Data Integration for the Relational Web

Michael J. Cafarella*
University of Washington
Seattle, WA 98195, USA
mjc@cs.washington.edu

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

Nodira Khoussainova
University of Washington
Seattle, WA 98195, USA
nodira@cs.washington.edu

ABSTRACT

The Web contains a vast amount of structured information such as HTML tables, HTML lists and deep-web databases; there is enormous potential in combining and re-purposing this data in creative ways. However, integrating data from this *relational web* raises several challenges that are not addressed by current data integration systems or mash-up tools. First, the structured data is usually not published cleanly and must be extracted (say, from an HTML list) before it can be used. Second, 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. Third, some of the important information regarding the data is only present in its enclosing web page and needs to be extracted appropriately.

This paper 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, 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 underlying each of these operators and experiments that demonstrate their efficacy.

1. INTRODUCTION

The Web contains a massive amount of structured data such as tables [4] and lists, and there is enormous potential in combining and re-purposing this data. For example, consider the tasks of creating a comprehensive table of VLDB PC members of the last ten years, or of collecting a list of cafes all over the world with various attributes such as customer-rating and wifi-availability. The relevant data is available on the Web, but performing these tasks today is a time-consuming and burdensome operation with little support from existing database and search systems.

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 [14, 21]. 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 *apriori*. This assumption is clearly unreasonable at the Web's vast scale. Thus, we need a system that integrates well with search and presents the user with relevant structured sources.

Second, most data sources are embedded in Web pages and must be "prepared" before they can be processed. This step involves extracting the relational data from the surrounding HTML. 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 offer Web-specific data-extraction operators.

Third, 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.

Finally, 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 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.) Hence, our system cannot require the user to spend much time on each data source.

This paper 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-

*Work done while all authors were at Google, Inc.

VLDB '09, August 24-28, 2009, Lyon, France

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 paper 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 paper 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 overview OCTOPUS and give definitions for each of the operators in Section 2. We describe the algorithms for implementing the operators in Section 3. Sections 4 and 5 describe our implementation and experiments, and evaluate the algorithmic tradeoffs on an implemented system. Finally, we conclude with a discussion of related and future work in Sections 6 and 7.

2. OCTOPUS AND ITS OPERATORS

We begin by describing the data that is manipulated by OCTOPUS (Section 2.1), and then provide the formal motivation for our operators (Section 2.2). In Section 2.3 we define the OCTOPUS operators.

2.1 Data model

OCTOPUS manipulates relations extracted from Web pages. The system currently uses data extracted from HTML tables and HTML lists, but in principle can manipulate data obtained from any information extraction technique that emits relational-style data (*e.g.*, we might use a transformed version of the outputs of [10, 17]). We extract data from HTML tables using techniques described in [3], and process HTML lists using techniques from [9].

Each extracted relation is a table T with k columns. There are no strict type or value constraints on the contents of a single column in T. However, the goal of OCTOPUS's extraction subsystem is for each T to "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 resulting data tables may contain extraction errors as well as any factual errors that were present in the original source 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 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 [3] 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 relational sources at its disposal.

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 use the GLAV approach [11] for describing sources on the Web, the pages from the 2008 and 2009 VLDB web sites would 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:

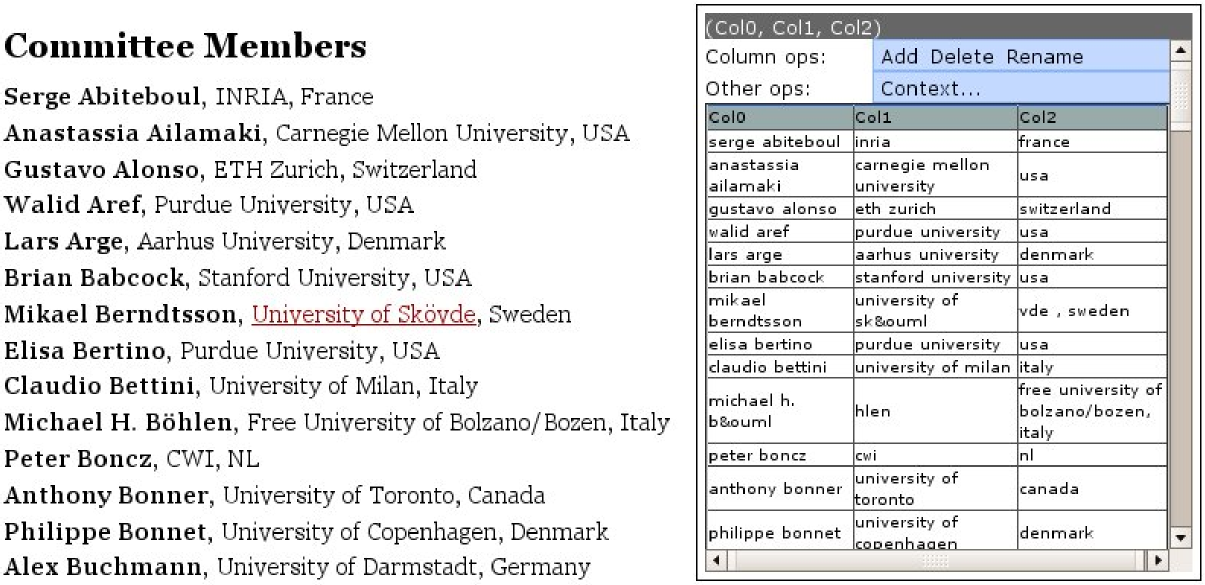
q(name, institution, year) :-
  PCMember(name, institution, "VLDB", year)

