Extracting and Managing Structured Web Data

Michael John Cafarella

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2009

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Michael John Cafarella

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

Oren Etzioni

Dan Suciu

Reading Committee:

Oren Etzioni

Alon Halevy

Dan Suciu

Date:

University of Washington

Abstract

Extracting and Managing Structured Web Data

Michael John Cafarella

Co-Chairs of the Supervisory Committee:
Professor Oren Etzioni
Computer Science and Engineering

Professor Dan Suciu
Computer Science and Engineering

The Web contains a large amount of structured data embedded in natural language text, two-dimensional tables, and other forms. This "Structured Web" of data is vast, messy, and diverse; it also promises new and compelling applications. Unfortunately, existing tools such as search engines and relational databases ignore Structured Web data entirely.

This dissertation identifies four design criteria for a successful Structured Web management system. Such systems are:

1. **Extraction-Focused** - They obtain structured data wherever it can be found.
2. **Domain-Independent** - They are not tied to one particular topic area.
3. **Domain-Scalable** - They can effectively manage many domains simultaneously.
4. **Computationally-Efficient** - They can handle the Web's enormous size.

We also describe three working Structured Web management systems that observe these criteria. TextRunner is an extractor for processing natural language Web text. WebTables extracts and provides applications on top of relations in HTML tables. Finally, Octopus provides integration services over extracted Structured Web data. Together, these three systems demonstrate that managing structured data on the Web is possible today, and also suggest directions for future systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

A good doctoral dissertation should sprawl within a very deep focus. The debts of gratitude one incurs in completing it are equally deep and much more sprawling.

I owe many thanks to my undergraduate professors at Brown University. I am grateful not only for Andy van Dam's years of teaching and advice, but also for his friendly insistence that I pursue an academic career. Doing so might not have been possible without the help of Ben Kimia and Philip Klein, who generously shared a research project with me when I decided to leave industrial work.

Much of this dissertation would not have been possible if I had not learned to program with some of the best in the business. I am especially thankful for the time I spent working with Doug Cutting, Adam Doppelt, and Arthur Van Hoff, all of whom are superb engineers and friends.

Google, Inc. graciously allowed me to publish the research I performed while employed there, much of which is present in this dissertation. I am thankful to Magda Balazinska and Efthi Efthimiadis for serving on my final exam committee.

The University of Washington has been a wonderful place to study, most of all because of the people who work there. Faculty members Steve Gribble, Ed Lazowska, Hank Levy, and Dan Weld were not my official academic advisors, but nonetheless have given me excellent advice over the years. My fellow students Eytan Adar, Michele Banko, Doug Downey, and Chris Ré have been great friends and collaborators.

This work would not have been possible without the help of three people. Alon Halevy at Google taught me how to turn inchoate ideas into actual published research papers and working systems. My thesis advisor Dan Suciu was an indispensable introductory guide to the world of database research and helped to sharpen the technical discussions of my work. My other thesis advisor, Oren Etzioni, taught me to write my first research papers, including how to frame a research question and how to evaluate the results. He also encouraged me to "swing for the fences" with my research, and to not be content with small papers. Working with Alon, Dan, and Oren has been hugely rewarding, both personally and professionally. All three have been extremely generous with their time and support.

In short, it would be impossible for any student to ask for more than what they offered me. The only way I can repay their help is to try to do the same for my own students in the future.

That said, I am indebted most to my family - my parents, my brother Pete, my wife Torri, and Charlie. They make it all worthwhile.

DEDICATION

For Torri

Chapter 1

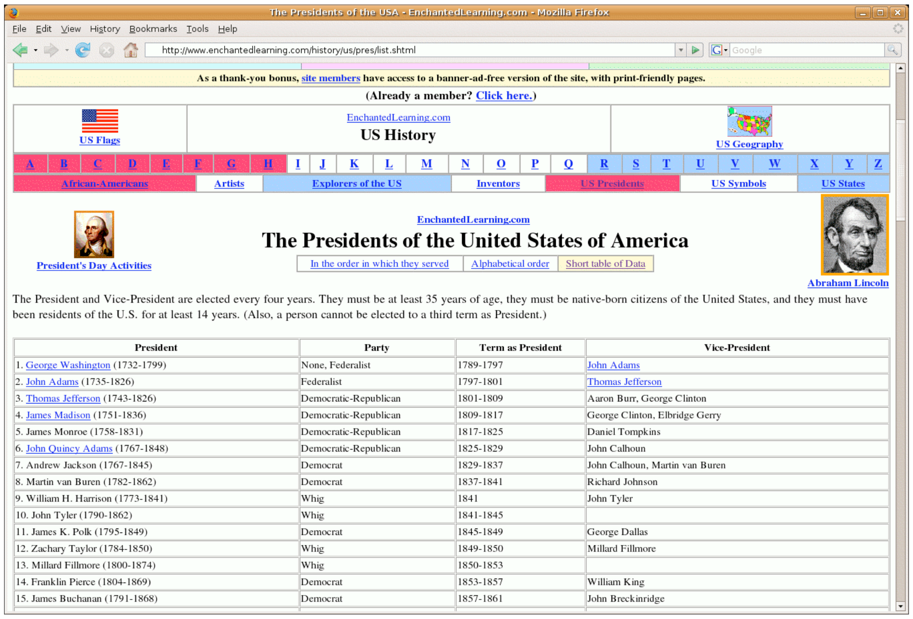
INTRODUCTION

A modern Web page often contains not just unstructured text but also a large amount of *structured* data. Consider a page like the one depicted in Figure 1.1. The lower part of this page contains an element that is very akin to a small relational database - it has a tuple-oriented row for each US President and a dimension-oriented column for each President's name, party, *etc.* The author of the page has even added a domain-specific label for each column. An interested Web user might examine this data and use it to compute the number of Presidents drawn from each political party. Consider also the page in Figure 1.2, which carries interesting factual assertions within natural language text, in this case, information about Einstein. By looking at other similar pages, a human user could easily compile a useful table of scientists and their birthdates. Unfortunately, there is no standard data tool that allows users to perform either of these straightforward tasks.

Traditional search engines are extremely good at document-finding, but they are limited to simple relevance-ranking queries. They cannot process queries that yield non-URL answers such as the name of a President, or involve multiple pieces of data from the same page such as the number of Presidents in each party, or combine information from multiple pages such as the set of all Presidents drawn from multiple countries. The search engines' query interfaces force users to treat each Web page as a unitary block of text, which can only be retrieved or not retrieved.

A relational database might appear to be a natural choice. However, such databases' import facilities assume that the inputs are clean and relational: the imported data should have an officially-declared relational schema, and the input data must abide by any constraints placed by the schema. But not even the cleanest HTML table on the Web can be imported by a standard relational database: the table cells are noisy, with many being empty or non-data-carrying; the metadata is declared only informally, often as simply the first row in the table; and of course the Web page contains many non-data elements. Data embedded in natural language text is even harder to obtain.

Further, relational databases make a number of assumptions about data scale that are not appropriate to the Structured Web context. For example, a traditional system forces users to refer to each relation by a unique human-understandable identifier. However, the Web contains more than 150M high-quality relations derived from HTML tables alone (see Section 4.2.1); no human being can be expected to remember so many unique names without some kind of aid akin to search engines that help manage text-centric document URLs.


The Presidents of the USA - EnchantedLearning.com - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.enchantedlearning.com/history/us/pres/list.shtml
Google

As a thank-you bonus, site members have access to a banner-ad-free version of the site, with print-friendly pages.

(Already a member? Click here.)

US Flags | EnchantedLearning.com US History | US Geography

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

African-Americans | Artists | Explorers of the US | Inventors | US Presidents | US Symbols | US States

President's Day Activities

EnchantedLearning.com

The Presidents of the United States of America

In the order in which they served | Alphabetical order | Short table of Data

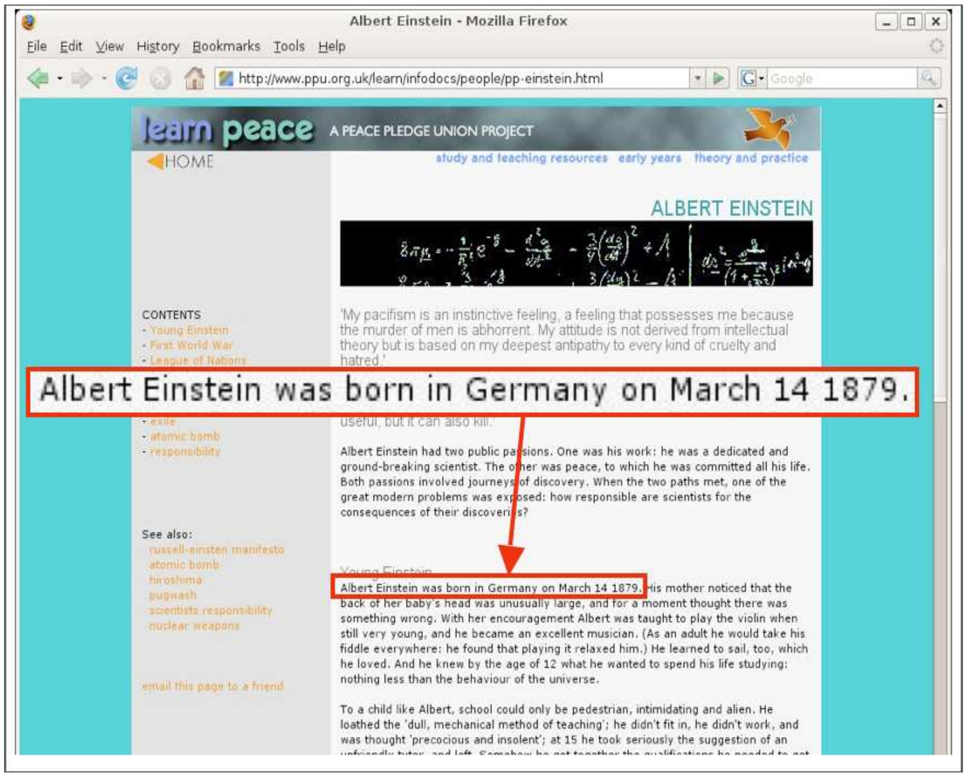
Abraham Lincoln

The President and Vice-President are elected every four years. They must be at least 35 years of age, they must be native-born citizens of the United States, and they must have been residents of the U.S. for at least 14 years. (Also, a person cannot be elected to a third term as President.)

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	Aaron 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	Daniel Tompkins
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 (1782-1862)	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 (1795-1849)	Democrat	1845-1849	George Dallas
12. Zachary Taylor (1784-1850)	Whig	1849-1850	Millard Fillmore
13. Millard Fillmore (1800-1874)	Whig	1850-1853	
14. Franklin Pierce (1804-1869)	Democrat	1853-1857	William King
15. James Buchanan (1791-1868)	Democrat	1857-1861	John Breckinridge

Figure 1.1: A typical use of the `table` tag to describe relational data. The data here has structure that is never explicitly declared by the author but is obvious to a human observer, including "metadata" that consists 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.

This dissertation examines how to extract and manage the *Structured Web*, an automatically-constructed structured version of information drawn from the Web. Given the Web's breadth and scope of coverage, we can also imagine it as a "database of everything." Such a dataset, combined with appropriate tools, promises a number of compelling applications.



Albert Einstein - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://www.ppu.org.uk/learn/infodocs/people/pp-einstein.html

learn peace A PEACE PLEDGE UNION PROJECT

HOME

study and teaching resources early years theory and practice

ALBERT EINSTEIN

CONTENTS
- Young Einstein
- First World War
- League of Nations
- exile
- atomic bomb
- responsibility

See also:
russell-einsten manifesto
atomic bomb
hiroshima
pugwash
scientists responsibility
nuclear weapons

email this page to a friend

'My pacifism is an instinctive feeling, a feeling that possesses me because the murder of men is abhorrent. My attitude is not derived from intellectual theory but is based on my deepest antipathy to every kind of cruelty and hatred.'

Albert Einstein was born in Germany on March 14 1879.

useful, but it can also kill.'

Albert Einstein had two public passions. One was his work: he was a dedicated and ground-breaking scientist. The other was peace, to which he was committed all his life. Both passions involved journeys of discovery. When the two paths met, one of the great modern problems was exposed: how responsible are scientists for the consequences of their discoveries?

Young Einstein

Albert Einstein was born in Germany on March 14 1879. His mother noticed that the back of her baby's head was unusually large, and for a moment thought there was something wrong. With her encouragement Albert was taught to play the violin when still very young, and he became an excellent musician. (As an adult he would take his fiddle everywhere: he found that playing it relaxed him.) He learned to sail, too, which he loved. And he knew by the age of 12 what he wanted to spend his life studying: nothing less than the behaviour of the universe.

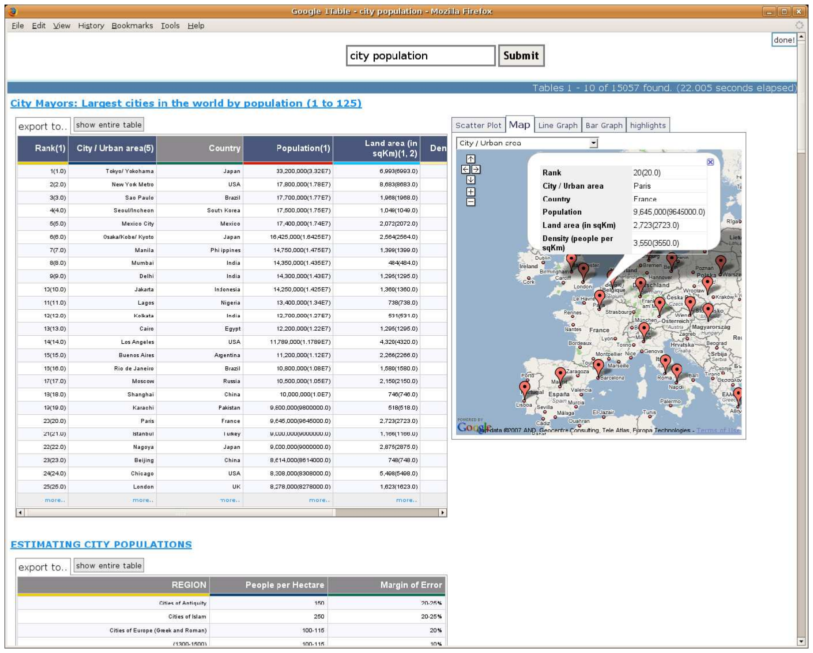
To a child like Albert, school could only be pedestrian, intimidating and alien. He loathed the 'dull, mechanical method of teaching'; he didn't fit in, he didn't work, and was thought 'precocious and insolent'; at 15 he took seriously the suggestion of an

Figure 1.2: A web page consisting of mainly natural language text, with no obvious relational table elements. Some sentences, such as the one highlit in the figure, contain a fact-like assertion. It is easy to imagine populating a database relation of scientists and birth information by reading a large number of documents and extracting facts of the type seen here.

For example:

- *Highly-Expressive Web Search* - Web searches and their results could be far more sophisticated than the current keyword queries. For example, a user might ask for a list of scientists, their inventions, and their years of death. Table 1.1 shows output for that query from the prototype ExDB system [24], which derived structured answers using natural language text from a 90M-page general Web crawl.

- *Easy Data Analysis* - Consider a spreadsheet user who would like to analyze some data, say for public policy reasons. Spreadsheet software provides a huge amount of


Google 1Table - city population - Mozilla Firefox
File Edit View History Bookmarks Tools Help
city population
Submit
done!
Tables 1 - 10 of 15057 found. (22.005 seconds elapsed)
Scatter Plot
Map
Line Graph
Bar Graph
highlights
City / Urban area
export to..
show entire table

City Mayors: Largest cities in the world by population (1 to 125)

Rank(1)	City / Urban area(5)	Country	Population(1)	Land area (in sqKm)(1, 2)	Den
1(1.0)	Tokyo/ Yokohama	Japan	33,200,000(3.32E7)	6,993(6993.0)	
2(2.0)	New York Metro	USA	17,800,000(1.78E7)	8,683(8683.0)	
3(3.0)	Sao Paulo	Brazil	17,700,000(1.77E7)	1,968(1968.0)	
4(4.0)	Seoul/Incheon	South Korea	17,500,000(1.75E7)	1,049(1049.0)	
5(5.0)	Mexico City	Mexico	17,400,000(1.74E7)	2,072(2072.0)	
6(6.0)	Osaka/Kobe/ Kyoto	Japan	16,425,000(1.6425E7)	2,564(2564.0)	
7(7.0)	Manila	Philippines	14,750,000(1.475E7)	1,399(1399.0)	
8(8.0)	Mumbai	India	14,350,000(1.435E7)	484(484.0)	
9(9.0)	Delhi	India	14,300,000(1.43E7)	1,295(1295.0)	
10(10.0)	Jakarta	Indonesia	14,250,000(1.425E7)	1,360(1360.0)	
11(11.0)	Lagos	Nigeria	13,400,000(1.34E7)	738(738.0)	
12(12.0)	Kolkata	India	12,700,000(1.27E7)	531(531.0)	
13(13.0)	Cairo	Egypt	12,200,000(1.22E7)	1,295(1295.0)	
14(14.0)	Los Angeles	USA	11,789,000(1.1789E7)	4,320(4320.0)	
15(15.0)	Buenos Aires	Argentina	11,200,000(1.12E7)	2,266(2266.0)	
16(16.0)	Rio de Janeiro	Brazil	10,800,000(1.08E7)	1,580(1580.0)	
17(17.0)	Moscow	Russia	10,500,000(1.05E7)	2,150(2150.0)	
18(18.0)	Shanghai	China	10,000,000(1.0E7)	746(746.0)	
19(19.0)	Karachi	Pakistan	9,800,000(9800000.0)	518(518.0)	
20(20.0)	Paris	France	9,645,000(9645000.0)	2,723(2723.0)	
21(21.0)	Istanbul	Turkey	9,000,000(9000000.0)	1,166(1166.0)	
22(22.0)	Nagoya	Japan	9,000,000(9000000.0)	2,875(2875.0)	
23(23.0)	Beijing	China	8,614,000(8614000.0)	748(748.0)	
24(24.0)	Chicago	USA	8,308,000(8308000.0)	5,498(5498.0)	
25(25.0)	London	UK	8,278,000(8278000.0)	1,623(1623.0)	
more..	more..	more..	more..	more..	

Rank	20(20.0)
City / Urban area	Paris
Country	France
Population	9,645,000(9645000.0)
Land area (in sqKm)	2,723(2723.0)
Density (people per sqKm)	3,550(3550.0)

ESTIMATING CITY POPULATIONS

REGION	People per Hectare	Margin of Error
Cities of Antiquity	150	20-25%
Cities of Islam	250	20-25%
Cities of Europe (Greek and Roman)	100-115	20%
(1300-1500)	100-115	10%

Figure 1.3: Results of a keyword query search for "city population", returning a relevance-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. The visualization to the right was generated automatically by the system, and shows the result of clicking on the "Paris" row. The title ("City Mayors...") links to the page where the original HTML table was found.

support for the mathematical side, but simply obtaining the relevant data can be a large burden. It should be possible to search for all structured datasets as easily as we can now find relevant unstructured text documents. Figure 1.3 depicts a sample query processed by a prototype version of such a structured search engine, where the user has typed the query `city population`. Instead of a ranked list of URLs, the system has returned a ranked list of databases. The first hit, shown in the figure, contains data about the 125 most-populous cities. The labeled columns contain information on the city name, the country where the city can be found, the population, and so on. Having issued just a single keyword query to find the data, the user could now easily import this data into her spreadsheet for further analysis.

- *Web Data Integration* - The sheer amount of Web data makes it an enticing subject for reuse and recombination. Consider the database of all program committee members for an academic conference. This information generally exists on the Web, but compiling it into a single database is tedious: a human being would probably use a search engine to locate each of the conference's year-specific Web sites, then manually combine the PC member information found on each. Instead, the user of a Structured Web-aware tool should be able to perform a simple *union* of PC member data found across the different sites, entailing very little manual work.

scientist	**invention**	**year-of-death**	**probability**
Kepler	log books	1630	0.8
Heisenberg	matrix mechanics	1976	0.8
Galileo	telescope	1642	0.7
Newton	calculus	1727	0.7

Table 1.1: The top-ranked results for an ExDB structured query, which asks for information about scientists, their inventions, and the years of their deaths. The query is expressed in Datalog-like syntax: `q(?scientist, ?invention, ?year-of-death) :- invented(?scientist, ?invention), died-in(?scientist, <year>?year-of-death)`. These answers were computed by extracting data from a crawled set of 90M Web pages. In the ExDB system, each extraction has an associated probability of being true or false, as do tuples in the answer set. Returned answers are sorted in descending order of probability.

The thesis of this dissertation is that managing and accessing the Structured Web poses unique difficulties which neither traditional database systems nor search engines address. Moreover, we can successfully manage the Structured Web by designing management tools that observe four critical design criteria: they should be *extraction-focused*, *domain-independent*, *domain-scalable* and *computationally-efficient*.

This chapter provides an overview of why managing the Structured Web data is difficult, and how we propose to handle it. We start in Section 1.1 by defining some necessary terminology. Section 1.2 describes several problems that distinguish Structured Web data management from more traditional text or relational data management issues, and we outline our general approach in Section 1.3. We then briefly discuss how our approach is made concrete in three distinct systems. Section 1.4 reviews our technical contributions. Finally, we provide an overview of the dissertation's general organization in Section 1.5.

1.1 What is Structured Data?

It is standard for database researchers to refer to data as either *structured*, *semi-structured*, or *unstructured*. Unfortunately, this usage is often imprecise. Generally, *structured data* refers to data expressed using the relational model; *semi-structured data* means XML data; and *unstructured data* refers to documents such as text, Web pages, spreadsheets, and presentations. Confusingly, semi-structured data is sometimes described as being contained in XML *documents*; we will reserve the word "document" to describe a piece of unstructured data.

These terms are unsatisfactory for a number of reasons. First, one might think that *semi-structured* XML data is somehow less formally-defined than *structured* relational data, but this is false. Indeed, it is also easy to give a formal definition for an *unstructured* document (*e.g.*, a text document is just a linear array of tokens drawn from the set of valid words and punctuation symbols).

Second, an unsuspecting reader may think that unstructured data lacks a formally-defined query language. But Web search queries are essentially Boolean selection queries that could be expressed in SQL. It is possible (though arguably unwise) to implement a search engine using a relational database.

Finally, it is tempting to believe these terms refer to whether the data is intended for machine use (structured) or human consumption (unstructured) or somewhere in-between (semi-structured). This usage is probably accurate to some extent - "unstructured" documents are often meant for human consumption. But a row in a structured relational database can be very easy for a person to read, and unstructured spreadsheets often contain abstruse statistical data.

There is one interpretation that appears to match general usage and is very relevant to this dissertation. We suggest the terms *structured*, *semi-structured*, and *unstructured* roughly describe the extent to which a dataset supports queries with domain-specific operations. (In this thesis, we will use the word *domain* in its non-mathematical sense, *i.e.*, as a synonym for *topic.*) For example, a very simple (structured) relational database about employees might support the following SQL query:

```
SELECT employeeName, yearsOfEducation, yearsOfService WHERE
       yearsOfEducation > 15 AND yearsOfService < 20
```

Writing such a query entails contributions from at least three different parties:

- The authors of the SQL language standard.
- The authors of the employee database schema, which includes concepts of `employeeName`, `yearsOfEducation` and `yearsOfService`.
- The author of the query, who has supplied the overall logical structure and the constants `15` and `20`.

A traditional document-centric search engine is not sufficient to support the above query. We could index this database with a search engine by first "flattening" the data into an unstructured format: we simply write out all the contents to a series of text files, with one tuple per file. We then use the search engine to index the resulting files. However, there would be no way to express the above query - a search user could issue the query `15 20`, but this would simply return all documents (tuples) that contain those terms, without regard to comparisons or the correct data attribute, *i.e.*, `yearsOfEducation` vs `yearsOfService`. The query language does not support comparison operators nor domain-specific notions like the data attribute names. Search engine query languages support multiple topic-insensitive operators (*e.g.*, testing term presence, testing phrase presence, testing the site's domain, possibly testing whether two terms are NEAR each other, *etc.*) and multiple "fields" of the data (such as the URL and the DNS domain mentioned earlier). But such systems do not support fields relevant to each document's actual subject.

Of course, it is possible to design a relational schema that hardly appears domain-specific at all. Instead of a table with columns for `employeeName`, `yearsOfEducation`, and `yearsOfService`, one could populate a three-column table with columns `tupleid`, `attrName`, and `value`, spreading a single tuple's data cross many rows. With such a design, it would still possible to pose questions against domain-specific elements by rewriting the user's query; topic-specific knowledge would be embedded in the query-rewriting system rather than the relational schema itself.

If *structured* data allows operations on domain-specific data elements, and *unstructured* data does not, then one might imagine that just a fraction of *semi-structured* data elements are domain-specific. Indeed, this is stereotypically the case for such canonical XML datasets as health records, which have a mix of relational-style and textual components.

This interpretation also gives us crisp definitions for some other terms relevant to this dissertation. The *Structured Web* is that portion of Web information that could usefully be queried using a domain-sensitive representation (even if it is currently indexed using just a search engine). For example, a list of upcoming musical tour dates should be part of the Structured Web, but a poem would not be. An *information extractor* takes an unstructured input and emits a more-structured representation of the information - that is, the extractor adds domain-sensitivity to the representation. For example, an extractor transforms the unstructured textual representation of musical tour dates into a structured, more-domain-specific, relational version.

With these definitional tasks complete, we can now discuss the challenges entailed in managing Structured Web data.

1.2 Challenges for Structured Web Data Tools

The Web is one of the most interesting and popular datasets ever assembled, covering a vast number of topics from a huge array of contributors. At least in the areas of document creation and distribution, the Web's distinctive qualities of enormous scale and a lack of central control, and the many document creators that those qualities engender, are now commonly-accepted facts of life. For example, although the fact was once startling, it is now taken for granted that erecting a Web site with global reach is inexpensive or free, that the Web contains information on almost every topic imaginable, and that there is no coordination among document-creators. However, note how remarkable these qualities are when applied to structured data management:

- We cannot assume that people will publish data "cleanly." It will often come in unstructured and often casual forms: inside natural language text, embedded in documents, or expressed with only minimal gestures toward machine-understanding. A traditional database, on the other hand, expects all "importable" data to be in one of a handful of extremely clean and unambiguous forms.

- Web data is published in any way the publisher prefers, with no guarantees about the data's structure/schema or quality. Contrast this scenario to the case of traditional relational data management, in which there is a centrally-designed schema that must be observed. The designers of the schema certainly choose the data's structural design, *e.g.*, what attributes may be expressed. They may choose to enforce data-quality constraints, *e.g.*, disallowing blank cells, or NULLs.

- The size and diversity of the Web audience means there will be a large amount of crawlable structured data on almost any topic. In contrast, traditional databases are difficult to access and focus heavily on a single domain at a time.

Most of these distinctions between traditional and Structured Web data management generally pose new burdens for the Web-centric case. However, Structured Web management is easier than traditional relational data management in at least one way: it is in general not transactional. Thus much of the traditional database work on transactional safety and performance is not applicable to the task at hand.

Transactions aside, the unique qualities of the Web pose several new challenges for Structured Web systems. In Section 1.3 below, we discuss the three design criteria that arise from these challenges, and how they are reflected in three different projects.

1.3 General Approach

The above-mentioned remarkable qualities of the Web - enormous scale with no central control - which have become so commonplace in modern document management practices, are in general not yet reflected in our structured data management tools. We believe that managing Structured Web data requires four design criteria:

1. **Extraction-focused**, rather than *data-loading-focused.* People on the Web will publish data in many different messy ways. It is unreasonable to expect that all interesting data will come in formats that are easy for relational databases to import, metadata and all (such as comma-separated lists). Instead, we should use information extraction (IE) techniques, which consume unstructured text and output clean, structured data. Extractors are not entirely universal - one may focus on text, another on tables, and so on - but they should attempt to cover as much of the Web as possible.

2. **Domain-independent**, rather than *sensitive to topic-specific data, rules, or schemas.* Our definitions above describe structured data as necessarily domain-dependent. But the mechanisms that extract and query this structured Web data cannot have domain knowledge "baked in." For IE systems, domain-independence means avoiding extraction rules or training data that are tailored to a specific topic. For example, an domain-dependent extractor that tries to recover corporate intelligence information might look only for sentences that contain the verb "acquired." Clearly, such an extractor will be useless for the vast majority of Web content.

3. **Domain-scalable**, in the system's ongoing operation. Most traditional structured databases manage single-domain datasets. The domain's metadata is not queryable - it is instead the environment in which users interact with the system. For example, relational databases assume that the user can easily name each table in the database - the table names themselves are not queryable. However, the Structured Web has a huge number of domains, each possibly with its own unique metadata. Users must be able to operate over data without incurring per-domain burdens.

   Domain-scalability may at first appear similar to domain-independence. The difference lies in whether a user of the system can not just extract or express data from many domains, but also effectively perform operations over many domains. A domain-scalable system must be domain-independent, but the reverse is not true.

4. **Computationally-efficient**, in non-traditional ways. Traditional database systems have been successful at handling large numbers of tuples, and search engines can handle large numbers of Web pages. But the Structured Web poses new efficiency challenges. Many of these issues concern information extraction components, which in the past have not often been designed for large document corpora. The explosion in the number of domains may also pose a new computational problem that previous systems have not needed to address.

