [21, 11], Xlog [35], and others [38, 32]. An emerging direction then focuses on building such frameworks and providing end-to-end management of a multitude of IE "blackboxes" in large-scale IE applications [18].

Information integration (II) is also a long-standing problem that has received much attention from the database, AI, and Web communities [23]. Much of early work focused on developing II architectures and languages, identifying key problems, and addressing II in various application domains. A recent trend started developing compositional solutions for certain II problems (e.g., schema matching and mention matching [34, 38]). Another emerging trend focuses on best-effort, approximate II (e.g., [23, 16]) (in addition to precise II commonly studied so far, e.g., in business applications).

Our work here builds on these emerging IE and II trends. We also advocate a compositional, best-effort approach to extracting and integrating data. In contrast to these prior works, however, we consider a tight combination of IE and II, within the context of Web communities.

The rise of the World-Wide Web in the past decade has also added new directions to IE and II research. One direction develops IE and II techniques to turn the whole Web (or a significant portion of it) into a structured database. Recent examples include KnowItAll [19], SemTag [15], and WebFountain [22]. These works have either focused on scaling to Web scale, or relied heavily on the redundancy of the Web, which is not always available in a Web community. Another important direction leverages the Web to assist with IE and II tasks, e.g., by querying search engines to discover relations between certain entities [28, 19]. Such techniques can potentially be applicable to Web community contexts.

Developing Structured Portals: This topic has also received much attention in the AI, database, Semantic Web, Web, and KDD communities. Recent examples include Libra [30], Rexa, DBLife [14], Citeseer, KA2 [4], OntoWeb [36], Flink, BlogScope [3], Strudel [20], and [27]. These works address two main issues: how to acquire structures and how to exploit structures.

In this work, we focus on how to acquire structures. A common solution is to apply IE and II techniques to acquire structures from raw community data pages, as we do here. Many works in the AI/Web communities (e.g., Citeseer, Rexa, Libra) develop such solutions, often using sophisticated learning techniques [9]. Similar works in the Semantic Web community include KA2 [4], SEAL (which is used to build OntoWeb) [37], and [10]. Several works (e.g., [28]) also leverage search engines for these purposes (as described earlier).

Only a few of these works (e.g., [30, 10]) have discussed a *methodology* to build structured portals. Even so, they have not addressed all salient aspects (e.g., matching mentions, matching entities across different days, and expanding portals), as we do here. In addition, as far as we know, none of these works has considered how to exploit common characteristics of Web communities in the portal building process.

Besides extracting and integrating from raw data pages, another major direction to acquiring structures is to manually provide them, either from a small set of builders, or from the multitude of users. Examples include Friend-of-a-Friend (*foaf-project.org*); MathNet, a portal for sharing math research and knowledge; and WebODE [2], a workbench for creating ontology-driven knowledge portals. Some recent works [39] build wiki portals, then allow community users to supply structured data, using an extension of the standard wiki language.

Clearly, the combination of automatically extracting and manually specifying structures can provide a powerful hybrid approach to developing structured portals. In [12] we provide an initial solution in this direction.

Community Information Management: Our work here is done in the context of *community information management (CIM)*. CIM studies how to effectively support information needs of a community by managing both the community data and users in a synergistic fashion, using IE, II, and mass collaboration techniques. We have provided

some initial discussion of CIM and DBLife in [17, 14]. This, however, is the first work to study in depth the problem of building structured portals, an important aspect of CIM.

8. CONCLUSION & FUTURE WORK

In this paper, we have argued that a top-down, compositional, and incremental approach is a good way to build structured Web community portals. We described Cimple, a workbench implementation of this approach, and DBLife, a portal built using this workbench. Our experience with DBLife suggests that our approach can effectively exploit common characteristics of Web communities to build portals quickly and accurately using simple extraction/integration operators, and to evolve them over time efficiently with little human effort.

Our work has only scratched the surface of research problems in this direction. Interesting problems include: How to develop a better compositional framework? How to make such a framework as declarative as possible? What technologies (e.g., XML, relational) should be used to store and manipulate the portal data? How to optimize both run-time and accuracy of data processing in portal construction? How to effectively process a large amount of portal data in a distributed fashion? How can users interact with the portal or mass collaborate to further evolve the portal? How to build effective user services that exploit the (extracted and integrated) structured data? How to apply Semantic Web technologies to capture and reason with community knowledge? And how to build a practical integrated development environment for portal developers? It is our hope that the initial work here will inspire other researchers to join us in addressing these questions.

9. REFERENCES

[1] E. Agichtein and S. Sarawagi. Scalable information extraction and integration (tutorial). In *KDD-06*.

