Answering Table Augmentation Queries from Unstructured Lists on the Web

[Extended Abstract]

Rahul Gupta
IIT Bombay, India
grahul@cse.iitb.ac.in

Sunita Sarawagi
IIT Bombay, India
sunita@iitb.ac.in

ABSTRACT

We present the design of a system for assembling a table from a few example rows by harnessing the huge corpus of information-rich but unstructured lists on the web. We developed a totally unsupervised end to end approach which given the sample query rows — (a) retrieves HTML lists relevant to the query from a pre-indexed crawl of web lists, (b) segments the list records and maps the segments to the query schema using a statistical model, (c) consolidates the results from multiple lists into a unified merged table, (d) and presents to the user the consolidated records ranked by their estimated membership in the target relation.

The key challenges in this task include construction of new rows from very few examples, and an abundance of noisy and irrelevant lists that swamp the consolidation and ranking of rows. We propose modifications to statistical record segmentation models, and present novel consolidation and ranking techniques that can process input tables of arbitrary schema without requiring any human supervision.

Experiments with Wikipedia target tables and 16 million unstructured lists show that even with just three sample rows, our system is very effective at recreating Wikipedia tables, with a mean runtime of around 20s.

1. INTRODUCTION

Consider a user who wishes to assemble a table of computer science concepts, their inventors and the year of invention. A keyword query "computer science concepts inventor year" retrieves mostly generic computer science pages. If instead he queries using a few example rows, say "turing machine relational databases codd supercomputer cray", he mostly gets pages relevant only to the query tokens, which do not contain new tuples, and any genuine new tuples are still spread across multiple pages and need to be integrated. However we know that numerous mini-compilations of target tuples do exist in tables and lists on the web, and an ideal system would extract and integrate them in a tabular format for the user. On the other hand a search engine only offers us whole documents.

Consider an alternate scenario where a site like Freebase[1] or Wikipedia wishes to build a table of multi-attribute records belonging to a new topic, say Oil spills, starting with a few editor-picked seed records. An ideal answer can be manually constructed by aggregating the oil-spill mentions present in various relevant web-sites, but this is time consuming. In fact, Freebase has several incomplete tables waiting to be completed by human volunteers and an automated tool for this task would be invaluable for them.

In both these scenarios, we begin with a "Show me more tuples like these" query and no other supervision, and our aim is to build a target relation with ranked tuples extracted from structured and unstructured relational data spread all over the web in the form of tables and lists. We call this problem the *table augmentation task*. We note that one highly special instance of this task is Google Sets[2], which only supports single-attribute queries.

In this paper we present a system called WWT[3] for tackling the table augmentation task. We focus on constructing the answer from one of the two major sources — HTML lists on the web. This paper deals only with lists primarily because list processing for our task is a more general and a technically harder problem than processing tables. Any table can be trivially reduced to a list, lists never have header information, and extracting multi-attribute tuples from list records is a significantly hard problem given that we only have a handful of examples. Our final goal is to extend WWT to integrate list and table sources. At this point we stress that this work is similar in spirit to the WebTables system [4], which uses an indexed corpus of 154 million HTML tables for a variety of applications, including ranking tables by relevance to a keyword query. However we go one step ahead and use our list corpus to extract and integrate tuples from general unstructured lists. We do this through a multi-step approach of extracting candidate tuples from individual lists, consolidating them across different lists and ranking them by relevance to the query.

Table augmentation from lists at a web scale poses various sets of challenges. First, it is difficult to determine the lists which are relevant to the user. Even inside a relevant list, not all records might be relevant. Not only do we need to ensure that irrelevant records are identified and ranked low in our final answer, we also need to minimize their effect on the other steps, say while training a tuple extractor or ag-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

[1] http://www.freebase.com
[2] http://labs.google.com/sets
[3] World Wide Tables

gregating with a genuine answer. Second, we do not assume that our lists enjoy the kind of regularity that machine generated lists do. This potentially arbitrary unstructured nature, coupled with the lack of nice column boundaries makes the tuple extraction task difficult, given that we only have a handful of examples in the query. In fact many a times some relevant lists have no syntactic overlap with the query at all! Consequently any noisy extractions will affect aggregation of results across lists. So we need robust extraction and aggregation mechanisms to minimize the loss in performance. Third, even with perfect extraction, aggregation across lists is riddled with problems like inexact duplicates and missing columns. In addition we have spurious matches where a common phrase like '1000' might match a desired query column e.g. Year, which can mislead the aggregation and ranking step. We need to ensure that such highly frequent chance matches do not end up with high ranks.

We now state our contributions in this paper:

- We present an end-end system, called WWT, that returns a table with ranked rows in response to a query. We show that even with a stringent evaluation criteria, WWT can construct more than half of the target in less than half a minute with just three query rows, and this increases to almost 70% with seven query rows.
- We show how to adapt existing statistical extraction models to example-driven table extraction from multiple web lists. We design a matching-based algorithm to create a training dataset from only a small set of query records so as to maximize match with the query records without introducing spurious matches. We show how to exploit consistency of style within a list using multiple sequence alignment features. Finally, we exploit content overlap across lists to supplement the limited information in the small query.
- We design a resolver that can handle arbitrary input tables, while being robust to extraction errors and missing data. The resolver uses a Bayesian network to compose row-level resolvers out of type-specific cell-level resolvers using intuitive parameters that can be set purely from the table statistics, without requiring any training data.
- We design a ranking strategy that can combine correctness scores from the extractors and support from various sources into a single scoring function for placing relevant and correct rows higher.

We describe WWT with a summary of its components in Section 2. In Section 3 we elaborate on the extraction problem of segmenting a list into columns of the target table, and in Section 4 we detail how the extracted results are consolidated and ranked into a single answer table. In Section 5 we present empirical results on the performance of WWT under real-life settings, and in Section 6 we discuss related literature.

2. ARCHITECTURE

Figure 1 shows the architecture of WWT. Here we describe the process flow followed by a brief overview of the various components. Individual components are described in detail in later sections. The execution pipeline in WWT can be split into the following steps:

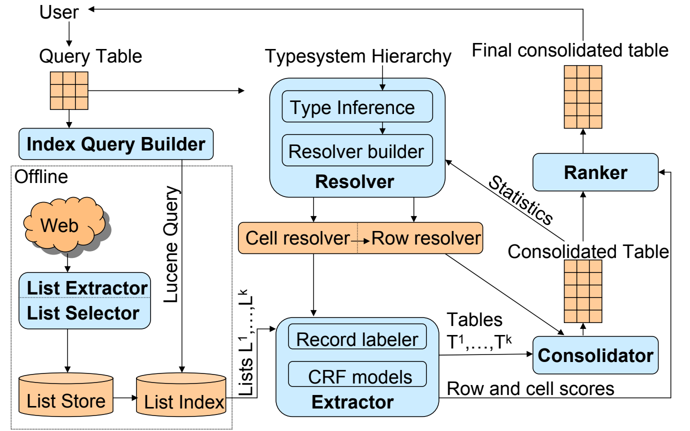


Figure 1: System architecture

1. WWT first builds an indexed repository of HTML lists by crawling the web. A list selection module extracts lists from web pages and throws away useless and navigational lists. This is a one-time offline step.
2. A typical query execution on WWT is as follows. A user first enters the query table of interest through a web interface. Then the query is processed by the resolver and list-index modules in parallel. The index returns HTML lists potentially relevant to the query, ordered by their relevance scores. These lists are called *source lists*. The resolver extracts meta data from the query, like inferring the type-system and constructing cell and row resolver objects (defined in Section 2.3) which are used to match an arbitrary string against query rows/cells.
3. The source lists returned by the list-index are passed to the extractor, which uses the constructed cell resolvers to learn statistical extraction models on the lists. Using these models, the extractor extracts segments from each list record and assigns the segments to query columns. Thus it transforms the lists into tables with the same schema as the query. The extractor also attaches confidence scores with the cells and rows in these tables.
4. The consolidator takes these constructed tables, and using the row resolvers, it clusters duplicate table rows to form a single consolidated table.
5. The ranker reorders the rows of the consolidated table so as to bring more relevant and highly supported rows on top. The top few rows of the re-ranked consolidated table are returned to the user.

