

# WebTables: Exploring the Power of Tables on the Web

Michael J. Cafarella<sup>\*</sup>  
University of Washington  
Seattle, WA 98107, USA  
mjc@cs.washington.edu

Alon Halevy  
Google, Inc.  
Mountain View, CA 94043, USA  
halevy@google.com

Daisy Zhe Wang  
UC Berkeley  
Berkeley, CA 94720, USA  
daisyw@cs.berkeley.edu

Eugene Wu  
MIT  
Cambridge, MA 02139, USA  
eugenewu@mit.edu

Yang Zhang  
MIT  
Cambridge, MA 02139, USA  
yaaang@gmail.com

## ABSTRACT

The World-Wide Web consists of a huge number of unstructured documents, but it also contains structured data in the form of HTML tables. We extracted 14.1 billion HTML tables from Google’s general-purpose web crawl, and used statistical classification techniques to find the estimated 154M that contain high-quality relational data. Because each relational table has its own “schema” of labeled and typed columns, each such table can be considered a small structured database. The resulting corpus of databases is larger than any other corpus we are aware of, by at least five orders of magnitude.

We describe the WEBTABLES system to explore two fundamental questions about this collection of databases. First, what are effective techniques for searching for structured data at search-engine scales? Second, what additional power can be derived by analyzing such a huge corpus?

First, we develop new techniques for keyword search over a corpus of tables, and show that they can achieve substantially higher relevance than solutions based on a traditional search engine. Second, we introduce a new object derived from the database corpus: the *attribute correlation statistics database* (AcsDB) that records corpus-wide statistics on co-occurrences of schema elements. In addition to improving search relevance, the AcsDB makes possible several novel applications: *schema auto-complete*, which helps a database designer to choose schema elements; *attribute synonym finding*, which automatically computes attribute synonym pairs for schema matching; and *join-graph traversal*, which allows a user to navigate between extracted schemas using automatically-generated join links.

---

<sup>\*</sup>Work done while all authors were at Google, Inc.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB ’08 Auckland, New Zealand

Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

## Categories and Subject Descriptors

H.3 [Information Storage and Retrieval]: Online Information Services; H.2 [Database Management]: Miscellaneous

## 1. INTRODUCTION

The Web is traditionally modelled as a corpus of unstructured *documents*. Some structure is imposed by hierarchical URL names and the hyperlink graph, but the basic unit for reading or processing is the unstructured document itself. However, Web documents often contain large amounts of relational data. For example, the Web page shown in Figure 1 contains a table that lists American presidents<sup>1</sup>. The table has four columns, each with a domain-specific label and type (*e.g.*, **President** is a person name, **Term as President** is a date range, etc) and there is a tuple of data for each row. This Web page essentially contains a small relational database, even if it lacks the explicit metadata traditionally associated with a database.

We extracted approximately 14.1 billion raw HTML tables from the English documents in Google’s main index, and used a series of techniques to recover those tables that are high-quality relations [5]. Recovery involves filtering out tables that are used for page layout or other non-relational reasons, and detecting labels for attribute columns. We estimate that the tested portion of our general web crawl contains 154M distinct relational databases - a huge number, even though it is just slightly more than 1.1% of raw HTML tables.

Previous work on HTML tables focused on the problem of recognizing good tables or extracting additional information from individual tables [26, 29, 30]. In this paper we consider a corpus of tables that is five orders of magnitude larger than the largest one considered to date [26], and address two fundamental questions: (1) what are effective methods for searching within such a collection of tables, and (2) is there additional power that can be derived by analyzing such a huge corpus? We describe the WEBTABLES system that explores these questions.

The main motivation for searching such a corpus of tables is to enable analysis and integration of data on the

---

<sup>1</sup><http://www.enchantedlearning.com/history/us/pres/list.shtml>

President	Party	Term as President	Vice President
1. George Washington (1732-1799)	None, Federalist	1789-1797	John Adams
2. John Adams (1735-1826)	Federalist	1797-1801	Thomas Jefferson
3. Thomas Jefferson (1743-1826)	Democratic-Republican	1801-1809	James Burr, George Clinton
4. James Madison (1751-1836)	Democratic-Republican	1809-1817	George Clinton, Elbridge Gerry
5. James Monroe (1758-1831)	Democratic-Republican	1817-1825	Elbridge Gerry
6. John Quincy Adams (1767-1848)	Democratic-Republican	1825-1829	John Calhoun
7. Andrew Jackson (1767-1845)	Democrat	1829-1837	John Calhoun, Martin van Buren
8. Martin van Buren (1767-1862)	Democrat	1837-1841	Richard Johnson
9. William H. Harrison (1773-1841)	Whig	1841	John Tyler
10. John Tyler (1790-1862)	Whig	1841-1845	
11. James K. Polk (1795-1846)	Democrat	1845-1849	George Dallas
12. Zachary Taylor (1784-1850)	Whig	1849-1850	Millard Fillmore
13. Millard Fillmore (1800-1874)	Whig	1850-1853	
14. Frank Pierce (1804-1889)	Democrat	1853-1857	William King
15. James Buchanan (1791-1868)	Democrat	1857-1861	John Breckinridge

Figure 1: A typical use of the table tag to describe relational data. The relation here has a schema that is never explicitly declared but is obvious to a human observer, consisting of several typed and labeled columns. The navigation bars at the top of the page are also implemented using the table tag, but clearly do not contain relational-style data. The automatically-chosen WEBTABLES corpus consists of 41% true relations, and contains 81% of the true relations in our crawl (as seen in Table 1). (The raw HTML table corpus consists of 1.1% true relations.)

Web. In particular, there is a recent flurry of tools for visualizing structured data and creating mashups on the Web (e.g., many-eyes.com swivel.com, Yahoo Pipes, Microsoft Popfly). Users of such tools often search the Web for good tabular data in a variety of domains. As further evidence of user demand for structured data, a scan over a randomly-chosen 1-day log of Google’s queries revealed that for close to 30 million queries, users clicked on results that contained tables from our filtered relational corpus. This is a very large absolute number of queries, even if the fraction of overall web queries is small (for commercial reasons, we cannot release the actual fraction).

Document search engines are commonplace, and researchers have studied the problem of keyword ranking for individual tuples within a database [2, 14]. However, to perform **relation ranking**, i.e., to sort relations by relevance in response to a user’s keyword search query, WEBTABLES must solve the new problem of ranking millions of individual databases, each with a separate schema and set of tuples. Relation ranking poses a number of difficulties beyond web document ranking: relations contain a mixture of “structural” and related “content” elements with no analogue in unstructured text; relations lack the incoming hyperlink anchor text that helps traditional search; and PageRank-style metrics for page quality cannot distinguish between tables of widely-varying quality found on the same web page. Finally, relations contain text in two dimensions and so many cannot be efficiently queried using the standard inverted index.

We describe a ranking method that combines table-structure-aware features (made possible by the index) with a novel query-independent table coherency score that makes use of corpus-wide schema statistics. We show that this approach gives an 85-98% improvement in search quality over a naïve approach based on traditional search engines.

To validate the power of WEBTABLES’s corpus, we de-

scribe the *attribute correlation statistics database*, (**ACSDb**), which is a set of statistics about schemas in the corpus. In addition to improving WEBTABLES’s ranking, we show that we can leverage the **ACSDb** to offer unique solutions to schema-level tasks. First, we describe an algorithm that uses the **ACSDb** to provide a *schema auto-complete* tool to help database designers choose a schema. For example, if the designer inputs the attribute **stock-symbol**, the schema auto-complete tool will suggest **company**, **rank**, and **sales** as additional attributes. Unlike set-completion (e.g., Google Sets) that has been investigated in the past, schema auto-complete looks for attributes that tend to appear in the same schema (i.e., horizontal completion).