Below, we describe how we applied these design considerations to three separate data systems. The first is TextRunner, an extraction system that operates over very large volumes of Web text. The second project, WebTables, extracts and then offers services on top of Web-embedded HTML tables that carry tabular relational-style data. The final project, Octopus, is focused relatively little on extraction and instead focuses on operations over extracted data; it allows users to effectively integrate data derived from HTML tables using WebTables, or from HTML lists.

1.3.1 Web-Scale Textual Extraction

TextRunner is an information extraction system that attempts to extract n-ary fact triples from natural-language Web text. For example, a biographical Web page about Einstein might yield `[Einstein, was-born-in, 1875]`. Textual fact extraction is not a new idea, but previous textual extractors, such as Agichtein and Gravano's Snowball [1], Brin's DIPRE [18], and Etzioni *et al.*'s KnowItAll [39] have had various qualities that make them unsuitable for processing all of the relations in a very large Web crawl; they have required domain-specific rules or training data, or alternatively have been extremely computationally expensive. TextRunner operates in three distinct steps:

1. **Learning an Extraction-Detector**. Using a small sample of the input corpus, a computationally-expensive natural language parser, and some heuristics about what makes a good extraction, TextRunner builds an efficient extraction-detector. For a sample set of input sentences, TextRunner runs a parser on each, obtaining a full parse tree. It then checks to see whether these parse trees match a set of heuristics

about linguistic forms that suggest fact tuples. For example, many fact-centric sentences have two *noun-phrases* linked by a *verb-phrase*, as in, "Einstein was born in 1875." TextRunner also obtains a set of inexpensively-computed features for the sentence, such as simple part-of-speech tagging and noun phrase chunking. It uses these heuristic-verdict / inexpensive-feature-vector pairs to train a classifier that uses only these inexpensive features to distinguish good from bad extractions.

2. **Single-Pass Extraction**. TextRunner then runs the trained extraction-classifier on every sentence in the entire corpus. Because the detector only requires inexpensively-computed features, as opposed to a full parse, this step is computationally efficient. If the classifier believes it has found a good extraction, it emits the resulting tuple and sends it to the next step.

3. **Redundancy-Based Assessment**. Using a model introduced by Downey *et al.* [37], TextRunner assigns a probability to each extraction based on how frequently it has been seen. The more frequently an extraction is observed, the greater the probability it is true.

TextRunner's design is meant to observe the above-mentioned Structured Web data design criteria. TextRunner is:

1. **Extraction-focused**. The main goal of TextRunner is to obtain all of the factual statements currently embedded in natural language text. These are transformed from unstructured text into a more-structured object/relation fact-triple model.

2. **Domain-independent**, as TextRunner requires no domain-specific extraction rules or training data. Note that this requirement is not observed by many previous extraction systems. For example, the previously-mentioned Snowball [1] and DIPRE [18] both require sample inputs, such as pairs of books and authors. The systems can then grow the sample inputs by finding additional tuples. Wrapper induction systems have required not just domain-specific training data, but website-specific data as well; see, *e.g.*, Kushmerick *et al.* [54] and Muslea *et al.* [66]. All of the rules and training

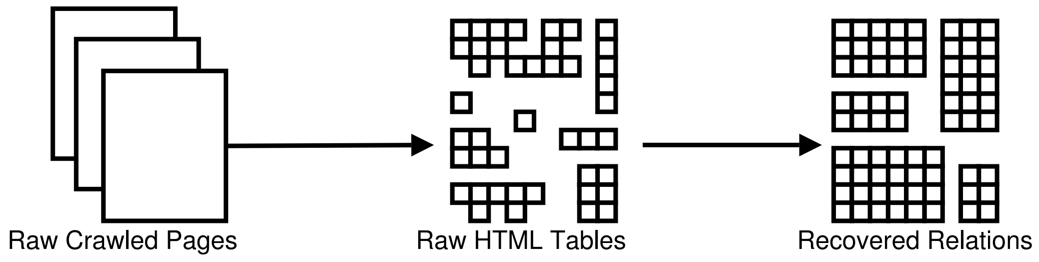


Figure 1.4: The WebTables relation extraction pipeline. About 1.1% of the raw HTML tables are true relations.

data used by TextRunner is not topical but linguistic in nature. The extraction heuristics and the natural language parser are language-specific. There is also some very weak linguistic tie to linguistic characteristics of the initial document sample.

3. **Domain-scalable**. TextRunner does not offer query facilities, whether domain-scalable or not. But it definitely does not require that the extraction task be parameterized by the user with any domain-specific information, unlike other extraction systems such as KnowItAll [39].

4. **Computationally-efficient**, as TextRunner avoids processing costs that scale with the number of domains. It also avoids the high per-document computational costs associated with using natural-language parsers at large scale.

We discuss TextRunner in additional detail in Chapter 3.

1.3.2 Table-Oriented Data Extraction and Mining

The WebTables project combines an information extractor for Web-embedded relational data with several applications built on top of that data. As mentioned at the start of this chapter, the HTML `table` tag is often used by Web authors to draw tabular structures in a browser window as in Figure 1.1. About 1.1% of these tables are used to display data that has tuple-oriented rows and dimension-oriented columns, as opposed to being used for page navigation or other purposes. In these tuple-oriented cases, the page's author is essentially publishing a simple relational database. These databases are far simpler than the transactional datasets managed by a traditional system such as Oracle or DB2. However, the WebTables-derived databases are also far more *numerous*. The sheer number of extracted databases allows WebTables to also build a number of novel applications on top of the data.

Figure 1.4 shows a simplified version of the WebTables processing pipeline. We start with a raw general Web crawl, then parse the HTML to find all of the raw HTML tables. WebTables then uses a series of classifiers to perform *relation recovery*, which entails distinguishing `relational` from `non-relational` tables, as well as detecting metadata in the form of column labels when present. After extracting a large number of databases, WebTables builds a number of interesting applications on top of the data: table search, synonym-finding, schema-autocomplete, and others.

WebTables observes the Structured Web data design criteria in the following ways. It is:

1. **Extraction-focused**, as it gathers structured relational data exclusively from extant HTML-embedded table structures. An HTML table is undeniably a flawed vehicle for expressing relational data: type information for data cells must be inferred, schema constraints are unenforceable, and metadata is both optional and sometimes difficult to detect. However, the qualities that make HTML tables difficult to process - the lack of central controller to enforce quality, a huge audience of creators with diverse ideas on how to design a structured database - are also what make them numerous and diverse. WebTables uses a series of classifiers to distinguish relational from non-relational tables, as well as detect when the author has inserted relevant metadata.

2. **Domain-independent**, as it requires no topic-specific information either when extracting structured data or when running applications on top of the resulting data. WebTables's extractor classifiers do not use any domain-specific features. It would be possible to test a candidate table's extracted schema against a dictionary of known good schemas, but this approach obviously constrains us to whatever is in the dictionary. Instead, the classifiers use only qualities that are present in every table, such as

the dimensions of the extracted tabular structures, the number of empty table cells, and the inferred data element types. Downstream applications, such as table search and attribute synonym finding, similarly require no domain-specific knowledge, rules, or data.

3. **Domain-scalable**, as it allows the user to easily search for relevant data regardless of domain.

4. **Computationally-efficient**, to the extent necessary. Most of the WebTables extraction and application components involve batch-oriented computation that is linear in the number of documents or tables being processed. Searching over the extracted tables incurs costs similar to Web search engines. Any expensive algorithms should be run only on small datasets, when possible.

Chapter 4 gives an in-depth description of WebTables, including its extraction pipeline and the applications that we have built on top of the resulting data.

1.3.3 Data Integration for the Structured Web

Octopus is a system for integrating extracted Web data. The problem of data integration has existed for a long time in a traditional database setting, for example, when two companies merge and must combine their employee databases. The scale and richness of the Web make integration very tantalizing - for example, the user may want to compile a database of all the program committee members who have ever served a given conference such as *VLDB*. But integrating Structured Web data also poses several difficulties. The relevant information exists, but is scattered across more than a dozen different sites and is not customarily published in a formal way. Further, there are a huge number of potential sources, none of which have formal metadata. Finally, there are complicated interactions among data elements on a single Web page that Octopus must navigate. For example, a *VLDB* Web page might enumerate its committee members using an HTML list, but the year of the conference is likely to be elsewhere, perhaps in the page's title.

The OCTOPUS system offers a workbench-style environment for users to construct a novel database from potentially dozens of source sites by issuing just two or three commands; the user never has to manually visit the source sites, and those sites never have to publish data in any kind of specialized format. The **search** command takes a user's keyword query as input, and returns clusters of relevant and structurally compatible extracted relations. Relations in a single cluster are effectively the system's candidates for union. The **context** operator examines the Web text that surrounds extracted data in order to obtain additional structured values. The **extend** operator is akin to the traditional *join* and adds extra columns to a table with data that are related to existing tuples and relevant to a user-indicated keyword.

Together, these operators enable a new and powerful way to repurpose existing Structured Web data while solving or sidestepping the new problems posed by that data. In particular, OCTOPUS solves these problems by observing the above-mentioned design criteria. It is:

1. **Extraction-focused**, as OCTOPUS uses only WEBTABLES-induced tables and HTML lists as its structured inputs. Table extraction is well-covered in our discussion of WEBTABLES in Section 4 below. HTML lists can be thought of as a form of HTML table in which the column boundaries have been lost and must be automatically recovered; OCTOPUS introduces algorithms for performing this recovery. In addition, the CONTEXT operator retrieves data elements embedded in the text that surrounds an extracted table in its source Web page, as in the VLDB conference case mentioned above. CONTEXT only makes sense in an environment where extracted data predominates.

2. **Domain-independent**, at the time of extraction as well as the user's application of operators. First, the extraction-rooted components of OCTOPUS (WEBTABLES table extraction, HTML list extraction, and the CONTEXT operator) rely exclusively on domain-independent features such as string lengths and word-frequencies. Similarly, the SEARCH and EXTEND operators are designed to, respectively, retrieve/cluster and join structured tables without access to reliable schema information. Note that

domain-independence can place a heavy computational and algorithmic burden. For example, a traditional join operator might simply examine a well-designed schema to see if two columns can be joined; our analogous operator, EXTEND, must apply a statistical test to many potential target tables in order to find a high-quality join between two columns.

3. **Domain-scalable**, throughout the application. As with WEBTABLES, users are never asked to address an extracted table by name. Rather, OCTOPUS's SEARCH always returns tables in response to a subject-matter keyword query. Further, the join-like EXTEND operator does not take a named join table as would be common with a relational database. It instead chooses a target join table based on a user-provided keyword.

4. **Computationally-efficient**, although a handful of operator algorithms pose most of the challenges. In these difficult cases, the algorithms must issue a huge number of Web search queries; we reduce the number of necessary queries via approximation.

Chapter 5 gives a detailed overview of OCTOPUS, especially its three novel operators and the algorithms that implement them.

We can now move to an overview of the technical contributions of this dissertation. First, however, it is helpful to briefly distinguish our Structured Web data from another nontraditional source of structured data: the Deep Web.

1.3.4 An Aside: The Deep Web

It is easy to confuse the Structured Web with what is often called the Deep Web. The Deep Web refers to the huge set of relational data stored in traditional databases that are connected to the Web, with an HTML-based front end. Most of the data in the Deep Web is not immediately crawlable, but is instead elicited by an HTTP form submission. Estimates of the size of the Deep Web differ, but some place it at up to 500 times the size of the traditional surface Web (see He *et al.* [45] and Madhavan *et al.* [59]). It is possible to automatically reveal part of the Deep Web by computing and submitting a form.

Unlike the Deep Web, our Structured Web data is easily crawlable: no special form submission is required. But, Structured Web elements typically involve substantial amounts of extraction effort, which may or may not apply to Deep Web data. The Structured Web is clearly distinct from the Deep Web when it comes to information extracted from natural language text. In the case of extracted HTML tables, we can detect overlap by checking whether an extracted table's source URL is parameterized, thus suggesting arguments that are passed to an application server and back-end Deep Web database. In our experiments, described in Chapter 4, we found that at least 60% of our URLs are not parameterized.

1.4 Technical Contributions

Structured data is prevalent on the Web, but no current data system can make use of it: search engines treat everything as unstructured text, and traditional databases make a number of assumptions (especially, but not only, around data import) that are not appropriate for Structured Web data. This dissertation identifies four design criteria that are critical for systems that successfully manage the Structured Web. They are: extraction-focused, domain-independent, domain-scalable and computationally-efficient. A system that lacks any of these four criteria will be unable to access most of the Structured Web, which is generally published in unstructured formats, covers a huge array of topics, and is "large" in surprising dimensions. We have applied these criteria to the design of three interesting systems. Their technical contributions are as follows:

- TextRunner - We introduce the TextRunner *open information extraction* system for obtaining tuples from natural language text on the Web. Unlike previous extraction systems, TextRunner's architecture allows it to operate without any domain-specific extraction rules or training data. Further, it has complexity $O(D)$, where D is the number of documents processed; other systems have complexity that is the product of the number of documents and topics, or even quadratic in the number of documents. We show that when TextRunner is tested against a more traditional information extraction system that requires *a priori* knowledge of the target relation, TextRunner can obtain results with a 33% lower error rate on a set of ten relations. On a 9M-page corpus that included 278,000 distinct relation strings, we show TextRunner offers a speedup of four orders of magnitude compared to KnowItAll.

- WebTables - We describe the WebTables system, which extracts relational databases from HTML tables, and then builds a number of novel data-centric applications on top of the resulting data. By processing a very large Web crawl, WebTables is thus able to obtain more than *125 million* distinct databases - a collection that is five orders of magnitude larger than any other corpus we are aware of. Based on the large amounts of metadata we can recover from this corpus, we introduce *attribute occurrence statistics*, which characterize the relationship between attribute labels observed in the database corpus. These statistics enable a number of novel applications: *schema autocomplete*, which suggests data attributes to novice database designers; an *attribute synonym tool* that automatically finds synonymous attribute label pairs; and others. We show experimentally that each of these applications give high-quality results on a test set of queries.

- Octopus - We introduce a workbench-like system so a user can integrate Structured Web sources with very little effort. Unlike previous systems, Octopus does not require a trained database administrator nor XML-formatted data nor hand-written extractors. We introduce three integration operators - **search, context**, and **extend**. Unlike traditional database operators, Octopus operators are "soft" and with different algorithmic implementations can yield high- or low-quality results. We provide multiple algorithmic implementations for each and demonstrate experimentally that they are effective on an externally-generated set of test queries. For example, with just a few Octopus clicks, we were able to integrate five distinct source tables from three distinct websites to create a single output database, which contained 243 tuples; 223 of these were completely correct. The inputs were never intended for reuse; indeed, some were many years old.

1.5 Outline of the Dissertation

The next chapter provides a brief background description of three areas of research relevant to this dissertation: information extraction, Web text processing, and traditional data management. These are all huge areas of work, so we necessarily focus on areas of the literature where they have intersected to some degree. Chapters 3, 4, and 5 cover the above-mentioned TextRunner, WebTables, and Octopus systems in detail. We discuss future work and conclude in Chapter 6. Together, these chapters present a coherent set of research contributions and a vision for future work on the Structured Web.

Parts of this dissertation have appeared in other venues. I have had student collaborators in all of these publications, but I was the "lead" researcher for all but the TextRunner project with Michele Banko [6]. In that work, we were peer collaborators, though with different areas of focus. In TextRunner-related chapters I will discuss primarily my own contributions, distinguishing our work where appropriate. Banko continued to work separately on TextRunner in later publications [7, 8] and in her doctoral dissertation [5]. The TextRunner-specific material in Chapter 3 appeared in the IJCAI conference in 2007 [6]. WebTables was first described in a WebDB paper in 2008, and later the same year in a longer VLDB paper [22, 23]. The Octopus work appears in the 2009 VLDB conference [21]. Finally, many of the ideas for future work first appeared in CIDR papers from 2007 [24] and 2009 [19].

Chapter 2
BACKGROUND DISCUSSION

Research into structured data on the Web is a relatively new development, published across AI, database, and Web-oriented venues. In addition, this work requires navigating three relevant and very large bodies of work: traditional structured database management, Web and information retrieval work, and more recent information extraction efforts. In this chapter, we provide background and context for the research in this dissertation. In Section 2.1 below, we give a very quick summary of these three areas of work for readers who may be unfamiliar with one or more of them. Then in Section 2.2, we take a closer look at more recent systems that involve structured data on the Web, including discussions of how they relate to the systems and design criteria that make up this dissertation. Finally, we conclude in Section 2.3.

2.1 Research Background

There are three large areas of research relevant to the Structured Web data management we pursue in this dissertation. They include traditional structured database management, information retrieval, and information extraction. In this section, we provide an extremely brief overview of each area, and encourage readers to examine citations where appropriate.

2.1.1 Structured Data Management

Business data management is one of the oldest computer applications, and indeed mechanized business data management predates electronic computers entirely. The oldest working system that is a recognizably "modern" database may be IBM's IMS, released in 1968. The system is discussed in Stonebraker and Hellerstein's overview [78]. Academic database research found its first home in the SIGMOD database conference, which started in 1975. SIGMOD grew out of the SIGFIDET workshops, which originated in 1970.

Database research from the 1960s to the 1980s were marked by strong debates over the best way to represent and model data. Candidates included IMS' Hierarchical model, the network-based CODASYL model [28], and Chen's entity-relationship model [26]. By far the most successful, however, was the relational model first proposed by Codd [29]. The relational model, which stores data in a series of two-dimensional tables, was able to represent many common datasets more naturally than other models. All of the most successful commercial systems of the 1980s and 1990s - including Oracle and IBM's DB2 - were based on the relational model. Other models have been suggested, but the relational model has retained its broad popularity, incorporating some modifications over the years, such as user-defined functions. As discussed in Chapter 1, today the phrase "structured data" almost always refers to relational data.

One critical element of a relational database is its domain-specific schema. The relational schema describes what the dataset will contain - which fields, with which types, under which constraints. It is impossible to load data into a relational database without first describing the database's schema.

Hellerstein *et al.* [47] provide an excellent architectural overview of a modern relational database management system.

2.1.2 Information Retrieval and the Web

Information retrieval is the second large research area and focuses on finding unstructured text documents within a large corpus. Early retrieval systems from the 1960s, 1970s, and 1980s existed primarily outside of public view - they were used at universities, newspapers, governments, and similar institutions. As an area of academic study, the first SIGIR conference was held in 1978, having grown out of a workshop of the same name, founded in 1971.

IR systems in the early days were quite different from the Web search engines we use today. Bourne and Hahn give a useful overview of these early systems [15]. Due to expensive hardware and lack of large electronic document sets, corpus sizes tended to be very, very small by today's standards. Most systems indexed just document metadata such as titles, rather than full texts. However, the basic shape of today's systems could be seen - IR systems took user search queries (generally Boolean ones) and returned a list of hits on the managed document set. They did not require any domain-specific information, as did database systems. Probably the most notable academic advances from this earlier period were the vector-space document model and the *tf-idf* scoring mechanism [74].

The popularization of the Web in the mid-1990s, and the search engines that accompanied it, brought substantial changes to the field. First, the huge corpus sizes made ranking of results, as opposed to strictly Boolean searching, much more important than it had been previously. Second, partially also due to lower hardware costs, full-text indexing became popular and expected. Finally, the Web's hypertext graph added an extra source of non-textual information for ranking purposes; algorithms on this graph became a popular academic topic for many years, *e.g.*, Kleinberg [52], Page *et al.* [68], and Pirolli *et al.* [70].

Information retrieval is extremely relevant to this dissertation in at least one way - a modern search engine is the only truly popular piece of software that even attempts to process all of the content on the Web. However, the field has historically ignored domain-specific structured information. Thus we find that while Web-centric IR informs our work at the architectural level, it remains largely silent on structured data questions.

2.1.3 Information Extraction

Information extraction (IE) has been an active area of research since at least the early 1990s, when interest in IE grew out of the DARPA Message Understanding Conferences [80]. Muslea provides a good survey of early work, which focused primarily on traditional textual forms [65]. Example applications include gathering corporate intelligence or extracting facts from news articles, as in Riloff [73].

The first attempt to extract information from Web pages appears to be the Webfoot system in 1997 [77], and many systems soon followed. Brin's DIPRE project [18] took as input a user-given "seed" 2-tuple (*e.g.*, "Of Mice and Men", "John Steinbeck") and generated extraction patterns that allowed it to obtain additional examples from downloaded Web pages. The Snowball [1] system worked in a similar fashion, adding new methods for generating extraction patterns.

The Web has several unusual qualities that were already somewhat present in these early systems. First, the Web is large and topically diverse, so it demands domain-independent techniques; DIPRE and Snowball required small but real domain-specific user inputs to obtain a large database. Some of these early systems handled the Web's scale and relative low quality only indirectly, by using search engines to obtain useful input documents. In most cases, authors of these early systems simply did not address computational efficiency at all.

2.2 Related Projects

Recently there has been a large amount of relevant research into some aspect of the Structured Web. A system may draw heavily on one or more of the above-listed bodies of work, but each tries to address an issue particular to data on the Web. We describe these projects below, with some commentary on their relationships to the contributions of this dissertation.

2.2.1 KnowItAll

Etzioni *et al.*'s KnowItAll system [39, 40] performs information extraction from natural language Web text. Its main technique is to apply domain-parameterized extraction patterns to Web pages. For example, the existence of phrases like *Y such as X* and *X is a Y* provide some evidence that *X* is a hypernym of *Y*. When parameterized with a class *Y*, KnowItAll can then find all examples of *Y* on the Web. It operates by first using a search engine to find a large number of relevant documents, downloading the documents, and running the extraction phrases over these downloaded documents. Finally, it uses the search engine to compute how often each extraction appears, and thresholds out those extractions that have a low PMI-IR score.

KnowItAll is an extractor, not a full Structured Web management system. Extracting information from text requires a textual relation (domain) name, though not domain-specific rules or training data. Each new extraction task must be parameterized by a new domain name. Thus it is probably most accurate to say that KnowItAll is *domain-independent*, but not *domain-scalable* - whoever runs it must explicitly provide a domain name for each execution.

It is not very *computationally-efficient*. For the single-domain case, Cafarella *et al.*'s KnowItNow system [20] obtained substantial speedups. In any plausible Web system, the set of domains to be extracted will be substantial, making any system that relies on per-domain execution terribly slow. We compare KnowItAll with the TextRunner system in Chapter 3.

2.2.2 CIMPLE

DeRose *et al.*'s CIMPLE system [31] is a data integration system tailored for Web use, being designed to construct "community web sites." For example, the CIMPLE DBLife site [32] is a compilation of information about the database research community, including researchers, their papers, conferences, and so on. There are several commercial Web sites that would arguably make good targets for CIMPLE, such as Metacritic.com, which compiles movie and other media reviews from multiple independent sources.

A CIMPLE site consists of a series of human-chosen data sources, hand-designed information extractors, and engineered mappings between data schemas. CIMPLE tools are designed to assist when possible, but it is still expected that a trained administrator will spend a relatively large amount of time designing each site. After the initial setup, the site is kept up-to-date largely automatically.

CIMPLE uses a data-centric model of Web content in order to construct each community web site: information extractors consume unstructured pages and emit structured, domain-specific data. However, unlike the approach we have taken in all three projects in this dissertation, CIMPLE requires substantial administrator interaction to write an extractor for a given domain, although the needed effort may be lessened by approaches such as the iFlex system [75]. CIMPLE also requires the administrator to design each output website. Thus, while CIMPLE is *extraction-focused*, it has heavy per-domain administrative costs that render it only partially *domain-independent*. CIMPLE provides some assistance to the administrator in finding good data sources, given an initial seed set; thus CIMPLE is somewhat *domain-scalable*. Published work on the system does not address *computational efficiency* issues.

2.2.3 YAGO and NAGA

The YAGO information extraction system, due to Suchanek *et al.* [79], produces a very high-quality ontology of objects consisting of *is-a* and 13 other fixed binary relations. For example, the object `Paris` is in the FAMILYNAMEOF relation along with `Priscilla Paris`, and in the MEANS relation along with `Paris, France`. It extracts the ontology from WordNet [63] and the structured parts of Wikipedia, such as the specialized "list pages" that assign type labels to other Wikipedia entries. It does not attempt to process Web pages in general, and does not even attempt to extract information from Wikipedia article content - it focuses only on the Wikipedia list construct. The main contribution of the YAGO work is that it can use Wikipedia information to populate an (admittedly limited) ontology with a large number of objects while retaining high quality.

The system does not attempt to process unstructured sources directly, but can be extended using a more traditional textual information extraction system. The authors showed that the domain-specific Leila extraction system could be used to add HEADQUARTEREDIN links to the existing YAGO dataset.

YAGO is not a full Structured Web management system, and does not truly fulfill *any* of our design criteria. It does not attempt to operate at any serious scale, and it is limited to a tiny number of binary relations. Its input is highly specialized, so it is only barely *extraction-focused*.

However, YAGO suggests a route for extracting the Structured Web quite unlike the one we have pursued in this dissertation. The system begins with highly-structured inputs, and then can adorn the resulting data with unstructured extractions. In contrast, the extractors in our TEXTRUNNER and WEBTABLES projects use no *a priori* information at all about relevant relations or data items. It is intriguing to imagine a Structured Web extractor that starts with a YAGO ontology at its core, expanding to a full *domain-independent* dataset by obtaining Web text on each known object, extracting information from each page. It is somewhat reminiscent of work by Mansuri and Sarawagi [61], who increased the size of a preexisting relational database with text extractions, using the already-structured data to guide future extraction.

Kasneci *et al.* extend the YAGO work with NAGA [49, 50], a query system that operates over the extracted information. NAGA allows users to pose structured queries against the extracted ontology. Query-writers can use regular expressions to refer to data objects, so it is not necessary to know the precise name of a desired item. NAGA will also use information from a background corpus of text documents to assist in ranking its query results.

We can think of the combined YAGO-NAGA system as somewhat *domain-scalable*, because of NAGA's flexibility in finding items in its potentially-massive dataset. NAGA's authors do not discuss its computational efficiency. It appears that a combined YAGO-NAGA system can model and query any number of domains, but without a proposed and evaluated extractor, it is hard to say whether the system is *domain-independent*.

2.2.4 Freebase and DBPedia

Metaweb's Freebase system [14] is a "wiki"-style collaborative structured database, designed primarily for human updates. Unlike traditional structured databases but like the Structured Web, Freebase covers a broad range of general-interest topics, roughly similar to the topics contained in Wikipedia. It offers both a user-friendly Web interface as well as a structured query language called MQL.

Although Freebase data is highly-structured, it does not conform to the relational model: the basic unit in Freebase is a typed object, which can contain attribute/value pairs or links to other Freebase objects. This graph-based system draws on features of several different data models studied in the database literature, in particular the entity-relation model [78]. Even though Freebase has been designed for updates by a community of human beings, a large amount of its current data has been extracted from Wikipedia and added by Metaweb. This extraction mechanism lives "outside" the official Freebase system.

The DBPedia project [4] is roughly similar, though with a more explicit reliance on extracted data, in particular Wikipedia information. It also includes data generated by the above-mentioned YAGO system. DBPedia stores everything as RDF triples.

These systems have ambitions to be "general-purpose" structured databases, roughly along the same lines as our vision for the Structured Web as described in Chapter 1. They are *extraction-focused*, although in Freebase's case the extraction mechanism is not an inherent part of the system design. Their query systems are *domain-independent*, but DBPedia uses a series of domain-specific extractors (and Freebase's system is not documented). Freebase relies on per-domain data type designs contributed by users, whereas DBPedia extracts them from Wikipedia and other sources.

Neither system attempts to manage data from the entire Web, but instead uses just a subset (*e.g.*, Wikipedia data). Their query systems are not documented, so it is impossible to say anything about their *computational efficiency.* By adopting an admirable *object-finding* interface that takes a keyword query and returns candidate data objects; and by avoiding a query system that takes per-domain information to operate effectively, Freebase is highly *domain-scalable.* DBPedia offers something similar - text search over the RDF store to make object-finding easier.

2.2.5 Purple SOX

Bohannon *et al.*'s proposed Purple SOX extraction management system [13] is designed to run a large number of Web extractors at very large scale. In particular, Purple SOX offers extractor developers a number of features. First, it offers a declarative extractor language that can be enhanced with developer-defined operators; this declarative language can also be executed and optimized automatically, much as SQL can be. Second, it keeps extensive lineage information for each extracted fact, so a developer can debug the output dataset. Finally, it maintains a score to represent the system's belief that a given extraction is true, which score can be adjusted using a variety of information sources (including socially-contributed information). The output of Purple SOX is an RDF-like triple store.

Purple SOX is clearly *extraction-focused*, though not *domain-independent* or *domain-scalable* - its entire goal in supporting developer-friendly features is to lower the bar to human-generated domain-specific information extractors. Purple SOX has not yet been completed and evaluated, so it is impossible to say whether its declarative extraction system will be *computationally efficient*.

2.3 Conclusion

There has been an explosion of interest in managing Structured Web data, combining work from several different areas of research. All of the systems discussed here have the goal of being as comprehensive as the Web, and they share many qualities in common. They claim to be computationally scalable, although there is little published information about their techniques or results. All of them (except, perhaps, Purple SOX) acknowledge that a user interacting with vast amounts of structured data cannot be expected to uniquely identify each data item, and thus offer assistance for locating data and/or relevant Web pages.