then the query would be reformulated into the following union:

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 and the number of data sources is very large. Therefore 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.

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.)



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 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

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:

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

or

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

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 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 [1]).

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.

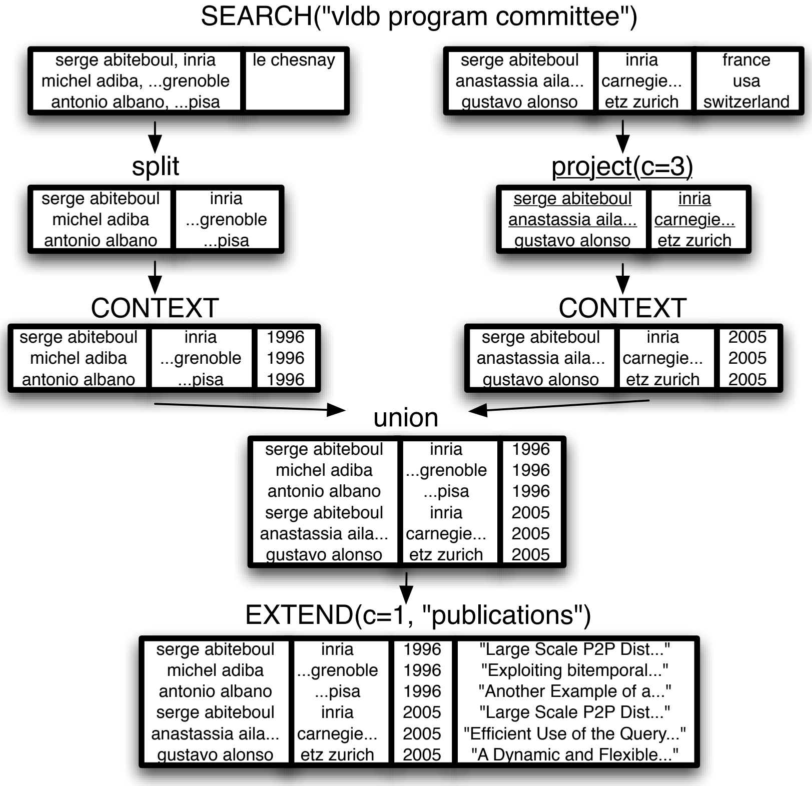
2.3 Integration operators

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

2.3.1 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 relations found on the 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



serge abiteboul, inria michel adiba, ...grenoble antonio albano, ...pisa	le chesnay

serge abiteboul	inria	france
anastassia aila...	carnegie...	usa
gustavo alonso	etz zurich	switzerland

serge abiteboul	inria
michel adiba	...grenoble
antonio albano	...pisa

serge abiteboul	inria
anastassia aila...	carnegie...
gustavo alonso	etz zurich

serge abiteboul	inria	1996
michel adiba	...grenoble	1996
antonio albano	...pisa	1996

serge abiteboul	inria	2005
anastassia aila...	carnegie...	2005
gustavo alonso	etz zurich	2005

serge abiteboul	inria	1996
michel adiba	...grenoble	1996
antonio albano	...pisa	1996
serge abiteboul	inria	2005
anastassia aila...	carnegie...	2005
gustavo alonso	etz zurich	2005

serge abiteboul	inria	1996	"Large Scale P2P Dist..."
michel adiba	...grenoble	1996	"Exploiting bitemporal..."
antonio albano	...pisa	1996	"Another Example of a..."
serge abiteboul	inria	2005	"Large Scale P2P Dist..."
anastassia aila...	carnegie...	2005	"Efficient Use of the Query..."
gustavo alonso	etz zurich	2005	"A Dynamic and Flexible..."