Second, we use the **ACSDb** to develop an *attribute synonym finding* tool that automatically computes pairs of schema attributes that appear to be used synonymously. Synonym finding has been considered in the past for text documents [15], but finding synonyms among database attributes comprises a number of novel problems. First, databases use many attribute labels that are nonexistent or exceedingly rare in natural language, such as abbreviations (e.g., **hr** for **home run**) or non-alphabetic sequences (e.g., **tel-#**); we cannot expect to find these attributes in either thesauri or natural text. Second, the context in which an attribute appears strongly affects its meaning; for example, *name* and *filename* are synonymous, but only when *name* is in the presence of other file-related attributes. If *name* is used in the setting of an address book, it means something quite different. Indeed, two instances of *name* will only be synonymous if their co-attributes come from the same domain. We give an algorithm that automatically detects synonymy with very high accuracy. For example, our synonym-finder takes an input domain and gives an average of four correct synonym pairs in its first five emitted pairs, substantially improving on performance by Lin and Pantel’s linguistic-synonym DIRT system [15].

Finally, we show how to use the **ACSDb** for *join-graph traversal*. This tool can be used to build a “schema explorer” of the massive WEBTABLES corpus that would again be useful for database designers. The user should be able to navigate from schema to schema using relational-style join links (as opposed to standard hypertext links that connected related documents).

Our extracted tables lack explicit join information, but we can create an approximation by connecting all schemas that share a common attribute label. Unfortunately, the resulting graph is hugely “busy”; a single schema with just two or three attributes can link to thousands of other schemas. Thus, our set of schemas is either completely disconnected (in its original state) or overly-connected (if we synthesize links between attribute-sharing schemas). It would be more useful to have a graph with a modest number of meaningful links. To address this problem, we introduce an **ACSDb**-based method that clusters together related schema neighbors.

A distinguishing feature of the **ACSDb** is that it is the first time anyone has compiled such large amounts of statistical data about relational schema usage. Thus we can take data-intensive approaches to all of the above-listed applications, similar in spirit to recent efforts on machine translation [4] and spell-correction that leverage huge amounts of data. We note that the idea of making use of a large number of schemas was initially proposed in [16] for the improving

schema matching. Our work is distinguished in that we consider a corpus that is several orders of magnitude larger, and we leverage the corpus more broadly. (Our synonym finder can be used for schema matching, but we do not explore that here.)

Before we proceed, we distinguish between the data we manage with WEBTABLES and the deep web. The WEBTABLES system considers HTML tables that are already surfaced and crawlable. The deep web refers to content that is made available through filling HTML forms. The two sets of data intersect, but neither contains the other. There are many HTML tables that are not behind forms (only about 40% of the URLs in our corpus are parameterized), and while some deep-web data is crawlable, the vast majority of it is not (or at least requires special techniques, such as those described in [13]). (Thus, we believe at least 60% of our data comes from non-deep-web sources.) In contrast to the work we describe in this paper, deep web research questions focus on identifying high quality forms and automatically figuring out how to query them in a semantically meaningful fashion. In addition to HTML tables and the deep web, there are many kinds of structure on the Web [18]. In this paper we will only consider the `table` tag.

This paper focuses on the extracted table corpus, how to provide search-engine-style access to this huge volume of structured data, and on the ACSDB and its applications. We do not study how to match or integrate the table data, though we have done so elsewhere [5].

The remainder of this paper is organized as follows. Section 2 describe our basic model and the ACSDB. In Section 3, we describe how to rank tables in response to keyword query on WEBTABLES. Section 4 covers our three novel ACSDB applications: *schema auto-complete*, *attribute synonym finding*, and *join-graph discovery*. We present experimental evaluations in Section 5, and conclude with discussions of related and future work (Sections 6 and 7).

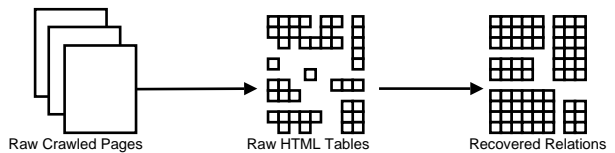
## 2. DATA MODEL

We begin by describing the set of relations we extract from the web crawl, and the statistics we store in the *attribute correlation statistics database* (ACSDB).

### 2.1 Extracted Relations

The corpus of relational databases that is contained within the set of raw HTML tables is hugely valuable, containing data drawn from millions of sites and across a vast array of topics. Unfortunately, most HTML tables are not used for relational data (but instead are used for, *e.g.*, page layout). There is no straightforward technical test (*e.g.*, constraint-checking) we can apply to an HTML table to determine whether it contains relational data - the question is one of human judgment. However, there are a number of machine-detectable features (*e.g.*, columns that have a consistent type, relatively few empty cells) that usually accompany a relational table.

We wrote a system, described in [5], that combines hand-written detectors and statistically-trained classifiers to filter *relational* tables from *non-relational* ones. We generated training and test data for our relational-table detection system by asking two independent judges to mark a series of HTML tables on a 1-5 “relational quality” scale; any table that received an average score of 4 or above was deemed *relational*, and a table with a lower score was deemed *non-*



**Figure 2:** The WEBTABLES relation extraction pipeline. About 1.1% of the raw HTML tables are true relations.

Relational Filtering		
true class	Precision	Recall
relational	0.41	0.81
non-relational	0.98	0.87
Metadata Detection		
true class	Precision	Recall
has-metadata	0.89	0.85
no-metadata	0.75	0.80

**Table 1:** A summary of the WEBTABLES relation extractor’s performance.

*relational*. We also built a separate trained detector that runs on all tables that have been classified as *relational* and extracts any embedded metadata (such as the first row in the table in Figure 1). The extraction pipeline is pictured in Figure 2.

Table 1 summarizes the performance of our relation-filter and metadata-detector, which yields more than 125M high-quality relations from our original web crawl. We tuned the relation-filter for relatively high recall and low precision, so that we lose relatively few true relations. We rely on downstream applications to gracefully handle the non-relational tables admitted by the filter<sup>2</sup>. The metadata-detector is tuned to equally weigh recall and precision. Except for the specifically-tuned true-relation precision, performance of our system is roughly similar to other domain-independent information extraction systems, such as KNOWITALL and SNOWBALL [11, 1].

As described in [5], the actual extraction process is quite involved, and is not germane to our current discussion. Here we simply assume we have a corpus,  $\mathcal{R}$ , of databases, where each database is a single relation. For each relation,  $R \in \mathcal{R}$ , we have the following:

- the url  $R_u$  and offset  $R_i$  within the page from which  $R$  was extracted.  $R_u$  and  $R_i$  uniquely define  $R$ .
- the schema,  $R_S$ , which is an ordered list of attribute labels. For example, the table in Figure 1 has the attributes  $R_S = [\text{President}, \text{Party}, \dots]$ . One or more

<sup>2</sup>The relational ranker should give low-quality relations a poor rank as a natural by-product of keyword-relevance ranking. For applications driven by attribute statistics, incorrectly-admitted relations do not have an organized “schema” of attribute labels and should not bias the statistics in a coherent way, even if the metadata-detector incorrectly recognizes the schema. Despite a flawed relational filter, a hand-examination of the 300 most-common metadata strings shows that all are reasonable attribute labels.

elements of  $R_S$  may be empty strings (*e.g.*, if the table’s schema cannot be recovered).