There is one way in which the systems differ strongly, and that is how they plan to populate their datasets. KnowItAll is focused entirely on automated domain-independent extraction, for a single domain at a time. Purple SOX and CIMPLE rely heavily on an administrator or developer. Freebase appears to want its user population to type in facts by hand, but in practice also relies heavily on what seem to be dataset-tailored extractors. YAGO/NAGA and DBPedia, meanwhile, rely on domain-dependent extractors; the published work on these systems does not offer any ideas on how to eventually process all information on the Web.

The work in this dissertation offers a solution for acquiring that is different from all of the above systems, that of using domain-independent information extraction. In the next Chapter, we will discuss our first domain-independent extractor, TextRunner, which operates over natural language text crawled from the Web.

Chapter 3
WEB-SCALE TEXTUAL EXTRACTION

Natural language text makes up a large and important portion of information on the Web, and extracting structured data from text should be a critical part of any Structured Web system. Unfortunately, traditional textual information extraction projects have generally attempted to gather data for specific predefined topics, using as input relatively small and homogeneous corpora. For example, an extractor might obtain the locations and times of academic seminars from a set of departmental email announcements. Changing the target domain has usually meant a substantial amount of work for a human, who must create new extraction rules or training data. Further, previous systems have often computationally scaled poorly with either the number of documents or the number of target topics. In short, previous textual extraction efforts have generally failed our design criteria of *domain-independence*, *domain-scalability*, and *computational efficiency*, and thus are not suitable for processing a Structured Web of billions of documents and a vast number of domains.

This chapter introduces TextRunner, an *Open Information Extraction* system that is *domain-independent*, *domain-scalable* and *computationally-efficient.* Most of the information in this chapter can be found in Banko, Cafarella *et al.* [6], a project in which Michele Banko and myself were the lead researchers. My contributions centered on TextRunner's architectural differences with previous IE systems, whereas Banko worked mainly on the NLP-based heuristics and features and the analysis of fact quality. This chapter will thus focus primarily on the system's architectural advantages. However, TextRunner cannot be understood without NLP-related details and an examination of the extraction quality and so we also discuss work done primarily by Banko where appropriate. We also separately cite her work on TextRunner if done outside the scope of our collaboration.

Unlike previous IE systems, TextRunner processes its input corpus in a single pass, without any per-domain training data or human guidance. Also, TextRunner is designed to have a complexity linear in the number of input documents, with a low constant factor, making it appropriate for processing input corpora of billions of Web pages. We show that on a test set of domains, our system can reduce extraction error by 33% when compared to KnowItAll, a high-quality non-open IE system mentioned previously. In addition, we demonstrate that TextRunner can process all of the domains in a corpus in several orders of magnitude less time than KnowItAll. By running it on a corpus of 9M Web pages, we were able to obtain 1M concrete facts. These attributes make TextRunner a valuable system for processing the portion of the Structured Web data embedded in natural language text.

3.1 Problem Overview

The large volumes of text on the Web, and the large number of topics that are covered by that text, pose a serious challenge to information extraction systems. IE projects have traditionally been applied to relatively small corpora, focusing on a small number of domains. They have generally required extensive human input for each domain, scaled poorly with the size of the corpus, and often suffered from brittle output quality when run over heterogeneous document collections. In all of these qualities, previous IE systems have been thoroughly inappropriate for processing text on the Structured Web, with its huge number of documents and a large and unknown number of topics.

Traditional IE systems' need for domain-specific hand-crafted data (whether in the form of training data, or extraction rules, or simply the target topic name) has implications for our Structured Web criteria. Many such systems are not *domain-independent* - they can only extract data for a single subject matter, or they require per-topic training data. Others may be domain-independent, but are not *domain-scalable* because they require a small amount of human input for each topic processed, even if just the topic name itself.

These qualities relating to processing many domains can lead to problems with *computational efficiency*. First, while the number of topics on the Web is hard to state concretely, it is undoubtedly large; as we describe in Section 3.4 below, we found 278,085 distinct relation strings (*e.g.*, `born-in`) in a 9M-page Web crawl. Compiling hand-made data for each of these strings would be extremely burdensome, perhaps prohibitively so. At any rate, even enumerating the target topics in advance undermines one of our goals with the Structured Web, which is to not just manage structured data that we know is on the Web, but to discover new topics as well.

Even in cases where it is possible to acquire human-created data for each topic, running the system on the entire Web's worth of topics now makes the overall extraction step scale with the number of domains, in addition to the extraction mechanism's existing complexity. Thus, if we treat the number of topics T as a parameter to consider in complexity analysis, then a single-domain extractor that is linear in the number of documents D goes from $O(D)$ to $O(D * T)$. If we treat the number of topics as large but fixed, then a single-domain extractor's algorithmic complexity stays at $O(D)$ but with an enormous added constant factor.

Traditional IE systems' heavy reliance on linguistic techniques has two ill effects. First, many linguistic technologies, such as dependency parsers and named-entity recognizers, are generally very computationally-heavyweight. These techniques were designed primarily for running over relatively small corpora (say, a set of newspaper articles), which obviously does not hold on the Web. This adds a high constant-time factor to the extractor's overall complexity. Second, these linguistic systems were themselves trained on relatively small and homogeneous text collections, so they can fail at higher rates when processing Web documents, which are diverse in type and in quality.

To avoid all of these problems, in this chapter we introduce TextRunner, an *Open Information Extraction* system that takes just the downloaded Web corpus as input, makes a single pass over that input, and emits extracted structured tuples as output. It takes no per-domain data of any kind. It is also designed to use only relatively low-overhead linguistic tools. In so doing, TextRunner meets our Structured Web criteria of *domain-independence*, *domain-scalability*, and *computationally-efficiency*. And as an extractor, it is of course *extraction-focused*.

Our goal for a textual extractor is to take a corpus of text as input and yield a set of accurate triples, each of the form $t = (e_i, r, e_j)$. In this tuple e_i and e_j represent real-world *entities* and r represents a binary *relation* that links the entities together. For example, the phrase "Einstein was born in Germany" might give rise to (`Einstein`, `was-born-in`, `Germany`). In this case, `Einstein` and `Germany` are entities, while the relation `was-born-in` links them together.

Other possible examples include (`Edison`, `invented`, `phonograph`), (`Paris`, `is-capital-of`, `France`), and so on. The relation r is the topic or domain of the extracted tuple. It is easy to imagine extracting not just binary relations, but 3-ary or n-ary ones, and emitting an appropriately larger tuple. *E.g.*, we might obtain (`Einstein`, `was-born-in`, `Germany`, `1875`). We focus only on binary relations, and thus 3-ary tuples, in this work.

The decision about whether or not to fix T for the sake of complexity analysis depends on assumptions about the input corpus. If the set of interesting relations can be identified easily ahead of time and is relatively small, then T can be fixed without any problem. If the set of relations grows rapidly over time, it may be more useful to retain T as part of the complexity analysis.

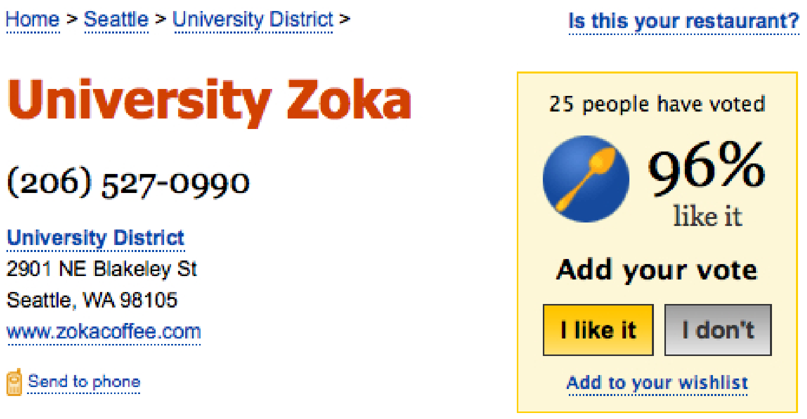
In the next section we will discuss in detail a number of previous IE projects and describe why none of them meet our needs for the Structured Web. Then in Section 3.3 we describe the TextRunner architecture in detail. We evaluate the system in Section 3.4, and then conclude in Section 3.5.

3.2 Related Work

There has been a substantial amount of research into textual information extraction, a few of which we describe here. As will become clear, we are unaware of any system that is both *domain-independent*, *domain-scalable*, and *computationally-efficient*.

3.2.1 Wrapper Induction

Wrapper-induction is a popular research area that attempts to extract structured tuples from text which is sometimes (confusingly) called *semi-structured*, although this sense of the word has nothing to do with XML data. A *wrapper* is a function that consumes semi-structured text and emits a structured extraction; *wrapper-induction* studies how to create these wrappers automatically or semi-automatically. Semi-structured text is generally text that has been generated algorithmically using information from a backend database. Examples might include product listings at Amazon.com that arise from a product database,


Home > Seattle > University District >

Is this your restaurant?

University Zoka

(206) 527-0990

University District
2901 NE Blakeley St
Seattle, WA 98105
www.zokacoffee.com

Send to phone

25 people have voted

96%
like it

Add your vote

I like it
I don't

Add to your wishlist

Figure 3.1: A sample page from Urbanspoon.com, and a good candidate for extraction by a wrapper-induction system.

or a list of available flights at Southwest.com that is backed by a reservations database. Figure 3.1 shows another example, a portion of a restaurant listing from Urbanspoon.com; in this case, a successful wrapper-induction system might obtain a tuple (`University-Zoka`, `206-527-0990`, `University-District`), for the restaurant's name, phone number, and neighborhood. Examples of work in this area include projects by Kushmerick *et al.* [54], and Muslea *et al.* [66], among others.

Wrapper-induction methods in general require new training data for each new domain or target website. For example, given a training set of restaurant-name/phone-number pairs which match entries on the Urbanspoon.com website, it may be possible to learn the site's general HTML templates that are used to format display restaurant names and phone-numbers. By using the learned templates, the system can then extract information from many novel pages on the site. Each topic requires a hand-made set of training data, so the induction systems can learn the relevant site-specific templates.

Thus, wrapper-induction methods are inappropriate for the entire Structured Web in two related ways. First, they are not fully *domain-independent*, as they require training data for every distinct target topic. Second, even if we had this data, we would still need to invoke the system T times. Assuming T is not fixed, we obtain a time complexity of at least $O(D*T)$. Thus, wrapper-induction fails the *computational efficiency* criterion as well.

3.2.2 Extraction Rule Learning

Extraction rule learning systems are roughly similar to some wrapper-induction methods, but can work on natural language text. Examples include the previously-mentioned DIPRE [18] and Snowball [1] systems. The goal of such systems is to take a small structured database as input, then expand the dataset using information extracted from Web text. These systems work by first locating examples of the current database tuples in Web pages, then learning linguistic patterns that appear to relate the fields of each tuple. For example, given a small dataset of company/headquarters pairs, an extraction rule learner might induce **X***'s headquarters in* **Y**, where **X** and **Y** are the extracted items.

The weaknesses of such an approach should be clear. Although these systems can process natural language text, they still require hand-made training data for each new topic, thus giving an algorithmic complexity of at least $O(D*T)$ when T is not fixed. Such systems are neither *domain-independent* nor *computationally-efficient*. It is not clear how *domain-scalability* applies to these systems.

3.2.3 KnowItAll

Etzioni *et al.*'s KnowItAll system [39] was discussed in Chapter 2. However, it must be modified to extract *all* of the topics in a corpus, not simply the one given by the user. In such a scenario, KnowItAll is invoked once for each domain, and must process all of the corpus documents for each invocation. Such an execution model yields an algorithmic complexity of $O(D*T)$ if the number of topics is an input to the system. If the number of topics is fixed, then KnowItAll obtains an algorithmic complexity of $O(D)$, but with a huge constant-time factor.

3.2.4 Unrestricted Relation Discovery

The work that comes the closest to TextRunner is probably that of Shinyama and Sekine [76], on "unrestricted relation discovery." Unlike most other efforts, their system is not topic-specific - it processes the entire corpus at once. Unfortunately, it uses a number of computationally-heavyweight linguistic components. Worse, it entails a document clustering step that requires time $O(D^2)$. So while Shinyama and Sekine's system does satisfy *domain-independence*, it is not even close to being *computationally-efficient* enough for the entire Web corpus.

As the above discussions should show, there has been not yet been a textual extractor that fits our requirements for textual Structured Web processing. In the next section, we discuss the architecture that allows TextRunner to meet them.

3.3 System Architecture

This section describes TextRunner's overall architecture, which allows us to meet the above-mentioned criteria. Recall that the goal is to start with a large Web crawl as input and extract as many accurate fact triples as possible.

There are three main components to the system:

1. The **Self-Supervised Learner** generates a classifier that labels an input candidate extraction triple as "good" or "bad." The Self-Supervised Learner takes as input just a small "training" portion of the crawled corpus; it does not require any human annotation of the input. In order to train its output classifier, the Learner must build a training set from its input training corpus, using the linguistic heuristics listed in Table 3.1 to obtain a "good" or "bad" label for each extraction in the training corpus. Because these heuristics require a full linguistic parse, they are fairly computationally intensive. However, the overall runtime impact of this stage is relatively small because relatively few documents are parsed.

2. The **Single-Pass Extractor** goes through each sentence in the *entire* input crawl, generating all candidate extractions for each sentence, then applying the previously-trained classifier to classify candidate as "good" or "bad." The inputs to this classifier, a large sample of which are listed in Table 3.2, are lexical and syntactic features of the candidate extraction and its source sentence. These features are very inexpensive

to compute, as they do not require a linguistic parse. Bad extractions are discarded; good tuples make it to the next step of processing.

3. The **Redundancy-Based Assessor** uses a tuple's frequency of extraction in order to compute a probability that the tuple is true. The probabilistic model we use is due to Downey *et al.* [37]. Any tuple with probability under a certain threshold is discarded.

We can now examine each of these components in turn. With each section below, we also discuss a small running example of how TextRunner eventually comes to emit the tuple (`Edison`, `invented`, `phonograph`).

3.3.1 Self-Supervised Learner

The goal of the Self-Supervised Learner is to create a classifier that will be applied to the entire TextRunner input corpus by the Single-Pass Extractor. This classifier takes a candidate tuple extraction and its source sentence as input and emits a simple "good" / "bad" verdict, describing whether the tuple is an accurate extraction from the text.

One way to build TextRunner would be to perform a deep linguistic parse of each sentence in the entire input crawl, inducing an extraction candidate from every plausible combination of *entity* and *relation* strings in the resulting parse. (For example, `Edison` and `phonograph` are both nouns and thus likely entity strings while the verb `invented` is a likely relation.) TextRunner could then test to see if the parsed sentence meets certain hard-coded domain-independent linguistic criteria that tend to describe a high-quality extraction. For example, if the two entities are not in the same clause, the extraction is not likely to be useful. Table 3.1 below lists the heuristics that would be appropriate when evaluating a candidate extraction. Depending on what the linguistic heuristics indicate, the system could emit "good" or "bad" as appropriate.

Unfortunately, because deep linguistic parsers have tended to be extremely computationally intensive, running the above system on the entire Web would be very burdensome. Further, as mentioned previously, these parsers have tended to be brittle when running on

The number of sentence tokens is not unreasonably large
The relation string contains a verb
The two entities appear at the same level of the parse tree
The two entities' lowest common parse tree ancestor is a clause
The left-hand entity is the clause subject
The left-hand entity is not the object of a prepositional phrase
The two entities do not belong to different clauses
The right-hand entity is not the object of a prepositional phrase
If the right-hand entity is the object of a prepositional phrase, it serves one of a handful of semantic roles regarding time, place, *etc.*.

Table 3.1: Boolean heuristic tests that indicate good extractions, used by the Self-Supervised Learner to construct a classifier training set. The Learner assumes that an expensive full parse tree is available. The full hand-designed decision algorithm that employs these tests can be seen in Figure 3.3 of Banko's dissertation [5].

the kind of non-standard text that is very common on the Web.

Rather than use the parser directly, we will instead use it to generate a training set for a Naive Bayes classifier (NBC). This classifier takes as input a series of lexical and syntactic features that can be computed over a candidate extraction and its source sentence. These features give some evidence as to whether an extraction candidate is accurate or not. For example, a capitalized entity string may indicate that it is a proper noun and thus a strong candidate for a legal entity. A large sample of which is shown in Table 3.2, which are very inexpensive to compute. The output of the NBC is the same simple "good"/"bad" verdict as the heuristics-driven approach described above.

TextRunner runs the parser (due to Klein and Manning [51]) on several thousand randomly-drawn sentences from the corpus. Each of these sentences yields at least one extraction candidate. For each candidate/sentence pair, we compute the heuristic-driven label and the inexpensive feature vector. We then use these labeled vectors to train the NBC. The parser is computationally expensive, but running it on a fixed-size input does not

The number of tokens in the relation string
The part-of-speech sequence in the relation string
The number of stopwords in the relation string
The punctuation sequence in the relation string
Whether the left-hand (right-hand) entity is a proper noun
Whether the left-hand (right-hand) entity is capitalized
The part-of-speech for the word to the left of the left-hand entity
The part-of-speech for the word to the right of the right-hand entity

Table 3.2: A portion of the features used as input to the extraction classifier. Unlike the heuristics in Table 3.1, none of these features require a linguistic parse in order to be computed. Instead, they only require low-overhead operations. Examples include regular expressions for testing capitalization, noun phrase chunking (*e.g.*, "light bulb" acts as a two-word noun), and part-of-speech lookups (*e.g.*, `phonograph` is a `noun`). The intuition is that these inexpensive local features give much of the discriminatory power of a full parse tree at much lower computational cost.

contribute to either TextRunner's algorithmic complexity or substantially to its practical running length. This can be seen in our experimental results, in which TextRunner spends an average of 0.036 CPU seconds per input sentence, even though a parser takes an average of 3 seconds per sentence.

By applying heuristics only to linguistically-derived features, the self-supervised learner step avoids using topic-specific data of any kind. It is, however, heavily reliant on NLP systems, so this step would not translate easily to languages where NLP technology may not be available.

The use of a small portion of the corpus for training the classifier could in theory pose a problem if it does not linguistically reflect the overall corpus. For example, imagine if the training corpus documents were drawn mainly from scientific articles while the overall corpus was mainly Web documents. It is possible that certain linguistic features could be present in the training corpus but absent in the overall corpus, and vice versa. If this were true, it would yield a very strange and ineffective classifier for the next TextRunner step.

To minimize this possibility, we draw training sentences wholly at random from the overall corpus.

The Naive Bayes classifier has two advantages. First, because such classifiers are resistant to noise in the training data, and because we are using a wide variety of lightweight features, the classifier should be comparatively better at handling mis-parses than simply examining the raw parse tree. Second, NBC systems are very quick to execute; the output of this step will enable us to efficiently process the entire corpus. It is easy and natural to fix the number of tokens in a natural language sentence to a relatively small constant, so choosing an NBC approach over the original parser-only approach gives a constant-factor, though substantial, speedup.

Running the NBC over the entire corpus text is the subject of the next section.

In-Depth Example

The input to the Self-Supervised Learner is a small section of the Web crawl, which is then used to construct a training set for the output classifier. Building a high-quality training set is the main challenge of this stage. Let us examine two sentences that might be found in the corpus: "Edison invented the phonograph" and "Von Neumann met Einstein's wife while teaching in New Jersey."

For each sentence, the Learner considers all possible tuple extractions. For each extraction, it computes a feature vector and uses heuristics to classify the extraction as good or bad. For example, the extraction (`Edison`, `invented`, `phonograph`) and the parse tree for "Edison invented the phonograph" demonstrates a typical subject-verb-object pattern, indicating a good extraction. For the sake of this example, assume that the overall set of parse-tree heuristics marks this extraction as good.

In contrast, consider the extraction (`Von Neumann`, `met`, `New-Jersey`) and the parse tree for "Von Neumann met Einstein's wife while teaching in New Jersey." The extraction has two objects that cross a clause boundary in the parse tree, matching a heuristic that indicates a bad extraction. Assume that this extraction is marked as bad.

We now have the verdicts for each of these sentences, and only have to compute the inexpensive features for each. For example, we test whether the entity objects are proper nouns (true in all four cases here), count the number of tokens in each relation string (3 for `was-born-in`, 1 for `met`), and count the number of stopwords in each relation string (2 for `was-born-in`, 0 in `met`). The resulting vectors of features and verdicts, one for each sentence in the small corpus sample, forms the training set for our Naive Bayes classifier. TextRunner uses this training set to create the classifier and then moves to the next stage.

3.3.2 Single-Pass Extractor

Once the self-supervised learner has emitted the classifier, we can process the entire corpus. TextRunner makes a single pass over the input corpus, breaking the text into sentences. For each sentence, the single-pass extractor performs the following steps:

1. Run a part-of-speech tagger and noun-phrase chunker over the sentence. These lightweight steps help us identify each of the sentence's *entities*. These two components, drawn from the OpenNLP toolkit, are computationally lightweight. Part-of-speech tagging and noun phrase chunking can be performed with high accuracy across topics and languages (as in Brill and Ngai [17] and Ngai and Florian [67]), so we can handle diverse material from the Web crawl effectively. For each word in a noun-phrase chunk, we also receive a probability that the word is part of the noun-phrase.

2. Compute all tuple extraction candidates from the sentence by finding all entity pairs plus the relation text string that falls between them.

3. Simplify the candidate by removing non-essential phrases and tokens. For example, the relation *definitely developed* is transformed to *developed*.

4. Throw out extraction candidates where the constituent entities have very low probability

5. Test the candidate with the previously-trained NBC, using lightweight features mentioned in the previous section. Discard extraction candidates that are classified as

"bad."

6. Append to a file all of the extracted tuples that have survived.

Because the average number of sentences per Web page does not change greatly over time or with repeated executions of TextRunner, we can say that the above sequence of steps is executed $O(D)$ times, where D is the number of crawled documents. For a sentence of k extracted entities, there are a maximal number of $\binom{k}{2}$ candidate tuples; in practice, real sentences contain few entities and many of the maximal set of candidate tuples fail to pass the above-listed tests. At any rate, as mentioned in the last section, it is reasonable to limit the number of tokens in a valid sentence and thus give fix the number of candidate tuples per sentence. Part-of-speech tagging and noun phrase chunking are not expensive operations, but their runtimes are also fixed by placing a limit on the sentence size. Thus, we can consider there to be a constant number of candidates per input sentence, and the overall complexity of the single-pass extractor to be $O(D)$.

Note that this step relies heavily on the trained classifier and some linguistic tools, but it requires no domain-specific data or human assistance.

By the end of the single-pass extractor we have obtained large number of tuples that we believe are accurate. However, given the diversity of Web text and our imperfect linguistic methods, it is inevitable that some inaccurate tuples have been retained. The next step is designed to further refine extraction quality.

In-Depth Example

Using the Naive Bayes Classifier from the previous step, we can now issue a good/bad verdict for each candidate extraction in the entire corpus. For example, after breaking the text at sentence boundaries, assume that we are faced with three sentences: "Edison invented the phonograph," "Edison reinvented Mankind," and "Morgan was frustrated with the progress of Edison's work."

For each of these sentences, we follow the same steps: find any candidate extractions, compute the inexpensive feature vector mentioned above, and then classify the extraction as good or bad using the NBC. For "Edison invented the phonograph" and the extraction (`Edison`, `invented`, `phonograph`), the resulting lightweight feature vector will probably elicit a "good" verdict from the NBC.

The sentence "Edison reinvented Mankind" contains the tuple (`Edison`, `reinvented`, `Mankind`). Despite this tuple's questionable worth and accuracy, it has many of the same features as a good extraction and probably also earns a "good" verdict from the classifier.

The sentence "Morgan was frustrated with the progress of Edison's work" and the tuple (`progress`, `of`, `Edison's work`) has some negative lightweight features, *e.g.*, a relation string that consists of just a single stopword. Thus, we can imagine it receives a "bad" verdict from the classifier.

This step thus allows the first two extractions to pass to the third and final stage, while throwing away the third extraction. After applying the classifier to every sentence and extraction in the corpus, TextRunner can execute the final step.

3.3.3 Redundancy-Based Assessor

TextRunner is able to generate a trustworthy probability for each extracted tuple using Downey *et al.*'s URNS method based on frequency of extraction. The intuition behind this system is that true tuples will appear in many sources and thus be repeatedly extracted; in contrast, false or incorrectly-extracted tuples will appear only rarely and thus be extracted infrequently. Using this method to find probabilities for the extracted tuples requires that we first compute how many times each unique extraction was obtained. TextRunner performs some simple normalization of extracted tuples, dropping unnecessary adjectives and adverbs (*e.g.*, converting the relation *was originally developed by* to *was developed by*).

The main contribution of the URNS model is not just to emit a probability that increases with observed extraction frequency, but to compute a probability that is also accurate. It is a "balls-and-urns" combinatorial model, in which a single textual extraction is modeled as a draw of a labeled ball from the urn. A labeled ball may represent a correct or an incorrect extraction. An information extraction system uses the model to determine the probability that an extraction is correct, given that it was observed k times in n draws from the urn. An urn is parameterized by the set of unique correct labels C, the set of unique error labels L, and a function $num(b)$ that gives the number of balls with a label b. With these parameters and an observed k, n, and b, it is possible to compute the probability that $b \in C$.

On the Web the model's parameters cannot be known ahead of time - for example, it is difficult or impossible to say how many correct extraction labels there are. Downey *et al.* estimate them using an expectation-maximization algorithm.

Having computed a probability for each tuple, we apply a threshold to remove tuples that do not meet a minimal level of plausibility. For the work described in this chapter, we chose a minimal probability of 0.8.

Prior to running the assessor, we sort the raw extracted tuples in order to convert them into a set of unique tuples with frequency counts. This adds a time complexity element of $O(T log T)$, where T is the number of raw extracted tuples.

In-Depth Example

There were two unique tuples from our running example that survived the previous step: (`Edison`, `invented`, `phonograph`) and (`Edison`, `reinvented`, `Mankind`). There is no necessary normalization for either tuple.

We imagine that the first tuple, (`Edison`, `invented`, `phonograph`), is commonplace on the Web and appears hundreds of times. When this count is entered into the probabilistic frequency model, we obtain a probability well above the 0.8 threshold. In contrast, for the sake of this experiment we can say that the tuple (`Edison`, `reinvented`, `Mankind`) is repeated just a single time, and so receives a probability well under 0.8.

Hence, the third and final TEXTRUNNER step will emit (`Edison`, `invented`, `phonograph`) as a valid extraction, but not (`Edison`, `reinvented`, `Mankind`).

3.3.4 Tying It All Together

We have now seen how TEXTRUNNER can extract information from *all* of the topics in a corpus of text while remaining *domain-independent*. By focusing only on linguistically-based heuristics and frequency-counts to distinguish good from bad extractions, we avoid requiring any domain-specific data. By using a Naive Bayes system, it also skips the brittleness associated with many NLP deep parses.

We can also see how TextRunner satisfies the *computational efficiency* criterion - the runtime of TextRunner is the sum of the second and third steps, $O(D + TlogT)$ (or simply $O(D)$ with a huge hidden constant). Because there is no interaction with the user - TextRunner is a batch job that takes the Web as its only input - interface-driven *domain-scalability* appears not to apply here.

The TextRunner design also has a number of limitations. First, it is strongly language-dependent. For example, TextRunner currently requires a full parser, part-of-speech tagger, and noun phrase chunker for the target language. Only the absolutely most-popular languages enjoy full linguistic technology support, so TextRunner may not be able to run over many interesting natural languages. Worse, the extraction-quality heuristics and the lightweight lexico-syntactic features may need to be rewritten for each new target language. Rewriting these heuristics may not even be possible - for example, a language in which the subject of each sentence is rarely or never named will be difficult for TextRunner to process. Addressing the issues surrounding alternate languages is a rich area for future work, which we discuss in Chapter 6.

Second, this strong tie to the source language limits TextRunner's expressiveness even in English, a language that lacks for nothing in terms of NLP technology support. Although all of our examples so far have focused on triples, it would be useful if TextRunner could easily obtain tuples of much greater cardinality. For example, instead of (`Edison`, `invented`, `phonograph`), we might have (`Edison`, `invented`, `phonograph`, `1879`, `Menlo Park`). There is nothing in the TextRunner design that prevents it from extracting such high-cardinality tuples, and indeed we have successfully collected some from the Web crawl. However, as the number of extractable data items increases, so does the complexity of the English language sentence needed to express them.

By requiring that all extractions appear within a single sentence, we practically limit the complexity of the data items that TextRunner even attempts to extract. For tuples of high-enough cardinality, there will be very few or no appropriate sentences on the Web. Non-linguistic extraction regimes (such as wrapper-induction systems) are not limited to what English commonly puts into a sentence. Alternatively, it may be possible to extract data that spans multiple sentences, at the cost of increased difficulty in tracking objects and facts across sentence boundaries.

Even with these limitations, it is still an open question whether TextRunner's architecture can actually obtain high-quality results. This issue is the subject of the next section.