We now give a brief overview of each component.

2.1 List Extractor and Selector

The list extractor applies a group of heuristics to every HTML list in every page to prune away navigational lists or lists which are too verbose or textual to contain multi-attribute records. A list is discarded if either of these tests holds: (a) it has less than 4 or more than 300 records, (b) it has even one record more than 300 bytes long, (c) more

than 20% records do not have delimiters (d) more than 70% of records are contained inside anchor tags. Using these heuristics, we get around 16 million lists from a web-crawl of 500 million pages.

2.2 List Index

We treat each list as a document and index it using Lucene[4]. A more expensive alternative is to store each list record as a separate Lucene document, which allows more precise row-level query answering. However, we prefer the list-as-document indexing model as it keeps the index more compact. Later stages of the pipeline prune away spurious matches.

In response to a table query, the index is probed by creating a bag of words query from the contents of the table. The top-K matching lists based on Lucene's inbuilt ranking criteria are returned.

2.3 Resolver

The resolver takes a query table and first infers the type of each column. For this it uses a user-configurable type hierarchy . The type information along with the contents of the table is then used to build two kinds of objects — row and cell resolvers. A cell resolver is built for each column in the query. Given a query row r, a column c, and a text s , the cell resolver for c returns a score of how well does s resolve to the cell c in r. The row resolver takes as input two rows and returns a score of them resolving to each other. It uses the cell resolvers as subroutines to resolve constituent columns of the input rows.

2.4 Extractor

The extractor module takes the source lists retrieved by the index, and trains extraction models on those lists. To generate labeled training data, it uses the cell resolvers to mark potential segments in the lists that match with some of the query cells. The extractor then trains statistical text segmentation models with this labeled data, and applies them to the lists, thus obtaining a table from each list. The extractor also attaches a confidence score with each row and cell entry for later use by the ranker.

2.5 Consolidator and Ranker

Given a set of tables output by the extractor along with row and cell confidence scores, the consolidator merges the tables into one table. A cell in the consolidated table contains a set of cell values that resolve to each other and, one of these is picked as the canonical entry based on cell probability scores output by the extractor.

The ranker combines the row and cell scores of a row into a single sorting metric. Intuitively, a high scoring consolidated cell/row is one which repeats in many lists, and which has a high confidence score of extraction. The top few ranked rows of the consolidated table are returned to the user as the answer.

We now comprehensively discuss the individual components of WWT.

3. EXTRACTOR

The input to the extractor module is a query table Q and a set $\mathcal{L} = \{L^1, \ldots, L^K\}$ of source lists potentially relevant to Q. For every record r in every list in $\mathcal{L}$, the extractor outputs a valid segmentation i.e. a partitioning of r into segments, with at most one segment mapped to each column in Q. Unmapped segments are assumed to map to the fictitious column -1, and unmapped columns are mapped to null segments. Thus a valid segmentation can be viewed as a table row, and the process of extraction viewed as transforming lists $L^i \in \mathcal{L}$ into tables T^i with the same schema as the query Q, but with possibly NULL entries. Figure 2(c) shows one possible set of tables obtained from the lists in Figure 2(a).

Now this list to table transformation task is a classical text segmentation problem. There are various possible segmentation schemes in the information extraction domain to choose from — list-specific wrappers, rule-based segmenters, or statistical segmentation models. We choose the semi-markov Conditional Random Field model (CRF) [19], which is a statistical segmentation model. There are various reasons for this choice. First, we do not assume that the list records are machine-generated. Thus they can contain noisy tokens and style variations. CRFs are more accurate and robust for such records as they can exploit an arbitrary number of rich and correlated segment-level properties. Second, a CRF outputs a well calibrated confidence score along with the best segmentation of a list record. Such scores are valuable during the later stages of WWT. Computing calibrated scores is impossible or very difficult with other extraction approaches.

We train separate CRF models for each source list to capture the specific regularities present within a source. The first challenge in training a model is the absence of explicitly labeled records from the source list. All we have are the handful of rows in Q. We present two different ways to use Q to generated labeled records for a particular list L, which lead to two different extraction mechanisms. The first method, which we called the default extractor or $E_{default}$, computes labeled records for each L separately using only Q. The second method , called E_{staged}, uses Q *and the extracted records of the other lists* to compute the labeled records of L. Thus E_{staged} exploits content overlap present across the source lists.

However even in $E_{default}$, generating labeled unstructured records from a few structured rows is a non-trivial task because the consequences of wrongly or inadequately labeled records are huge, as detailed in Section 3.1. So we divide the functionality of the extractor into a non-trivial labeler component, and a CRF trainer which uses the labeled records generated by the labeler to learn a model. For ease of explanation, we first describe $E_{default}$, whose labeler generates labeled records using only Q. The labeler computes a matching between the rows of L and Q, with the interpretation that a matched record in L is an occurrence of the corresponding row in Q. For each matched record in L, the labeler also outputs a segmentation that maps to the cells of its query row. For example in Figure 2(b), the first three records in the first list are matched with corresponding rows of the query in Figure 2(a) (although the last record is wrongly matched with the third row). For the first record, the output segmentation is {Name = "Arthur Charles Clarke", Place = "Somerset", Year = "1917"}, which roughly matches with the cell contents of the first query row.

Algorithm 1 formally describes $E_{default}$. In Sections 3.1

[4]http://lucene.apache.org

Query Q:

Arthur C. Clarke	1917	Somerset, UK
David "Dave" Barry	1947	Armonk, New York
Isaac Asimov	1920	Petrovichi, Russia

Source list 1:

- *Arthur Charles Clarke*, born in Somerset, 1917.
- *Dave Barry*, born in Armonk, 1947.
- *Frank Herbert*, born in 1920.
- *Dame Agatha Christie*, born in Devon (UK), 1890.
- *Noam Chomsky*, born in Philadelphia.

Source list 2:

- Noam Chomsky -- 7 December 1928.
- Agatha Christie -- 15 September 1890.
- John R. R. Tolkien -- 3 January 1892.
- Salman Rushdie -- 19 June 1947.

(a) A query with two result lists

List 1: Labeled records before pruning

- *Arthur Charles Clarke*, born in Somerset in 1917.
- *Dave Barry*, born in Armonk in 1947.
- *Frank Herbert*, born in 1920.

After pruning (columns = {Name,Year,Place})

- *Arthur Charles Clarke*, born in Somerset in 1917.
- *Dave Barry*, born in Armonk in 1947.

List 2: Labeled records before pruning

- Salman Rushdie -- 19 June 1947.

After pruning (columns = {Year})

- Salman Rushdie -- 19 June 1947.

(b) $E_{default}$: Labeled data

Tables output by $E_{default}$

Table T^1 (= E_{staged} : Table T^1)

Arthur Charles Clarke	1917	Somerset
Dave Barry	1947	Armonk
Frank Herbert	1920	---
Dame Agatha Christie	1890	Devon (UK)
Noam Chomsky	---	Philadelphia

Table T^2

---	1928	---
---	1890	---
---	1892	---
---	1947	---

(c) Tables output by $E_{default}$

E_{staged}: CT_2 = Consolidate(Q, T^1)

Arthur C. Clarke Arthur Charles Clarke	1917	Somerset Somerset, UK
Dave Barry David "Dave" Barry	1947	Armonk Armonk, New York
Frank Herbert	1920	---
Dame Agatha Christie	1890	Devon (UK)
Noam Chomsky	---	Philadelphia
Isaac Asimov	1920	Petrovichi, Russia

(d) E_{staged}: Consolidated table after first stage

Labeled records of list 2 before pruning

- Noam Chomsky -- 7 December 1928.
- Agatha Christie -- 15 September 1890.

After pruning (columns = {Name,Year})

- Agatha Christie -- 15 September 1890.