- a list of tuples,  $R_T$ . A tuple  $t$  is a list of data strings. The size of a tuple  $t$  is always  $|R_S|$ , though one or more elements of  $t$  may be empty strings.

In Section 3 we describe how to find relevant tables in such a huge corpus.

## 2.2 Attribute Correlation Statistics

The sheer size of our corpus also enables us to compute the first large-scale statistical analysis of how attribute names are used in schemas, and to leverage these statistics in various ways.

For each unique schema  $R_S$ , the **ACSDb** contains a frequency count that indicates how many relations express that schema. We assume two schemas are identical if they contain same attributes (regardless of their order of appearance in the source page). In other words, the **ACSDb**  $\mathcal{A}$  is a set of pairs of the form  $(R_S, c)$ , where  $R_S$  is a schema of a relation in  $\mathcal{R}$ , and  $c$  is the number of relations in  $\mathcal{R}$  that have the schema  $R_S$ .

Extracting the **ACSDb** given the corpus  $\mathcal{R}$  of extracted relations is straightforward, as described below. Note that if a schema appears multiple times under URLs with a single domain name, we only count the schema once; we thus prevent a single site with many similar pages from swamping the schema statistics. (We also perform some small data-cleaning steps, such as canonicalizing punctuation and stripping out obvious non-attribute strings, such as URLs.)

**Function** createACS( $\mathcal{R}$ ):

```

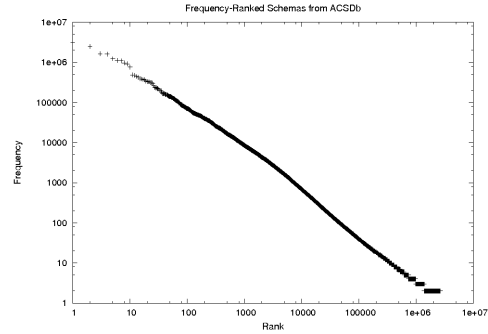
 $\mathcal{A} = \{\}$ 
 $seenDomains = \{\}$ 
for all  $R \in \mathcal{R}$  do
  if  $getDomain(R.u) \notin seenDomains[R.S]$  then
     $seenDomains[R.S].add(getDomain(R.u))$ 
     $\mathcal{A}[R.S] = \mathcal{A}[R.S] + 1$ 
  end if
end for

```

After removing all attributes and all schemas that appear only once in the entire extracted relation corpus, we computed an **ACSDb** with 5.4M unique attribute names and 2.6M unique schemas. Unsurprisingly, a relatively small number of schemas appear very frequently, while most schemas are rare (see the distribution of schemas in Figure 3).

The **ACSDb** is simple, but it critically allows us to compute the probability of seeing various attributes in a schema. For example,  $p(address)$  is simply the sum of all counts  $c$  for pairs whose schema contains *address*, divided by the total sum of all counts. We can also detect relationships between attribute names by conditioning an attribute’s probability on the presence of a second attribute. For example, we can compute  $p(address|name)$  by counting all the schemas in which “address” appears along with “name” (and normalizing by the counts for seeing “name” alone). As we will see in Sections 3.1 and 4, we can use these simple probabilities to build several new and useful schema applications.

We next describe the **WEBTABLES** relation search system, which uses features derived from both the extracted relations and from the **ACSDb**. Afterwards, in Section 4, we will discuss **ACSDb** applications that are more broadly applicable to traditional database tasks. Indeed, we believe



**Figure 3:** Distribution of frequency-ordered unique schemas in the **ACSDb**, with rank-order on the x-axis, and schema frequency on the y-axis. Both rank and frequency axes have a log scale.

```

1: Function naiveRank( $q, k$ ):
2: let  $\mathcal{U} =$  urls from web search for query  $q$ 
3: for  $i = 0$  to  $k$  do
4:   emit  $getRelations(\mathcal{U}[i])$ 
5: end for

```

**Figure 5:** **Function** naïveRank: it simply uses the top  $k$  search engine result pages to generate relations. If there are no relations in the top  $k$  search results, naïveRank will emit no relations.

the **ACSDb** will find many uses beyond those described in this paper.

## 3. RELATION SEARCH

Even the largest corpus is useless if we cannot query it. The **WEBTABLES** search engine allows users to rank relations by relevance, with a search-engine-style keyword query as input. Figure 9 shows the **WEBTABLES** search system architecture, with the index of tables split across multiple back-end servers.

As with a web document search engine, **WEBTABLES** generates a list of results (which is usually much longer than the user wants to examine). Unlike most search engines, **WEBTABLES** results pages are actually useful on their own, even if the user does not navigate away. Figure 4 shows a sample results page for the query “city population.” The structured nature of the results allows us to offer search services beyond those in a standard search engine.

For example, we can create query-appropriate visualizations by testing whether the tuples  $R.T$  contain a column of geographic placenames. If so, **WEBTABLES** will place all of each tuple’s data at the correct locations on the map (see, *e.g.*, the “Paris” tuple in Figure 4). If two columns of  $R.T$  contain interesting numerical data, **WEBTABLES** will suggest a scatterplot visualization to show the relationship between the two variables. The user can also manually choose a visualization. Finally, **WEBTABLES** search offers traditional structured operations over search results, such as selection and projection.

Of course, none of these extensions to the traditional search application will be useful without good search relevance. In the section below we present different algorithms for ranking individual databases in relation to a user’s query. Unfortu-

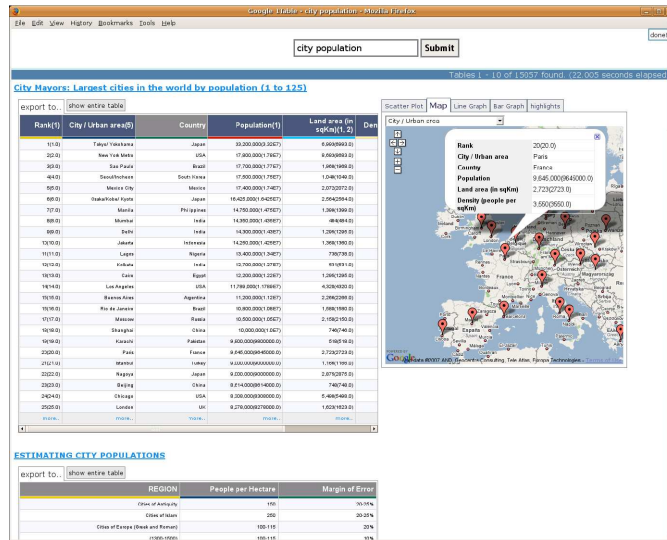


Figure 4: Results of a WEBTABLES keyword query for “city population”, showing a ranked list of databases. The top result contains a row for each of the most populous 125 cities, and columns for “City/Urban Area,” “Country,” “Population,” “rank” (the city’s rank by population among all the cities in the world), etc. WEBTABLES automatically extracted the data from an HTML table and applied table ranking and formatting (such as the schema-row highlight). It also computed the visualization to the right, which shows the result of clicking on the “Paris” row. The title (“City Mayors...”) links to the page where the original HTML table was found.

```

1: Function filterRank(q, k):
2: let U = ranked urls from web search for query q
3: let numEmitted = 0
4: for all u in U do
5:   for all r in getRelations(u) do
6:     if numEmitted >= k then
7:       return
8:     end if
9:     emit r; numEmitted ++
10: end for
11: end for

```

Figure 6: Function filterRank: similar to naïveRank, it will go as far down the search engine result pages as necessary to find  $k$  relations.

nately, the traditional inverted text index cannot support these algorithms efficiently, so in Section 3.2 we also describe additional index-level support that WEBTABLES requires.