Figure 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. This extra information is very useful after unioning the two tables together. (Otherwise, the two Serge Abiteboul tuples here would be indistinguishable.) Finally, she executes EXTEND to adorn the table with publication information for each PC member.

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.)

2.3.2 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 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.

2.3.3 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 (for example, a list of PC member names), and k describes a desired attribute of c to add to T (for example, 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. The user can express her information need in whatever terminology is natural to her.

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 1 shows two tables in a single SEARCH cluster after the user has issued a request for `vldb program committee`. Figure 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 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 (that may exist in a single well-extracted table), and then recover the year in which the data was collected.

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. We evaluate both result quality and runtime performance for each algorithm in Section 5.

```
function SimpleRank(keywords):
urls = searchengine(keywords) //in search-engine-rank order
tables = []
for url ∈ urls do
    for table ∈extract_tables(url): //in-page order do
        tables.append(table)
    end for
end for
return tables
```

Figure 3: The SimpleRank algorithm.

```
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 4: The SCP and tablescore algorithms.

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.

3.1.1 Ranking

A strawman state-of-the-art algorithm for ranking is to leverage the ranking provided by a search engine. The **SimpleRank** algorithm (see Figure 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 1 does not contain the terms `vldb program committee`. Further, any "metadata" about an HTML 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. (Cafarella *et al.* attempted this approach when ranking extracted HTML tables [4], 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 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 for finding multiword units (such as bigrams or trigrams) in text [5]. It is very similar to Pointwise Mutual Information [19]. 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 data-extraction 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 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 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 engines 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.

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

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 4 below, we substantially reduce the search engine load by approximating SCP scores on a small subset of the Web corpus.

3.1.2 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 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) 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 (that is, 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 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 **ColumnTextCluster** 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.

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 (thus having 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 6 and operates roughly as follows. 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 $s = 10$).

```
function RVPContext(table, source_page):
sig_terms = getSignificantTerms(source_page, table)
list_of_tables = []
for row ∈ table do
    list_of_tables.append(getRVPTables(row, sig_terms))
end for
terms = all terms that occur in list_of_tables
sort terms in descending order of # of tables each term occurs in
return terms

function getRVPTables(row, sig_terms):
tables = SEARCH(row, topk = 5).extractTables()
return tables that contain at least one term from sig_terms
```

Figure 6: The RVP algorithm.

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.

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 [15] 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 (*e.g.*, it may be that k =*publications* while an extracted table says *papers*). Second, it must solve a *reference reconciliation problem* [8] 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 (*e.g.*, 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 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

```
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 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.

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 7, attempts to automate this laborious process.

MultiJoin addresses the issue of schema matching via the column-matching clustering algorithm described in Section 3.1.2 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 [7].

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); search engines incorporate incoming anchor text that will naturally give data on a page multiple aliases; and 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.

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 paper 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 [2]. 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 5.3 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. 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.

5.1 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.

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

Table 1: List of queries used.

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 summarizing.

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 (and thus engaging in another form of cherry-picking). Also, by forcing each query to be supported by two example tables, we guaranteed that each suggestion is at least roughly "structured."

5.2 SEARCH

As we described in Section 3.1, there are two steps in implementing Search: ranking and clustering. To compare the different algorithms for each, we used the aforemen-

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

Table 2: Fraction of top-k sets that contain at least one "relevant" table.

tioned test set of 52 queries, each representing a starting point for a complex information gathering task. See Table 1 for the list of queries.

5.2.1 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 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.

5.2.2 Clustering

Next, we evaluate the three clustering algorithms described in Section 3.1.2. 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 2.3.1, 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 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 (*i.e.*, that the cluster is at all useful). 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.

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

Table 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 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.

The results in Table 3 show that even when the requested cluster has just two elements (and thus the system has 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 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 very large consumer of search engine queries and so there is little computational efficiency to be gained here.

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 1 (skipping over any results where there was either no relevant table or where all relevant tables were single-column). Next, two of the authors 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).