E_{staged}'s output: Table T^2

Noam Chomsky	1928	
Agatha Christie	1890	---
John R. R. Tolkien	1892	---
Salman Rushdie	1947	---

(e) E_{staged}: Using CT_2 to label data in L^2, and its output table

E_{staged}: Consolidation of T^1 and T^2

Arthur Charles Clarke	1917	Somerset
Dave Barry	1947	Armonk
Frank Herbert	1920	---
Dame Agatha Christie Agatha Christie	1890	Devon (UK)
Noam Chomsky	1928	Philadelphia
John R. R. Tolkien	1892	---
Salman Rushdie	1947	---

(f) Consolidator's output on tables T^1 and T^2 built by E_{staged}

Figure 2: Execution of $E_{default}$ and E_{staged} for a sample query.

and 3.2 we describe the labeler and the CRF model in detail.

Algorithm 1: Default Extractor ($E_{default}$)

Input: Set of lists $\mathcal{L} = \{L^1, \ldots, L^K\}$, Query table Q
Output: Tables $T^1, \ldots, T^K$
for $i = 1 \ldots K$ **do**
  $(D, \hat{\mathbf{s}}) \leftarrow$ Labeler(Q, L^i); `/* Sec 3.1 */`
  Train a CRF with labeled data $(D, \hat{\mathbf{s}})$;
  Initialize T^i to an empty table;
  foreach *record* $r \in L^i$ **do**
    Find the best segmentation $\mathbf{s}^*(r)$ from the CRF;
    Append $\mathbf{s}^*(r)$ to T^i;
  end
end
return $T^1, \ldots, T^K$

3.1 Labeled Data Generation

The labeler takes as input a query table Q and a list L, and its task is to output a matching of the records of L with the rows of Q. Denote this matching by ϕ. If a labeler deems that record $r \in L$ is an occurrence of a query row q, then as evidence it also outputs a segmentation $\hat{\mathbf{s}}(r)$ such that segments in $\hat{\mathbf{s}}(r)$ resolve to the columns of q to which they are assigned. All matched records are output as the set D, i.e. $D \triangleq \{r \in L | \phi(r) \text{ is defined}\}$. The complete output of the labeler is $(D, \{\hat{\mathbf{s}}(r)\}_{r \in D})$.

Observe that a matching-based approach is needed because we cannot just independently mark the occurrences of individual query cells in list records. This is mainly because we need to preserve the row structure of query rows when we look for segments inside list records.

Now if we enforce a exact string matching criteria to identify crisp occurrences of query cells in L, then we might get too few matches and hence too few labeled records in L. On the other hand, if we soften the criteria and generate a few wrongly labeled records, we will severely impact the accuracy of the CRF trained on these records. Besides this choice of criteria, our task is further complicated by the following observations: First, a query cell might be present in a noisy form in a list record. This is usually true for cell values containing dates, names and places. Second, list records may possess only a few columns of Q. Then the resulting ambiguity can make it difficult to map a list record to a unique query row. Third, some query cells might occur multiple times across list records, e.g. cells containing only state names or only the year part of a date. Only one of these occurrences is relevant for the query cell, when seen in the context of the entire query row. Last, a list record might contain disjoint segments that match with cells belonging to different query rows. We need to ensure that only one query row is matched with the record.

To tackle these issues, we model the labeling task as a maximum weight matching problem in a bipartite graph with the two sides representing rows of Q and L. There is a node in the graph for every row of Q and L, and the weight of an edge (q, r) is the score of the segmentation $\hat{\mathbf{s}}(r)$ *assuming that* r *is matched with* q. If r is matched with q, then the output segmentation $\hat{\mathbf{s}}(r)$ will align as much as possible with the cells of q. For now assume that a subroutine **SegmentAndMatch**(r, q) produces such a segmentation along with its score. Later we will present two options for **SegmentAndMatch** that lead to two different labelers.

For the sake of performance, we do not construct a complete bipartite graph but use a simple heuristic to limit the number of edges. We demand that if for the vertex pair (q, r), no cell of q approximately matches any substring of r, then the edge (q, r) be absent. An approximate match, implemented by an **ApproxMatch**(q, r) subroutine, permits a limited amount of token padding/deletion/permutation. In practice, this heuristic reduces the number of edges in the graph from $O(|Q||L|)$ to $O(\min(|Q|, |L|))$, where $|.|$ denotes the number of rows.

The row map ϕ and hence D are trivially obtained from the maximum matching. The output segmentation set is simply $\{\hat{\mathbf{s}}(r) \triangleq \textbf{SegmentAndMatch}(r, \phi(r)) | r \in D\}$. However, this is not the final output. The segmentations in $\{\hat{\mathbf{s}}(r)\}$ will in general not agree on the set of query columns present in them. For example, in Figure 2(b), segmentations for the second and third records have column sets {Name,Place,Year} and {Year}. Using all such segmentations for training a CRF often leads to an inferior model because the model does not learn the presence of a column even though it might exist in a record. For example, in the third record, the model will wrongly learn that "Frank Herbert" is not a name. So to keep our model clean, we prune the labeled set by first computing the biggest column set present in any $\hat{\mathbf{s}}(r)$. In Figure 2(b), this will be {Name,Place,Year}. Then we conservatively throw out every r from D whose column set in $\hat{\mathbf{s}}(r)$ does not match this maximum column set. For any r omitted thus, we also remove the corresponding segmentation $\hat{\mathbf{s}}(r)$. These pruned values of D and $\hat{\mathbf{s}}$ are returned as the labeler's output.

The full labeling algorithm is described in Algorithm 2.

Algorithm 2: Labeler

```
Input: List L, Query table Q
Output: Labeled set D, Segmentations {ŝ(r)}_{r∈D}
G ← edge-less bipartite graph with nodes from Q ∪ L;
foreach record r ∈ L, row q ∈ Q do
    if ApproxMatch(q, r) then
        (ŝ(r), score) ← SegmentAndMatch(r, q);
        Add edge (q, r) with weight=score to G;
    end
end
φ ← MaxWeightMatching(G);
D ← {r ∈ L|φ(r) is defined};
foreach record r ∈ D do
    (ŝ(r), .) ← SegmentAndMatch(r, φ(r));
end
Y ← MaximumColumnSet({ŝ(r)}_{r∈D});
foreach record r ∈ D s.t. ColumnSet(ŝ(r)) ≠ Y do
    Remove r from D;
end
return D, {ŝ(r)}_{r∈D}
```

We now discuss two possibilities for **SegmentAndMatch**, which leads to two different kinds of edge weights, and hence two different labeler implementations.

3.1.1 Hard Labeler

A natural way for segmenting a record r so as to match the cells of a query row q is to look for an approximate occurrence of every cell of q in r and declare those occurrences as segments with appropriate column assignments. This version of **SegmentAndMatch** does precisely this with a small but essential modification. It checks for approximate matches of bigger query cells first. This ensures that matches for bigger cells are available and not eaten up by smaller cells which might have overlapping content. For example, for the two column query row "New York City | New York", matches for "New York City" are searched first.

This implementation of **SegmentAndMatch** reuses **ApproxMatch** to figure out if there are matches for a cell or not. Approximate match queries can be easily answered by constructing an in-memory index on the tokens of r. The score of the output segmentation $\hat{\mathbf{s}}(r)$ is simply the number of cells of q which had an approximate match in r.

3.1.2 Soft Labeler

The **ApproxMatch** algorithm used by the hard labeler is based on hard thresholds on the leeway allowed for approximate matches, e.g. maximum number of token insertions/deletions allowed. Also, the notion of a match is still based on string exactness — some query cell tokens *have* to occur in the segment.

For this reason, the hard labeler usually outputs high precision labelings, but might miss out on close matches, such as "Arthur Charles Clarke" and "Arthur C. Clarke" (Figure 2(b)). If **ApproxMatch** demands the presence of all segment tokens among the query cell tokens, then it might match only 'Arthur' or 'Clarke' with "Arthur C. Clarke". However if we know that this query cell has a type 'PersonName', then we can expand the match to include the entire name correctly.

Keeping this intuition in mind, we define a soft scoring function $\textbf{SoftScore}(s, q, c) \rightarrow [-1, 1]$, which given a candidate record segment s, a query row q and a particular column c, outputs a normalized similarity score of the query cell q_c matching s. **SoftScore** takes into account the type of the column c. Recall that column type inference is done by the resolver when the query is posed. The functionality of **SoftScore** is provided by the cell-resolver object of the resolver module, which is discussed in detail in Section 4.1.