3.4 Experiments

To evaluate TextRunner's output quality, we compared it to the above-mentioned KnowItAll, a modern unsupervised Web extraction system that still requires per-domain information from the user. We also evaluated the standalone accuracy of TextRunner's output. Finally, because one of the main goals with TextRunner is to obtain a *computationally-efficient* text extraction system, we examined its runtime performance.

3.4.1 Output Quality

One way to evaluate TextRunner's domain-independent output is to compare it with a domain-dependent system, such as KnowItAll. KnowItAll requires that a human contribute at least the name of each target domain (relation) it attempts to extract. We ran both KnowItAll and TextRunner on the same corpus of 9M crawled Web pages.

KnowItAll requires that we enumerate the target domains prior to execution. We randomly chose 10 relations (listed in Table 3.3) that were found in at least 1,000 sentences in the corpus. We manually removed vague relations such as "includes."

Table 3.4 shows the average error rates for each system, and the total number of correct extractions for each. TextRunner achieves an average error rate that is 33% lower than KnowItAll, while finding a near-equal number of correct extracted tuples. Most of TextRunner's improvement over KnowItAll comes from being able to better identify the "endpoints" of each relation string. Thus, even when we know the targets *a priori*, the TextRunner open IE system has better output quality than the closed KnowItAll system.

X *acquired* **Y**
X *graduated from* **Y**
X *is author of* **Y**
X *is based in* **Y**
X *studied* **Y**
X *studied at* **Y**
X *was developed by* **Y**
X *was formed in* **Y**
X *was founded by* **Y**
X *worked with* **Y**

Table 3.3: Ten randomly-chosen topics/domains/relations for comparing TextRunner and KnowItAll.

	Average Error Rate	Correct Extractions
TextRunner	12%	11,476
KnowItAll	18%	11,631

Table 3.4: Output quality results for TextRunner, compared to KnowItAll. TextRunner does not know the target domains in advance, unlike KnowItAll. Nonetheless, TextRunner obtains an average error rate that is 33% lower than KnowItAll's, while extracting a near-equal number of correct tuples.

A second way to evaluate output quality is simply to evaluate the entire set of TextRunner outputs, without pre-choosing the target topics. We randomly selected four hundred tuples from the overall 11.3M output set to manually test by three different human judges. The judges examined each tuple and answered several questions:

- Is the tuple's relation string "well-formed?" *I.e.*, is there some pair of entities that could possibly be valid for the relation string? For example, `specializes-in` is a well-formed relation, whereas `of-securing` is not. Of the 11.3M output tuples, 9.3M have a well-formed relation.

- If the relation is well-formed, do the tuple's entities plausibly fit with the relation/topic (even if falsely)? For example, (`Fred`, `dogcatcher`) is a plausible pair of entities for `was-elected`. The entity pair (`29`, `company`) is not. Of the 9.3M well-formed relation tuples, 7.8M have well-formed entities.

- Of the tuples that pass the above two steps, the judges found that 80.4% are correct.

- If both of the above tests are passed, is the tuple *concrete* or *abstract*? A concrete tuple is one where the truth of the statement can be grounded in the present entities, *e.g.*, (`Einstein`, `was-born-in`, `Germany`). An abstract tuple might be (`Einstein`, `invented`, `theory`). Abstract tuples may initially appear worthless, but they could be used for ontology learning or some similar model-learning task. The judges found that 14% of tuples with well-formed relations and entities are concrete, with the remainder being abstract.

- The concrete tuples were correct 88.1% of the time, whereas abstract tuples were correct 79.2% of the time.

We thus see that not only does TextRunner's output compare favorably to a more traditional closed IE system when tested on a controlled set of topics. TextRunner also retains high output quality across tuples that are selected at random, a quality that is very useful when attempting to extract all of the topics on the Structured Web.

In separate work, Banko found that extra redundancy had a small but real impact on extraction quality [5]. Extracted tuples with at least two appearances in a different general Web crawl had precision of 84.3%, while tuples that appeared at least 5 times had precision 86.9%, tuples of frequency 50 or higher had precision of 88.8%, and tuples of frequency 200 or higher had precision 92.3%. Further, there is some evidence that when using high-quality corpora (*e.g.*, Wikipedia) it is possible to obtain high precision extractions without any repetitions at all.

3.4.2 Runtime Performance

Even superb output quality is not useful to us unless we can run it at very large scale. We evaluated runtimes of TextRunner and KnowItAll using a small cluster of 20 nodes, on the same corpus of 9M Web pages.

Running TextRunner on our test corpus extracted every relation and took a total of 85 CPU hours to complete. KnowItAll, meanwhile, required 6.3 CPU hours for each relation (domain). There may be some application scenarios in which we know the set of target relations ahead of time; in these cases, it may be advantageous to choose KnowItAll. However, it only takes 14 targets before TextRunner is the faster choice.

When processing the Structured Web, we assume we do not know the set of relations ahead of time. After running TextRunner on the test corpus, we found 278,085 distinct relation strings among extracted tuples that enjoy high probability and at least 10 distinct source sentences in the corpus. If these relations make up the target set, then we estimate that KnowItAll would take 1,751,400 CPU hours to complete; in this case, TextRunner is four orders of magnitude faster.

Of course, not all of these relations are actually distinct topics. Banko's later work [5] estimated there to be 10,000-20,000 textual relations, based on the number of entries in hand-crafted resources such as WordNet [63]. If we assume there are 10,000 relations, then the KnowItAll running time is still 63,000 CPU hours; that is two orders of magnitude larger than the TextRunner time.

3.5 Conclusions

We introduced TextRunner, a natural-language-text information extraction system that is *domain-independent* and *domain-scalable* and can obtain all of its extractions in a single pass over the input corpus. In doing so, TextRunner achieves a performance advantage over competing systems of several orders of magnitude, while retaining comparable or better extraction quality, and thus showing that TextRunner is *computationally-efficient* as well.

Although TextRunner performs admirably in its goal of domain-independence, it has a serious weakness which this dissertation does not address in depth. It is strongly language-dependent. TextRunner currently requires a full parser, part-of-speech tagger, and noun phrase chunker for the target language. Only the absolutely most-popular languages enjoy full linguistic technology support, so TextRunner may not be able to run over many interesting natural languages. Worse, the extraction-quality heuristics and the lightweight lexico-syntactic features are somewhat language-dependent, and may need to be rewritten for each new target language. Rewriting these heuristics may not even be possible - for example, a language in which the subject of each sentence is rarely or never named will be difficult for TextRunner to process. Addressing issues surrounding alternate languages is a rich area for future work, which we discuss in Chapter 6.

Despite these weaknesses, TextRunner meets our design criteria for natural language text on the Structured Web. In the next chapter, we move beyond text to Web-embedded tabular structures.

Chapter 4
TABLE-ORIENTED DATA EXTRACTION AND MINING

One notable form of data on the Structured Web is that of HTML tables. The HTML table tag can be used to present a two-dimensional relational-style data table. Because each such relational table has its own "schema" of typed and possibly-labeled columns, it can be considered a small structured database. The scale of the Web enables us to collect a massive number of individual databases - from 14.1 billion raw HTML tables from a general-purpose crawl our WebTables system extracted roughly 154M tables with high-quality relational data. The resulting corpus of databases is larger than any other corpus we are aware of, by at least five orders of magnitude.

In this chapter we describe the WebTables system for processing table-oriented data from the Structured Web.[1] As with the TextRunner system, WebTables is *extraction-focused*, relying entirely on data that is already extant. We present a *domain-independent*, *domain-scalable*, and *computationally-efficient* extraction technique for obtaining this huge and largely-untapped set of structured databases.

We also explore several fundamental questions about managing them. First, what are good techniques for searching through a massive number of structured datasets? To answer this question, 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, what additional power can be derived by analyzing such a huge collection of databases? 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 com-

[1]Much of this work was done while the author was at Google, Inc.

putes attribute synonym pairs for schema matching; and *join-graph traversal*, which allows a user to navigate between extracted schemas using automatically-generated join links.

Together, these contributions - a method for extraction, a search system, and novel applications built on top of the data - demonstrate that table-oriented data is an important and feasible component of any Structured Web system.

4.1 Problem Overview

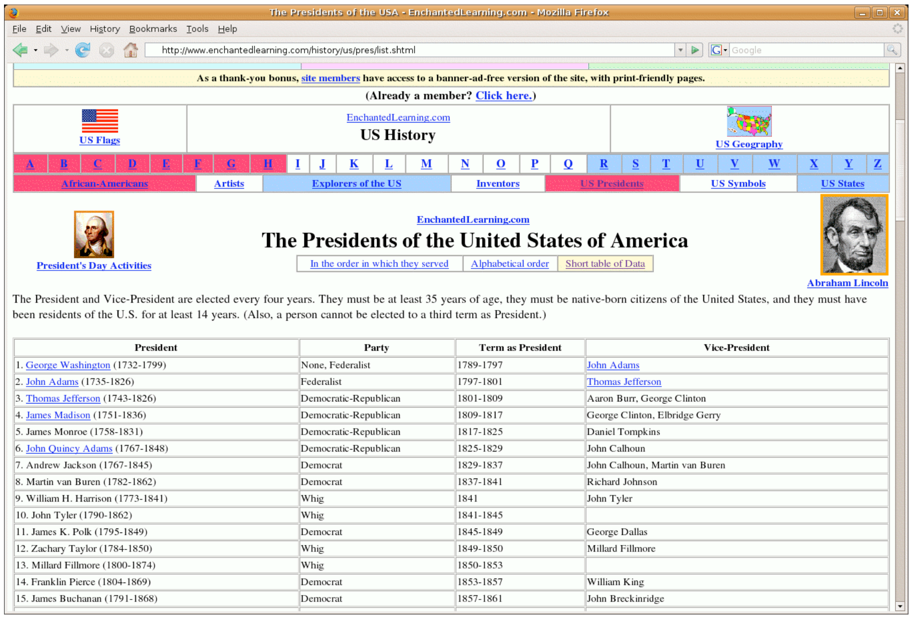
Even Web documents that are thought of as purely textual can contain large amounts of relational data. For example, the Web page shown in Figure 4.1 (and shown previously in Section 1.3.2) contains a table that lists American presidents.[2] The table has four columns, each with a domain-specific label 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 that could be managed with Oracle or DB2, even if it lacks the explicit metadata traditionally associated with a database.

The first major research question addressed by WebTables is how to best recover relational tables even though they are intermixed with HTML tables being used for page layout or other non-relational reasons. As mentioned above, by running the WebTables extractor over a large Web crawl, we are able to obtain a set of relational databases that is five orders of magnitude larger than any other collection that we are aware of (due to Wang *et al.* [84]).

The scale of this collection motivates two additional research questions: (1) what are *computationally-efficient*, *domain-independent*, and *domain-scalable*) techniques for searching within such a collection of tables, and (2) is there additional power that can be derived by analyzing such a huge corpus? The extractor plus answers to these two questions form the contributions of the WebTables system.

Table search is a critical component in effectively addressing all of the distinct databases recovered by WebTables. Of course, today SQL is the most popular method for querying a structured relational database. However, a traditional relational system assumes that the

[2] `http://www.enchantedlearning.com/history/us/pres/list.shtml`


The Presidents of the USA - EnchantedLearning.com - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.enchantedlearning.com/history/us/pres/list.shtml
Google

As a thank-you bonus, site members have access to a banner-ad-free version of the site, with print-friendly pages.

(Already a member? Click here.)

US Flags | EnchantedLearning.com US History | US Geography

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

African-Americans | Artists | Explorers of the US | Inventors | US Presidents | US Symbols | US States

President's Day Activities

EnchantedLearning.com

The Presidents of the United States of America

In the order in which they served | Alphabetical order | Short table of Data

Abraham Lincoln

The President and Vice-President are elected every four years. They must be at least 35 years of age, they must be native-born citizens of the United States, and they must have been residents of the U.S. for at least 14 years. (Also, a person cannot be elected to a third term as President.)

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	Aaron 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	Daniel Tompkins
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 (1782-1862)	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 (1795-1849)	Democrat	1845-1849	George Dallas
12. Zachary Taylor (1784-1850)	Whig	1849-1850	Millard Fillmore
13. Millard Fillmore (1800-1874)	Whig	1850-1853	
14. Franklin Pierce (1804-1869)	Democrat	1853-1857	William King
15. James Buchanan (1791-1868)	Democrat	1857-1861	John Breckinridge

Figure 4.1: This is the same figure seen in Chapter 1, in Figure 1.1. 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. (The raw HTML table corpus consists of 1.1% true relations.)

set of relation names is fairly small; although it is possible to find relational systems that have thousands of tables, even tens of thousands would be a startling number. Users of a traditional data management system are expected to uniquely name each interesting table, and so WebTables's massive set of extracted relations would place an huge burden on the database user attempting to find just the right one. The difference in difficulty between finding a table in a traditional relational database and finding one in WebTables's data is akin to the difference between finding a file on a single computer and finding a file anywhere on the Web.

We thus believe that managing a large corpus of relational databases has many more similarities to Web search than it does to traditional relational query processing. Therefore, WebTables has a component to perform **relation ranking**, *i.e.*, to sort relations by relevance in response to a user's keyword search query. Note that researchers like Agrawal *et al.* [2] and Hristidis and Papakonstantinou [48] have also studied the quite-different problem of keyword ranking for tuples within a single database. We also believe solving the relation ranking problem has utility beyond just extracted HTML tables: large collections of spreadsheets and non-HTML-table Web relations should also benefit.

WebTables must thus 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 are useless when tables of widely-varying quality can be 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 a substantial improvement in search quality over a naïve approach based on traditional search engines.

Finally, to demonstrate the power of WebTables's corpus, we also describe 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, *e.g.*, by Lin and Pantel [55]. 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 for automatically detecting synonymy that uses real-world labels and incorporates coattribute context information. It finds synonyms 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.

Third, 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.

All of the above tools are examples of how web-scale data can be used to solve problems that are otherwise very hard. They are similar in spirit to recent efforts on machine translation such as Brants *et al.* [16], that leverage huge amounts of data. The distinguishing feature of the **ACSDb** is that it is the first time such large statistics have been collected for structured data schema design. We note that the idea of leveraging a large number of schemas was initially proposed by Madhavan *et al.* [56] for 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.

As mentioned in Section 1.3.4, it is easy to confuse 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 overlap, 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 by He *et al.* [46]). In contrast to the work we describe in this chapter, 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, including tagged items, ontologies, XML documents, spreadsheets, and even extracted language parses. Madhavan *et al.* [58] discuss some of these forms. In this chapter we will only consider the `table` tag.

This chapter focuses on how to extract the 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, which we save for Chapter 5.

The remainder of this chapter is organized as follows. We start by covering the table extraction pipeline and our basic model of table-embedded data in Section 4.2. We use Section 4.3 to describe the **ACSDb**. In Section 4.4, we describe how to rank tables in response to keyword query on WebTables. Section 4.5 covers our three novel **ACSDb** applications: *schema auto-complete*, *attribute synonym finding*, and *join-graph discovery.* We present experimental evaluations in Section 4.6, discuss related work in Section 4.7, and finally conclude in Section 4.8.

4.2 HTML Table Extraction

This section covers our techniques for HTML table extraction, which distill the original raw HTML tables into a much smaller set of high-quality relations. Most HTML tables are used for page layout, form layout, or other non-relational data presentation (such as "property sheet" lists that focus on a single data item); these non-relational tables must be filtered out. Further, even the correctly-detected relations lack explicit metadata such as column labels and types. WebTables detects this metadata when it is embedded in the HTML.

In order to better understand the extraction task, we start by characterizing the raw crawled HTML table corpus. We then move onto the extractor mechanics.

4.2.1 The HTML Table Corpus

We applied an HTML parser to a multi-billion-page crawl of the English-speaking web to obtain about 14.1B instances of the `table` tag. Presenting relational-style data is perhaps the most "obvious" use of the tag, but non-relational uses are far more numerous.

A few use cases make up the bulk of all HTML tables and are easy to detect:

- *Extremely small* tables are those with fewer than two rows or two columns. We assume that these tables carry no interesting relational information.

- Many tables are embedded inside *HTML forms* and are generally used for visual layout of user input fields.

- Some tables are used to draw a *calendar* onscreen, and consist of nothing but a column for each day of the week and a number in each cell.

We wrote parsers that reliably find each of these use cases. As seen in Table 4.1, these three table types make up more than 88% of the HTML tables in our raw corpus.[3] Any web-embedded relations must be found in the remaining portion.

[3]Remarkably, 0.88% of all HTML tables in our raw crawl (more than 122M) are "no-ops," containing *zero rows and zero columns.*

Table type	**% total**	**count**
Extremely small	88.06	12.34B
HTML forms	1.34	187.37M
Calendar	0.04	5.50M
Obviously non-relational, total	89.44	12.53B
Other non-relational (est.)	9.46	1.33B
Relational (est.)	**1.10**	154.15M

Table 4.1: Various types of HTML tables found in the crawl. Types are non-overlapping; we only examine tables that were not eliminated by tests higher on the list. The rate of *other non-relational* tables is estimated from a human-judged sample.

However, even this remainder consists primarily of non-relational tables. These non-relations include tables used for page layout, tables with an enormous number of blank cells, tables that are really simple lists presented in two dimensions, and "property sheets" that consist of attribute value pairs for a single data entity (*e.g.*, MySpace personal preference lists). Unlike the HTML forms and calendars listed above, it is difficult to automatically detect these non-relational types. Two tables may have identical HTML structure, but only one may contain good relational data. Nor can traditional tests for relational well-formedness (*e.g.*, testing whether a table obeys schema constraints, or testing a schema to see whether it is in a certain normal form) be applied here. Not only is there no explicit metadata associated with the tables, many traditional relational constraints (*e.g.*, foreign key constraints) make no sense in the web-publishing scenario.

Answering whether a table contains relational data usually means understanding the table's data, and thus is unavoidably one of human judgment. So we asked two human judges to examine a sample of the remaining tables and mark each as *relational* or *non-relational*. The results, seen in Table 4.1, show that 10.4% of the tables in the remainder are classified as relational (1.1% of the original raw crawl), meaning our 14B-table crawl contains roughly 154M high-quality relations. While the percentage is relatively small, the

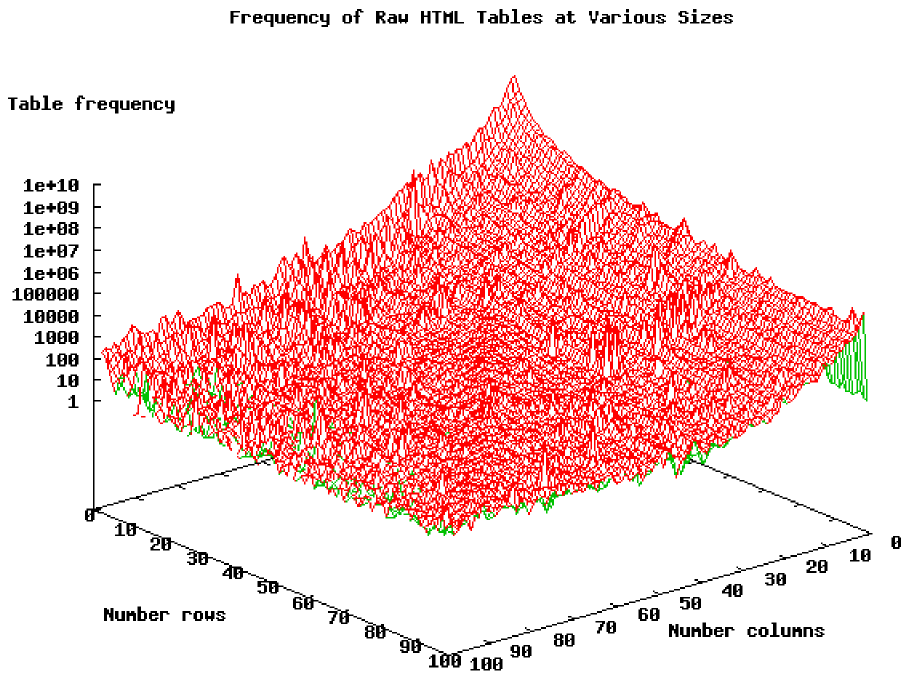


Figure 4.2: Frequency of raw HTML tables/relations of various row and column sizes. The most frequently-seen table in the raw crawl (seen 3.8B times, accounting for 27.32% of all tables) contains a single cell, and is represented in the plot by the point at `rows=1, cols=1`. The green lines visible at the right-hand edge indicate the "underside" of the 3D plot.

vast number of tables in our crawl means that the resulting set of relations is still enormous.

Figures 4.2 and 4.3 show the relative number and sizes of tables in the crawl. The first figure shows the frequency of the raw HTML tables, and the second figure shows the frequency of the extracted relations obtained from the raw corpus by the extractor described below in Section 4.2.2.

Table 4.2 compares a few selected pieces of data from Figures 4.2 and 4.2. Note that tables with between 2 and 9 columns make up 55% of the raw corpus, but more than 93% of the recovered relations; as intuition would suggest, there are relatively few high-quality relations with a very large number of attributes. In contrast, there is a much greater diversity of row counts among the recovered relations.

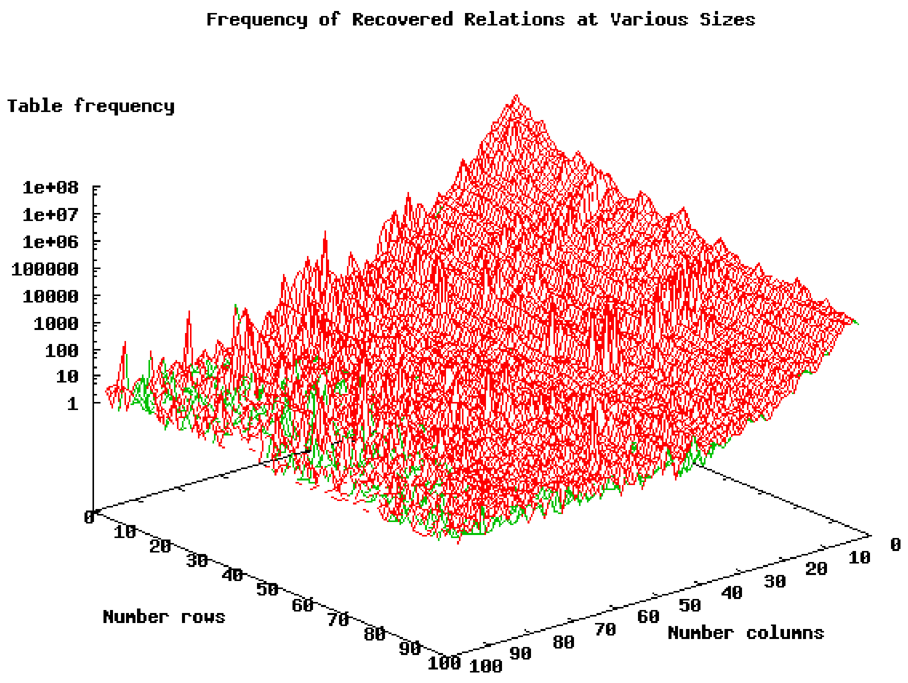


Figure 4.3: Frequency of high-quality recovered HTML tables/relations of various row and column sizes. The plot shows no tables with fewer than 2 rows or 2 columns, and many fewer tables in general than plot in Figure 4.2. Also note that there is much less variation with the number of rows than in the raw plot; the high-quality plot varies mainly with the number of columns. This observation matches the intuition that there should be relatively many high-quality relations with few columns, and relatively few relations with many columns; at the same time, the number of rows should be relatively unimportant as an indicator of relation quality.

Having described the set of relational tables that are available via HTML, we can now turn to the actual extraction mechanism.

4.2.2 Extractor Mechanics

Recovering relations from the raw HTML tables consists of two steps. First, WebTables attempts to filter out all non-relational tables. Second, WebTables attempts to recover embedded metadata (in the form of attribute labels) for the now-filtered relations (such as

Cols	Raw %	Recovered %
0	1.06	0
1	42.50	0
2-9	55.00	93.18
10-19	1.24	6.17
20-29	0.19	0.46
30+	0.02	0.05

Rows	Raw %	Recovered %
0	0.88	0
1	62.90	0
2-9	33.06	64.07
10-19	1.98	15.83
20-29	0.57	7.61
30+	0.61	12.49

Table 4.2: Frequency of raw tables and recovered relations at selected sizes, as a percentage of each dataset.

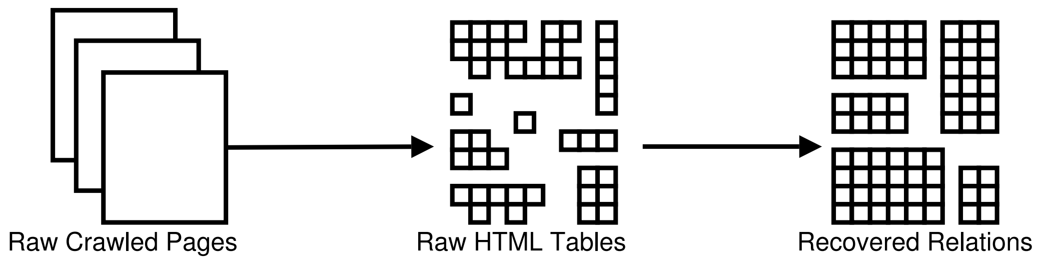


Figure 4.4: The WebTables relation extraction pipeline, first seen in Chapter 1, in Figure 1.4.

the first row in the table in Figure 4.1). The extraction pipeline is pictured in Figure 4.4.

More formally, the goal of the extractor is to obtain 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, $R_{\mathcal{S}}$, which is an ordered list of attribute labels. For example, the table in Figure 4.1 has the attributes $R_{\mathcal{S}} = [\textbf{President}, \textbf{Party}, \ldots]$. One or more elements of $R_{\mathcal{S}}$ may be empty strings (*e.g.*, if the table's schema cannot be recovered).

- a list of tuples, $R_{\mathcal{T}}$. A tuple t is a list of data strings. The size of a tuple t is always $|R_{\mathcal{S}}|$, though one or more elements of t may be empty strings.

Relation Filtering

After applying the hand-written filters described above, we can treat relational filtering as a machine learning classification problem, similar to the work of Wang and Hu [84]. We asked two human judges to classify a large number of HTML tables as *relational* or not. We paired these classifications with a set of automatically-extracted features for each table (listed in Figure 4.5(a)) to form a supervised training set for a statistical learner.

Like Wang and Hu, our features describe the table's layout as well as datatypes in each column. Table layout qualities include, for example, the number of rows in the table, the number of columns, the number of empty cells (or NULLs), the average cell string length, and so on. The motivation behind these features is that high-quality relational tables tend to have certain statistics in common: say, relatively few empty cells. WebTables discovers each column's datatype by examining whether almost all of the values in a single column observe the same type (*e.g.*, integer or date). If so, then the column appears to be well-typed; we imagine a relational table probably consists mainly of many well-typed columns.

In order to remain *domain-independent*, we do not use techniques that rely on specific values in the tables. This approach is unlikely to work with a large diversity of data tuples, and indeed it showed almost no gains beyond the techniques above, when attempted by Wang and Hu on a relatively small dataset of several thousand tables.

Only results that our classifier declares to be *relational* will make it into downstream applications; good relations that are labeled as *non-relational* will be discarded. Non-relations that are incorrectly labeled as relational are imperfect but acceptable, as there is still a chance that the downstream WebTables application will be able to handle a bad table. So, we tuned the relational classifier to give very high recall at the cost of lower precision. Our experiments in Section 4.6 show that we can recover the estimated set of 154M relations with acceptable precision, and with recall that is comparable to other domain-independent Web extraction systems.

rows
cols
% rows w/mostly NULLS
cols w/non-string data
cell strlen avg. μ
cell strlen stddev. σ
cell strlen $\frac{\mu}{\sigma}$

(a) Relational Filtering

rows
cols
% cols with lower-case in *row_1*
% cols with punctuation in *row_1*
% cols with non-string data in *row_1*
% cols with non-string data in *body*
% cols with type clashes between *row_1* and *body*
% cols with $\lvert len(row\_1) - \mu \rvert > 2\sigma$ *(i.e., where first row's strlen is more than 2 standard deviations from average)*
% cols with $\sigma \leq \lvert len(row\_1) - \mu \rvert \leq 2\sigma$ *(i.e., where first row's strlen is between 1 and 2 standard deviations from average)*
% cols with $\sigma > \lvert len(row\_1) - \mu \rvert$ *(i.e., where first row's strlen is less than 1 standard deviation from average)*

(b) Header Detection

Figure 4.5: Selected features used in relational ranking and header detection. Relational Filtering requires statistics that help it distinguish relational tables, which tend to contain either non-string data, or string data with lengths that do not differ greatly. Header detection relies on both syntactic cues in the first row and differences in data type and string length between the first row and the remainder of each column. The string length measures help in detecting a header even when the body of the column contains strings and thus does not generate a type clash with the header.

Metadata Recovery

Even the correctly-filtered relations still lack formal metadata. However, attribute names and types in a good relation will often be obvious to a human observer (sometimes because labels are directly embedded in the HTML). WebTables is only concerned with metadata as far as it consists of these per-attribute labels. Attribute labels are very inexpressive compared to the the metadata that is standard in a traditional relational database. However, much of the standard relational metadata applies only to multiple-relation databases, *e.g.*, foreign-key constraints. Moreover, many of the single-table constraints in relational metadata would be too restrictive if applied to WebTables data, which we assume will always be somewhat dirty, even when recovered correctly. For the title of each extracted relation we currently use the title of the HTML page where the table was found; an interesting topic for future work is to compute a relevant synthetic human-readable title for each relation. Recovering the metadata is still quite difficult, even when limited to these labels.