### 3.1 Ranking

Keyword ranking for documents is well-known and understood, and there has been substantial published work on keyword access to traditional relational databases. But keyword ranking of individual databases is a novel problem, largely because no one has previously obtained a corpus of databases large enough to require search ranking.

Ranking for web-extracted relations poses a unique set of challenges: relations do not exist in a domain-specific schema graph, as with relational keyword-access systems (e.g., DBXplorer[2], DISCOVER [14]), page-level features like word frequencies apply ambiguously to tables embedded in the page (e.g., it is not clear which table in the page

```

1: Function featureRank(q, k):
2: let R = set of all relations extracted from corpus
3: let score(r in R) = combination of per-relation features in Table 2
4: sort r in R by score(r)
5: for i = 0 to k do
6:   emit R[i]
7: end for

```

Figure 7: Function featureRank: score each relation according to the features in Table 2. Rank by that score and return the top  $k$  relations.

is described by a frequent-word; further, attribute labels for a relational table are extremely important, even if they appear infrequently), and even a high-quality page may contain tables of varying quality (e.g., a table used for interface element layout). Relations also have special features that may reveal their subject matter: schema elements should provide good summaries of the subject matter, tuples may have a key-like element that summarizes the row, and we may be able to discern relation quality by looking at the relation size and the distribution of NULLs.

To rank our extracted WEBTABLES relations, we created a series of ranking functions of increasing complexity, listed in Figures 5, 6, 7, and 8. Each of these functions accept as input a query  $q$  and a top- $k$  parameter  $k$ . Each invokes the emit function to return a relation to the user.

The first, naïveRank, simply sends the user’s query to a search engine and fetches the top- $k$  pages. It returns extracted relations in the URL order returned by the search engine. If there is more than one relation extracted per page, we return it in document-order. If there are fewer than  $k$  ex-

```

1: Function cohere( $R$ ):
2:  $totalPMI = 0$ 
3: for all  $a \in attrs(R), b \in attrs(R), a \neq b$  do
4:    $totalPMI = PMI(a, b)$ 
5: end for
6: return  $totalPMI / (|R| * (|R| - 1))$ 

```

---

```

1: Function pmi( $a, b$ ):
2: return  $\log(\frac{p(a, b)}{p(a) * p(b)})$ 

```

Figure 8: The *coherency score* measures how well attributes of a schema fit together. Probabilities for individual attributes are derived using statistics in the ACSDB.

tracted relations in these pages, **naïveRank** will not go any deeper into the result list. Although very basic, **naïveRank** roughly simulates what a modern search engine user must do when searching for structured data. As we will see in the experimental results in Section 5.1, using this algorithm to return search results is not very satisfactory.

Algorithm **filterRank** is similar to **naïveRank**, but slightly more sophisticated. It will march down the search engine results until it finds  $k$  relations to return. The ordering is the same as with **naïveRank**. Because search engines may return many high-ranking pages that contain no relational data at all, even this basic algorithm can be a large help to someone performing a relation search.

Figure 7 shows **featureRank**, the first algorithm that does not rely on an existing search engine. It uses the relation-specific features listed in Table 2 to score each extracted relation in our corpus. It sorts by this score and returns the top- $k$  results.

We numerically combined the different feature scores using a linear regression estimator trained on more than a thousand ( $q, relation$ ) pairs, each scored by two human judges. Each judge gave a pair a quality score between 1 and 5. The features from Table 2 include both query-independent and query-dependent elements that we imagined might describe a relevant relation. The two most heavily-weighted features for the estimator are the number of hits in each relation’s schema, and the number of hits in each relation’s leftmost column. The former fits our intuition that attribute labels are a strong indicator of a relation’s subject matter. The latter seems to indicate that values in the leftmost column may act something like a “semantic key,” providing a useful summary of the contents of a data row.

The final algorithm, **schemaRank**, is the same as **featureRank**, except that it also includes the ACSDB-based *schema coherency score*, which we now describe. Intuitively, a *coherent schema* is one where the attributes are all tightly related to one another in the ACSDB schema corpus. For example, a schema that consists of the attributes “make” and “model” should be considered highly coherent, and “make” and “zipcode” much less so. The coherency score is defined formally in Figure 8.

The core of the coherency score is a measure called Pointwise Mutual Information (or PMI), which is often used in computational linguistics and web text research, and is designed to give a sense of how strongly two items are related [8, 11, 25]. PMI will be large and positive when two

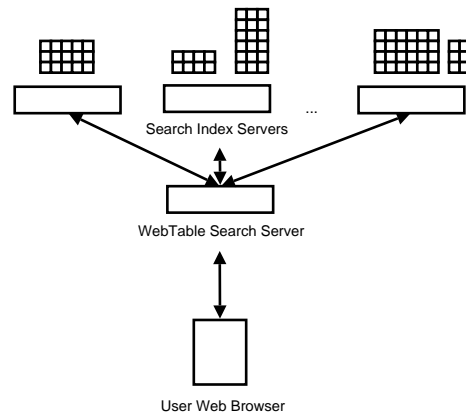


Figure 9: The WEBTABLES search system. The inverted table index is segmented by term and divided among a pool of search index servers. A single front-end search server accepts the user’s request, transmits it to all of the index servers, and returns a reply.

variables strongly indicate each other, zero when two variables are completely independent, and negative when variables are negatively-correlated.  $pmi(a, b)$  requires values for  $p(a)$ ,  $p(b)$ , and  $p(a, b)$ , which in linguistics research are usually derived from a text corpus. We derive them using the ACSDB corpus.

The coherency score for a schema  $s$  is the average of all possible attribute-pairwise PMI scores for the schema. By taking an average across all the PMI scores, we hope to reward schemas that have highly-correlated attributes, while not overly-penalizing relations with a single “bad” one.

We will see in Section 5.1 that **schemaRank** performs the best of our search algorithms.

To complete our discussion, we now describe the systems-level support necessary to implement the above algorithms. Unfortunately, the traditional inverted index cannot support operations that are very useful for relation ranking.

### 3.2 Indexing

Traditional search engines use a simple inverted index to speed up lookups, but the standard index cannot efficiently retrieve all the features listed in Table 2.

Briefly, the inverted index is a structure that maps each term to a sorted *posting list* of (docid, offset) pairs that describe each occurrence of the term in the corpus. When the search engine needs to test whether two search terms are in the same document, it simply steps through the terms’ inverted posting lists in parallel, testing to see where they share a docid. To test whether two words are adjacent, the search engine also checks if the words have postings at adjacent offset values. The offset value may also be useful in ranking: for example, words that appear near the top of a page may be considered more relevant.

Unlike the “linear text” model that a single offset value implies, WEBTABLES data exists in two dimensions, and the ranking function uses both the horizontal and vertical offsets to compute the input scoring features. Thus, we adorn each element in the posting list with a two-dimensional ( $x, y$ )

# rows
# cols
has-header?
# of NULLs in table
document-search rank of source page
# hits on header
# hits on leftmost column
# hits on second-to-leftmost column
# hits on table body

Table 2: Selected text-derived features used in the search ranker.

offset that describes where in the table the search term can be found. Using this offset WEBTABLES can compute, for example, whether a single posting is in the leftmost column, or the top row, or both.