For the purposes of our labeler it suffices to assume that a score of -1 denotes extreme dissimilarity between s and q_c, and a score of 1 denotes exact string match, and inexact matches take intermediate values. Now if we assume that row q is indeed matched with record r, then the best segmentation $\hat{\mathbf{s}}(r)$ and its score are given by:

$$\hat{\mathbf{s}}(r) = \underset{\mathbf{s}:\mathbf{s} \text{ valid for } r}{\arg\max} \sum_{(s,c)\in\mathbf{s}} \textbf{SoftScore}(s, q, c) \quad (1)$$

where we trivially extend **SoftScore** to output zero for unmapped segments s, i.e. $\textbf{SoftScore}(s, q, -1) = 0$. Observe that this objective is decomposable over individual segments of $\mathbf{s}$, so it can be easily computed using dynamic programming by recursing over the length of r. Thus the soft labeler's implementation of **SegmentAndMatch** returns a segmentation and score after maximizing Equation 1.

3.2 CRF-based extraction models

The extractor uses CRFs to segment list records, so we provide a concise overview of CRFs here. Let r be a list record r and $\mathbf{s}(r) = \{(s_i, c_i)\}_i$ be a candidate segmentation where s_i is the ith segment in $\mathbf{s}(r)$ and c_i is the query column it maps to (-1 if unmapped). Then the CRF models the

likelihood of $\mathbf{s}(r)$ as:

$$P(\mathbf{s}(r)|r,\lambda) = \frac{\exp \lambda^T \cdot \sum_i \mathbf{f}(s_i, c_i, c_{i-1})}{Z_\lambda(r)} \quad (2)$$

where $\mathbf{f}(s_i, c_i, c_{i-1})$ is a vector of features of the labeled segment (s_i, c_i) and the previous segment's label c_{i-1} (which leads to the model being Markovian). λ is a weight vector that establishes the relative importance of each feature in $\mathbf{f}$, and Z_λ is a normalizing factor. The weight vector λ is learnt during the training phase.

Feature Set

The CRF allows arbitrary and correlated segment-level features in the feature vector $\mathbf{f}$. So traditional deployments of CRFs use many feature templates like word occurrences, dictionary matches, regex patterns etc. These are usually boolean features e.g. c_i is a name column and s_i is short segment or not. However with only a few labeled records available, we cannot use too many features for the risk of overfitting. So we use only one feature template, derived from delimiters and HTML tokens (called separator tokens):

$$\begin{aligned} f_{j,j',t,t',c}(s_i, c_i, c_{i-1}) &= (c = c_i) \\ &\wedge \ (s_i \text{ is } j' \text{ tokens to the right of } t') \\ &\wedge \ (s_i \text{ is } j \text{ tokens to the left of } t) \quad (3) \end{aligned}$$

where $j, j' \in \{1, 2, 3\}$, t, t' vary over the separators seen in the list containing this row, and c varies over the columns of Q. For example, if in a source, the column 'Name' is frequently surrounded by an anchor tag, then for any segment s surrounded by anchor tags, we will have:

$$f_{1,1,,<a>,\text{Name}}(s, c_i, .) = 1 \text{ if } (c_i = \text{Name}) \text{ else } 0$$

So any such s will be biased towards the column Name, which is also the desired output.

Training the CRF

Given a feature template $\mathbf{f}$ and labeled records $(D, \hat{\mathbf{s}})$, we learn the weight vector λ that maximizes the log-likelihood of the labeled data:

$$\max_\lambda \sum_{r \in D} \log P(\hat{\mathbf{s}}(r)|r,\lambda) - \frac{\lambda^2}{2\sigma^2} \quad (4)$$

where the second term regularizes λ and prevents over-fitting. This objective is concave in λ and can be optimized using standard gradient-based methods [17].

Computing the Best Segmentation

Once the weight vector λ has been trained, computing the best segmentation for a list record r means solving:

$$\begin{aligned} \mathbf{s}^*(r) &= \underset{\mathbf{s}:\, \mathbf{s} \text{ valid for } r}{\arg\max} \ P(\mathbf{s}|r,\lambda) \\ &= \underset{\mathbf{s}:\, \mathbf{s} \text{ valid for } r}{\arg\max} \ \lambda^T \cdot \sum_{(s_i,c_i) \in \mathbf{s}} \mathbf{f}(s_i, c_i, c_{i-1}) \quad (5) \end{aligned}$$

Equation 5 can be solved recursively using the Viterbi algorithm [19] with some restrictions to allow only valid segmentations.

3.3 Improving the Default Extractor

We now present two modifications to $E_{default}$: one that exploits the consistency of records *inside* a list, and another which uses any content overlap across different lists. These modifications are orthogonal to the labeler implementation used for generating the labeled data.

3.3.1 Alignment Features based Extractor

The default feature template in Equation 3 works well in practice as long as we have a good set of separators in the list. To handle lists which do not have such separators, we propose an additional feature template that captures the consistency of the authoring style generally seen in lists. To compute these features, we first perform a multiple sequence alignment of all the records in the list. We use the standard center-star approximation algorithm of [12] for minimizing the pairwise distances of the record alignments. Note that here the alignment is purely syntactic, query oblivious and unsupervised. Our pairwise distance metric rewards matching delimiters with each other, exact HTML token matches, and similarly-typed substrings with each other.

The multiple sequence alignment outputs an alignment column id for each token of each record. Then given a segment s we can get its starting and ending alignment column ids from the first and last tokens of s. Using these ids, we define an additional feature template:

$$\begin{aligned} f^{align}_{c,\alpha,\beta}(s_i, c_i, c_{i-1}) &= (c = c_i) \\ &\wedge \ (\text{StartAlignId}(s_i) = \alpha) \\ &\wedge \ (\text{EndAlignId}(s_i) = \beta) \quad (6) \end{aligned}$$

In Section 5, we shall see the empirical benefit of using these extra features.

3.3.2 Multi-stage Extractor

As stated before, $E_{default}$ independently transforms each list in $\mathcal{L}$ to the corresponding table, without exploiting the valuable overlap that is present among lists. For example, in Figure 2(a) the two source lists have two records in common but L^2 has only one cell in common with the query ("1947"). So $E_{default}$ can extract at most one column from L^2 (Figure 2(c)).

To rectify this, we propose the following multi-stage approach: First, we run the labeler independently on each list and obtain (# columns in the labeled records, # labeled records) for every list. This metric is used to sort the lists in descending order. Then we learn the CRF for the list on top of this sorted order . Such a list will yield a more robust model than the others. Now using Q *and* the confident segmentations of this list, we compute the labeled records in the next best list, learn its model and segment its records, and move on. At stage i, we use Q *and* the confident rows of tables $T^1, \ldots, T^{i-1}$ extracted upto stage $i-1$ to compute the labeled records for the ith best list in our sort order. Besides generating robust models, our sorting criteria also helps in creating a bigger and fuller set of tables $\{Q, T^1, \ldots, T^{i-1}\}$ to label the records in the ith stage. Algorithm 3 formally describes this approach. We use a probability threshold of 0.8 to denote confident segmentations.

The key step in this approach is the computation of labeled records for L^i using $\{Q, T^1, \ldots, T^{i-1}\}$. For this we keep a running consolidated table CT_i, which contains an amalgamation of Q with the confident rows of $T^1, \ldots, T^{i-1}$. Figure 2(d) illustrates a consolidated table CT_2 obtained from merging Q with T^1. A cell in CT_i generally contains multiple values, each of which denotes a variant of the cell's canonical value.

Now recall that the sub-routine **SegmentAndMatch**(q, r) was responsible for labeling a record r assuming it was matched to the query row q. Here q is a consolidated row from CT_i, so we make a minor modification to **SegmentAndMatch**: for a column c of q and a segment s of r, **SegmentAndMatch** will look at all the variants in the cell q_c and pick the variant which is best suited for s. With this modification the labeler can use Algorithm 2 as is to compute labeled records at any given stage.