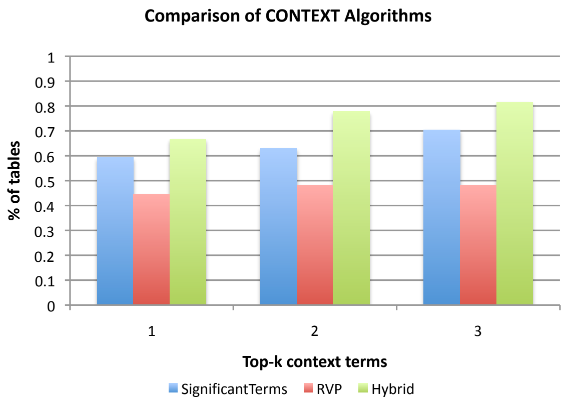


Figure 8: A comparison of the Context algorithms. Shows the percentage of tables (y-axis) for which the algorithm returns a correct context term within the top-k (x-axis) context terms.

Figure 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.

5.4 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 1, many of which do not have much plausible "extra" information. (*E.g.*, 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 (*i.e.*, proper names, such as 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.

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: Test queries for Extend, derived from results from queries in Table 1.

We thus chose a small number of queries from Table 1 that appear to be Extend-able. For each, we chose as the source table T the top-ranked "relevant" table (as marked by a human reviewer) 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 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 **Mul-**

tiJoin 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.

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 related work in the textual information extraction area that we would like to build on using our system [13].

6. RELATED WORK

Data integration on the web, often called a "mashup," is an increasingly popular area of work. The Yahoo! Pipes project 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 [21]. There are other mashup tools available, including the Marmite system [20]. Karma [18] automatically populates a user's database, but still requires sources with formal declarations.

The CIMPLE system is a data integration system tailored for web use, being designed to construct "community web sites [6]." A CIMPLE site consists of a series of human-chosen data sources, extractors, and schema mappings. CIMPLE tools are designed to assist in the task, but an administrator still spends a relatively large amount of time (hours) on a site's initial design. Of course, OCTOPUS is designed to have as little up-front cost as possible, and can be used by untrained users.

Raman and Hellerstein's Potter's Wheel emphasizes live interaction between a data cleaner and the system [16]. 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. There are other cleaning systems with operators that could be useful to OCTOPUS [12].

7. CONCLUSIONS

We described OCTOPUS, a system that engages users to integrate data from many structured sources on the Web. Unlike traditional data integration systems, OCTOPUS offers access to orders of magnitude more data sources because it does not require writing site-specific wrappers, and frees the user from having to design or even know about a mediated schema and mappings between it and data sources. We described a set of basic operators that enable OCTOPUS to provide time-saving services to the user, and described effective algorithms for implementing each of them. In the future we plan to add several improvements to OCTOPUS, including index structures to support real-time user interaction.

8. REFERENCES

[1] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] M. J. Cafarella, A. Y. Halevy, Y. Zhang, D. Z. Wang, and E. Wu. Uncovering the Relational Web. In *WebDB*, 2008.

[4] M. J. Cafarella, A. Y. Halevy, Y. Zhang, D. Z. Wang, and E. Wu. WebTables: Exploring the Power of Tables on the Web. *PVLDB*, 1(1):538–549, 2008.

[5] J. F. 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.

[6] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building Structured Web Community Portals: A Top-Down, Compositional, and Incremental Approach. In *VLDB*, pages 399–410, 2007.

[7] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *SIGMOD Conference*, pages 509–520, 2001.

[8] X. Dong, A. Y. Halevy, and J. Madhavan. Reference Reconciliation in Complex Information Spaces. In *SIGMOD Conference*, pages 85–96, 2005.

[9] H. Elmeleegy, J. Madhavan, and A. Halevy. Harvesting Relational Tables from Lists on the Web. *PVLDB*, 1(3), 2009.

[10] O. Etzioni, M. J. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale Information Extraction in KnowItAll: (Preliminary Results). In *WWW*, pages 100–110, 2004.

[11] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational Plans for Data Integration. In *AAAI/IAAI*, pages 67–73, 1999.

[12] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *VLDB*, pages 371–380, 2001.

[13] S. Kok and P. Domingos. Extracting Semantic Networks from Text Via Relational Clustering. In *ECML/PKDD (1)*, pages 624–639, 2008.

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

[15] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB J.*, 10(4):334–350, 2001.

[16] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*, pages 381–390, 2001.

[17] S. Sarawagi and W. W. Cohen. Semi-Markov Conditional Random Fields for Information Extraction. In *NIPS*, 2004.

[18] R. Tuchinda, P. A. Szekely, and C. A. Knoblock. Building Data Integration Queries by Demonstration. In *Intelligent User Interfaces*, pages 170–179, 2007.

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

[20] J. Wong and J. I. Hong. Making Mashups with Marmite: Towards End-User Programming for the Web. In *CHI*, pages 1435–1444, 2007.

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