Interestingly, the user-exposed search query language can also take advantage of this new index style. WEBTABLES users can issue queries that include various spatial operators like `samecol` and `samerow`, which will only return results if the search terms appear in cells in the same column or row of the table. For example, a user can search for all tables that include **Paris** and **France** on the same row, or for tables with **Paris**, **London**, and **Madrid** in the same column.

#### 4. ACSDB APPLICATIONS

The ACSDB is a unique dataset that enables several novel pieces of database software, applicable beyond the recovered relations themselves. In this section we describe three separate problems, and present an ACSDB-based solution for each. First, we show how to perform *schema autocomplete*, in which WEBTABLES suggests schema elements to a database designer. *Synonym discovery* is useful for providing synonyms to a schema matching system; these synonyms are more complete than a natural-language thesaurus would be, and are far less expensive to generate than human-generated domain-specific synonym sets. Finally, we introduce a system for *join-graph traversal* that enables users to effectively browse the massive number of schemas extracted by the WEBTABLES system.

All of our techniques rely on attribute and schema probabilities derived from the ACSDB. Similar corpus-based techniques have been used successfully in natural language processing and information extraction [4, 11, 19]. However, we are not aware of any similar technique applied to the structured-data realm, possibly because no previous database corpus has been large enough.

##### 4.1 Schema Auto-Complete

Inspired by the word and URL auto-complete features common in word-processors and web browsers, the **schema auto-complete** application is designed to assist novice database designers when creating a relational schema. We focus on schemas consisting of a single relation. The user enters one or more domain-specific attributes, and the schema auto-completer guesses the rest of the attribute labels, which should be appropriate to the target domain. The user may accept all, some, or none of the auto-completer’s suggested attributes.

```

1: Function SchemaSuggest( $I, t$ ):
2:  $S = I$ 
3: while  $p(S - I|I) > t$  do
4:    $a = \max_{a \in A - S} p(a, S - I|I)$ 
5:    $S = S \cup a$ 
6: return  $S$ 
7: end while

```

Figure 10: The SchemaSuggest algorithm repeatedly adds elements to  $S$  from the overall attribute set  $A$ . We compute attribute probabilities  $p$  by examining counts in the ACSDB (perhaps conditioning on another schema attribute). The threshold  $t$  controls how aggressively the algorithm will suggest additional schema elements; we set  $t$  to be 0.01 for our experiments.

For example, when the designer enters `make`, the system suggests `model`, `year`, `price`, `mileage`, and `color`. Table 3 shows ten example input attributes, followed by the output schemas given by the auto-completer. Note that this problem is quite different from that described by Nandi and Jagadish [22], who described an end-user autocomplete-style interface for querying an existing database. In contrast, our system is for schema design (not query processing). Accordingly, our system has no input database to use for ranking guidance, and has no reason to perform query operations such as selection, projection, etc.

We can say that for an input  $I$ , the best schema  $S$  of a given size is the one that maximizes  $p(S - I|I)$ . The probability of one set of attributes given another set can be easily computed by counting attribute cooccurrences in the ACSDB schemas.

It is possible to find a schema using a greedy algorithm that always chooses the next-most-probable attribute, stopping when the overall schema’s probability goes below a threshold value. (See Figure 10 for the formal algorithm.) This approach is not guaranteed to find the maximal schema, but it does offer good interaction qualities; when proposing additional attributes, the system never “retracts” previous attribute suggestions that the user has accepted.

The greedy approach is weakest when dealing with attributes that occupy two or more strongly-separated domains. For example, consider the “name” attribute, which appears in so many distinct domains (*e.g.*, address books, file listings, sports rosters) that even the most-probable response is not likely to be useful to the end-user, who may not know her target schema but certainly has a subject area in mind. In such situations, it might be better to present several thematic options to the user, as we do in “join graph traversal” described below in Section 4.3.

##### 4.2 Attribute Synonym-Finding

An important part of schema matching is finding synonymous column labels. Traditionally, schema-matchers have used a synonym-set from a hand-made thesaurus [17, 24]. These thesauri are often either burdensome to compile or contain only natural-language strings (excluding, say, `tel-#` or `num-employees`). The ACSDB allows us to automatically find synonyms between arbitrary attribute strings.

The synonym-finder takes a set of *context attributes*,  $C$ , as input. It must then compute a list of attribute pairs

Input attribute	Auto-completer output
name	name, size, last-modified, type
instructor	instructor, time, title, days, room, course
elected	elected, party, district, incumbent, status, opponent, description
ab	ab, h, r, bb, so, rbi, avg, lob, hr, pos, batters
stock-symbol	stock-symbol, securities, pct-of-portfolio, num-of-shares, mkt-value-of-securities, ratings
company	company, location, date, job-summary, miles
director	director, title, year, country
album	album, artist, title, file, size, length, date/time, year, comment
sqft	sqft, price, baths, beds, year, type, lot-sqft, days-on-market, stories
goals	goals, assists, points, player, team, gp

Table 3: Ten input attributes, each with the schema generated by the WEBTABLES auto-completer.

Input context	Synonym-finder outputs
name	e-mail email, phone telephone, e-mail address email address, date last-modified
instructor	course-title title, day days, course course-#, course-name course-title
elected	candidate name, presiding-officer speaker
ab	k so, h hits, avg ba, name player
stock-symbol	company company-name, company-name securities, company option-price
company	phone telephone, job-summary job-title, date posted
director	film title, shares-for shares-voted-for, shares-for shared-voted-in-favor
album	song title, song track, file song, single song, song-title title
sqft	bath baths, list list-price, bed beds, price rent
goals	name player, games gp, points pts, club team, player player-name

Table 4: Partial result sets from the WEBTABLES synonym-finder, using the same attributes as in Table 3.

```

1: Function SynFind( $C, t$ ):
2:  $R = []$ 
3:  $A =$  all attributes that appear in ACSDb with  $C$ 
4: for  $a \in A, b \in B$ , s.t.  $a \neq b$  do
5:   if  $(a, b) \notin \text{ACSDb}$  then
6:     // Score candidate pair with syn function
7:     if  $\text{syn}(a, b) > t$  then
8:        $R.\text{append}(a, b)$ 
9:     end if
10:  end if
11: end for
12: sort  $R$  in descending syn order
13: return  $R$ 

```

Figure 11: The SynFind algorithm finds all potential synonym pairs that have occurred with  $C$  in the ACSDb and have not occurred with each other, then scores them according to the *syn* function

$P$  that are likely to be synonymous in schemas that contain  $C$ . For example, in the context of attributes `album`, `artist`, the ACSDb synonym-finder outputs `song/track`. Of course, our schemas do not include constraints nor any kind of relationship between attributes other than simple schema-co-membership.

Our algorithm is based on a few basic observations: first, that synonymous attributes  $a$  and  $b$  will never appear together in the same schema, as it would be useless to duplicate identical columns in a single relation (*i.e.*, it must be true that the ACSDb-computed probability  $p(a, b) = 0$ ).

Second, that the odds of synonymity are higher if  $p(a, b) = 0$  despite a large value for  $p(a)p(b)$ . Finally, we observe that two synonyms will appear in similar contexts: that is, for  $a$  and  $b$  and a third attribute  $z \notin C$ ,  $p(z|a, C) \cong p(z|b, C)$ .