Figure 2(e) illustrates that the multi-stage extractor is able to label and extract more columns, including the important 'Name' column, thus boosting the accuracy of the system. We perform an empirical study of this extractor, called E_{staged}, along with other extractors in Section 5.

Algorithm 3: Multi-stage Extractor (E_{staged})

Input: Set of lists $\mathcal{L} = \{L^1, \ldots, L^K\}$, Query table Q
foreach $i = 1 \ldots K$ **do**
  $(D^i, \hat{\mathbf{s}}^i) \leftarrow$ Labeler(Q, L^i);
end
$\{L^1, \ldots, L^K\} \leftarrow$ Sort $\mathcal{L}$ by the criteria in Section 3.3.2;
$CT_1 \leftarrow Q$;
foreach $i = 1 \ldots K$ **do**
  $(D^i, \hat{\mathbf{s}}^i) \leftarrow$ Labeler(CT_i, L^i);
  Train a CRF with $(D^i, \hat{\mathbf{s}}^i)$;
  Apply the CRF to L^i to get T^i as in $E_{default}$;
  $CT_{i+1} \leftarrow$ Consolidate$(CT_i$, confident rows of $T_i)$;
end
Output: Tables $T^1, \ldots, T^K$

To summarize, we presented the following orthogonal options for extractor design: (a) $E_{default}$ vs E_{staged}, depending on whether content overlap is to be exploited while generating labeled records, (b) Soft vs hard implementations of **SegmentAndMatch**, depending on whether we want more but possibly slightly noisy cell matches or less but crisp matches, (c) Additional alignment based features, which are useful if there is consistency across records inside a source.

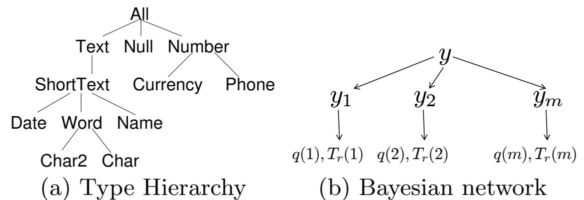
4. CONSOLIDATION AND RANKING

The output of extraction is a set of tables $T^1, \ldots, T^K$, which we consolidate into a single table R such that all duplicate rows across the tables are merged into a single row. Figure 2(f) shows the consolidation of two tables output by E_{staged}. The rows are then ranked so that correct, useful, and relevant rows appear earlier in R. There are several advantages to presenting a single consolidated table to the user instead of individual tables. Consolidation allows us to create more complete rows by merging cells from different sources, hide mistakes in individual extractions, and make sound row-level relevance judgments.

A key building block of the consolidator is a resolver that given two rows, outputs a score indicating if they are duplicates of each other. We discuss the design of the resolver in Section 4.1. Next in Section 4.2 we present how the consolidator works on the K input tables to produce the consolidated table. Finally, in Section 4.3 we discuss strategies for ranking rows in the result.

4.1 Resolver

We attach two kinds of resolvers with any table T:



(a) Type Hierarchy (b) Bayesian network

Figure 3: The type hierarchy and the Bayesian network used by the resolver.

1. A row-resolver that given an arbitrary tuple q, and a row T_r of the table, outputs a real score such that when the score is positive the row pairs are considered duplicates, with the magnitude of the score indicating the confidence in the decision.

2. A cell-resolver is attached to each column c, such that given an arbitrary string $\mathbf{x}$ and a cell $T_r(c)$, it outputs a real score whose sign and magnitude are interpreted the same way as above. The cell-resolver, in addition to forming the building block of the row-resolver, is used during extraction to create labeled data as discussed in Section 3.1.2.

In spite of the extensive research on entity resolution, designing such resolvers turned out to be a surprisingly complex task for several reasons: First, the sheer diversity of types and data formats on the web makes it hard to hand tune any specific rule-based system, or find a representative dataset to train a statistical model. Second, the resolution tasks have to be performed on possibly erroneously extracted entities. For example, it is unclear how many extra tokens before or after a cell should be allowed before a string stops being a duplicate of that cell. Third, a row-resolver will have to work on row pairs with arbitrary combination of cells missing in the two rows. Many of the earlier methods of finding a row-level score by taking a weighted sum of the scores of the cells and thresholding fail on incomplete data.

We followed a three layered strategy to tackle the above challenges. Instead of attempting to design resolvers for generic string types, we incorporate a configurable type hierarchy in WWT. For each column we infer its most specific type node in the hierarchy. Each type comes with a default resolver for columns of that type. We elaborate on this in Section 4.1.1. Given a column of a table, we derive its cell resolver by adapting the type-specific resolver based on the contents of the table as described in Section 4.1.2. Finally, we create the row-level resolvers out of the cell-level resolvers and a Bayesian network to intuitively parameterize the interaction among the various cell resolvers. We describe row-resolvers in Section 4.1.3.

4.1.1 Type hierarchy

Our system can take as input any hierarchy of column types provided each type is associated with the following methods: (a) A recognizer for whether a cell is of that type, (b) An indexer that can support exact and top-k searches on a column of values of the type. (c) A default cell-resolver for pairs of cells of the type.

An example default hierarchy of our system is shown in Figure 3(a). To infer the type of a table column we first

find for each of its cell the most specific node from the type hierarchy that recognizes it to be of that type, then in a single bottom-up sweep, we find the most specific type that covers almost all of its cells, barring a few outliers.

Once we have associated a type with a column, we can use the default resolver of that type to answer cell-resolution queries on that column. For example, in the type hierarchy of Figure 3(a), all different text types define the resolver score between two cells c_1, c_2 as $2(\text{similarity}(c_1, c_2) - 0.5)$ where $\text{similarity}(c_1, c_2)$ returns a value between 0 and 1 chosen appropriately from the extensive set of options in the literature [3]. In our case, Text types use Cosine similarity with TF-IDF weights on words, ShortText types use Soft-cosine with an error tolerant similarity function like Jaro-Winkler to match any token left unmatched based on exact string equality, Name types use a custom similarity function that can handle typical name variants in initials and last names, and Word types use Jaccard on character 3-grams.

4.1.2 Creating a cell resolver for a column

We adapt a default resolver to a given column of values by exploiting the observation that typically two cells of a table column from a given source either are exact duplicates detectable by string equality, or are non-duplicates. Given a column, we first collapse these obvious duplicates and generate training pairs of top-k most similar non-duplicates from the rest. This set along with a perfect duplicate pair is used to train a binary classifier where the features are one or more similarity functions attached with that column type. During training, we make suitable adjustments to handle class skew, and safeguard against over-fitting. We skip the details here due to lack of space. This training is very fast because the number of training records is typically below 10.

4.1.3 Creating a row resolver for a table

A row-level resolver can be composed out of cell resolvers in many different ways. A standard method is to use the weighted sum of cell similarities as the row similarity score [7, 18]. However, this scheme does not easily adapt to our scenario because we have to handle tables with widely varying number of column types and arbitrary number of nulls.

We use a Bayesian network to capture how the row-level duplicate decision interacts with cell-level decisions. We designed the network so that its parameters can be easily set from either the cell-level resolvers, or observed statistics of a table T. Let y denote the row-level decision variable of whether rows q and T_r are duplicates. Variable y directly influences whether the cell pairs in each column c are duplicates of each other. We denote these decision variables as $y_1, \ldots y_m$. When $y_c = 1$, the cells $T_r(c), q(c)$ are resolved to be duplicates of each other, which induces a certain joint distribution on their values. The final Bayesian network is shown in Figure 3(b). It has two kinds of parameters:

The* $\Pr(y_c|y)$ *distribution. This distribution denotes the probability that the cell pairs in column c are duplicates conditioned on whether the row pairs are duplicates. Consider first the case when $y = 1$, that is, the rows are duplicates. Normally, we expect that when two rows are duplicates, the cell pairs in all columns should also be duplicate, that is, $\Pr(y_c = 1|y = 1)$ should be 1. However, in real-life it is possible for duplicate rows to disagree on a column due to either random errors, or phenomenal reasons (change of jobs can cause duplicate person rows to disagree on affiliation). We capture this possibility by assigning a small probability of ϵ to cell pairs disagreeing even when overall rows agree. Thus $\Pr(y_c = 1|y = 1) = 1 - \epsilon$. (We use $\epsilon = 0.02$).