High-quality labels for each column can have a number of good downstream effects in WebTables. In the context of the relation-ranking query tool, good labels allow relations to appear correctly on-screen, labels improve rank quality, and labels are helpful when applying structured data services (such as XML export or data visualization). Further, good labels allow for the very existence of the **ACSDb**, a collection of statistics about schema attributes that we describe in Section 4.3.

There are two cases to consider for meta-data recovery. In the first case, there is already a "header" row in the table with column labels. However, this header is difficult to distinguish, as relations often contain alphabetic string data. A relational column contains strings either because its intended type really is `string`, or because the column was intended to have a different type (say, `numeric`), but includes a few stray strings as a side-effect of HTML extraction (*e.g.*, a column might contain a misplaced copyright notice). Based on a hand-marked sample, we believe that 71% of the true relations have a header row.

To obtain the metadata contained in header rows, we developed the **Detect** classifier that declares whether a relational header is *present* or not. **Detect** uses the features listed in Figure 4.5(b), trained on approximately six thousand hand-marked samples by two separate judges. The two most heavily-weighted features for header-detection are the number of columns and the percentage of columns with non-string data in the first row.

The second case covers the remaining 29% of true relations, where the data is good but there is no header row. In these cases, we can only hope to synthesize column labels that make sense. We tested an algorithm called **ReferenceMatch**, which attempted to create synthetic column labels by matching the contents of an unlabeled column to a separate dataset where we already know a correct label. For example, an anonymous column that contains Casablanca, Vertigo, and other movies may match find a large number of entries to a preexisting movie database, allowing us to apply the **movie** label. Unfortunately, we found extremely few tables with clean enough string data to match our controlled database of 6.8M tuples in 849 separate domains. For now, synthetic schema generation is still an area for future work.

As seen in the experimental results in Section 4.6, **Detect** is quite successful at recovering header-embedded metadata, especially when combined with the **ACSDb** that we describe next.

4.3 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.

The **ACSDb** lists each unique set of attributes $\mathcal{S}$ found in the corpus of relations, along with a count that indicates how many relations contain the given $\mathcal{S}$. We assume two schemas are identical if they have the same set of attributes (regardless of their order). The **ACSDb** $\mathcal{A}$ is a set of pairs of the form $(\mathcal{S}, c)$, where $\mathcal{S}$ is a schema of a relation in $\mathcal{R}$, and c is the number of relations in $\mathcal{R}$ that have the schema $\mathcal{S}$.

Extracting the **ACSDb** given the corpus $\mathcal{R}$ of extracted relations is straightforward, as described below. Note that a given domain name can only provide one count toward a single schema, preventing a single site with many tables from swamping the schema statistics. (The *seenDomain* collection of sets stores all of the observed domain names that have ever contributed a count toward a given schema.)

```
Function createACS($\mathcal{R}$):
$\mathcal{A} = \{\}$
$seenDomains = \{\}$
for all $R \in \mathcal{R}$ do
  if $getDomain(R.u) \notin seenDomains[R.\mathcal{S}]$ then
    $seenDomains[R.\mathcal{S}].add(getDomain(R.u))$
```

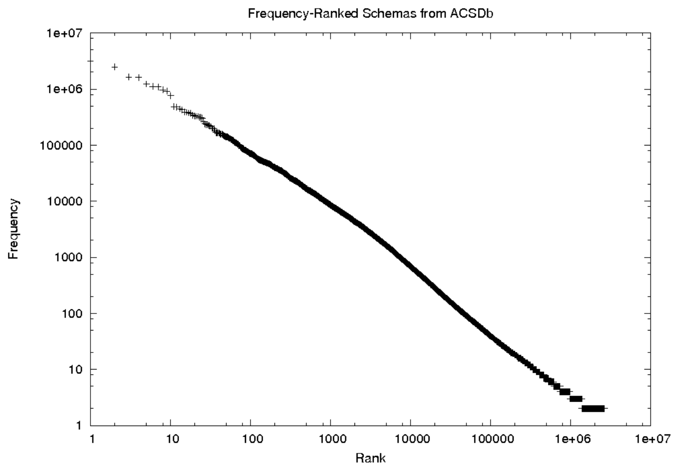
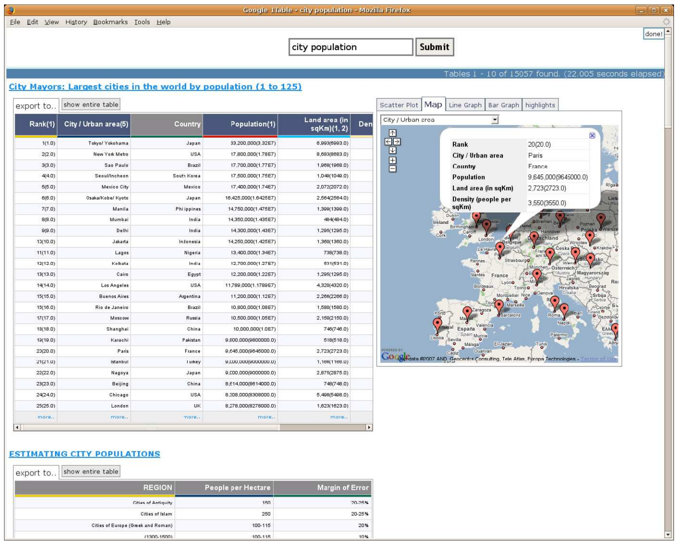


Figure 4.6: Distribution of frequency-ordered unique schemas in the **ACSDb**, with rank-order on the x-axis, and schema frequency on the y-axis. Both rank and frequency axes have a log scale.

```
        $\mathcal{A}[R.\mathcal{S}] = \mathcal{A}[R.\mathcal{S}] + 1$
    **end if**
**end for**
```

In order to guarantee that each attribute and each schema has nontrivial support, we removed all attributes and schemas that appear only once. The remaining data allowed us to compute 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 4.6).

The **ACSDb** is simple, but it 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," then normalizing by the counts for seeing "name" alone. As we will see in Sections 4.4.1 and 4.5, we can use these simple probabilities to build several new and useful schema applications.



Tables 1 - 10 of 15057 found. (22.005 seconds elapsed)

City Mayors: Largest cities in the world by population (1 to 125)

Rank(1)	City / Urban area(5)	Country	Population(1)	Land area (in sqKm)(1, 2)	Den
1(1.0)	Tokyo/ Yokohama	Japan	33,200,000(3.32E7)	6,993(6993.0)	
2(2.0)	New York Metro	USA	17,800,000(1.78E7)	8,683(8683.0)	
3(3.0)	Sao Paulo	Brazil	17,700,000(1.77E7)	1,968(1968.0)	
4(4.0)	Seoul/Incheon	South Korea	17,500,000(1.75E7)	1,049(1049.0)	
5(5.0)	Mexico City	Mexico	17,400,000(1.74E7)	2,072(2072.0)	
6(6.0)	Osaka/Kobe/ Kyoto	Japan	16,425,000(1.6425E7)	2,564(2564.0)	
7(7.0)	Manila	Philippines	14,750,000(1.475E7)	1,399(1399.0)	
8(8.0)	Mumbai	India	14,350,000(1.435E7)	484(484.0)	
9(9.0)	Delhi	India	14,300,000(1.43E7)	1,295(1295.0)	
10(10.0)	Jakarta	Indonesia	14,250,000(1.425E7)	1,360(1360.0)	
11(11.0)	Lagos	Nigeria	13,400,000(1.34E7)	738(738.0)	
12(12.0)	Kolkata	India	12,700,000(1.27E7)	531(531.0)	
13(13.0)	Cairo	Egypt	12,200,000(1.22E7)	1,295(1295.0)	
14(14.0)	Los Angeles	USA	11,789,000(1.1789E7)	4,320(4320.0)	
15(15.0)	Buenos Aires	Argentina	11,200,000(1.12E7)	2,266(2266.0)	
16(16.0)	Rio de Janeiro	Brazil	10,800,000(1.08E7)	1,580(1580.0)	
17(17.0)	Moscow	Russia	10,500,000(1.05E7)	2,150(2150.0)	
18(18.0)	Shanghai	China	10,000,000(1.0E7)	746(746.0)	
19(19.0)	Karachi	Pakistan	9,600,000(9600000.0)	518(518.0)	
20(20.0)	Paris	France	9,645,000(9645000.0)	2,723(2723.0)	
21(21.0)	Istanbul	Turkey	9,000,000(9000000.0)	1,166(1166.0)	
22(22.0)	Nagoya	Japan	9,000,000(9000000.0)	2,875(2875.0)	
23(23.0)	Beijing	China	8,614,000(8614000.0)	748(748.0)	
24(24.0)	Chicago	USA	8,308,000(8308000.0)	5,498(5498.0)	
25(25.0)	London	UK	8,278,000(8278000.0)	1,623(1623.0)	
more..	more..	more..	more..	more..	

Rank	20(20.0)
City / Urban area	Paris
Country	France
Population	9,645,000(9645000.0)
Land area (in sqKm)	2,723(2723.0)
Density (people per sqKm)	3,550(3550.0)

Map data ©2007 AND, Geocentre Consulting, Tele Atlas, Europa Technologies - Terms of Use

ESTIMATING CITY POPULATIONS

REGION	People per Hectare	Margin of Error
Cities of Antiquity	150	20-25%
Cities of Islam	250	20-25%
Cities of Europe (Greek and Roman)	100-115	20%
(1300-1500)	100-115	10%

Figure 4.7: This is the same figure seen in Chapter 1, in Figure 1.3. 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. The visualization to the right was generated automatically by WebTables, and shows the result of clicking on the "Paris" row. The title ("City Mayors...") links to the page where the original HTML table was found.

We next describe the WebTables relation search system, which uses features derived from both the extracted relations and from the **ACSDb**. Afterwards, in Section 4.5, we will discuss **ACSDb** applications that are more broadly applicable to traditional database tasks. Indeed, we believe the **ACSDb** will find many uses beyond those described in this chapter.

4.4 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. As discussed in Section 4.1 above, most relational database expect a human being to uniquely address each table, placing a heavy cognitive burden on users when the number of distinct tables grows large. Thus, because WebTables manages more than a hundred million relations, relation search is a crucial part of making WebTables *domain-scalable*.

Figure 4.12 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 4.7 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.

```
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 4.8: 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.

One exciting area of structure-driven services is in the area of data visualization. WebTables does not make any research contributions in this area yet, but it can currently create a query-appropriate visualization by testing whether the tuples $R.\mathcal{T}$ contain a column of geographic placenames. We test the column by checking each cell against a hand-crafted geographic database; if more than 95% of the cells are present in the database, then we mark the column as geographic. If the column is so marked, WebTables will place all of each tuple's data at the correct locations on the map (see, *e.g.*, the "Paris" tuple in Figure 4.7). The user can also manually choose a visualization. In the future we hope to construct a general and completely automated tool that suggests appropriate visualizations based on the statistical relationship between table columns. Finally, WebTables search also 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

```
1: Function filterRank($q, k$):
2: let $\mathcal{U}$ = ranked urls from web search for query $q$
3: let $numEmitted = 0$
4: for all $u \in \mathcal{U}$ 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 4.9: Function **filterRank**: similar to **naïveRank**, it will go as far down the search engine result pages as necessary to find k relations.

without good search relevance. In the section below we present different algorithms for ranking individual databases in relation to a user's query. Unfortunately, the traditional inverted text index cannot support these algorithms efficiently, so in Section 4.4.2 we also describe additional index-level support that WebTables requires.

4.4.1 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.

Web-extracted relations pose a unique set of difficult ranking challenges. Relations do not exist in a domain-specific schema graph, as with relational keyword-access systems (*e.g.*, DBXplorer [2], DISCOVER [48]). Page-level features like word frequencies apply ambiguously to tables embedded in the page. Even a high-quality page may contain tables of varying quality; consider a page that has several good relations as well as a page-layout table that is incorrectly believed to be relational.

However, relations also have special features that may make ranking easier. Most tables have schema elements that provide good, if partial, summaries of the subject matter. Rows often put key-like values in the first column; for example, a row about the city `Paris` will usually but the string "Paris" in the leftmost column. Finally, good relations tend to be amenable to human-consumption: there are relatively few empty cells, and the number of columns is reasonably small. All of these qualities suggest features that are useful in ranking.

To rank our extracted WebTables relations, we created a series of ranking functions of increasing complexity, listed in Figures 4.8, 4.9, 4.10, and 4.11. 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.

```
1: Function featureRank($q, k$):
2: let $\mathcal{R}$ = set of all relations extracted from corpus
3: for $r \in \mathcal{R}$ do
4:     let $score(q, r)$ = combination of per-relation features in Table 4.3
5: end for
6: sort $r \in \mathcal{R}$ by descending order of $score(q, r)$
7: for $i = 0$ to $k$ do
8:     emit $\mathcal{R}[i]$
9: end for
```

Figure 4.10: Function **featureRank**: score each relation according to the features in Table 4.3. Rank by that score and return the top k relations.

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 extracted 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 4.6.2, 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 4.10 shows **featureRank**, the first algorithm that does not rely on an existing search engine. It uses the relation-specific features listed in Table 4.3 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 4.3 include both query-independent and query-dependent elements that we imagined might describe a relevant relation. Cues to a relevant relation include rough indicators of relation quality (*e.g.*, the number of rows, columns, and empty cells) and indicators of the relation's direct relevance to the search terms (*e.g.*, the number of hits on the relation's metadata, the number of hits to the leftmost column, and the number of hits on the table's cells). 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. Schema coherency is defined formally in Figure 4.11. Because the coherency score

```
1: **Function** cohere($R$):
2: $totalPMI = 0$
3: **for all** $a \in attrs(R), b \in attrs(R), a \neq b$ **do**
4:     $totalPMI = totalPMI + PMI(a, b)$
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 4.11: The *coherency score* measures how well attributes of a schema fit together. Probabilities for individual attributes are derived using statistics in the **ACSDb**.

is query-independent, it can serve as a measure of a standalone schema's quality. Thus it is very useful as a component of table ranking. It is also possible to use the score to improve metadata recovery; this technique is described in Section 4.6.1.

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. (See works by Church and Hanks [27], Etzioni *et al.* [39], and Turney [82].) PMI will be large and positive when two 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. Note that the coherency score is query-independent.

We will see in Section 4.6.2 that **schemaRank** performs the best of our search algorithms.

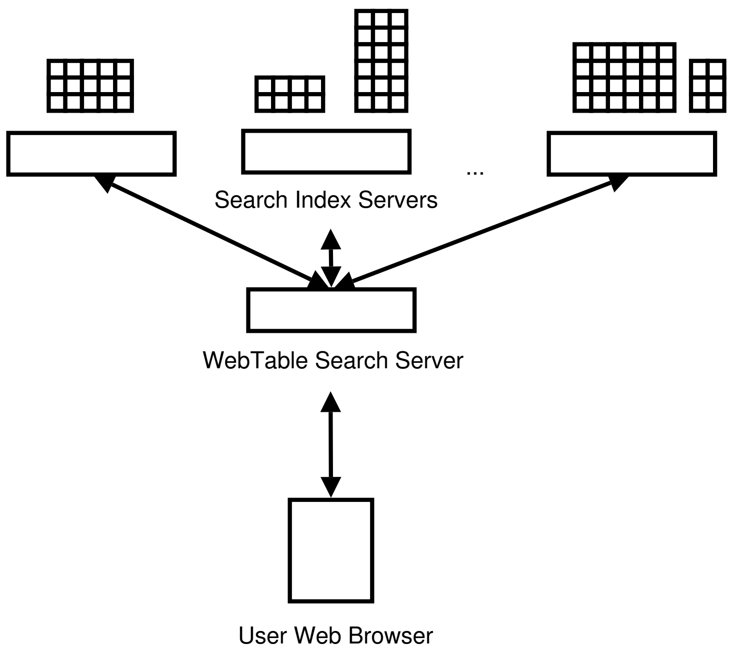


Figure 4.12: The WebTables search system. The inverted table index is segmented by term and divided among a pool of search index servers. A single front-end search server accepts the user's request, transmits it to all of the index servers, and returns a reply.

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.

4.4.2 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 4.3.

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

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

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)` 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.

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 4.4: Ten input attributes, each with the schema generated by the WebTables auto-completer.

4.5 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 (*e.g.*, Brants *et al.* [16] and various techniques described by Schütze and Manning [60]) and information extraction (such as KnowItAll [39]). However, we are not aware of any simi-

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 4.5: Partial result sets from the WebTables synonym-finder, using the same attributes as in Table 4.4.

lar technique applied to the structured-data realm, possibly because no previous database corpus has been large enough.

4.5.1 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 database users when designing a novel 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.

For example, when the user enters `make`, the system suggests `model, year, price, mileage,` and `color`. Table 4.4 shows ten example input attributes, followed by the output schemas given by the auto-completer.

We can say that for an input I, the best schema S of a given size is the one that

```
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 4.13: 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.

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 4.13 for the formal algorithm.) A greedy approach is not guaranteed to find the maximal schema, but it does offer good interaction qualities - once an attribute has been accepted, it is added to I and cannot be "retracted." In contrast, consider an algorithm that attempts to present a ranked list of schemas sorted strictly by overall probability. Any user examining such a sorted list could find that schemas ranked nearby may not have anything at all in common.

The greedy approach is weakest when dealing with attributes that occupy two or more strongly-separated domains. For example, consider the "address" attribute, which appears in multiple domains (*e.g.*, real-world street addresses as well as IP addresses) so the most-probable response may be inappropriate for the user's intended subject area a large amount of the time. 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 4.5.3.

4.5.2 Attribute Synonym-Finding

An important part of schema matching is finding synonymous column labels. Unfortunately, obtaining a high-quality set of label synonyms can be burdensome. The **ACSDb** allows us to automatically find attribute strings synonyms from WebTables's large volume of schema data.

The synonym-finder takes a set of *context attributes*, C, as input. It must then compute a list of attribute pairs 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, if it is true that $p(a, b) = 0$, the two attributes may never cooccur either because they are completely substitutable (*i.e.*, synonymous) or because they are independent and simply never happened to appear together. If $p(a)p(b)$ is high, then the likelihood of a and b failing to cooccur through simple chance is lower.

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:

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

The value of $syn(a, b)$ 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 $syn(a, b)$ will be high when the attributes frequently appear in similar contexts and thereby drive the denominator lower.

Although our current system requires that $p(a, b) = 0$, this constraint may be too aggressive. It may be sufficient simply that $p(a, b)$ be very small in comparison to $p(a)p(b)$.

Examining the quality and synonym-yield tradeoffs between the strict $p(a, b) = 0$ constraint and a more relaxed version is a subject for future work.

Our **SynFind** algorithm (see Figure 4.14) 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. There are a few reasons the number of considered pairs does not grow to enormous size. First, the algorithm only considers attributes that appear with *all* members of the set C, so as C grows, the set of pairs to be scored should shrink. Second, the *syn()* function is commutative, so SynFind saves time by only considering attribute pairs where a is lexigraphically less than b. Third, the limited size of each schema and the distribution of data within schemas limits the potential number of pairs; in our data, even the most-popular attribute (`name`) cooccurs with only 62,000 other attribute labels. Finally, the definition for *syn()* makes it easy to throw away pairs that are likely to have a very low score, *e.g.*, those in which at least one attribute has a very low marginal probability. This allows us to save a huge amount of time if we are willing to retrieve just the top-k pairs.

Table 4.5 lists synonyms found by WebTables for a number of input contexts.

4.5.3 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 describing 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 4.15. We never materialize the full join

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

Figure 4.14: 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. The inequality comparison between attributes a and b is a small optimization - for correctness, a and b simply should not be identical.