We can use these observations to describe a *syn* score for attributes  $a, b \in A$ , with context attributes  $C$ :

$$\text{syn}(a, b) = \frac{p(a)p(b)}{\epsilon + \sum_{z \in A} (p(z|a, C) - p(z|b, C))^2}$$

The value of  $\text{syn}(a, b)$  will naturally be higher as the numerator probabilities go up and there is a greater “surprise” with  $p(a, b) = 0$  at the same time that  $p(a)p(b)$  is large. Similarly, the value of  $\text{syn}(a, b)$  will be high when the attributes frequently appear in similar contexts and thereby drive the denominator lower.

Our SynFind algorithm (see Figure 11) takes a context  $C$  as input, ranks all possible synonym pairs according to the above formula, and returns pairs with score higher than a threshold  $t$ . Table 4 lists synonyms found by WEBTABLES for a number of input contexts.

### 4.3 Join Graph Traversal

The library of schemas extracted by WEBTABLES should be very helpful to a schema designer looking for advice or examples of previous work. Unfortunately, there is no explicit join relationship information in the schemas we extract, so WEBTABLES must somehow create it artificially. The goal is not to “reproduce” what each schema’s designers may have intended, but rather to provide a useful way of navigating this huge graph of 2.6M unique schemas. Navigating the schemas by join relationship would be a good way of describ-



```

1: Function ConstructJoinGraph( $\mathcal{A}, \mathcal{F}$ ):
2:  $\mathcal{N} = \{\}$ 
3:  $\mathcal{L} = \{\}$ 
4: for  $(\mathcal{S}, c) \in \mathcal{A}$  do
5:    $\mathcal{N}.add(\mathcal{S})$ 
6: end for
7: for  $\mathcal{S}, c \in \mathcal{A}$  do
8:   for  $attr \in \mathcal{F}$  do
9:     if  $attr \in \mathcal{S}$  then
10:       $\mathcal{L}.add((attr, \mathcal{F}, \mathcal{S}))$ 
11:     end if
12:   end for
13: end for
14: return  $\mathcal{N}, \mathcal{L}$ 

```

**Figure 12:** ConstructJoinGraph creates a graph of nodes ( $\mathcal{N}$ ) and links ( $\mathcal{L}$ ) that connect any two schemas with shared attributes. We only materialize the locally-viewable portion, from a focal schema  $\mathcal{F}$ ; this is sufficient to allow the user access to any of its neighbors. The function also takes the ACSDB  $\mathcal{A}$  as input.

ing relationships between domains and is a well-understood browsing mode, thanks to web hypertext.

We construct the basic join graph  $\mathcal{N}, \mathcal{L}$  by creating a node for each unique schema, and an undirected join link between any two schemas that share a label. Thus, every schema that contains `name` is linked to every other schema that contains `name`. We describe the basic join graph construction formally in Figure 12. We never materialize the full join graph at once, but only the locally-viewable portion at a focal schema  $F$ .

A single attribute generally links to many schemas that are very similar. For example, `size` occurs in many filesystem-centric schemas: `[description, name, size]`, `[description, offset, size]`, and `[date, file-name, size]`. But `size` also occurs in schemas about personal health (`[height, size, weight]`) and commerce (`[price, quantity, size]`). If we could cluster together similar schema neighbors, we could dramatically reduce the “join graph clutter” that the user must face.

We can do so by creating a measure for *join neighbor similarity*. The function attempts to measure whether a shared attribute  $D$  plays a similar role in its schemas  $X$  and  $Y$ . If  $D$  serves the same role in each of its schemas, then those schemas can be clustered together during join graph traversal.

$$neighborSim(X, Y, D) = \frac{1}{|X||Y|} \sum_{a \in X, b \in Y} \log\left(\frac{p(a, b|D)}{p(a|D)p(b|D)}\right)$$

The function *neighborSim* is very similar to the *coherency score* in Figure 8. The only difference is that the probability inputs to the PMI function are conditioned on the presence of a shared attribute. The result is a measure of how well two schemas cohere, apart from contributions of the attribute in question. If they cohere very poorly despite the shared attribute, then we expect that  $D$  is serving different roles in each schema (e.g., describing filesystems in one, and commerce in the other), and thus the schemas should be kept in separate clusters.

Clustering is how *neighborSim* helps with join graph traver-

sal. Whenever a join graph user wants to examine outgoing links from a schema  $S$ , WEBTABLES first clusters all of the schemas that share an attribute with  $S$ . We use simple agglomerative clustering with *neighborSim* as its distance metric. When the user chooses to traverse the graph to a neighboring schema, she does not have to choose from among hundreds of raw links, but instead first chooses one from a handful of neighbor clusters.

## 5. EXPERIMENTAL RESULTS

We now present experimental results for relation ranking and for the three ACSDB applications.

### 5.1 Relation Ranking

We evaluated the WEBTABLES ranking algorithms described in Section 3.1: **naïveRank**, **filterRank**, **featureRank**, and **schemaRank**.

Just as with the training set described in Section 3, we created a test dataset by asking two human judges to rate a large set of (query, relation) pairs from 1 to 5 (where 5 denotes a relation that is perfectly relevant for the query, and 1 denotes a completely irrelevant relation). We created 1000 pairs, divided over a workload of 30 queries. These queries were chosen by hand to plausibly yield structured data, but they were frozen prior to evaluation. For **featureRank** and **schemaRank**, which incorporate a series of clues about the relation’s relevance to the query, we chose feature weights using a trained linear regression model (from the WEKA package) [27].

We composed the set of query, relation pairs by first sending all the queries to **naïveRank**, **filterRank**, **featureRank**, and **schemaRank**, and recording all the URLs that they emitted. We then gathered all the relations derived from those URLs, and asked the judges to rate them. Obviously it is impossible for human judges to consider every possible relation derived from the web, but our judges did consider all the “plausible” relations - those generated by any of a number of different automated techniques. Thus, when we rank all the relations for a query in the human-judged order, we should obtain a good approximation of the “optimal” relation ranking.

We will call a relation “relevant” to its query if the table scored an average of 4 or higher by the judges. Table 5 shows the number of relevant results in the top- $k$  by each of the rankers, presented as a fraction of the score from the optimal human-judged list. Results are averaged over all queries in the workload.

There are two interesting points about the results in Table 5. First, **Rank-ACSDB** beats **Naïve** (the only solution for structured data search available to most people) by 78-100%. Second, all of the non-**Naïve** solutions improve on the optimal solution as  $k$  increases, suggesting that we are doing relatively well at large-grain ranking, but more poorly at smaller scales.

### 5.2 Schema Auto-Completion

Output schemas from the auto-completion tool are almost always coherent (as seen with the sample outputs from Table 3), but it would also be desirable if they cover the most relevant attributes for each input. We can evaluate whether the tool recalls the relevant attributes for a schema by testing how well its outputs “reproduce” a good-quality test schema.

k	Naïve	Filter	Rank	Rank-ACSDb
10	0.26	0.35	0.43	0.47
20	0.33	0.47	0.56	0.59
30	0.34	0.59	0.66	0.68

Table 5: Fraction of high-scoring relevant tables in the top-k, as a fraction of “optimal” results.

Input	1	2	3
name	0	0.6	0.8
instructor	0.6	0.6	0.6
elected	1.0	1.0	1.0
ab	0	0	0
stock-symbol	0.4	0.8	0.8
company	0.22	0.33	0.44
director	0.75	0.75	1.0
album	0.5	0.5	0.66
sqft	0.5	0.66	0.66
goals	0.66	0.66	0.66
<b>Average</b>	0.46	0.59	0.62

Table 6: Schema auto-complete’s rate of attribute recall for ten expert-generated test schemas. Auto-complete is given three “tries” at producing a good schema.