Next consider the case when $y = 0$. $\Pr(y_c = 1|y = 0)$ denotes the probability that two non-duplicate rows have duplicate cell values on column c. Row q could be a non-duplicate of T_r in two cases: either q is a valid row of T or q is an arbitrary row irrelevant to T. In the former case, we can interpret probability $\Pr(y_c = 1|y = 0)$ as the probability that an arbitrary other row of T will repeat a value in column c. This is easy to estimate based on the observed count of distinct values $D(c)$ over the N rows of T as $p_1 = 1 - \frac{D(c)+1}{N+2}$ where the constants 1 and 2 are used for Laplace smoothing of the estimates due to the possibly small value of N. In the latter case, where q can be an arbitrary irrelevant row, we need to depend on corpus-level statistics, or some type-derived biases to estimate the probability p_2 that an irrelevant row will repeat value $T_r(c)$. We currently depend on the type of the column to set p_2 to a high value for number columns and very small value for text columns (these are set to 0.5 and 0.002 in our case). Finally, we combine these two cases to estimate $\Pr(y_c = 1|y = 0) = 1 - (1 - p_1)(1 - p_2)$.

The* $\Pr(T_r(c), q(c)|y_c)$ *distribution. The scores of the cell resolvers can be easily converted to obtain $\Pr(y_c|T_r(c), q(c))$ the probability that two given cells are duplicates of each other. We calculate $\Pr(T_r(c), q(c)|y_c)$ from $\Pr(y_c|T_r(c), q(c))$ by applying Bayes rule and imposing equal priors on the two value of y_c to get: $\Pr(T_r(c), q(c)|y_c) \propto \Pr(y_c|T_r(c), q(c))$. When any of the two cells are null, $\Pr(y_c|T_r(c), q(c))$ returns 0.5, the state of zero knowledge.

Using the above parameters and an equal prior on y, it can be derived that the Bayesian network of Figure 3(b) will give rise to:

$$\Pr(y|T_r, q) = \kappa \prod_c \sum_{y_c=0,1} \Pr(y_c|T_r(c), q(c)) \Pr(y_c|y) \quad (7)$$

κ is a normalization constant. Using the above conditional distribution on y, we return the score that T_r is a duplicate of q as $\Pr(y = 1|T_r, q) - \Pr(y = 0|T_r, q)$. This score is negative when the probability of the rows being non-duplicates is higher than the probability of their being duplicates.

4.2 Consolidator

We now discuss how we use the row-resolver scores to consolidate the K extracted tables. A straightforward algorithm is to consolidate tables iteratively as follows: Start with an empty consolidated table. Go over each table T in turn and merge it into R. In the merge step, go over each row T_r of T, use an index-based filter to pick candidate matches, invoke the resolver on each to find the highest scoring row R_s of R. If this score is positive merge R_s and T_r, else create a new row in R using T_r.

We improve upon this method by exploiting an observation on the form of duplicates in real data. *Intra-source duplicates exhibit significantly lower variance than inter-source duplicates.* Within a source, duplicates are either not present or are trivial to detect based on exact string similarity. Once the obvious duplicates are removed, the rows within a source should be treated as distinct and in the consolidated table they should not be merged in the same row. We exploit this observation to consolidate tables as follows:

An optimal consolidated table would maximize the sum of *positive* resolver scores of row pairs merged together while ensuring that non-duplicate rows of the same list are kept in separate rows. When $K = 2$ and there are no duplicates within a table, we can find the optimal solution by finding the maximal matching between the two row sets. When either of the two conditions are violated, it is easy to verify the NP-hardness of optimizing this objective by reduction from tripartite matching and graph partitioning.

We outline the approximate algorithm used in our consolidator in Algorithm 4. The consolidator merges each table T^i in turn with an intermediate result table R such that each distinct row of T^i is in a different entry as follows: First, it builds a new table $\bar{T}^i$ by merging obvious duplicates in T^i. Most sources do not contain duplicate rows, so this step has no effect in most cases. Next, to merge $\bar{T}^i$ into R it uses an index on R to find close matches for each row $\bar{T}^i_r$, invokes the resolver on the close matches and retains those rows pairs where the similarity is greater than a threshold δ. Threshold δ is chosen so that unsure duplicates are not merged until all tables have had a chance to find a good match. The candidate pairs are used to define edges of a weighted bipartite graph and the maximal matching in the graph provides an optimal solution to the best way to merge two tables with distinct rows. The mapped row pairs in the maximal matching are merged and unmapped rows define new entries in R. The index on R and resolver parameters are updated before moving on to the next table. After all tables are merged using the bipartite criteria, it makes a final pass to merge rows in R with positive resolver scores while ensuring that rows from the same source are kept separate.

Algorithm 4: : Consolidating tables $T^1, \ldots, T^K$, query table: q, result table R

```
Initial consolidated table R = φ
Initialize resolver parameters based on query table q
for i = 1...K do
   T̄^i ← T^i with obvious duplicates merged.
   Form a bipartite graph G with left nodes as rows of R,
   right nodes as rows of T̄^i and weighted edges on row
   pairs with resolver score > δ.
   Find the maximal matching M in G.
   Merge into R, those rows of T̄^i that are matched in M
   and create new rows in R for the rest.
   Update index on R and resolver parameters.
end for
Merge row pairs (r_i, r_j) of R with positive resolver score
so that no member of r_i belongs to the same source
table as r_j.
```

4.3 Row Ranker

Since the only form of input in the user-query is a set of example rows of the target table, it is impossible to make hard judgments on the relevance of a row. We therefore follow the standard IR practice of ranking rows in the answer table so that the user finds high-quality entries at the top.

A natural sort order is based on the number of sources that support a row. A row that is contained in many lists is likely to be more relevant since each list was selected based on overlap with the query table, This scheme, that we call **Additive** performs poorly mainly because it does nothing about errors in the extraction process. Since we use a probabilistic model to extract rows from unstructured lists, it is possible to get well-calibrated confidence scores indicating the probability of correctness of that extraction. These can be combined to get a probability of the correctness of a consolidated row as follows:

Let R_i be row i of R and let M_i denote the set of (list ℓ, record r) pairs that have been merged to form R_i. Let $\Pr(T^\ell_r|L^\ell_r)$ indicate the probability of the correctness of extraction of the rth record of the ℓth list. We calculate the probability that R_i is correct by assuming independence of the correctness probability of its members, and thus

$$\Pr(R_i|M_i) = 1 - \prod_{(\ell,r)\in M_i} (1 - \Pr(T^\ell_r|L^\ell_r))$$

We call this the **Softmax** scheme. It rewards correctness and mildly favors redundancy.

A second problem in both this and the earlier scheme is that they ignore the number of useful non-empty cells in a row. Different lists might provide different number of columns of the query table, so it is quite possible that common columns like year and state names match many lists and their support overwhelms the support of rows which provide a larger set of columns. In fact, the Softmax approach favors rows with fewer columns because such rows tend to have higher correctness probability.

One way to tackle this is to attach a notion of importance $I(c)$ to each column c of a query. The importance score of a column should ideally be given by the user. In the absence of such an input, we depend on type-specific scores — e.g. number columns are considered one-fifth as important as text columns.

Column importance along with the row correctness probability, can be combined to associate a score that indicates the amount of useful and correct information that a row provides. One intuitive scoring function is this:

$$S(R_i|M_i) = \Pr(R_i|M_i) \sum_{c:R_i(c)\neq \text{null}} I(c)$$

We call this the **Softmax_ColImp** scheme.