graph at once, but only the locally-viewable portion at a focal schema F.

```
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 4.15: **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 takes focal schema $\mathcal{F}$ and **ACSDb** database $\mathcal{A}$ as inputs. The input $\mathcal{A}$ consists of a series of pairs (S, c), which describe a schema and its observed count.

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

true class	Precision	Recall
relational	0.41	0.81
non-relational	0.98	0.87

Table 4.6: Test results for filtering true relations from the raw HTML table corpus.

during join graph traversal.

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

The function *neighborSim* is very similar to the *coherency score* in Figure 4.11. 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. Note that normalizing the output of a log function here is not an ideally-principled approach, but gave good results in practice. In the future we would like to experiment with more cleanly-defined measures.

Clustering is how *neighborSim* helps with join graph traversal. 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.

4.6 Experimental Results

We now present experimental results for extraction, relation ranking and for the three **ACSDb** applications.

Detector	header?	Precision	Recall
Detect	has-header	0.79	0.84
	no-header	0.65	0.57
Detect-ACSDb	has-header	0.89	0.85
	no-header	0.75	0.80

Table 4.7: Test results for detecting a header row in true relations. We correctly detect most of the true headers, but also mistakenly detect headers in a large number of non-header relations. Incorporating **ACSDb** information improves both precision and recall.

4.6.1 Table Extraction

Recall that table extraction consists of two stages: filtering out non-relational tables, and recovering metadata for each relation that passes the filter.

Filtering

We asked human judges to classify six thousand HTML tables as *relational* or not. We then trained a rule-based classifier using the extracted features listed in Table 4.5(a), using the WEKA package [85]. We cross-validated the trained classifier by splitting the human-judged data into five parts, using four parts for training and the fifth for testing. We trained five different classifiers, rotating the testing set each time, and averaged performance numbers from the resulting five tests.

Table 4.6 shows the results. As mentioned previously, we tuned the training procedure to favor recall over precision, hoping that downstream applications will "cover our mistakes." The rule-based classifier retains 81% of the truly relational tables, though only 41% of the output is relational. These results mean we retain about 125M of the 154M relations we believe exist in the raw crawl, at a cost of sending 271M tables downstream. Our filter thus raises the "relational concentration" from 1.1% in the raw HTML table corpus up to 41%

It is interesting to examine the ways in which our relational filtering mechanism failed. Tables that are misclassified by the filter fall into a handful of categories:

Year/Color							
	1994			1995			
Model	Black	White	’94 total	Black	White	’95 total	Grand Total
Chevy	50	40	90	85	115	200	290
Ford	50	10	60	85	75	160	220
Grand Total	100	50	150	170	190	360	510

Table 4.8: A relational table like the one pictured here, with multiple layers of metadata, will probably not be classified correctly by the WebTables extractor’s relation filter. It would be possible to present this information in a more traditional format that WebTables could handle easily - this is the “pivot table” representation. The example is drawn from Gray *et al.* [43]

- **Ambiguous Tables** are somewhat unclear even to humans whether they should be called relational or not. For example, judges disagreed about tables that were technically well-formed relations but which has a huge number of empty cells may. This group of close calls is a difficult category to improve on.

- **Unusual Layout Tables** contain far more empty cells than is customary for a high-quality relation, usually for graphical layout purposes. Our filter usually takes high numbers of empty cells as a sign of low relational quality. While this signal is usually reliable, there are a number of tables where it is perverse. We might be able to avoid this problem by using a more complicated “visual model” of the table, estimating when a cell is empty for visual-layout reasons as opposed to data-quality reasons.

- **Model Mismatch Tables** are too complicated or otherwise unexpected for our filter to process. We assume that all tables are simply two-dimensional grids, with possibly a single row of metadata. However, consider Table 4.8. First, there are many values that span multiple columns, making some cells appear misleadingly “empty,” and thus lower our estimate of the table’s relational quality. Second, this table has multiple rows of metadata, all but one of which the WebTables relational filter will interpret as actual data. By misinterpreting metadata as data values, there will appear to be several columnar type clashes, again lowering the chances of deciding that this is a relational database. In other words, the WebTables model of how HTML tables describe relational data (2D, with a single row of metadata) is too simplistic to accurately describe the relation in this example.

While some of these errors are probably unavoidable (*e.g.*, the ambiguous tables), reducing errors in other categories, especially model-based errors, seems to be quite feasible and is a promising area for future work.

All remaining experiments in this paper use the output relations from this classifier.

Metadata Recovery

Recovering metadata entails first detecting when there is a header for the relation, and then generating a synthetic header when the raw HTML does not give one.

We created two different header-detectors. The first, **Detect**, uses the features from Figure 4.5(b), as described in Section 4.2.2. The second, **Detect-ACSDb**, also incorporates the **ACSDb**-derived coherency score from Figure 4.11. Previously used to measure the quality of competing tables' schemas during relation ranking quality, the coherency score should also be able to serve as a "bad metadata-extraction detector." A false positive during metadata recovery will lead to a real data row being interpreted as a set of label attributes. We expect that an arbitrarily-chosen row of data from the table will receive a relatively low coherency score, so a bad coherency score may lead **Detect-ACSDb** to detect no metadata at all.

We took the filtered output from the previous stage, and marked a large sample of 1000 relations as either *has-header*, *no-header*, or *non-relational* (in case of a mistake made by the relational filter). We then used all of the *has-header* or *no-header* relations to train and test our rule-based classifier, again using five-fold cross-validation (as above).

Table 4.7 shows the results. Unlike the relational-filtering case, there is no obvious recall/precision bias we should impose. As we expected, we can improve the performance of header-detection by including the schema coherency score. Both precision and recall are good for metadata recovery, and are consistent with other published extraction results (*e.g.*, Agichtein *et al.* [1] and Etzioni *et al.* [39]).

4.6.2 Relation Ranking

We evaluated the WebTables ranking algorithms described in Section 4.4.1: **naïveRank**, **filterRank**, **featureRank**, and **schemaRank**.

Just as with the training set described in Section 4.4, we created a test dataset by asking two human judges to rate a set of one thousand `(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 divided the pairs over a workload of 30 queries. 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 (again from the WEKA package [85]).

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 4.9 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 4.9. 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.

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 4.9: Fraction of high-scoring relevant tables in the top-k, as a fraction of "optimal" results.

4.6.3 Schema Auto-Completion

Output schemas from the auto-completion tool are almost always coherent (as seen with the sample outputs from Table 4.4), 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.

To generate these test schemas, we found six humans who are familiar with database schemas and gave them only the single-attribute input contexts listed in Tables 4.4. We then asked the human judges to create an attribute list (*i.e.*, a schema) for each of ten inputs. 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. We allowed the auto-completion tool to run multiple times; after the algorithm from Section 4.5.1 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 4.10 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

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
Average	0.46	0.59	0.62

Table 4.10: 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.

schemas except one,[4] 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 4.4 (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.

4.6.4 Synonym-Finding

We tested the synonym-finder's accuracy by asking it to generate synonyms in ten different areas. We gave the system the single-attribute input contexts listed in Table 4.5 - the same attributes discussed in the previous section. For example, when given `instructor`,

[4]No test designer recognized `ab` as an abbreviation for "at-bats," a piece of baseball terminology. WebTables gave exclusively baseball-themed outputs.

the synonym-finder emitted `course-title / title`, `course-name / course-title`, and so on. The synonym-finder's output is ranked in descending order of 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 human judge to declare whether each given synonym pair is accurate or not.

The results in Table 4.11 show that the synonym-finder's ranking is very good, with an average of 80% accuracy in the top-5. The average number of correct results declines as the rank increases, as expected.

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
Average	0.8	0.64	0.55	0.5

Table 4.11: Fraction of correct synonyms in top-k ranked list from the synonym-finder.

4.6.5 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 4.16. 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 annotated 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 4.16, 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.

4.7 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.* [42] attempted to discover tabular structure without the HTML `table` tag, through cues such as onscreen data placement. Their approach could make a useful table extractor for WebTables. Chen *et al.* [25] tried to extract tables from ASCII text. Penn *et al.* [69] attempted to reformat existing web information for handheld devices. 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 4.1, Wang and Hu [84] detected "good" tables with a classifier, using features that involved both content and layout. They operated on a corpus of just over ten thousand pages and found a few thousand true relations. Zanibbi *et al.* [90] offered a survey of table-parsing papers, almost all focused on processing a single

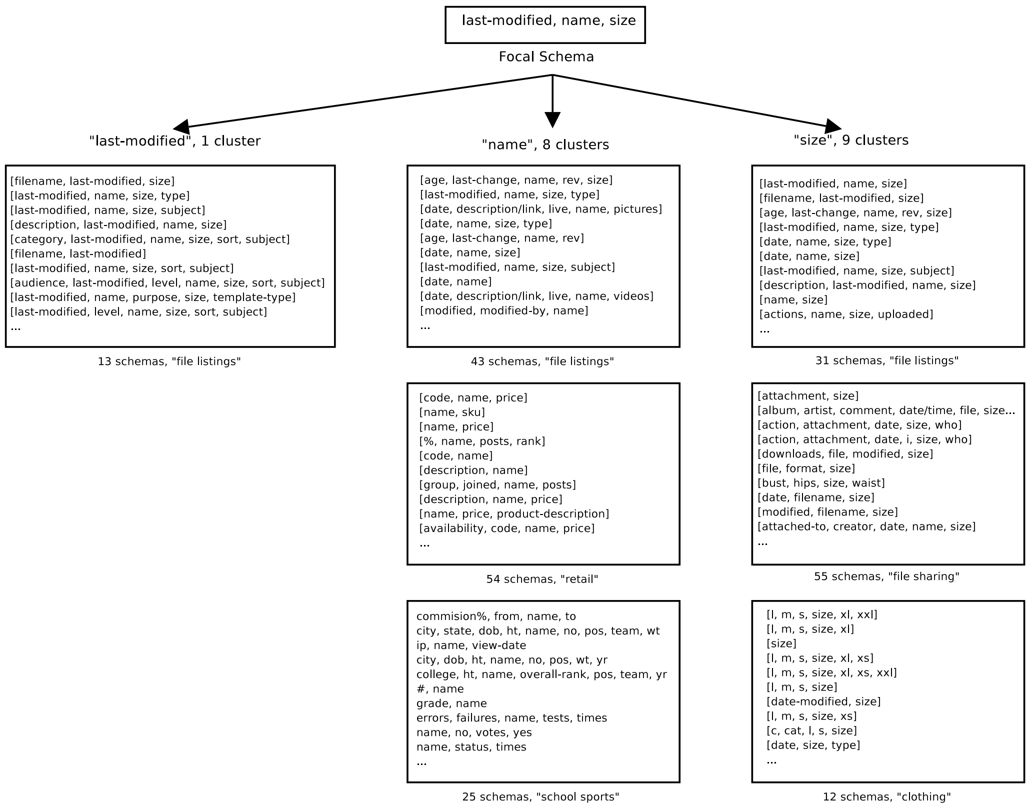

last-modified, name, size
Focal Schema
"last-modified", 1 cluster
"name", 8 clusters
"size", 9 clusters
[filename, last-modified, size]
[last-modified, name, size, type]
[last-modified, name, size, subject]
[description, last-modified, name, size]
[category, last-modified, name, size, sort, subject]
[filename, last-modified]
[last-modified, name, size, sort, subject]
[audience, last-modified, level, name, size, sort, subject]
[last-modified, name, purpose, size, template-type]
[last-modified, level, name, size, sort, subject]
...
13 schemas, "file listings"
[age, last-change, name, rev, size]
[last-modified, name, size, type]
[date, description/link, live, name, pictures]
[date, name, size, type]
[age, last-change, name, rev]
[date, name, size]
[last-modified, name, size, subject]
[date, name]
[date, description/link, live, name, videos]
[modified, modified-by, name]
...
43 schemas, "file listings"
[last-modified, name, size]
[filename, last-modified, size]
[age, last-change, name, rev, size]
[last-modified, name, size, type]
[date, name, size, type]
[date, name, size]
[last-modified, name, size, subject]
[description, last-modified, name, size]
[name, size]
[actions, name, size, uploaded]
...
31 schemas, "file listings"
[code, name, price]
[name, sku]
[name, price]
[%, name, posts, rank]
[code, name]
[description, name]
[group, joined, name, posts]
[description, name, price]
[name, price, product-description]
[availability, code, name, price]
...
54 schemas, "retail"
[attachment, size]
[album, artist, comment, date/time, file, size...
[action, attachment, date, size, who]
[action, attachment, date, i, size, who]
[downloads, file, modified, size]
[file, format, size]
[bust, hips, size, waist]
[date, filename, size]
[modified, filename, size]
[attached-to, creator, date, name, size]
...
55 schemas, "file sharing"
commision%, from, name, to
city, state, dob, ht, name, no, pos, team, wt
ip, name, view-date
city, dob, ht, name, no, pos, wt, yr
college, ht, name, overall-rank, pos, team, yr
#, name
grade, name
errors, failures, name, tests, times
name, no, votes, yes
name, status, times
...
25 schemas, "school sports"
[l, m, s, size, xl, xxl]
[l, m, s, size, xl]
[size]
[l, m, s, size, xl, xs]
[l, m, s, size, xl, xs, xxl]
[l, m, s, size]
[date-modified, size]
[l, m, s, size, xs]
[c, cat, l, s, size]
[date, size, type]
...
12 schemas, "clothing"

Figure 4.16: 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.

table. 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 suggested by Halevy *et al.* [44]. Madhavan *et al.* [56] 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, the authors 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 (*e.g.*, Dhamankar *et al.* [33], Madhavan *et al.* [57], Madhavan *et al.* [56], and Rahm and Bernstein [71]). There has been some work on corpus-driven linguistic synonym-finding in the machine learning community. Pantel and Lin [55] found predicate-like synonyms (*e.g.*, *is-author-of* $\cong$ *wrote*) from text by examining statistics about use of objects (*e.g.*, *Mark Twain, book*) near the synonym candidates; there is no natural analogue to these objects in a relational schema setting. Turney [82] 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. Cong and Jagadish [89] used data statistics and notions of schema-graph-centrality to create a high-quality summary of a very complicated relational schema graph, but this was designed to apply to an existing highly-engineered schema, not a synthetically-produced graph as in our join traversal problem.

A number of tools have taken advantage of data statistics, whether to match schemas (Dhamankar *et al.* [33] and Doan *et al.* [34]), to find dependencies (Bell and Brockhausen [9] and Wong *et al.* [87]), or to group into tables data with many potentially-missing values (Cafarella *et al.* [24] and Miller and Andritsos [64]). 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.

4.8 Conclusions

We described the WebTables system, which is the first large-scale attempt to extract, manage, and leverage the relational information embedded in HTML tables on the Web. Self-evidently *extraction-focused*, WebTables also has a *domain-independent* extraction mechanism and a *domain-scalable* table search mechanism.

We 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*. 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. All of these applications are *domain-independent*.

Now that we can extract and manage tables for the Structured Web, we can turn to combining them in new and interesting ways. That is the subject of the next chapter.

Chapter 5
DATA INTEGRATION FOR THE STRUCTURED WEB

In previous chapters we demonstrated how to extract multiple type of Structured Web data, and in the case of tabular data, how to search and perform operations over it. With such massive and diverse data available, recombining and repurposing the data is a very tantalizing idea. For example, we could compile an authoritative list of conference program committee members from data on many years' worth of websites. However, integrating data from the Structured Web raises several challenges that are not addressed by current data integration systems or mashup tools. First, due to the vastness of the corpus, a user can never know all of the potentially-relevant databases ahead of time, much less write a wrapper or mapping for each one; the source databases must be discovered during the integration process, making the system *domain-scalable.* Second, the data is usually not published cleanly; the integration system must be *extraction-focused* and *domain-independent.* Finally, it must *computationally efficient.*

This chapter describes Octopus, a system that combines search, extraction, data cleaning and integration, and enables users to create new data sets from those found on the Structured Web.[1] The key idea underlying Octopus is to offer the user a set of best-effort operators that automate the most labor-intensive tasks. For example, the Search operator, which has some similarities to the table search application from WebTables, takes a search-style keyword query and returns a set of relevance-ranked and similarity-clustered structured data sources on the Web; the Context operator helps the user specify the semantics of the sources by inferring attribute values that may not appear in the source itself, and the Extend operator helps the user find related sources that can be joined to add new attributes to a table. Octopus executes some of these operators automatically, but always allows the user to provide feedback and correct errors. We describe the algorithms

[1]Much of this work, like WebTables, was done while the author was at Google, Inc.

underlying each of these operators and experiments that demonstrate their efficacy and computational scalability. All of these operators process tabular data that was extracted using the WebTables system from Chapter 4, as well as tabular data derived from HTML lists.

The result is a system that brings much of the power of traditional data integration, focused on limited domains and relatively few users, to the much broader dataset and userbase of the Web.

5.1 Problem Overview

There is enormous potential in combining and repurposing the data on the Structured Web. In principle, we could approach this challenge as a data integration problem, but at a bigger scale. In fact, some tools exist today for lightweight combination of sources found on the Web, such as Microsoft's Popfly [62] and Yahoo Pipes [88]. However, these tools and the traditional data integration approach fall short in several ways.

First, simply *locating* relevant data sources on the Web is a major challenge. Traditional data integration tools assume that the relevant data sources have been identified *a priori*. With Structured Web data, *domain-scalability* is a critical concern; clearly, it is unreasonable to ask humans to know about each of hundreds of millions of data sources without some kind of aid. Thus, we need a system that integrates well with search and presents the user with relevant structured sources.

Second, and perhaps obviously, Structured Web data sources must be extracted before they can be processed. Traditional systems and Web tools that assume XML-formatted inputs assume datasets that are already reasonably clean and ready to be used. On the Web, assuming XML data hugely limits the number of possible data sources, and thus the usefulness of the overall data integration tool. Hence, our system needs to be *extraction-focused* and *domain-independent*.

There is an added dimension to information extraction that applies in the data integration setting, and amounts to a third failing of traditional approaches. The semantics of Web data are often implicitly tied to the source Web page and must be made explicit prior to integration. For example, there are multiple tables on the Web with VLDB PC members, but the year of the conference is in the Web page text, not the table itself. In contrast, traditional data integration systems assume that any extra data semantics are captured by a set of hand-made semantic mappings, which are unavailable for a generic data source on the Web. Thus, our system must be able to recover any relevant and implicit columns for each extracted data source.

Data integration systems are customarily run on relational databases that have large numbers of tuples, but comparatively few tables and schema elements. Because the number of distinct potential data sources is so large, our solution must pay attention to *computational efficiency.*

There is one final distinction between traditional data integration and the kind of Structured Web-centric system we propose here. Data integration systems have been tailored for the use case in which many similar queries are posed against a federation of stable data sources. In contrast, our aim is to support a more Web-centric use case in which the query load consists of many transient tasks and the set of data sources is constantly changing. Indeed, a particular set of sources may be combined just once. It may be useful to apply an integration repeatedly to a single data source, but it is not necessary. The overriding assumption is that a user will simply perform each "online" integration as needed. Hence, our system cannot require the user to spend much time on each data source.

From the user's perspective, OCTOPUS integrations are onetime transient operations that are the product of a graphical user interface. However, under the covers user interactions with OCTOPUS produce declarative GLAV-style mappings that could in principle be exported to a more formal integration system, executed multiple times, or used toward other ends.

This chapter describes OCTOPUS, a system that combines search, extraction, data cleaning, and integration, and enables users to create new data sets from those found on the Web. The key idea underlying OCTOPUS is to offer the user a set of *best-effort operators* that automate the most labor-intensive tasks. For example, the SEARCH operator takes a search-style keyword query and returns a set of relevance-ranked and similarity-clustered structured data sources on the Web. The CONTEXT operator helps the user specify the semantics of the sources by inferring attribute values that may not appear in the source itself. The Extend operator helps the user find related sources that can be joined to add new attributes to a table. Octopus executes some of these operators automatically, but always allows the user to provide feedback and correct errors. The choice of operators in Octopus has a formal foundation in the following sense: each operator, as reflected by its intended semantics, is meant to recover some aspect of source descriptions in data integration systems. Hence, together, the operators enable creating integrations with a rich and well-defined set of descriptions.

This chapter describes the overall Octopus system and its integration-related operators. Octopus also has a number of extraction-related operators that we do not discuss. The specific contributions of this chapter are:

- We describe the operators of the Octopus system and how it enables data integration from Web sources.

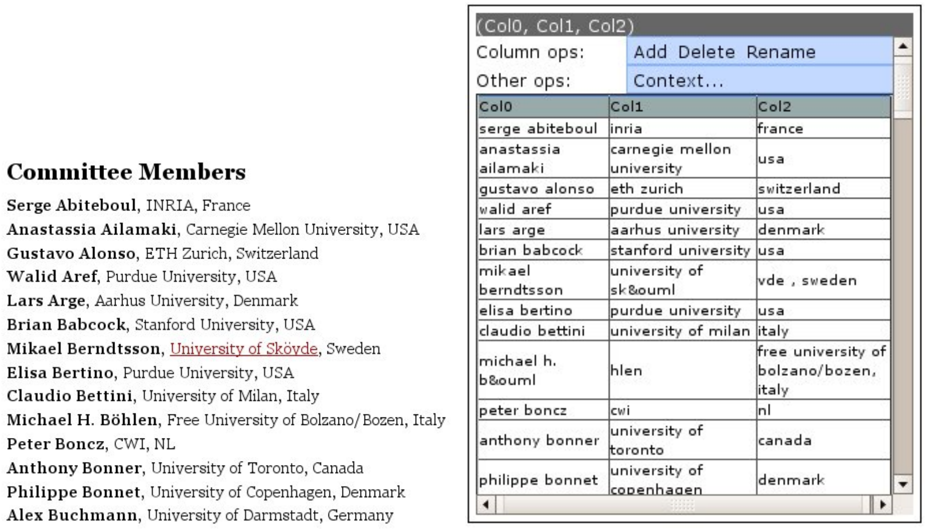
- We consider in detail the Search, Context, and Extend operators and describe several possible algorithms for each. The algorithms explore the tradeoff between computational overhead and result quality.

- We evaluate the system experimentally, showing high-quality results when run on a test query load. We evaluate most system components using data gathered from a general Web audience, via the Amazon Mechanical Turk.

We provide an overview of Octopus and give definitions for each of the operators in Section 5.2. We describe the algorithms for implementing the operators in Section 5.3. Sections 5.4 and 5.5 describe our implementation and experiments, and evaluate the algorithmic tradeoffs on an implemented system. Finally, we discuss related work in Section 5.6, and conclude in Section 5.7.

5.2 Octopus and its operators

We begin by describing the data that is manipulated by Octopus in Section 5.2.1, and then provide the formal motivation for our operators in Section 5.2.2. In Section 5.2.3 we define the Octopus operators.



Committee Members

Serge Abiteboul, INRIA, France
Anastassia Ailamaki, Carnegie Mellon University, USA
Gustavo Alonso, ETH Zurich, Switzerland
Walid Aref, Purdue University, USA
Lars Arge, Aarhus University, Denmark
Brian Babcock, Stanford University, USA
Mikael Berndtsson, University of Skövde, Sweden
Elisa Bertino, Purdue University, USA
Claudio Bettini, University of Milan, Italy
Michael H. Böhlen, Free University of Bolzano/Bozen, Italy
Peter Boncz, CWI, NL
Anthony Bonner, University of Toronto, Canada
Philippe Bonnet, University of Copenhagen, Denmark
Alex Buchmann, University of Darmstadt, Germany

Col0	Col1	Col2
serge abiteboul	inria	france
anastassia ailamaki	carnegie mellon university	usa
gustavo alonso	eth zurich	switzerland
walid aref	purdue university	usa
lars arge	aarhus university	denmark
brian babcock	stanford university	usa
mikael berndtsson	university of sk&ouml	vde , sweden
elisa bertino	purdue university	usa
claudio bettini	university of milan	italy
michael h. b&ouml	hlen	free university of bolzano/bozen, italy
peter boncz	cwi	nl
anthony bonner	university of toronto	canada
philippe bonnet	university of copenhagen	denmark

Figure 5.1: The screenshot to the left shows a region of the VLDB 2005 website; the extracted table to the right contains the corresponding data. This table was returned by Octopus after the user issued a Search command for `vldb program committee`. In this case, the table was extracted from an HTML list and the column-boundaries automatically recovered.

5.2.1 Data model

Octopus manipulates relations extracted from Web pages. The system currently uses WebTables-derived tables as well as HTML lists; but in principle it can operate on data obtained from any information extraction technique that emits relational-style data. For example, we might compile a two-column `born-in` relation from multiple TextRunner tuple extractions. We process HTML lists using techniques from Elmeleegy *et al.* [38].

Each extracted relation is a table T with k columns. As with WebTables, there are no strict type or value constraints on the contents of a single column in T. However, the goal of each extraction subsystem is for each table T should "look good" by the time the user examines the relation. A high-quality relation tends to have multiple domain-sensitive columns, each appearing to observe appropriate type rules. That is, a single column will tend to contain only strings which depict integers, or strings which are drawn from the same domain (*e.g.*, movie titles). Of course, the data tables may contain extraction errors as well as any factual errors that were present in the original material.

Each relation T also preserves its extraction lineage - its source Web page and location within that page - for later processing by Octopus operators such as Context. A single portion of crawled HTML can give rise to only a single table T.

To give an idea for the scale of the data available to Octopus, we found 3.9B HTML lists in a portion of the Google Web crawl. For tables, we estimated in Chapter 4 that 154M of 14B extracted HTML tables contain high-quality relational data, which is a relatively small percentage, slightly more than 1%. However, while HTML tables are often used for page layout, HTML lists appear to be used fairly reliably for some kind of structured data; thus we expect that a larger percent of them contain good tabular data. By a conservative estimate, our current Octopus prototype has at least 200M source relations (154M of them from WebTables) at its disposal.

5.2.2 Integrating Web Sources

To provide the formal motivation for Octopus's operators, we first imagine how the problem would be solved in a traditional data integration setting. We would begin by creating a mediated schema that could be used for writing queries. For example, when collecting data about program committees, we would have a mediated schema with a relation PCMember(name, institution, conference, year).

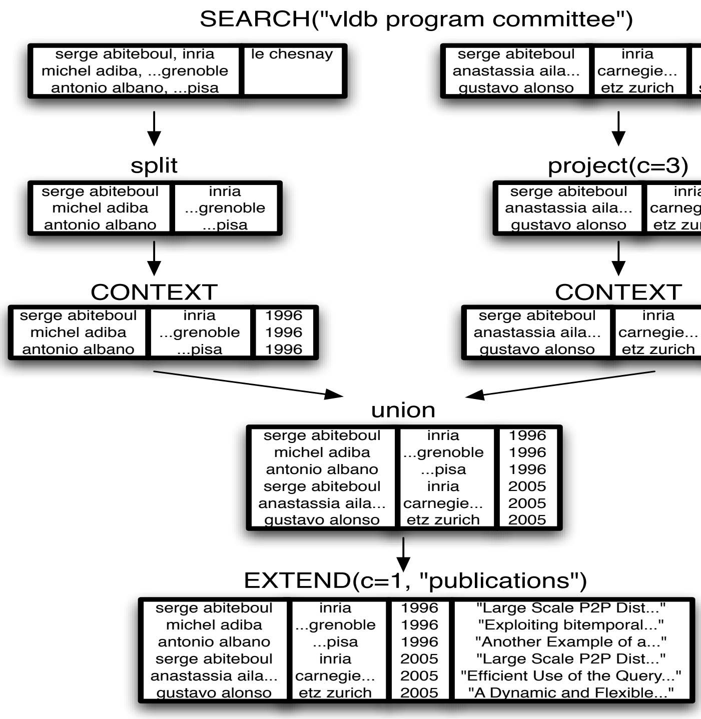
The contents of the data sources would be described with semantic mappings. For example, if we were to use the GLAV approach (described in Friedman *et al.* [41]) for describing sources on the Web, the pages from the 2008 and 2009 VLDB web sites might be described as follows:

VLDB08Page(N,I) $\subseteq$ PCMember(N,I,C,Y), C="VLDB", Y=2008
VLDB09Page(N,I) $\subseteq$ PCMember(N,I,C,Y), C="VLDB", Y=2009

Now if we queried for all VLDB PC members:

q(name, institution, year) :- PCMember(name, institution, "VLDB", year)

then the query would be reformulated into the following union:



SEARCH("vldb program committee")
serge abiteboul, inria
michel adiba, ...grenoble
antonio albano, ...pisa
le chesnay
serge abiteboul
anastassia aila...
gustavo alonso
inria
carnegie...
etz zurich
france
usa
switzerland
split
project(c=3)
CONTEXT
CONTEXT
1996
2005
union
EXTEND(c=1, "publications")
"Large Scale P2P Dist..."
"Exploiting bitemporal..."
"Another Example of a..."
"Efficient Use of the Query..."
"A Dynamic and Flexible..."

Figure 5.2: A typical sequence of Octopus operations. The data integration operators of Octopus are in upper-case type, while other operations in lower-case. The user starts with Search, which yields a cluster of relevant and related tables. The user selects two of these tables for further work. In each case, she removes the rightmost column, which is schematically inconsistent and irrelevant to the task at hand. On the left table she verifies that the table has been split correctly into two columns, separating the name and the institution. If needed, she may manually initiate an operator that will split a column into two. She then executes the Context operator on each table, which recovers the relevant VLDB conference year. Without this extra information, the two Serge Abiteboul tuples would be indistinguishable after union. Finally, she executes Extend to adorn the table with publication information for each PC member.

q'(name, institution, 2008) :- VLDB08Page(name, institution),

q'(name, institution, 2009) :- VLDB09Page(name, institution)

However, as we pointed out earlier, our data integration tasks may be transient or one-time affairs, and the number of data sources is very large. Therefore doing some kind of elaborate offline integration ahead of time, or otherwise preparing the data source descriptions in advance is infeasible. Our operators are designed to help the user effectively and quickly combine data sources by automatically recovering different aspects of an implicit set of source descriptions. (Of course, the recovered source descriptions will not have the nice human-understandable names we use in the examples in this chapter.)

Finding the relevant data sources is an integral part of performing the integration. Specifically, SEARCH initially finds relevant data tables from the myriad sources on the Web; it then clusters the results. Each cluster yielded by the SEARCH operator corresponds to a mediated schema relation, *e.g.*, the PCMember table in the example above. Each member table of a given cluster is a concrete table that contributes to the cluster's mediated schema relation. Members of the PCMember cluster correspond to VLDB08Page, VLDB09Page, and so on.

The CONTEXT operator helps the user to discover selection predicates that apply to the semantic mapping between source tables and a mediated table, but which are not explicit in the source tables themselves. For example, CONTEXT recovers the fact that VLDB08Page has a year=2008 predicate even though this information is only available via the VLDB08Page's embedding Web page. CONTEXT only requires a single concrete relation, along with its lineage, to operate on.

These two operators are sufficient to express semantic mappings for sources that are projections and selections of relations in the mediated schema. The EXTEND operator will enable us to express joins between data sources. Suppose the PCMember relation in the mediated schema is extended with another attribute, adviser, recording the Ph.D adviser of PC members. To obtain tuples for the relation PCMember we now have join the tables VLDB08Page and VLDB09Page with other relations on the Web that describe adviser relationships.

The EXTEND operator will find tables on the Web that satisfy that criterion. *I.e.*, it will find tables T such that:

$$\mathsf{PCMember(N,I,C,Y,\ Ad)} \subseteq \mathsf{VLDB08Page(N,I),\ T(N,Ad),\ C="VLDB",\ Y=2008}$$

or

$$\mathsf{PCMember(N,I,C,Y,\ Ad)} \subseteq \mathsf{VLDB09Page(N,I),\ T(N,Ad),\ C="VLDB",\ Y=2009}$$

Note that the adviser information may come from many more tables on the Web. At the extreme, each adviser tuple may come from a different source.

It is important to keep in mind that unlike traditional relational operators, Octopus's operators are not defined to have a *single* correct output for a given set of inputs. Consequently, the algorithms we present in Section 5.3 are also best-effort algorithms. On a given input, the output of our operators cannot be said to be "correct" vs "incorrect," but instead may be "high-quality" vs "low-quality." In this way, the Octopus operators are similar to traditional Web search ranking, or to the *match* operator in the model management literature, *e.g.*, Bernstein [11].

In principle, Octopus can also include cleaning operators such as data transformation and entity resolution, but we have not implemented these yet. Currently, with the operators provided by Octopus, the user is able to create integrations that can be expressed as select-project-union and some limited joins over structured sources extracted from the Web.

5.2.3 Integration operators

This chapter focuses on the three integration-related operators of Octopus: Search, Context and Extend. We now describe each one.

Search

The Search operator takes as input an extracted set of relations **S** and a user's keyword query string q. It returns a sorted list of clusters of tables in **S**, ranked by relevance to q. In our case, the set of relations can be considered all the tabular relations recovered from the Structured Web.

A relevance ranking phase of Search allows the user to quickly find useful source relations in **S** and is evaluated in a similar fashion to relevance in web search. A secondary clustering step allows the user to find relations in **S** that are similar to each other. Intuitively, tables in the same cluster can be described as projections and selections on a single relation in the mediated schema, and are therefore later good candidates for being unioned. Tables in a single cluster should be *unionable* with few or no modifications by the user. In particular, they should be identical or very similar in column-cardinality and their per-column attribute labels. OCTOPUS will present in a group all of the tables in a single cluster, but the user actually applies the union operation, removing tables or adjusting them as needed.

The output of SEARCH is a list **L** of table sets. Each set $\mathbf{C} \in \mathbf{L}$ contains tables from **S**. A single table may appear in multiple clusters **C**. The SEARCH operator may sort **L** for both relevance and diversity of results. As in web search, it would be frustrating for the user to see a large number of highly-ranked clusters that are only marginally different from each other.

Context

CONTEXT takes as input a single extracted relation T and modifies it to contain additional columns, using data derived from T's source Web page. For example, the extracted table about 2009 VLDB PC members may contain attributes for the member's `name` and `institution`, but not the `year`, `location`, or `conference-name`, even though this information is obvious to a human reader. The values generated by CONTEXT can be viewed as the selection conditions in the semantic mappings first created by SEARCH.

The CONTEXT operator is necessary because of a design idiom that is very common to tabular Structured Web data. Data values that hold for *every* tuple are generally "projected out" and added to the Web page's surrounding text. Indeed, it would be very strange to see a Web-embedded relation that has `2009` in every single tuple; instead, the `2009` value simply appears in the page's title or text. Consider that when a user combines several extracted data tables from multiple sources, any PC members who have served on multiple committees from the same institution will appear as duplicated tuples. In this scenario, making `year` and `location` explicit for each tuple would be very valuable.

Extend

EXTEND enables the user to add more columns to a table by performing a join. EXTEND takes as input a column c of table T, and a topic keyword k. The column c contains a set of values drawn from T, *e.g.*, a list of PC member names. k describes a desired attribute of c to add to T, *e.g.*, the "publications" that correspond to each name. EXTEND returns a modified version of T, adorned with one or more additional columns whose values are described by k.

EXTEND differs from traditional *join* in one very important way: any new columns added to T do not necessarily come from the same *single* source table. It is more accurate to think of EXTEND as a join operation that is applied independently between each row of T and *some other relevant tuple from some extracted table*; each such tuple still manages to be about k and still satisfies the join constraint with the row's value in c. Thus, a single EXTEND operation may involve gathering data from a large number of other sources. Note that join can be distributed over union and therefore OCTOPUS has the flexibility to consider individual sources in isolation.

Finally, note that k is merely a keyword describing the desired new column. It is not a strict "schema" requirement, and it is possible to use EXTEND to gather data from a table that uses a different label from k or no label at all. Hence, we are not requiring the user to know any mediated schema in advance. This decision reflects again the *domain-scalability* we believe is critical to any Structured Web tool. The user can express her information need in whatever terminology is natural to her.

5.2.4 Putting It All Together

OCTOPUS operators provide a "workbench"-like interactive setting for users to integrate Web data. From the user's perspective, each application of an operator further refines a working set of relations on the screen. We can also think of each operator as modifying an underlying semantic mapping that is never explicitly shown, but the on-screen data set reflects this mapping to the user.

In addition to the three data-integration oriented OCTOPUS operators, the system allows users to make "manual" changes to the tables and hence to the semantic mappings. For example, if the user disagrees with SEARCH's decision to include a table in a cluster she can simply delete the table from the cluster. Other currently supported operations include selection, projection, column-add, column-split, and column-rename. However, the architecture and user interaction are flexible and therefore we can add other data cleaning and transformation operators as needed.

Figure 5.1 shows two tables in a single SEARCH cluster after the user has issued a request for `vldb program committee`. Figure 5.2.2 shows the application's state in the form of a transition diagram, taking as input the user's decision to execute one of the integration operators. In short, an interactive session with OCTOPUS involves a single SEARCH followed by any combination of CONTEXT, EXTEND, and other cleaning and transformation operators. It is important to note, as illustrated in Figure 5.1, that different sequences of operator invocations will be appropriate depending on the data at hand. For example, if the user starts with a table of US cities, she may want to apply the EXTEND operator *before* the CONTEXT to first add mayor data to the table, and then recover the year in which the data was collected.

5.3 Algorithms

In this section we describe a series of algorithms that implement the OCTOPUS operators. The utility of a user's interaction with OCTOPUS largely depends on the quality of the results generated by SEARCH, CONTEXT, and EXTEND. We propose several novel algorithms for each operator, and describe why the current state of the art techniques do not suffice. None of our operator implementations are time-consuming in the sense of traditional algorithmic complexity, but some require data values (*e.g.*, word usage statistics) that can be burdensome to compute and pose some problems for computational efficiency. We further address computational issues in Section 5.4 below. We evaluate both result quality and runtime performance for each algorithm in Section 5.5.

5.3.1 SEARCH

The SEARCH operator takes a keyword query as input and returns a ranked list of table clusters. There are two challenges in implementing SEARCH. The first is to rank the tables by relevance to the user's query, and the second is to cluster other related tables around the top-ranking SEARCH results.