To generate these test schemas, we asked six humans who are familiar with database schemas to create an attribute list for each of ten databases, given only the inputs listed in Tables 3 and 4. For example, when given the prompt `company`, one user responded with `ticker`, `stock-exchange`, `stock-price`. We retained all the attributes that were suggested at least twice. The resulting test schemas contained between 3 and 9 attributes.

We then compared these test schemas against the schemas output by the WEBTABLES auto-completion tool when given the same inputs. To allow the tool to guess multiple correct schemas for a single input topic, we allowed the auto-completion tool to attempt multiple “tries”; after the algorithm from Section 4.1 emitted a schema  $S$ , we simply removed all members of  $S$  from the ACSDb, and then reran the algorithm. We gave the auto-completion tool three “tries” for each input.

Table 6 shows the fraction of the input schemas that WEBTABLES was able to reproduce. By its third output, WEBTABLES reproduced a large amount of all the test schemas except one<sup>3</sup>, and it often did well on its first output. Giving WEBTABLES multiple tries allows it to succeed even when an input is somewhat ambiguous. For example, the WEBTABLES schema listed in Table 3 (its first output) describes filesystem contents. But on its second run, WEBTABLES’s output contained address-book information (*e.g.*, `office`, `phone`, `title`); the test schema for `name` contained exclusively attributes for the address-book domain.

### 5.3 Synonym-Finding

<sup>3</sup>No test designer recognized `ab` as an abbreviation for “at-bats,” a piece of baseball terminology. WEBTABLES gave exclusively baseball-themed outputs.

Input	5	10	15	20
name	1.0	0.8	0.67	0.55
instructor	1.0	1.0	0.93	0.95
elected	0.4	0.4	0.33	0.3
ab	0.6	0.4	0.33	0.25
stock-symbol	1.0	0.6	0.53	0.4
company	0.8	0.7	0.67	0.5
director	0.6	0.4	0.26	0.3
album	0.6	0.6	0.53	0.45
sqft	1.0	0.7	0.53	0.55
goals	1.0	0.8	0.73	0.75
<b>Average</b>	0.8	0.64	0.55	0.5

Table 7: Fraction of correct synonyms in top-k ranked list from the synonym-finder.

We tested the synonym-finder by asking it to generate synonyms for the same set of inputs as seen previously, in Table 4. The synonym-finder’s output is ranked by quality. An ideal ranking would present a stream of only correct synonyms, followed by only incorrect ones; a poor ranking will mix them together. We consider only the accuracy of the synonym-finder and do not attempt to assess its overall recall. We asked a judge to determine whether a given synonym pair is accurate or not, and used this information to compute the system’s average accuracy for various top-k values. We did not assess a traditional recall measure because there is no reasonable “answer set” to compare against; even a hand-compiled thesaurus will not contain many of the unexpected synonyms (especially typographic ones) that our system can find (*e.g.*, `tel` and `tel-#`). (Our accuracy data does show raw counts of correct synonyms.)

The results in Table 7 show that the synonym-finder’s ranking is very good, with an average of 80% accuracy in the top-5. Attribute synonym-finding is a somewhat different problem from that of linguistic synonym-finding, but our results compare favorably to those of Lin and Pantel’s DIRT system, which achieved an accuracy of 57.5% or lower for eight of nine tested topics [15]. The average number of correct results from our system declines as the rank increases, as expected.

### 5.4 Join Graph Traversal

Most clusters in our test set (which is generated from a workload of 10 focal schemas) contained very few incorrect schema members. Further, these “errors” are often debatable and difficult to assess reliably. It is more interesting to see an actual portion of the clustered join graph, as in Figure 13. In this diagram, the user has visited the “focal schema”: [`last-modified`, `name`, `size`], which is drawn at the top center of the diagram. The user has applied join graph clustering to make it easier to traverse the join graph and explore related schemas.

By definition, the focal schema and its neighbor schemas share at least one attribute. This figure shows some of the schemas that neighbor [`last-modified`, `name`, `size`]. Neighbors connected via `last-modified` are in the left-hand column, neighbors via `name` are in the center column, and neighbors who share `size` are at the right.

Every neighbor schema is part of a cluster. (We have an-

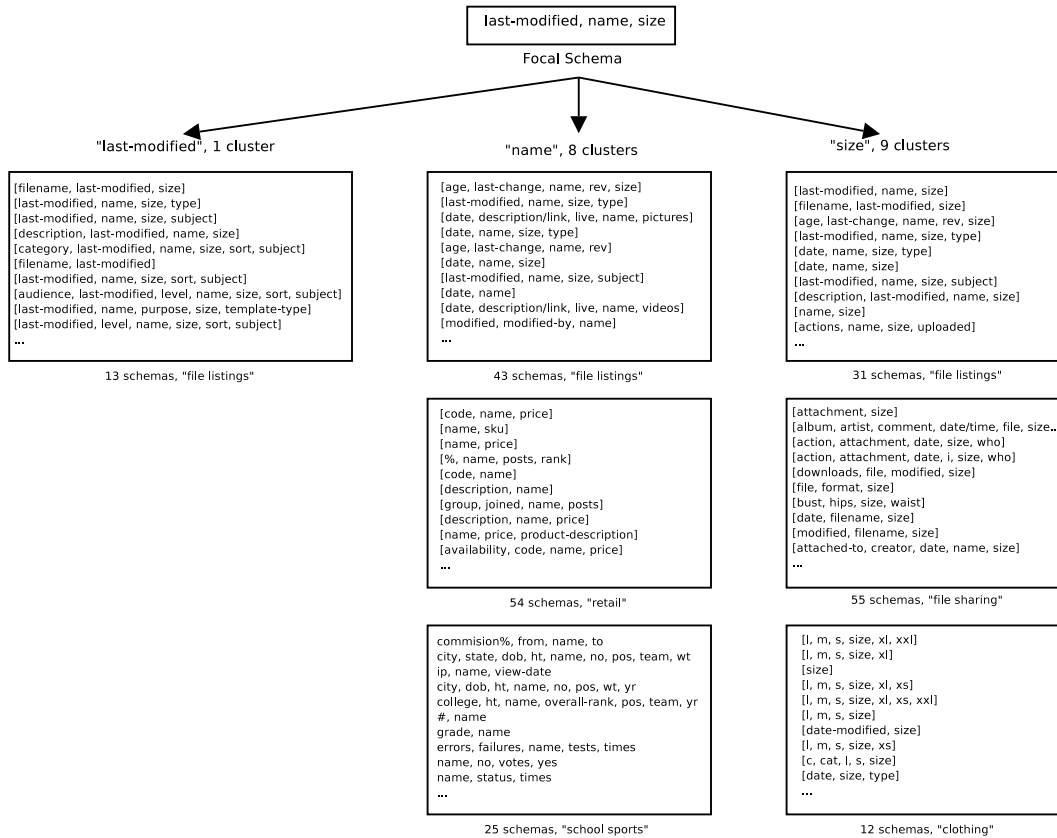


Figure 13: A neighbor-clustered view of the join graph, from focal schema [last-modified, name, size]. Schemas in the left-hand column share the last-modified attribute with the focal schema; schemas in the center column share name, and schemas at right share size. Similar schemas are grouped together in cluster boxes. The annotation under each box describes the total number of schemas and the theme of the cluster.

notated each cluster with the full cluster size and the rough subject area that the cluster seems to capture.) Without these thematic clusters in place, the join graph user would have no reasonable way to sort or choose through the huge number of schemas that neighbor the focal schema.