Even this scoring function has a bias toward rows with fewer columns that are supported by many lists. Consider a query table to collect mottos of universities. The columns of the query are "University name", "Location" and "Motto". In the absence of any other information, assume that all columns have equal importance. It is easy to find many lists on the web containing the first two columns, and assuming that the extraction task is also easy, rows which contain the first two fields will easily get a score of 2. A row that extracts all three fields needs to have a correctness probability of at least 2/3 to out-rank the two-column rows. This is less likely to happen because extracting mottos is harder, and there are not many lists to soft-max over. We solve this by associating correctness probability with each cell of the consolidated row instead of a single row-level probability. We exploit the power of the full-fledged probability distribution available with each list record, to calculate the marginal probability of the correctness of each column of the extracted record. These probabilities can be calculated using a straightforward extension of the message passing algorithm used during training [11]. We obtain a correctness probability of a cell in the consolidated row by combining the cell-level correctness probability from member lists using

the same soft-max function as follows:

$$\Pr(R_i(c)|M_i) = 1 - \prod_{(\ell,r)\in M_i \quad T_r^\ell(c)\neq \text{null}} (1 - \Pr(T_r^\ell(c)|L_r^\ell))$$

Now we define the score of a consolidated row as the sum of the correctness probability of its cells weighted by the cell's importance:

$$S(R_i) = \sum_{c:R_i(c)\neq\text{null}} I(c)\Pr(R_i(c)|M_i)$$

We call this the **Cell_Softmax** scheme. This scheme does not impose any penalty on larger rows. For example, in the previous example it is likely that the marginal probability of the extraction of University name and Location is close 1 even if the joint probability of the full (University,Location,Motto) combination is smaller. This will cause the total score to be greater than 2.

5. EXPERIMENTAL EVALUATION

We first demonstrate that WWT is more accurate than simpler approaches, especially for 'difficult' queries (defined later). This justifies the use of advanced statistical techniques for extraction, consolidation and ranking. Second, we show via timing results that WWT takes only a few seconds to answer a typical user query. Third, we study the impact of the various extractors discussed in Section 3, and show that exploiting content overlap across lists has a significant payoff, again more so for difficult queries. We also establish the importance of going beyond pure support-based methods for ranking rows in the final result.

Source Lists

We use a web crawl of almost 500 million webpages, from which the list extractor found around 16 million HTML lists after applying the selection heuristics. These were indexed into nine Lucene index shards, with a total size of roughly 10GB, which were probed in parallel for every query.

Query tables

We created a query benchmark by simulating the kind of queries that would be posed by someone trying to automatically compile the wealth of tables in Wikipedia. The tables cover a wide variety of topics ranging from Danny DeVito's filmography, Newbery Medal winners, US university mottos, Kings of Thailand, and so on. Since we were limited by the need to collect ground truth for our queries, we selected a set of 65 Wikipedia tables as our hidden target tables. The number of columns in a target table ranged from three to six. Then we constructed query tables by sampling rows from the target tables. In our experiments, we consider samples with $3, 4, 5, 7, 10$ rows, with five samples taken per size. We use 'query size' to denote the number of rows in a query.

Ground truth and evaluation

We created ground truth for each of the 65 target tables as follows. We used the *entire* Wikipedia table to find the top-20 lists from the list-index. On each returned list we collected human input on: (a) whether the list is relevant to the query or not, (b) for every list record, the target table row it maps to or if it is a new record or irrelevant record, and (c) the correct segmentation of every relevant record. We create a ground truth consolidated table by merging together rows mapped to the same Wikipedia target row. We call this the true consolidated table (CT*). Overall this recovered roughly 75% of the Wikipedia rows and there were 25% new rows that were judged as relevant but did not appear in Wikipedia. This is interesting because our input sources were lists and exact mirroring of content between the Wikipedia tables and our lists was not possible.

Here we stress that even though the index reports the lists as relevant, on an average 45% of the lists turn out to be irrelevant to the target on manual inspection. This brings in many consolidation and ranking challenges for WWT. Also, we categorize a query as *difficult* if each of its relevant lists in the ground truth has less than 60% of the target rows. To answer such queries it is essential to integrate the results from multiple lists through robust consolidation. Roughly 27% of our queries are difficult.

Using the true consolidated table (CT*), we evaluate our system via *recall*, i.e. the number of correct rows found in the top-M rows of the result table R where M is the number of rows in CT*. This is a very stringent metric because we demand the full construction of CT* in spite of the inherent inadequacy of the small query.

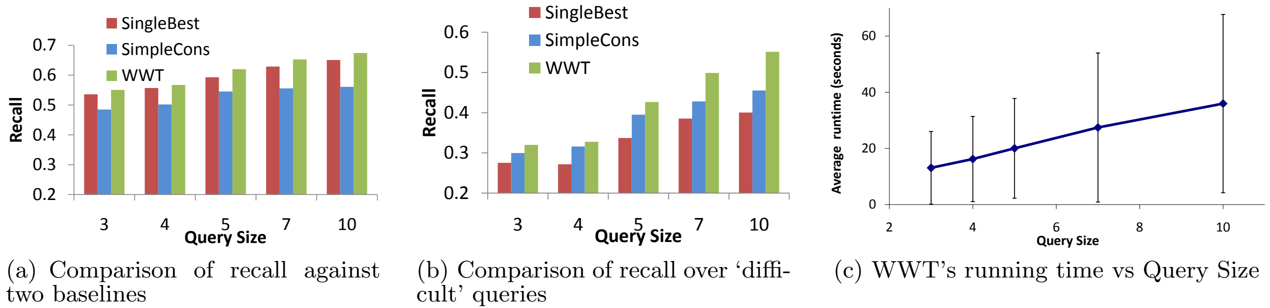
5.1 Overall Performance

Our first set of experiments looks at the end to end performance of WWT — in terms of the recall over all queries and over the difficult queries, and the total time required for computing the answer. We fix WWT to use the following: alignment-based features (Section 3.3.1), Soft Labeler, the default extractor, and the Cell_Softmax ranker. This setup is compared against two baselines: (a) SingleBest, which *magically* picks the source list with the highest recall and ignores the others, and (b) SimpleCons, which is the same as WWT but uses a naïve consolidator — it only merges exact duplicate rows together with allowance for null cells.

These baselines are interesting because first, they will tell us the importance of correctly integrating multiple lists together for our task. If a single list is enough to answer a query, then WWT can gain little by integrating various lists, and effort is better spent in identifying those single lists. Second, the baselines will reveal the need for complex but accurate consolidation techniques.

Figure 4(a) shows the effect of increasing the query size on the overall recall of WWT and the two baselines. We first observe that WWT has a recall of 55% even for a query size of three, in spite of our stringent evaluation criteria. This is significant, as recall on web-scale retrieval tasks are usually poor. As expected, recall increases when more rows are included in the query. WWT is significantly better than SimpleCons, illustrating the need for a good consolidator and ranker. However the gap is not too big with SingleBest, because many of the queries were answerable with a single list, and we post facto selected the single best list. So in Figure 4(b) we perform the comparison on only the difficult queries. Here all the recalls are lower, but WWT outperforms the rest, and SingleBest is the worst.

To demonstrate the usability of WWT, in Figure 4(c) we plot its running time (in seconds) as a function of the query size. Although the average running time increases linearly, we observe a huge variance. This is so because queries of same size may fetch source lists with a widely varying number of rows. Other factors like the number of irrelevant



(a) Comparison of recall against two baselines

(b) Comparison of recall over 'difficult' queries

(c) WWT's running time vs Query Size

Figure 4: Overall performance of WWT vs Query size.

records and number of columns also have a significant effect on the runtime. However for all practical purposes, a small query is typically answered in around 20-30 seconds.

5.2 Extraction

We now establish that the default extractor $E_{default}$ can be improved by using the modifications discussed in Section 3.3. For this we report the extractor's accuracy, which is represented by its F1 score. F1 is the harmonic mean of the cell precision and recall, defined as — the fraction of segments output by the extractor that are correct, and the fraction of true segments output respectively.

We use $E_{default}$ as a baseline, and make the following cumulative modifications: (a) soft labeler instead of hard labeler (b) using the multi-staged extractor E_{staged} (c) adding alignment features. Figure 5(a) plots the F1 vs query size for all the variants. As the query size increases, the F1 of all the variants increase and converge as the benefits of the modifications diminish. But for small query sizes like three, the modifications boost the F1 from 76.8% to 80.8%. Surprisingly, the benefit from using E_{staged} is not huge. The reason is that first, a significant fraction of the queries have only one or two relevant source lists so we can execute only very few stages of E_{staged}. Second, any errors during the initial stages of E_{staged} are compounded in the later stages hurting its performance. However for difficult queries, say of size three, E_{staged} posts an F1 of 76.8% vis-a-vis an F1 of 73.5% of its nearest competitor (plot omitted).