```
1: function SimpleRank(keywords):
2: urls = searchengine(keywords) //in search-engine-rank order
3: tables = []
4: for url ∈ urls do
5:     for table ∈extract_tables(url): //in-page order do
6:         tables.append(table)
7:     end for
8: end for
9: return tables
```

Figure 5.3: The **SimpleRank** algorithm.

Ranking

A strawman state-of-the-art algorithm for ranking is to leverage the ranking provided by a search engine. The **SimpleRank** algorithm, seen in Figure 5.3 simply transmits the user's SEARCH text query to a traditional Web search engine, obtains the URL ordering, and presents extracted structured data according to that ordering. For pages that contain multiple tables, **SimpleRank** ranks them according to their order of appearance on the page.

SimpleRank has several weaknesses. First, search engines rank individual whole pages, so a highly-ranked page that contains highly-relevant raw text can easily contain irrelevant or uninformative data. For example, a person's home page often contains HTML lists that serve as navigation aids to other pages on the site. Another obvious weakness is when multiple datasets are found on the same page, and **SimpleRank** relies on the very possibly misleading in-page ordering.

It would be better to examine the extracted tables themselves, rather than ranking the overall page where the data appears. Search engines traditionally rely on the tf-idf cosine measure, which scores a page according to how often it contains the query terms, and how unusual those terms are. Tf-idf works because its main observation generally holds true in practice – pages that are "about" a term t generally contain repetitions of t. However, this observation does not strongly hold for HTML lists: *e.g.*, the list from Figure 5.1 does not contain the terms `vldb program committee`. Further, any "metadata" about an HTML

```
function SCPRank(keywords):
// returns in search-engine-rank order
urls = search_engine(keywords)
tables = []
for url ∈ urls do
    for table ∈ extract_tables(url) do
        //returns in-page order
        tables.append((TableScore(keywords, table), table))
    end for
end for
return tables.sort()

function table_score(keywords, table)
column_scores = []
for column ∈ table do
    column_score = $\sum_{c \in column.cells} scp(keywords, c)$
end for
return max(column_scores)
```

Figure 5.4: The SCP and tablescore algorithms.

list exists only in the surrounding text, not the table itself, so we cannot expect to count hits between the query and a specific table's metadata. We attempted this approach in Chapter 4 when ranking extracted HTML tables, which do often carry metadata in the form of per-column labels.

An alternative is to measure the *correlation* between a query phrase and each element in the extracted database. Our **SCPRank** algorithm, seen in Figure 5.4 uses symmetric conditional probability, or SCP, to measure the correlation between a cell in the extracted database and a query term.

In the **SCPRank** algorithm we use the following terminology. Let s be a term. The value $p(s)$ is the fraction of Web documents that contain s:

$$p(s) = \frac{\#\ web\ docs\ that\ contain\ s}{total\ \#\ of\ web\ docs}$$

Similarly, $p(s_1, s_2)$ is the fraction of documents containing both s_1 and s_2:

$$p(s_1, s_2) = \frac{\#\ web\ docs\ that\ contain\ both\ s_1\ and\ s_2}{total\ \#\ of\ web\ docs}$$

The symmetric conditional probability between a query q and the text in a data cell c is defined as follows:

$$scp(q, c) = \frac{p(q, c)^2}{p(q)p(c)}$$

This formula determines how much more likely q and c appear together in a document compared to chance. For example, it is reasonable to believe that `Stan Zdonik` appears with `vldb program committee` on the same page much more frequently than might an arbitrarily-chosen string. Thus, *scp*(`Stan Zdonik`, `vldb program committee`) will be relatively large.

Symmetric conditional probability was first used by Lopes and DaSilva [30] for finding multiword units, such as bigrams or trigrams, in text. It is very similar to Pointwise Mutual Information [83]. In the measure generally used in text analysis, however, the $p(q, c)$ value measures the probability of q and c occurring in *adjacent* positions in a document. In our tabular Structured Web setting, a data cell is generally only adjacent to other data cells. Thus our **SCPRank** employs *non-adjacent* symmetric conditional probability, and only requires that q and c appear together in the same document.

Of course, the **SCPRank** algorithm scores tables, not individual cells. As Figure 5.4 shows, it starts by sending the query to a search engine and extracting a candidate set of tables. For each table, **SCPRank** computes a series of per-column scores, each of which is simply the sum of per-cell SCP scores in the column. A table's overall score is the maximum of all of its per-column scores. Finally, the algorithm sorts the tables in order of their scores and returns a ranked list of relevant tables. In Section 5.5 we show that **SCPRank** is time-consuming due to the huge number of required SCP scores, but that its result quality is very high.

Unfortunately, naïvely computing the SCP scores for **SCPRank** can pose a runtime performance issue. The most straightforward method is to use a search engine's inverted index to compute the number of indexed documents that contain both q and c. A single inverted index lookup is equivalent to an in-memory merge-join of "posting lists" - the integers that represent the documents that contain q and c. This operation can be done quickly - a classically-designed search engine performs an inverted index lookup for every Web search. But it is not completely trivial, as each posting list may run in the tens or even hundreds of millions of elements. We must merge a posting list for each *token* in q and c, so multiple-term queries or data values are more expensive.

The number of possible unique SCP scores is $O(kT^k)$, where T is the number of unique tokens in the entire Web corpus, and k is the number of tokens in q and c. Because T is likely to be in the millions, precomputing SCP is not feasible. We are unaware of any indexing techniques beyond the inverted index for computing the size of a document set, given terms in those documents. To make matters worse, **SCPRank** requires a large number of SCP scores: one for every data value in every extracted candidate table. A single table can contain hundreds of values, and a single search may elicit hundreds of candidate tables. Thus, to make **SCPRank** more tractable, we make two optimizations. First, we only compute scores for the first r rows of every candidate table. Second, as described in Section 5.4 below, we substantially reduce the search engine load by approximating SCP scores on a small subset of the Web corpus.

Clustering

We now turn to the second part of the SEARCH operator, clustering the results by similarity. Intuitively, we want the tables in the same cluster to be "unionable." Put another way, they should represent tables derived from the same relation in some notional mediated schema. For example, a good cluster that contains the two VLDB PC member tables from Figure 5.1 roughly corresponds to a mediated schema describing all PC members from all tracks across multiple VLDB conferences.

We frame clustering as a simple similarity distance problem. For a result table t in a ranked list of tables T, $cluster(t, T-t)$ returns a ranked list of tables in $T-t$, sorted in order of decreasing similarity to t. The generic $cluster()$ algorithm, seen in Figure 5.5, computes $dist(t, t')$ for every $t' \in T - t$. Finally, it applies a similarity score threshold that limits the size of the cluster centered around t. The difference between good and bad cluster results, *i.e.*, the difference between clusters in which the tables are *unionable* and those in which the tables have little to do with each other, lies in the definition for $dist()$.

Our first and simplest $dist()$ function is **TextCluster**, which is identical to a very

```
**function** cluster($T, thresh$):
clusters = []
**for** $t \in T$ **do**
    $cluster$ = singlecluster($t, T, thresh$)
    $clusters$.append($sizeof$($cluster$), $cluster$)
**end for**
return $clusters.sort()$

**function** singlecluster($t$, $T$, $thresh$):
clusteredtables = []
**for** $t' \in T$ **do**
    $d = dist(t, t')$
    **if** $d > thresh$ **then**
        $clusteredtables$.append($d, t'$)
    **end if**
**end for**
return $clusteredtables.sort()$
```

Figure 5.5: The generic cluster algorithm framework. Possible implementations of *dist*() are **TextCluster**, **SizeCluster**, and **ColumnTextCluster**.

popular and simple document clustering algorithm. **TextCluster** just computes the tf-idf cosine distance between the texts of table a and the text of table b. It does not preserve any column or row information.

Unfortunately, related tables may have few, if any, words in common. As an example, consider two sets of country names derived from a single underlying table, where one set covers the countries starting with ”A-M” and the other set covers ”N-Z”. These tables are two disjoint selections on the overall relation of country names. With no text necessarily in common, it is difficult to determine whether two data sets are related or not.

While similar tables may not contain overlapping text, data strings from the same data type will often follow roughly similar size distributions. **SizeCluster**, the second *dist*() function, computes a column-to-column similarity score that measures the difference in mean string length between them. The overall table-to-table similarity score for a pair of tables is the sum of the per-column scores for the best possible column-to-column matching. The best column-to-column matching maximizes the sum of per-column scores.

The final distance metric is **ColumnTextCluster**. Like **SizeCluster**, the **Column-**

TextCluster distance between two tables is the sum of the tables' best per-column match scores. However, instead of using differences in mean string length to compute the column-to-column score, **ColumnTextCluster** computes a tf-idf cosine distance using only text found in the two columns.

5.3.2 CONTEXT

The CONTEXT operator has a very difficult task: it must add data columns to an extracted table that are suggested by the table's surrounding text. For example, recall that for a listing of conference PC members, the conference's `year` will generally not be found in each row of the table - instead, it can be found in the page text itself. When the OCTOPUS user wants to adorn a table with this information, she only has to indicate the target table T, which already contains the necessary extraction-lineage information.

We developed three competing algorithms for CONTEXT. **SignificantTerms** is very simple. It examines the source page where an extracted table was found and returns the k terms with the highest tf-idf scores that do not also appear in the extracted data. We hope that terms that are important to a page, *e.g.*the VLDB conference year, will be repeated relatively often within the page in question and thus have a high term-frequency while being relatively rare on the Web as a whole.

The second algorithm is **Related View Partners**, or **RVP**. It looks beyond just the table's source page and tries to find supporting evidence on other pages. The intuition is that some Web pages may have already needed to perform a form of the CONTEXT operator and published the results. Recall that an extracted table of VLDB PC members is likely to contain `(name, institution)` data. Elsewhere on the Web, we might find a homepage for a researcher in the PC member table, identified by a `name` value. If that homepage lists the researcher's professional services, then it might contain explicit structured `(conference, year)` data. We can think of the researcher's professional services data as another view on a notional underlying mediated-schema relation, which also gave rise to the VLDB PC member data.

The **RVP** algorithm is described formally in Figure 5.6 and operates roughly as follows.

```
1: function RVPContext(table, source_page):
2: sig_terms = getSignificantTerms(source_page, table)
3: list_of_tables = []
4: for row ∈ table do
5:     list_of_tables.append(getRVPTables(row, sig_terms))
6: end for
7: terms = all terms that occur in list_of_tables
8: sort terms in descending order of # of tables each term occurs in
9: return terms
10:
11: function getRVPTables(row, sig_terms):
12: tables = SEARCH(row, topk = 5).extractTables()
13: return tables that contain at least one term from sig_terms
```

Figure 5.6: The RVP algorithm.

When operating on a table T, it first obtains a large number of candidate related-view tables, by using each value in T as a parameter to a new Web search and downloading the top-10 result pages. There is one such Web search for each cell in T. Because T may have a very large number of tuples, **RVP** limits itself to a sample of s rows. In our experiments below, we used $s = 10$.

RVP then filters out tables that are completely unrelated to t's source page, by removing all tables that do not contain at least one value from **SignificantTerms**(T). **RVP** then obtains all data values in the remaining tables and ranks them according to their frequency of occurrence. Finally, **RVP** returns the k highest-ranked values.

Our last algorithm, **Hybrid**, is a hybrid of the above two algorithms. It leverages the fact that the **SignificantTerms** and **RVP** algorithms are complementary in nature. **SignificantTerms** finds context terms that **RVP** misses, and **RVP** discovers context terms that **SignificantTerms** misses. The **Hybrid** algorithm returns the context terms that appear in the result of *either* algorithm. For the ranking of the context terms, the **Hybrid** interleaves the results starting with the first result of the **SignificantTerms** algorithm. We show in our experiments that **Hybrid** outperforms the **SignificantTerms** and **RVP** algorithms.

5.3.3 EXTEND

Recall that EXTEND attempts to adorn an existing table with additional relevant data columns derived from other extracted data sources. The user indicates a source table T, a join column c, and a topic keyword k. The result is a table that retains all the rows and columns of the original source table, with additional columns of row-appropriate data that are related to the topic.

It is important to note that any EXTEND algorithm must address two hard data integration problems. First, it must solve a *schema matching* problem, described in Rahm and Bernstein [71]. The task is to verify that new data added by EXTEND actually focus on the topic k, even if the terminology from the candidate page or table is different. For example, it may be that $k =$*publications* while an extracted table says *papers*. Second, it must solve a *reference reconciliation problem*, as in Dong *et al.* [36]. The goal is to ensure that values in the join column c match up if they represent the same real-world object, even if the string representation differs. For example, realizing that Alon Halevy and Alon Levy as the same person but both are different from Noga Alon.

We developed two algorithms for EXTEND that solve these problems in different ways, and extract the relevant data from sources on the Web. The algorithms largely reflect two different notions of what kinds of Web data exist.

The first, **JoinTest**, looks for an extracted table that is "about" the topic and which has a column that can join with the indicated join column. Schema matching for **JoinTest** relies on a combination of Web search and key-matching to perform schema matching. It assumes that if a candidate join-table was returned by a search for k, and the source table T and the candidate are joinable, then it's reasonable to think that the new table's columns are relevant to the EXTEND operation. The join test in this case eliminates from consideration many unrelated tables that might be returned by the search engine simply because they appear on the same *page* as a high-quality target table. Reference reconciliation for **JoinTest** is based on a string edit-distance test.

The **JoinTest** algorithm assumes that for each EXTEND operation, there is a single high-value "joinable" table on the Web that simply needs to be found. For example, it is

```
function MultiJoin($column$, $keyword$):
urls = []
for $cell \in column$ do
    $urls$ += searchengine($cell + keyword$)
    $tables$ = []
    for $url \in urls$ do
        for $table \in$ extract_tables($url$) do
            $table$.setscore(table_score($keywords$, $table$))
            $tables$.append($table$)
        end for
    end for
end for
sort $tables$
$clusters$ = []
for $table \in tables$ do
    $cluster$ = makecluster($table$)
    $cluster$.setscore(join_score($table$.getScore(),$column$, $cluster$))
    $clusters$.append($cluster$)
end for
sort $clusters$
return $clusters$