In Figure 13, most of the schemas in any given cluster are extremely similar. However, there is one schema (namely, the [group, joined, name, posts] schema) that has been incorrectly allocated to its cluster. This schema appears to be used for some kind of online message board, but has been badly placed in a cluster of retail-oriented schemas.

## 6. RELATED WORK

A number of authors have studied the problem of information extraction from a single table, though most have not tried to do so at scale. Gatterbauer, *et al.* attempted to discover tabular structure without the HTML `table` tag, through cues such as onscreen data placement [12]. Their approach could make a useful table extractor for WEBTABLES. Chen, *et al.* tried to extract tables from ASCII text [7]. Penn, *et al.* attempted to reformat existing web information for handheld devices [23]. Like WEBTABLES, they had to recognize “genuine tables” with true two-dimensional semantics (as opposed to tables used merely for layout). Similarly, as discussed in Section 1, Wang and Hu detected

“good” tables with a classifier, using features that involved both content and layout [26]. They operated on a corpus of just over ten thousand pages and found a few thousand true relations. Zanibbi, *et al.* offered a survey of table-parsing papers, almost all focused on processing a single table [30]. None of the experiments we found involved corpora of more than a few tens of thousands of tables.

We are not aware of any other effort to extract relational tables from the web at a scale similar to WEBTABLES. The idea of leveraging a corpus of schemas was first considered in [16]. That work considered collections of 40-60 schemas in *known domains* extracted from various sources, and showed that these schemas can be used to improve the quality of automatically matching pairs of disparate schema. As part of that, [16] used statistics on schema elements similar to ours.

We do not know of any work on automated attribute-synonym finding, apart from simple distance metrics used as part of schema matching systems [9, 16, 17, 24]. There has been some work on corpus-driven linguistic synonym-finding in the machine learning community. Turney used language cooccurrence statistics from the Web to answer standardized-test-style synonym questions, but relies on word-proximity statistics that seem inapplicable to structured data [25].

A number of tools have taken advantage of data statistics,

whether to match schemas [9, 10], to find dependencies [3, 28], or to group into tables data with many potentially-missing values [6, 21]. All of these systems rely on the dataset to give cues about the correct schema. We believe a hybrid of the **ACSDB** schema data and these data-centric approaches is a very promising avenue for future work.

## 7. CONCLUSIONS AND FUTURE WORK

We described the **WEBTABLES** system, which is the first large-scale attempt to extract and leverage the relational information embedded in HTML tables on the Web. We described how to support effective search on a massive collection of tables and demonstrated that current search engines do not support such search effectively. Finally, we showed that the recovered relations can be used to create what we believe is a very valuable data resource, the *attribute correlation statistics database*.

In this paper we applied the **ACSDB** to a number of schema-related problems: to improve relation ranking, to construct a schema auto-complete tool, to create synonyms for schema matching use, and to help users in navigating the **ACSDB** itself.

We believe we are just starting to find uses for the statistical data embodied in our corpus of recovered relations. In particular, by combining it with a “row-centric” analogue to the **ACSDB**, in which we store statistics about collocations of tuple keys rather than attribute labels, we could enable a “data-suggest” feature similar to our schema auto-completer. Of course, there are tremendous opportunities for creating new data sets by integrating and aggregating data from **WEBTABLES** relations, and enabling users to combine this data with some of their private data.

The **WEBTABLES** relation search engine is built on the set of recovered relations, and still offers room for improvement. An obvious path is to incorporate a stronger signal of source-page quality (such as PageRank) which we currently include only indirectly via the document search results. We would like to also include relational data derived from non-HTML table sources, such as deep web databases and HTML-embedded lists.

## 8. REFERENCES

- [1] E. Agichtein, L. Gravano, V. Sokolovna, and A. Voskoboinik. Snowball: A prototype system for extracting relations from large text collections. In *SIGMOD Conference*, 2001.
- [2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.
- [3] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. In *European Conference on Machine Learning*, 1995.
- [4] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Language Learning*, pages 858–867, 2007.
- [5] M. Cafarella, A. Halevy, Z. Wang, E. Wu, and Y. Zhang. Uncovering the relational web. In under review, 2008.
- [6] M. J. Cafarella, D. Suci, and O. Etzioni. Navigating extracted data with schema discovery. In *WebDB*, 2007.
- [7] H. Chen, S. Tsai, and J. Tsai. Mining tables from large scale html texts. In *18th International Conference on Computational Linguistics (COLING)*, pages 166–172, 2000.
- [8] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. In *Proceedings of the 27th Annual Association for Computational Linguistics*, 1989.
- [9] R. Dhamankar, Y. Lee, A. Doan, A. Y. Halevy, and P. Domingos. imap: Discovering complex mappings between database schemas. In *SIGMOD Conference*, 2004.
- [10] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.
- [11] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Web-scale information extraction in knowitall (preliminary results). In *Thirteenth International World Wide Web Conference*, 2004.
- [12] W. Gatterbauer, P. Bohunsky, M. Herzog, B. Krüpl, and B. Pollak. Towards domain-independent information extraction from web tables. In *Proceedings of the 16th International World Wide Web Conference (WWW 2007)*, pages 71–80, 2007.
- [13] B. He, Z. Zhang, and K. C.-C. Chang. Knocking the door to the deep web: Integration of web query interfaces. In *SIGMOD Conference*, pages 913–914, 2004.
- [14] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.
- [15] D. Lin and P. Pantel. Dirt: Discovery of inference rules from text. In *KDD*, 2001.
- [16] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based schema matching. In *ICDE*, 2005.
- [17] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, 2001.
- [18] J. Madhavan, A. Y. Halevy, S. Cohen, X. L. Dong, S. R. Jeffery, D. Ko, and C. Yu. Structured data meets the web: A few observations. *IEEE Data Eng. Bull.*, 29(4):19–26, 2006.
- [19] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
- [20] I. R. Mansuri and S. Sarawagi. Integrating unstructured data into relational databases. In *ICDE*, 2006.
- [21] R. Miller and P. Andritsos. Schema discovery. *IEEE Data Eng. Bull.*, 26(3):40–45, 2003.
- [22] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD Conference*, pages 1156–1158, 2007.
- [23] G. Penn, J. Hu, H. Luo, and R. McDonald. Flexible web document analysis for delivery to narrow-bandwidth devices. In *International Conference on Document Analysis and Recognition (ICDAR01)*, pages 1074–1078, 2001.
- [24] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
- [25] P. D. Turney. Mining the web for synonyms: Pmi-ir versus lsa on toefl. In *Proceedings of the Twelfth European Conference on Machine Learning*, 2001.
- [26] Y. Wang and J. Hu. A machine learning based approach for table detection on the web. In *Eleventh International World Wide Web Conference*, 2002.
- [27] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufman, San Francisco, 2nd edition edition, 2005.
- [28] S. Wong, C. Butz, and Y. Xiang. Automated database schema design using mined data dependencies. *Journal of the American Society of Information Science*, 49(5):455–470, 1998.
- [29] M. Yoshida and K. Torisawa. A method to integrate tables of the world wide web. In *Proceedings of the 1st International Workshop on Web Document Analysis*, pages 31–34, 2001.
- [30] R. Zanibbi, D. Blostein, and J. Cordy. A survey of table recognition: Models, observations, transformations, and inferences, 2003.