5.3 Ranking

Our last experiment studies the effect of different ranking methods on the overall recall. Figure 5(b) and 5(c) present the ROC curves for query sizes three and ten for four ranking methods discussed in Section 4.3: Additive, Softmax, Softmax_ColImp and Cell_Softmax. The ROC curve is drawn by picking increasing values of n and for each n we pick the first n rows in the result table R and find the number of matched rows $p(n)$ in CT^*. Let P be the total number of matched rows in the entire table R and N be the total size of R. Using these, we calculate for each n the true positive rate $\frac{p(n)}{P}$ which forms the y-coordinate and false positive rate $\frac{n-p(n)}{N-P}$ which forms the x-coordinate. An ideal ranking strategy would place all matched rows before any unmatched row as shown by the dotted lines in the figure. The better a ranking strategy, the closer it is to the ideal curve. From figures 5(b) and 5(c) we observe that the Additive and Softmax methods are inferior to Cell_Softmax and Softmax_ColImp and the gap is as big as 20 points for small values of n.

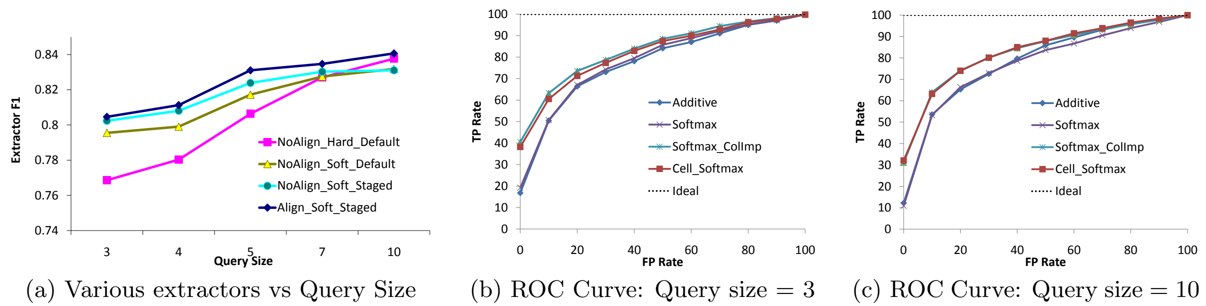
6. RELATED WORK

We are not aware of any prior work on answering ad hoc table completion queries over open-domain and unstructured list sources such as on the web. Our table completion queries is a generalization of set completion queries such as Google Sets which take as input a set of examples such as city names and expand the set to more members of that type. The key challenge here is establishing relevance of a list based on co-occurrence of instances across sources. Several approaches have been used including the Bayesian sets approach of [10], page rank approach of [22], and graph labeling approach of [21]. Extending these to creating multi-column instances from unstructured lists raises new challenges of extraction, consolidation, and relevance not encountered in the single column setting.

The extraction of records from unstructured lists is an extensively researched topic spanning many communities, diverse input types, and usage scenarios [17]. A vast majority of these methods assume the presence of labeled unstructured records and treat structured records as extra information beyond labeled unstructured data [14, 5, 19]. Even among methods [16, 1, 9] that do not require labeled data, the success of the proposed methods depend on the presence of a *large* database of structured records. In our case, the query gives us only a few seed structured records. Therefore, it becomes essential to match the unstructured data with the seeds as carefully as possible for creating reliable training data. We are not aware of any work that addresses this problem.

Another related area is extracting records by exploiting regularities of web pages. We classify the extensive literature on this topic into two kinds: the first kind extracts regular lists,e.g. [2, 6], and the second extract multi-attribute records from lists e.g. [23, 13, 8, 9]. Our focus in this paper is the second problem. Most approaches for that problem assume machine generated lists and learn the template behind them before extracting data. We make no such assumptions as our lists are predominantly human generated. When regularities do hold, we exploit them in a more error tolerant way through alignment features in the statistical model.

Our approach of creating row-resolvers from cell-resolvers using a Bayesian network is similar to the approaches used in [15] and [20] of using a graphical model to express the interaction between attribute-level and record-level de-duplication decisions. One crucial difference in our case is that we can-



(a) Various extractors vs Query Size (b) ROC Curve: Query size = 3 (c) ROC Curve: Query size = 10

Figure 5: Accuracies of different variants of the extractor (5(a)) and ranker (5(b) and 5(c)).

not learn the parameters of the network from any labeled training data. Hence, we presented a design which allows us to interpret these parameters intuitively and determine their values from the input table statistics.

7. CONCLUSION AND FUTURE WORK

We introduced the task of augmenting a table with very few example rows by constructing new rows from unstructured lists on the web. Our key contribution was the WWT system, which extracts rows from lists using statistical models, consolidates and ranks them in the face of huge noise and irrelevance present in the data. We showed that WWT can compute large portions of Wikipedia tables with very few examples in a matter of seconds.

As future work, we plan to extend WWT to incorporate other structured web sources such as HTML tables and logical lists such as web-based catalogs, in a unified way.

Acknowledgments. The work reported here was supported by a research grant from Yahoo! Research and an IBM Faculty award. We thank Soumen Chakrabarti and Webaroo for the Web corpus.

8. REFERENCES

[1] E. Agichtein and V. Ganti. Mining reference tables for automatic text segmentation. In *SIGKDD*, 2004.

[2] M. Álvarez, A. Pan, J. Raposo, F. Bellas, and F. Cacheda. Extracting lists of data records from semi-structured web pages. *Data Knowl. Eng.*, 64(2):491–509, 2008.

[3] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive name-matching in information integration. *IEEE Intelligent Systems*, 2003.

[4] M. Cafarella, N. Khoussainova, D. Wang, E. Wu, Y. Zhang, and A. Halevy. Uncovering the relational web. In *WebDB*, 2008.

[5] A. Chandel, P. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, 2006.

[6] C. Chang. and S. Lui. Iepad: Information extraction based on pattern discovery. In *WWW*, 2001.

[7] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[8] W. W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *WWW*, 2002.

[9] H. Elmeleegy, J. Madhavan, and A. Halevy. Harvesting relational tables from lists on the web. In *VLDB*, 2009.

[10] Z. Ghahramani and K. A. Heller. Bayesian sets. In *NIPS*, 2005.

[11] R. Gupta and S. Sarawagi. Curating probabilistic databases from information extraction models. In *VLDB*, 2006.

[12] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[13] K. Lerman, C. Knoblock, and S. Minton. Automatic data extraction from lists and tables in web sources. In *Workshop on Advances in Text Extraction and Mining (ATEM)*, 2001.

[14] M. Michelson and C. A. Knoblock. Creating relational data from unstructured and ungrammatical data sources. *Journal of Artificial Intelligence Research (JAIR)*, 31:543–590, 2008.

[15] P. Ravikumar and W. W. Cohen. A hierarchical graphical model for record linkage. In *UAI*, 2004.

[16] E. Riloff and R. Jones. Learning dictionaries for information extraction by multi-level bootstrapping. In *AAAI*, 1999.

[17] S. Sarawagi. Information extraction. *FnT Databases*, 1(3), 2008.

[18] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, 2002.

[19] S. Sarawagi and W. W. Cohen. Semi-markov conditional random fields for information extraction. In *NIPS*, 2004.

[20] P. Singla and P. Domingos. Entity resolution with markov logic. In *ICDM*, 2006.

[21] P. P. Talukdar, J. Reisinger, M. Pasca, D. Ravichandran, R. Bhagat, and F. Pereira. Weakly-supervised acquisition of labeled class instances using graph random walks. In *EMNLP*, 2008.

[22] R. C. Wang and W. W. Cohen. Language-independent set expansion of named entities using the web. In *ICDM*, 2007.

[23] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *WWW*, 2005.