function join_score($tableScore$, $column$, $cluster$):
// Weight $w$ is a parameter of the system
$scoreCount = len(cluster$.getUniqueJoinSrcElts())
$score = scoreCount / len(column)$
return $(w * tableScore) + (1 - w) * score$
```

Figure 5.7: The **MultiJoin** algorithm. Take particular note of the `join_score()` function. The `getUniqueJoinSrcElts()` function returns, for a given cluster, the set of distinct cells from the original query *column* that elicited tables contained in the cluster. The size of its output, when normalized by the size of *column*, measures the degree to which a cluster "covers" data from the query column.

plausible that a source table that describes major US cities could thus be extended with the city column and the topic keyword `mayor`. On the other hand, it appears unlikely that we can use this technique to extend the set of VLDB PC members on the PC member column, with topic `publication`; this single table simply does not appear anywhere in our Web crawl even though the information is available scattered across many different tables.

JoinTest works by finding the table that is "most about k" while still being joinable to T on the c column. Because Web data is always dirty and incomplete, we can never expect a perfect join between two tables; instead, we use Jaccardian distance to measure the compatibility between the values in T's column c and each column in each candidate table. If the distance is greater than a constant threshold t, we consider the tables to be joinable. All tables that pass this threshold are sorted in decreasing order of relevance to k, as measured by a traditional Web search query. If there is *any* extracted table that can pass the join-threshold, it will be returned and used by EXTEND.

The second algorithm is **MultiJoin**. **MultiJoin** attempts to join each tuple in the source table T with a potentially-different table. It can thus handle the case when there is no single joinable table, as with VLDB PC members' publications. The algorithm resembles what a human search-user might do when looking to adorn a table with additional information. The user could extend a table piecemeal by performing a series of Web searches, one for each row in the table. Each search would include the the topic-specific keyword (*e.g.*, publications) plus the individual value for that row in column c (*e.g.*, a PC member's name). The user could then examine the huge number of resulting tables, and check whether any are both topic-appropriate and effectively join with the value in c. **MultiJoin**, shown in detail in Figure 5.7, attempts to automate this laborious process.

MultiJoin addresses the issue of schema matching via the column-matching clustering algorithm described in Section 5.3.1 above. Multiple distinct tables that are all about the same topic should appear in the same cluster. For each cluster, **MultiJoin** computes how many distinct tuples from the source table T elicited a member of the cluster; the algorithm then chooses the cluster with the greatest "coverage" of T. This clustering-based approach is roughly similar to data-sensitive schema-matching techniques, *e.g.*, Doan *et al.* [35].

Reference reconciliation in **MultiJoin** is partially solved as a by-product of using a search engine to find a separate table for each joinable-tuple. For example, a search for "Alon Levy" will yield many of the same results as a search for "Alon Halevy." This works for several reasons: pages that embed the tables will sometimes contain multiple useful labels, as in the "Alon Levy" case here. Also, search engines incorporate incoming anchor text that will naturally give data on a page multiple aliases. Finally, search engines include some amount of spelling correction, string normalization, and acronym-expansion.

Note that **MultiJoin** is similar to the SEARCH search-and-cluster framework, with two important differences:

1. When generating the list of raw web pages, **MultiJoin** issues a distinct web search query for every pair (v_c, q), where v_c is a value in column c of T. Because of how these queries are constructed, we can think of each elicited result table as having a "source join element" to which it is related, *i.e.*, v_c.

2. When ranking the resulting clusters, **MultiJoin** uses a combination of the relevance score for the ranked table, and a join score for the cluster. The join score counts how many unique values from the source table's c column elicited tables in the cluster via the web search step. This gives higher rank to clusters that extend the source table T more completely.

5.4 Implementation At Scale

The OCTOPUS system provides users with a new way of interacting deeply with the corpus of Web documents. As with a traditional search engine, OCTOPUS will require a lot of hardware and software in order to scale to many users. The main goal of this chapter is to show that the system can provide good-quality results, not to build the entire OCTOPUS back end software stack. That said, it is important to see whether OCTOPUS can ever provide low latencies for a mass audience. In this section, we step through a few of the special systems problems that OCTOPUS poses beyond traditional relevance-based Web search and show that with the right infrastructure, building a large-scale OCTOPUS service is feasible.

There are two novel operations executed by the algorithms from the section above, each of which could reasonably require a new piece of back-end infrastructure software were OCTOPUS to be widely deployed. They include **non-adjacent** SCP computations from SEARCH's **SCPRank** and **multi-query Web searches** from the CONTEXT's **RVP** algorithm and EXTEND's **MultiJoin** algorithm. All of these steps can be implemented using standard Web searches using just the hitcounts in the **SCPRank** case, but this is not a good solution. Search engines can afford to spend a huge amount of resources in order to quickly process a single query, but the same is unlikely to be true when a single OCTOPUS user yields tens of thousands of queries. Some OCTOPUS-specific infrastructure, however, can hugely reduce the required computational resources. In the first two cases, we implemented small prototypes for back-end systems. In the final case, we relied exclusively on approximation techniques to make it computationally feasible.

The first challenge, **non-adjacent** SCP statistics, are required by SEARCH's **SCPRank** algorithm. Unfortunately, we cannot simply precompute word-pair statistics, as we could if we focused only on adjacent words; each sampled document in the nonadjacent case would yield $O(w^2)$ unique token-combinations, even when considering just pairs of tokens. Therefore, we created a "miniature" search engine that would fit entirely in memory for fast processing. Using about 100 GB of RAM over 100 machines, we searched just a few million Web pages. We do not require absolute precision from the hitcount numbers, so we saved memory by representing document sets using Bloom Filters [12]. This solution is usable, but quantifying how much worse it is than a precise answer is a matter for future work.

The second challenge, **multi-query Web searches**, arises from the **RVP** and **MultiJoin** algorithms. The **Naïve RVP** implementation requires rd Web searches, where r is the number of tables processed by CONTEXT, and d is the average number of sampled non-numeric data cells in each table. For reasonable values of $r = 100$, $d = 30$, **RVP** may require several thousand search queries. Luckily, **RVP** computes a score that is applied to the table as a whole, so it may be reasonable to push d to fairly low values, drastically reducing the number of searches necessary. Further, as we will see in Section 5.5.3 below, **RVP** offers a real gain in quality, but whether it is enough to justify the extra cost of its Web search load is not clear. Exploring alternate index schemes for **RVP** is another interesting area for future work. **MultiJoin** has a similar, but smaller, problem of issuing a large number of search queries for each source table. It only needs to issue a single query per row.

5.5 Experiments

We now evaluate the quality of results generated by each of our operators: Search, Context, and Extend. We begin with a description of our query collection technique.

5.5.1 Collecting Queries

It is not obvious how to choose a sample set of queries for testing Octopus. Ideally, we would have drawn the test queries from real user data. Of course, Octopus is a research project with no user base beyond its developers, so there is no such data to obtain. We also considered using query logs from traditional Web search, but only a fraction of searches are meant to obtain structured data, and any mechanism to choose only the "structured" ones would have entailed the risk of "cherry-picking" queries that would work well. We do not know of a popular data-centric application with an appropriate query stream.

So, we chose to use the Amazon Mechanical Turk service to obtain a diverse query load suggested by Web users. It is a service that allows a *requester* to post an "intelligence task" along with an offered payment. Meanwhile, a *worker* can examine the offered tasks and choose to perform zero or more of them, earning payment upon completion. Example tasks include image labeling and text summarization.

We posted an intelligence task that asked each worker to "Suggest [one or two] topics for a useful data table (*e.g.*, **used cars** or **US presidents**)." We also asked each worker to supply two distinct URLs that provide an example table for the topic. Finally, we also included a nontechnical description of what makes a good table of data. We paid between twenty-five and fifty cents per task completed, and all submitted answers are included in our test query load. We removed all data suggested by one worker who did not even attempt to supply meaningful support URLs, and removed a few queries that were essentially duplicates. We otherwise kept the queries exactly as entered by the workers.

Of course, queries suggested by Turk workers may not be representative of what Octopus users will actually enter. But by collecting a test set in this manner, we avoided formulating queries ourselves, thus engaging in another form of cherry-picking. Also, by

- state capitals and largest cities in us	
- cigarette use among high school students	
- business expenditure on research and development	
- international educational exchange in the united states	
- usa pizza	- currencies of different countries
- fast cars	- 2008 beijing olympics
- mlb world series winners	- bittorrent clients
- phases of the moon	- australian cities
- video games	- usa population by state
- used cellphones	- science discoveries
- composition of the sun	- running shoes
- pain medications	- stock quote tables
- company income statements	- periodic table of elements
- world's tallest buildings	- north american mountains
- pre production electric vehicles	- pga leaderboard
- nba scoreboard	- ipod models
- olympus digital slrs	- 2008 olympic gold medal winners
- professional wrestlers	- exchange rates for us dollar
- fuel consumption	- wimbledon champions
- top grossing movies	- world religions
- us cities	- economy gdp
- car accidents	- stocks
- clothing sizes	- fifa world cup winners
- nutrition values	- dog breeds
- prime ministers of england	- country populations
- academy awards	- black metal bands
- ibanez guitars	- kings of africa
- world interest rates	

Table 5.1: List of queries used.

forcing each query to be supported by two example tables, we guaranteed that each suggestion is at least roughly "structured."

5.5.2 SEARCH

As we described in Section 5.3.1, there are two steps in implementing SEARCH: ranking and clustering. To compare the different algorithms for each, we used the aforementioned test set of 52 queries, each representing a starting point for a complex information gathering

Algorithm	Top 2	Top 5	Top 10
SimpleRank	27%	51%	73%
SCPRank	47%	64%	81%

Table 5.2: Fraction of top-k sets that contain at least one "relevant" table.

task. See Table 5.1 for the list of queries.

Ranking

We ran the ranking phase of SEARCH on each of the above 52 queries, first using the **SimpleRank** algorithm, and then **SCPRank**. For each input text query, the system outputs a ranked list of tables, sorted in order of relevance. We asked two judges, again drawn from the Amazon Mechanical Turk service, to independently examine each table/query pair and label the table's relevance to the query on a scale from 1-5. We mark a table as relevant only when both examiners give a score of 4 or higher.

We measure a ranking algorithm's quality by computing the average percentage of "relevant" results in the top-2, top-5, and top-10 emitted query results. Table 5.2 summarizes our results: on a plausible user-chosen workload, OCTOPUS returns a relevant structured table within the top-10 hits more than 80% of the time. Almost half the time, there is a relevant result within the top-2 hits. Note that **SCPRank** performs substantially better than **SimpleRank**, especially in the top-2 case. The extra computational overhead of **SCPRank** clearly offers real gains in result quality.

Clustering

Next, we evaluate the three clustering algorithms described in Section 5.3.1. The clustering system takes two inputs: a "center" table to cluster around, and a set of cluster candidates. A good cluster will contain tables that are similar to the center table, in both structure and data. As mentioned in Section 5.2.3, the component tables of a cluster should be unionable with few or no modifications. The working OCTOPUS system presents results by invoking the cluster algorithm once for each item in the ranked table results, with the remaining tables provided as the candidate set. Octopus then takes the resulting clusters and ranks them according to the average in-cluster relevance score. Without effective clustering, each resulting table group will be incoherent and unusable.

We tested the competing clustering algorithms using the queries in Table 5.1. We first issued each query and obtained a sorted list of tables using the **SCPRank** ranking algorithm. We then chose by hand the "best" table from each result set, and used it as the table center input to the clustering system. We ignore cluster performance on irrelevant tables; such tables are often not simply irrelevant to the query, but also of general poor quality, with few rows and empty values. Further, clusters centered on such tables are likely to be very low in the final output, and thus never seen by the user.

We assessed cluster quality by computing the percentage of queries in which a k-sized cluster contains a table that is "highly-similar" to the center. This value is computed for $k = 2$, $k = 5$, and $k = 10$. This number reveals how frequently clustering helps to find even a single related table. We determine whether a table pair is "highly-similar" by again asking two workers from the Amazon Mechanical Turk to rate the similarity of the pair on a scale from 1 to 5. If both judges give a similarity rating of 4 or higher, the two tables are marked as highly-similar.

The results in Table 5.3 show that even when the requested cluster has just two elements, giving the system only two "guesses" to find a table that is similar to the center, Octopus can generate a useful cluster 70% of the time. If the requested cluster size is larger ($k = 10$), then Octopus finds a useful cluster 97% of the time.

There is surprisingly little variance in quality across the algorithms. The good performance from the **Naïve SizeCluster** algorithm is particularly interesting. To make sure that this "single useful table" test was not too simple, we also computed the overall average user similarity score for tables in clusters of size $k = 2$, $k = 5$, and $k = 10$. As seen in Table 5.4, the user quality ratings for a given cluster size are very close to each other.

In the case of clustering, it appears that even very simple techniques are able to obtain good results. We conjecture that unlike a short text query, a data table is a very elaborately-described object that leaves relatively little room for ambiguity. It appears that many similarity metrics will obtain the right answer; unfortunately, no clustering algorithm is a

Algorithm	**k=2**	**k=5**	**k=10**
SizeCluster	70%	88%	97%
TextCluster	67%	85%	91%
ColumnTextCluster	70%	88%	97%

Table 5.3: Percentage of queries that have at least one table in top-k-sized cluster that is "very similar" to the cluster center. The percentage should generally increase as k grows, giving the algorithm more chances to find a good table.

Algorithm	**k=2**	**k=5**	**k=10**
SizeCluster	3.17	2.82	2.57
TextCluster	3.32	2.85	2.53
ColumnTextCluster	3.13	2.79	2.48

Table 5.4: Average user similarity scores between cluster center and cluster member, for clusters of size k. Higher scores are better. The average score should generally decrease as k increases, and the clustering algorithm must find additional similar tables.

very large consumer of search engine queries and so there is little computational efficiency to be gained here.

5.5.3 CONTEXT

In this section we compare and contrast the three Context algorithms described above. To evaluate the algorithms' performance, we first created a test set of *true context values*. First, we again took the first relevant table per query listed in Table 5.1, skipping over any results where there was either no relevant table or where all relevant tables were single-column. Next, two judges manually and independently reviewed each table's source Web page, noting terms in the page that appeared to be useful context values. Any context value that was listed by *both* reviewers was added to the test set. This process left a test set of 27 tables with a non-empty set of test context values. Within the test set, there is a median of three test context values per table, and thus, per query.

Figure 5.8 shows the results of our experiments for the Context operator. For each algorithm (**SignificantTerms**, **RVP**, and **Hybrid**), we measure the percentage of tables where a true context value is included in the top 1, top 2 or top 3 of the context terms generated by the algorithm. Because the OCTOPUS user generally examines CONTEXT results by hand, it is acceptable if CONTEXT's output contains some incorrect terms. The main goal for CONTEXT is to prevent the user from examining the source pages by hand.

We see that CONTEXT can adorn a table with useful data from the surrounding text over 80% of the time, when given 3 "guesses." Even in the top-1 returned values, CONTEXT finds a useful value more than 60% of the time.

The **Naïve SignificantTerms** algorithm does a decent, if not spectacular, job. For example, it finds a true context term in the top 3 results for 70% of the tables. Although the **RVP** algorithm does not outperform **SignificantTerms**, we can see from the **Hybrid** algorithm's performance that **RVP** is still helpful. Recall that the **Hybrid** algorithm interleaves the results of the **SignificantTerms** and **RVP** algorithms. Although the **RVP** and **SignificantTerms** results are not disjoint, **RVP** is able to discover new context terms that were missed by **SignificantTerms**. For example, looking again at the top-3, the **Hybrid** algorithm outperforms **SignificantTerms** algorithm by more than 16%; thus achieving an accuracy of 81%.

Even though **SignificantTerms** does not yield the best output quality, it is efficient and very easy to implement. Combining it with **RVP** algorithm results in improved quality, but because **RVP** can entail very many search engine queries, deciding between the two is likely to depend on the amount of computational resources at hand.

5.5.4 EXTEND

When evaluating the performance of EXTEND algorithms, we can imagine that the combination of each source tuple in T and the query topic k forms a "question set" that EXTEND attempts to answer. We compare **JoinTest** and **MultiJoin** by examining what percentage of T's rows were adorned with correct and relevant data. We do not distinguish between incorrect and nonexistent extensions.

Our test query set is necessarily more limited than the set shown in Table 5.1, many of

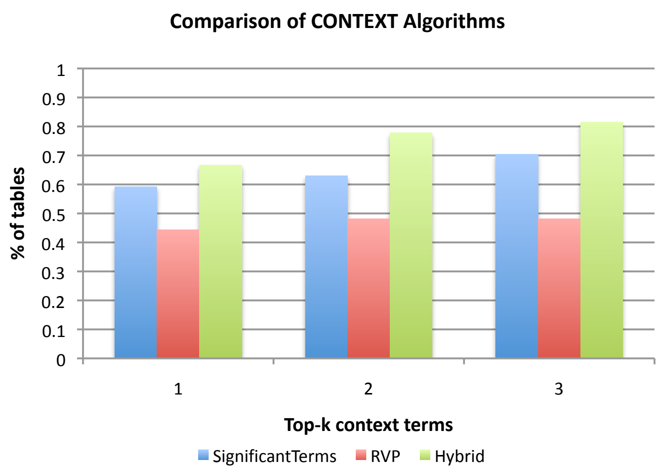


Figure 5.8: **A comparison of the CONTEXT algorithms. Shows the percentage of tables (y-axis) for which the algorithm returns a correct context term within the top-k (x-axis) context terms.**

Description of join column	**Topic query**
countries	universities
us states	governors
us cities	mayors
film titles	characters
UK political parties	member of parliament
baseball teams	players
musical bands	albums

Table 5.5: Test queries for EXTEND, derived from results from queries in Table 5.1.

which do not have much plausible "extra" information. For example, it is unclear how a user might want to add to the `phases of the moon` table. Further, the set of useful join keys is more limited than in a traditional database setting, where a single administrator has designed several tables to work together. Although numeric join keys are common and reasonable for a traditional database, in the Web setting they would suggest an implausible degree of cooperation between page authors. Labels, *e.g.*, place or person names, are more useful keys for our application. In addition, some table extensions might not be possible because the data simply does not exist on the Web.

We thus chose a small number of queries from Table 5.1 that appear to be EXTEND-able.

For each, we chose as the source table T the top-ranked human-marked "relevant" table returned by SEARCH. We chose the join column c and topic query k by hand, opting for values that appeared most amenable to EXTEND processing. For example, in the case of VLDB PC members, c is the name of the reviewer, not the reviewer's home institution; the topic query is `publications`. Table 5.5 enumerates the resulting test set.

The **JoinTest** algorithm only found extended tuples in three cases - countries, cities, and political parties. Recall that **JoinTest** tries to find a *single* satisfactory join table that covers all tuples in the source table. In these three cases, 60% of the tuples were extended. The remaining 40% of tuples could not be joined to any value in the join table. Each source tuple matched just a single tuple in the join table, except in the political parties case, where multiple matches to the party name are possible.

In contrast, the **MultiJoin** algorithm found EXTEND data for **all** of the query topics. On average, 33% of the source tuples could be extended. This rate of tuple-extension is much lower than in cases where **JoinTest** succeeds, but arguably shows the flexibility of **MultiJoin**'s per-tuple approach. Tables that are difficult to extend will be impossible to process with **JoinTest**, as a complete single table extension is simply unlikely to exist for many queries. With **MultiJoin**, fewer rows may be extended, but at least *some* data can be found.

The most remarkable difference between the two algorithms, however, is the sheer number of extensions generated. As mentioned, **JoinTest** generally found a single extension for each source tuple. In contrast, **MultiJoin** finds an average of *45.5 correct extension values* for every successfully-extended source tuple. For example, **MultiJoin** finds 12 *albums* by led zeppelin and 113 distinct *mayors* for new york. In this latter case, *mayor* extensions to new york obviously reflect mainly past office-holders. However, detecting the current mayor is an interesting area for future research.

In retrospect, this difference between **JoinTest** and **MultiJoin** is not surprising - if **JoinTest** could extend large numbers of tuples in a single table *and simultaneously* find many different values for each source tuple, it would suggest the existence of extremely massive and comprehensive tables. **MultiJoin** only requires that topic-and-tuple relevant data be discoverable on some page somewhere, not that all the source tuples will have all their topic data in exactly the same place. Because it appears that choosing between **JoinTest** and **MultiJoin** should depend on the underlying nature of the data being joined, in the future we would like to combine them into a single algorithm; for example, we might first attempt **JoinTest** and then move to **MultiJoin** if **JoinTest** fails to find a "good enough" joinable table.

Experimental Summary

Overall, our experiments show that it is possible to obtain high-quality results for all three Octopus operators discussed here. Even with imperfect outputs, Octopus already improves the productivity of the user, as generally the only alternative to these operators is to manually compile the data.

There are also promising areas for future research. Not only are there likely gains in output quality and algorithmic runtime performance, there are also interesting questions about reasoning about the data, as in the case of finding New York's current mayor. There has been some work by Kok and Domingos [53] in the textual information extraction area that we would like to build on using our system.

5.6 Related Work

Data integration on the Web is an increasingly popular area of work. The Yahoo! Pipes project [88] allows the user to graphically describe a "flow" of data, but it works only with structured data feeds and requires a large amount of work to describe data operations and schema matches between sources. There are other mashup tools available, including the Marmite system [86]. Karma [81] automatically populates a user's database, but still requires sources with formal declarations.

The CIMPLE system [31], mentioned in Chapter 2, allows administrators to design "community web sites" that incorporate data from many other sources. Its Web-centric data integration task appears very similar to the one faced by Octopus. However, CIMPLE's integration scenario is very traditional compared to the one that Octopus pursues: data integrations are assumed to be relatively static and very expensive to compute. In contrast, we believe that Octopus integrations will be quick to compile and easy to execute.

Raman and Hellerstein's Potter's Wheel [72] emphasizes live interaction between a data cleaner and the system. They offer several special cleaning operators, many of which are useful in a web setting, but do nothing to solve Web-centric problems such as data-finding.

5.7 Conclusions

We described the Octopus, a system for Structured Web data integration. We also presented the three novel Octopus operators, which observe our design criteria for Structured Web data tools. Unlike traditional data integration systems, the Search operator enables a user to find data sources indirectly by keyword, allowing for *domain-scalability* with respect to the large number of tables under management. The *extraction-oriented* Context operator allows the user to find relevant information from a data table's source Web page. The Extend allows users to express join requests over Structured Web sources, even when the join target is indicated only via keyword, again illustrating the importance of *domain-scalability*.

All of these operators are *domain-independent* and are built on the domain-independent extraction infrastructure from Chapter 4. Finally, we presented algorithms for all of these operators and showed that they can be implemented in *computationally efficient* ways, although there is room for future improvement in this area with certain algorithms.

We have now presented the three major parts of our Structured Web tool suite - TextRunner, WebTables, and Octopus. In the next and final chapter, we discuss areas for future work and conclude the dissertation.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This dissertation has described the challenges associated with managing Structured Web data, and presented three working Structured Web data systems: TextRunner, WebTables, and Octopus. These projects make substantial steps toward addressing the challenges of the Structured Web, which we summarize in Section 6.1 below. They also point the way to interesting future work, involving extensions to each system as well as new overall approaches. We discuss this future work in Section 6.2.

6.1 Contributions

In Chapter 1, this dissertation introduced four design criteria for Structured Web data management tools. We now briefly examine them and discuss the contributions made by three different systems that embody those criteria.

First, because we cannot assume that authors on the Web will ever agree to any standard format for data publishing, the tools should be *extraction-focused*. An extraction-focused tool finds structured data in whatever format it may currently be embodied, even a messy or incomplete one. Second, because of the number of topics on the Web and the Web's constant evolution, domain-dependent extractors or management techniques will inevitably miss interesting data. The tools must be *domain-independent* in order to apply to the Web's full breadth. Third, because there are so many distinct domains under management, the systems must be *domain-scalable*, requiring no undue amount of user work for each domain. Finally, the tools must be *computationally-efficient* enough to process data at the Web's size. It is difficult to imagine a system that ignores one or more of these criteria and yet successfully manages the entire Structured Web.

6.1.1 TextRunner Contributions

TextRunner is a *domain-independent*, *domain-scalable*, and *computationally-efficient* information extractor that operates over natural language Web text. The key contribution of the TextRunner system is its architecture, which allows us to extract all of the facts in a corpus in a single pass. It thus obtains a performance advantage over competing systems of several orders of magnitude. We described its three-part design:

- The **Self-Supervised Learner** uses a deep natural language parse and some TextRunner heuristics to distinguish good from bad extractions. This process yields high-quality results and is domain-independent. However, it is also extremely expensive, so we instead run this parse/heuristic combination over just a small random portion of the corpus. Instead of using these results directly, they are used to build a training dataset for a much more efficient extraction classifier.

- The **Single-Pass Extractor** is that inexpensive classifier applied to the entire corpus.

- The **Redundancy-Based Assessor** uses frequency counts to improve the Single-Pass Extractor's output quality.

These three steps allow TextRunner to efficiently obtain extractions from natural language Web text across many domains. It achieved a performance advantage of several orders of magnitude in comparison to the competing KnowItAll system, which requires the enumerated target relations ahead of time. TextRunner does so while retaining similar extraction quality.

6.1.2 WebTables Contributions

The WebTables system introduced three components for managing the relational information embedded in Web HTML tables. The first was a *domain-independent* extraction mechanism for recovering high-quality relations from the mass of HTML tables. We showed that this *extraction-oriented* system obtained results for table extraction with quality similar to other domain-independent extractors; it also enabled us to assemble the largest corpus of independent databases that we know of, by several orders of magnitude.

The second component was a table search engine that allowed a user to find extracted data in a *domain-scalable* way. We demonstrated that our search system retrieved tables with substantially better result quality than the Google commercial search engine.

Finally, we used the extracted tables to compile an unique data resource: the *attribute correlation statistics database*, or **ACSDb**. The **ACSDb** allowed us to construct several novel data-oriented services, such as synonym finding and schema autocomplete. Together, these three elements made major steps toward managing the massive set of tabular datasets available on the Web.

6.1.3 Octopus Contributions

Octopus introduced three novel operators for performing data integration among extracted Structured Web data tables. The *domain-scalable* Search operator enables a user to find data sources indirectly by keyword, using both relevance-ranking and clustering techniques. The *extraction-oriented* and *domain-independent* Context uses hints from the data tables and the user to extract relevant data values from each data table's source Web page. The Extend allows users to express join requests over the extracted tables. By allowing users to indicate the join target via keyword, Extend fulfills the goal of *domain-scalability.* We also provided *computationally-efficient* algorithms for implementing these operators.

We showed that each operator can obtain high-quality results on real Web data, across an independently-suggested and evaluated test set. The end result of Octopus was a system that allows a user to describe and execute a data integration task involving potentially dozens of sites drawn from a set of hundreds of millions. Because of the power of the Octopus operators, the user can do so without any professional training and using just a handful of clicks.

6.2 Future Work

Although each Structured Web project in this thesis has made substantial contributions, each has remaining weaknesses that can be improved in future work. Also, the experience gained through working with all three systems suggests several brand-new directions not tied to any one piece of preexisting work.

6.2.1 TextRunner

Michele Banko continued the extraction and linguistic work in TextRunner [8, 5], in particular focusing on replacements for the Naive Bayes Classifier that yielded improved extraction quality. However, there are areas where the TextRunner architecture could also be improved. One ripe target for future work is TextRunner's "single-sentence assumption," wherein each extracted tuple must be derived from a single standalone sentence. Although the model is simple and avoids many problems, it also leaves too much information behind that is obvious to a human reader. This is not surprising - natural language sentences are written to appear in an ordered stream, and a given sentence is often only sensible in light of the sentences that came before. As mentioned in Chapter 3, the single-sentence assumption strongly limits the complexity of a tuple that TextRunner can reasonably expect to obtain.

By allowing the author more space to describe an idea, tuples that are derived from multiple sentences could be much more complex. For example, timestamped tuples would probably be easier to obtain. It should also increase the triple yield, by admitting sentences that contain anaphora that is locally-resolvable. For example, the following text might yield two tuples instead of just one:

Colin Powell went to Brazil. He is a Republican.

Unfortunately, processing the multi-sentence text would be substantially more difficult. The current TextRunner translates from text to clean entities and relations by applying semantically-charged heuristics to the linguistic information. These heuristics are strongly tied to a simple declarative fact model that is well-matched to single English-language sentences. Processing multiple sentences would entail a much more complicated set of heuristics, which would cover not just anaphor resolution and object reconciliation, but also inter-sentence notions such as time and causality. All of these problems are major research questions in Natural Language Processing, making future work along these lines seem quite daunting. However, it may be possible to use the scale of Web data and clever use of frequencies to make progress in this area.

6.2.2 WebTables

There are two major directions for future work in WebTables.

The first is to examine the WebTables relation search engine application. Ranking may be improved by incorporating a stronger signal of source-page quality (such as PageRank). We currently include page quality information only very indirectly, by using the page ranking given by a traditional non-table-aware search engine. A separate and very exciting direction is to expose additional structured operations in the search engine results page. For example, a system that automatically computes relevant visualizations, a demonstration of which can be seen in Figure 4.7, would dramatically improve access to a very useful data management tool.

The second direction is to expand work with the **ACSDb**-based applications into a larger class of *semantic services* that can serve as infrastructure for other data-oriented applications. The schema autocomplete and synonym-generation systems are two examples. It should be possible to construct several other services, as well:

- Given a name of an attribute, the system should return a set of values that could plausibly populate a column with the same name. Such a system could be useful for testing data quality in databases; it could also be used to automatically fill out forms in order to surface Deep Web content.

- Given a set of data, the system should return a schema that fits the data well. This would be useful in "schema-light" data scenarios like spreadsheets, where the data may come before any rigorous data design.

- Given an entity, the system should return a set of properties associated with the entity. For example, given `University-of-Washington`, the system should return its location, date of founding, list of departments, etc. Such a service would be useful for query expansion.

Clearly, these semantic services will require looking at the extracted tabular content as well as the schema information.

6.2.3 Octopus

Improvements to Octopus fall into two major categories: improved algorithms for interacting with the existing system, and additional reasoning techniques to apply to the managed data.

An important step for Octopus is to enable entirely interactive-speed user operations. Currently, some algorithms are computationally very intensive and can take up to a few minutes. For non-adjacent SCP statistics, used by the **SCPRank** algorithm, the system requires that the data be entirely memory-resident for performance reasons. Even with a budget of 100GB of RAM, the entire Web statistics are too large; Octopus approximates these statistics by using a subset of the overall corpus as well as noisy Bloom filters. In the future we should quantify the tradeoff between memory size and result quality.

The **RVP** algorithm, which finds join relationships between extracted tables, also poses substantial performance problems. Research into join-indexing techniques may be very effective at improving the **RVP** runtime.

Finally, it would be interesting to examine the semantics of Octopus data in more detail. For example, recall the *mayor* of *new york* join operation in Section 5.5.4 that yielded 113 distinct mayors; obviously, there are not 113 *current* mayors of New York City. If the system could automatically detect when data is temporally-sensitive, it could present those 113 mayors in a way that makes the time relationship obvious. There may be other useful relationships to detect, such as many-to-one relationships (*e.g.*, cities and their containing state) or functional relationships (*e.g.*, a state and its capital city).

6.2.4 Multiple Extractors

One of the central assumptions of this dissertation has been that domain-independent extraction is sufficient to obtain a comprehensive version of the data in the Structured Web. If this were true, then it should be possible to take any two domain-independent extractors, apply them each to crawls of the Web that are roughly similar, and obtain roughly similar results.

However, there is some evidence that suggests this is not the case. Table 6.1(a) shows the ten most-popular textual relations extracted from TextRunner data. They focus largely on biographical and "person-centric" data. Table 6.1(b) shows the topics of the ten most-popular relational schemas from the WebTables data. They focus mainly on technical and transactional-style information. The stark difference in data types between the two lists suggests something that is quite plausible: that when a user chooses a topic, she may also choose an appropriate data model with which to express the data.

(a) TextRunner
be/is
ask/call
arrive/come/go
join/lead
born-in
created-in
is-not
is-city-in
beat/defeat
located-in

(b) WebTables
web access logs
file listings
forum posts
album listings
phone numbers
search queries
product catalog
real estate listing
auction data
manufacturing data

Table 6.1: On the left, the most popular relation strings extracted by TextRunner. On the right, the most popular topics for schemas from WebTables-extracted relations. Even though we ran two domain-independent extractors on a large general web crawl, the outputs are very different.

The initial goal for future work in this area must be to quantify more finely how much these two extractors agree or disagree. If the extractors truly diverge as much as they appear to, then obtaining a truly comprehensive Structured Web data set will be much harder than it first appeared. One possible solution is to operate many extractors in parallel and combine their results, an approach we first sketched in Cafarella [19]. Combining extractions from multiple sources entails a very large deduplication challenge. There has been substantial recent work in this area which may apply, *e.g.*, Arasu *et al.* [3], Benjelloun *et al.* [10], and Dong *et al.* [36].

6.2.5 Extractor Execution Model

All of the extractors in this dissertation have followed a simplistic execution model: each extractor runs over source material, computes its output, and terminates. There are several weaknesses with this approach that we would like to fix in the future.

First, real-world content is updated over time and so there may be quality or efficiency advantages in designing extractors that "remember" previous executions. At the very least, it seems hugely wasteful to re-extract data from Web pages that remain unchanged or change in only minor ways in between extractor executions.

Second, in many deployment scenarios, users will want to correct the extraction errors that arise from even the best systems. These cannot simply be applied to a previously-extracted database: it will be enormously frustrating for users if their corrections are clobbered by future extraction runs. In addition, the extractor should incorporate the user feedback for future runs. One approach to solving both problems is to use a much more sophisticated extraction model, in which database facts are backed by extractions from a large number of sources, both automated and human. In many aspects, such a system would resemble the multiextractor problem mentioned above.

We experimented to a small degree with extractions from multiple sources in the ExDB system [24]. The motivation of ExDB was to allow general queries over data extracted from a small number of extraction mechanisms - it was a very simple version of the multiextractor problem. However, our experience with ExDB may offer lessons for future work involving multiple extractors as well as human/extractor interaction.

ExDB allowed users to write Datalog-style queries over extractions stored in a probabilistic database; these extractions came from KnowItAll, TextRunner, and a mechanism for finding attribute synonyms. Unlike queries written over a traditional database, the user did not have to know about a schema ahead of time - she could simply use a binary predicate in the query and assume that ExDB's large extracted dataset would contain information for that predicate. The probabilities associated with each extraction allowed the system to incorporate multiple sources of information when computing each potential output tuple. For example, a tuple that lists `Edison` as a `scientist` and as having `invented` the `lightbulb` might have a probability that is the product of two extractions: the TextRunner-style fact triple and the KnowItAll-derived `scientist` hypernym.

The system had two major weaknesses that any future work will have to address. Both problems involved the "synthesized" type of tuple mentioned above. First, even though the concrete stored tuples had probabilities that appeared to be accurate, and ExDB's probabilistic inference method was correct, the synthesized tuples often suffered from inaccurate probabilities. It appears that this problem was at least partly due to an overly-simplistic probabilistic model, which treated each individual concrete tuple as an independent event. Thus, a "wide" output tuple that combined many concrete source tuples would often have a low probability, simply by virtue of being the product of a large number of terms that were close to, but less than, 1. A clear area to explore is a model that does not treat all extractions as independent, particularly if the extractions are derived from the same source document or website.

The second major problem with ExDB was the computational burden of computing the answer to a query. Even simple queries on the extraction database (compiled from just 90M pages) can take over a minute to run. Luckily, the domain area may offer a straightforward solution - it is not clear that users can distinguish between the quality of two extractions that differ quite substantially in their probabilities. The output of an extraction inference system like ExDB, whether run on its own; embedded inside a multiextraction system; or as the core of an extractor/user interaction system; may be robust to inferred probabilities that are very roughly approximated. We would like to examine improving query processing time by aggressively limiting the accuracy of the computed probabilities, and examining the impact on the above-mentioned applications.

BIBLIOGRAPHY

[1] Eugene Agichtein, Luis Gravano, Viktoriya Sokolovna, and Aleksandr Voskoboynik. 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] Arvind Arasu, Christopher Ré, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, pages 722–735, 2007.

[5] Michele Banko. *Open Information Extraction for the Web*. PhD thesis, University of Washington, Seattle, 2009.

[6] Michele Banko, Michael J. Cafarella, Stephen Soderland, Matthew Broadhead, and Oren Etzioni. Open Information Extraction from the Web. In *IJCAI*, pages 2670–2676, 2007.

[7] Michele Banko and Oren Etzioni. Strategies for lifelong knowledge extraction from the web. In *K-CAP*, pages 95–102, 2007.

[8] Michele Banko and Oren Etzioni. The tradeoffs between traditional and open relation extraction. In *HLT-NAACL*, 2008.

[9] Siegfried Bell and Peter Brockhausen. Discovery of data dependencies in relational databases. In *European Conference on Machine Learning*, 1995.

[10] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 2009.

[11] Philip A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[13] Philip Bohannon, Srujana Merugu, Cong Yu, Vipul Agarwal, Pedro DeRose, Arun Iyer, Ankur Jain, Vinay Kakade, Mridul Muralidharan, Raghu Ramakrishnan, and Warren Shen. Purple sox extraction management system. *SIGMOD Record*, 37(4):21–27, 2008.

[14] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD Conference*, pages 1247–1250, 2008.

[15] Charles P. Bourne and Trudi Bellardo Hahn. *A History of Online Information Services, 1963-1976*. The MIT Press, 2003.

[16] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey 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.

[17] Eric Brill and Grace Ngai. Man* vs. machine: A case study in base noun phrase learning. In *ACL*, 1999.

[18] Sergey Brin. Extracting Patterns and Relations from the World Wide Web. In *WebDB*, pages 172–183, 1998.

[19] Michael J. Cafarella. Extracting and querying a comprehensive web database. In *CIDR*, 2009.

[20] Michael J. Cafarella, Doug Downey, Stephen Soderland, and Oren Etzioni. Knowitnow: Fast, scalable information extraction from the web. In *HLT/EMNLP*, 2005.

[21] Michael J. Cafarella, Alon Halevy, and Nodira Khoussainova. Data Integration for the Relational Web. *PVLDB*, 2(1), 2009.

[22] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. WebTables: Exploring the Power of Tables on the Web. *PVLDB*, 1(1):538–549, 2008.

[23] Michael J. Cafarella, Alon Y. Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. Uncovering the Relational Web. In *WebDB*, 2008.

[24] Michael J. Cafarella, Christopher Re, Dan Suciu, and Oren Etzioni. Structured querying of web text data: A technical challenge. In *CIDR*, pages 225–234, 2007.

[25] 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.

[26] Peter P. Chen. The entity-relationship model - toward a unified view of data. *Transactions on Database Systems*, 1(1):9–36, 1976.

[27] Kenneth Ward Church and Patrick Hanks. Word association norms, mutual information, and lexicography. In *Proceedings of the 27th Annual Association for Computational Linguistics*, 1989.

[28] *CODASYL: Feature Analysis of Generalized Data Base Management Systems.* ACM, May 1971.

[29] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[30] J Ferreira da Silva and G. P. Lopes. A local maxima method and a fair dispersion normalization for extracting multi-word units from corpora. *Sixth Meeting on Mathematics of Language*, 1999.

[31] Pedro DeRose, Warren Shen, Fei Chen, AnHai Doan, and Raghu Ramakrishnan. Building structured web community portals: A top-down, compositional, and incremental approach. In *VLDB*, pages 399–410, 2007.

[32] Pedro DeRose, Warren Shen, Fei Chen, Yoonkyong Lee, Douglas Burdick, AnHai Doan, and Raghu Ramakrishnan. Dblife: A community information management platform for the database research community (demo). In *CIDR*, pages 169–172, 2007.

[33] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Y. Halevy, and Pedro Domingos. imap: Discovering complex mappings between database schemas. In *SIGMOD Conference*, 2004.

[34] AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.

[35] AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, pages 509–520, 2001.

[36] Xin Dong, Alon Y. Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, pages 85–96, 2005.

[37] Doug Downey, Oren Etzioni, and Stephen Soderland. A probabilistic model of redundancy in information extraction. In *IJCAI*, pages 1034–1041, 2005.

[38] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. Harvesting Relational Tables from Lists on the Web. *PVLDB*, 1(3), 2009.

[39] Oren Etzioni, Michael J. Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-Scale Information Extraction in KnowItAll: (Preliminary Results). In *WWW*, pages 100–110, 2004.

[40] Oren Etzioni, Michael J. Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artif. Intell.*, 165(1):91–134, 2005.

[41] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *AAAI/IAAI*, pages 67–73, 1999.

[42] Wolfgang Gatterbauer, Paul Bohunsky, Marcus Herzog, Bernhard Krüpl, and Bernhard 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.

[43] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[44] Alon Y. Halevy, Oren Etzioni, AnHai Doan, Zachary G. Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *CIDR*, 2003.

[45] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep web. *Commun. ACM*, 50(5):94–101, 2007.

[46] Bin He, Zhen Zhang, and Kevin Chen-Chuan Chang. Knocking the door to the deep web: Integration of web query interfaces. In *SIGMOD Conference*, pages 913–914, 2004.

[47] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[48] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[49] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. Naga: harvesting, searching and ranking knowledge. In *SIGMOD Conference*, pages 1285–1288, 2008.

[50] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. Naga: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[51] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *ACL*, pages 423–430, 2003.

[52] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[53] Stanley Kok and Pedro Domingos. Extracting semantic networks from text via relational clustering. In *ECML/PKDD (1)*, pages 624–639, 2008.

[54] Nicholas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *IJCAI (1)*, pages 729–737, 1997.

[55] Dekang Lin and Patrick Pantel. Dirt: Discovery of inference rules from text. In *KDD*, 2001.

[56] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Y. Halevy. Corpus-based schema matching. In *ICDE*, 2005.

[57] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *VLDB*, 2001.

[58] Jayant Madhavan, Alon Y. Halevy, Shirley Cohen, Xin Luna Dong, Shawn R. Jeffery, David Ko, and Cong Yu. Structured data meets the web: A few observations. *IEEE Data Eng. Bull.*, 29(4):19–26, 2006.

[59] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Y. Halevy. Google's deep web crawl. *PVLDB*, 1(2):1241–1252, 2008.

[60] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* MIT Press, 1999.

[61] Imran R. Mansuri and Sunita Sarawagi. Integrating unstructured data into relational databases. In *ICDE*, 2006.

[62] Microsoft Popfly. http://www.popfly.com/.

[63] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[64] Renee Miller and Periklis Andritsos. Schema discovery. *IEEE Data Eng. Bull.*, 26(3):40–45, 2003.

[65] Ion Muslea. Extraction patterns for information extraction tasks: A survey. In *In AAAI-99 Workshop on Machine Learning for Information Extraction*, pages 1–6, 1999.

[66] Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93–114, 2001.

[67] Grace Ngai and Radu Florian. Transformation based learning in the fast lane. In *NAACL*, 2001.

[68] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[69] 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.

[70] Peter Pirolli, James E. Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structures from the web. In *CHI*, pages 118–125, 1996.

[71] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[72] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[73] Ellen Riloff. Automatically constructing a dictionary for information extraction tasks. In *AAAI*, pages 811–816, 1993.

[74] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[75] Warren Shen, Pedro DeRose, Robert McCann, AnHai Doan, and Raghu Ramakrishnan. Toward best-effort information extraction. In *SIGMOD Conference*, pages 1031–1042, 2008.

[76] Yusuke Shinyama and Satoshi Sekine. Preemptive information extraction using unrestricted relation discovery. In *HLT-NAACL*, 2006.

[77] Stephen Soderland. Learning to extract text-based information from the world wide web. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, 1997.

[78] Michael Stonebraker and Joseph M. Hellerstein, editors. *Readings in Database Systems, Fourth Edition.* The MIT Press, 2005.

[79] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[80] Beth M. Sundheim and Nancy A. Chinchor. Survey of the message understanding conferences. In *HLT '93: Proceedings of the workshop on Human Language Technology*, pages 56–60, Morristown, NJ, USA, 1993. Association for Computational Linguistics.

[81] Rattapoom Tuchinda, Pedro A. Szekely, and Craig A. Knoblock. Building data integration queries by demonstration. In *Intelligent User Interfaces*, pages 170–179, 2007.

[82] Peter D. Turney. Mining the web for synonyms: Pmi-ir versus lsa on toefl. In *ECML*, pages 491–502, 2001.

[83] Peter D. Turney. Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL. *CoRR*, 2002.

[84] Yalin Wang and Jianying Hu. A machine learning based approach for table detection on the web. In *WWW*, pages 242–250, 2002.

[85] I.H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufman, San Francisco, 2nd edition edition, 2005.

[86] Jeffrey Wong and Jason I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI*, pages 1435–1444, 2007.

[87] S.K.M. Wong, C.J. 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.

[88] Yahoo Pipes. http://pipes.yahoo.com/pipes/.

[89] Cong Yu and H. V. Jagadish. Schema summarization. In *VLDB*, pages 319–330, 2006.

[90] R. Zanibbi, D. Blostein, and J.R. Cordy. A survey of table recognition: Models, observations, transformations, and inferences, 2003.