[2] J. C. Arprez, O. Corcho, M. Fernández-López, and A. Gómez-P'erez. WebODE in a nutshell. *AI Magazine*, 24, 2003.

[3] N. Bansal, F. Chiang, N. Koudas, and F. W. Tompa. Seeking stable clusters in the Blogosphere. *VLDB-07*.

[4] V. R. Benjamins, D. Fensel, S. Decker, and A. G. Perez. (KA)2: Building ontologies for the Internet: a mid term report. *IJHCS-99*, 51.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 30:107, 1998.

[6] S. Chakrabarti, K. Punera, and M. Subramanyam. Accelerated focused crawling through online relevance feedback. *WWW-02*.

[7] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. *ICDE-06*.

[8] P.-A. Chirita, D. Olmedilla, and W. Nejdl. Finding related pages using the link structure of the WWW. *Web Intelligence-04*.

[9] W. Cohen and A. McCallum. Information extraction from the World Wide Web (tutorial). In *KDD-03*.

[10] J. Contreras, V. R. Benjamins, M. Blázquez, S. Losada, R. Salla, J. L. Sevillano, J. R. Navarro, J. Casillas, A. Momp, D. Patón, O. Corcho, P. Tena, and I. Martoyo. A semantic portal for the international affairs sector. *EKAW-04*.

[11] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. *ACL-02*.

[12] P. DeRose, X. Chai, B. J. Gao, W. Shen, A. Doan, P. Bohannon, and J. Zhu. Building community wikipedias: A machine-human partnership approach. Technical report, UW-Madison, 2007.

[13] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured Web community portals: A top-down, compositional, and incremental approach. Technical report, UW-Madison, 2007.

[14] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A community information management platform for the database research community [demo]. *CIDR-06*.

[15] S. Dill, N. Eiron, D. Gibson, D. Gruhl, and R. Guha. SemTag and Seeker: Bootstrapping the semantic Web via automated semantic annotation. *WWW-03*.

[16] A. Doan. Best-effort data integration. *NSF Workshiop on Data Integration*, 2006.

[17] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Engineering Bulletin*, 29(1), 2006.

[18] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction: state of the art and research directions (tutorial). In *SIGMOD-06*.

[19] O. Etzioni, M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 165(1), 2005.

[20] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of Web sites with Strudel. *VLDB J.*, 9(1), 2000.

[21] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4), 2004.

[22] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a WebFountain: An architecture for very large-scale text analytics. *IBM Systems Journal*, 43(1), 2004.

[23] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: the teenage years. *VLDB-06*.

[24] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl?: towards a query optimizer for text-centric tasks. *SIGMOD-06*.

[25] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms (tutorial). In *SIGMOD-06*.

[26] A. Kruger, C. L. Giles, F. Coetzee, E. Glover, G. Flake, S. Lawrence, and C. Omlin. DEADLINER: Building a new niche search engine. *CIKM-00*.

[27] R. Lara, S. H. Han, H. Lausen, M. T. Stollberg, Y. Ding, and D. Fensel. An evaluation of Semantic Web portals. *IADIS-04*.

[28] Y. Matsuo, J. Mori, M. Hamasaki, K. Ishida, T. Nishimura, H. Takeda, K. Hasida, and M. Ishizuka. POLYPHONET: an advanced social network extraction system from the web. *WWW-06*.

[29] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. A machine learning approach to building domain-specific search engines. *IJCAI-99*.

[30] Z. Nie, J.-R. Wen, and W.-Y. Ma. Object-level vertical search. *CIDR-07*.

[31] F. Peng and A. McCallum. Accurate information extraction from research papers using conditional random fields. *HLT-NAACL-04*.

[32] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. Technical report, IBM Almaden, 2007.

[33] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11), 1975.

[34] W. Shen, P. DeRose, L. Vu, A. Doan, and R. Ramakrishnan. Source-aware entity matching: A compositional approach. *ICDE-07*.

[35] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. *VLDB-07*.

[36] P. Spyns, D. Oberle, R. Volz, J. Zheng, M. Jarrar, Y. Sure, R. Studer, and R. Meersman. OntoWeb - a Semantic Web community portal. *PAKM-02*.

[37] N. Stojanovic, A. Maedche, S. Staab, R. Studer, and Y. Sure. SEAL: a framework for developing SEmantic PortALs. *K-CAP-01*.

[38] A. Thor and E. Rahm. MOMA - a mapping-based object matching system. *CIDR-07*.

[39] M. Völkel, M. Krötzsch, D. Vrandecic, H. Haller, and R. Studer. Semantic Wikipedia. *WWW-06*.

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