S. Seth, G. Nagy. Segmenting Tables via Indexing of Value Cells by Table Headers, Procs. ICDAR'13, Washington, D.C., August 2013.

Segmenting Tables via Indexing of Value Cells by Table Headers

Sharad Seth
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA 68588-0115
seth@cse.unl.edu

George Nagy
Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering
Troy, NY, USA 12180
nagy@ecse,rpi.edu

***Abstract*— Correct segmentation of a web table into its component regions is the essential first step to understanding tabular data. Our algorithmic solution to the segmentation problem relies on the property that strings defining row and column header paths uniquely index each data cell in the table. We segment the table using only "logical layout analysis" without resorting to any appearance features or natural language understanding. We start with a CSV table that preserves the 2-dimensional structure and contents of the original source table (e.g., an HTML table) but not font size, font weight, and color. The indexing property of table headers implies a four-quadrant partitioning of the table about a minimum index point. The algorithm finds the index point through an efficient guided search. Experimental results on a 200-table benchmark demonstrate the generality of the algorithm in handling a variety of table styles and forms.**

Keywords— indexing by header strings; minimum indexing point; table segmentation

I. INTRODUCTION

Research on processing tables has moved from the earliest work on finding the underlying grid structure of scanned and ASCII tables [1, 6, 12] to locating and bounding HTML tables [17, 18], and more recently, to end-to-end conversion of visually meaningful web tables to relational databases and data stores amenable to online query and search [3, 5, 7, 13, 14, 15].

Our recent efforts focused on realizing and improving on the original conception of TANGO [14], an end-to-end system for generating ontology from tables. Grounding our analytical work on syntactical table analysis of X. Wang [16], we target the efficient extraction of the relations of row and column header cells to content cells. The two-dimensional indexing of each content cell by corresponding row and column header paths is essential for understanding individual tables as well as combining related facts from different tables.

Fig. 1 shows the data flow of our overall system. We first convert a *source table* in HTML to a corresponding *grid table* in comma-separated-values (CSV) format using Excel/VBA programs. The spanning cells in the HTML table are divided into atomic cells in the CSV table, such that every row (column) has the same number of aligned cells, i.e., the CSV table is a rectangular array of atomic cells defined only by their grid coordinates and their content as a text string. Although almost all format information is lost in the conversion, the CSV format serves a broad range of applications and provides a standardized cell-based representation for downstream processing.

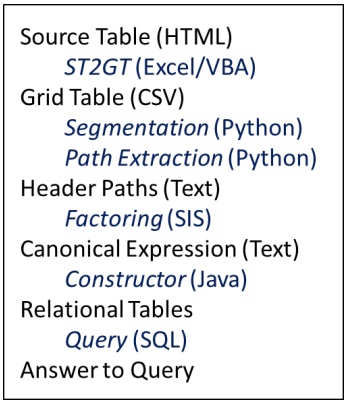

Fig. 1. Data flow of TANGO [14]

The interpretation of the 2D grid table starts with its five-way segmentation into *stub head*, *row header*, *column header*, *data* (or *delta*) *region*, and a composite *auxiliary region* at the top and bottom and occasionally between the column headers and data cells of the table, conveying such information as the title, units, and notes of various kinds. The segmentation of the table chosen as our running example is shown in Fig. 2. This example shows why the stub-head and the data-cell regions are sufficient to determine the other three parts of the segmentation.

Four *critical cells* that bound the stub-head and data regions completely define the segmentation [10]: CC1 and CC2 correspond to the top-left and bottom-right cells of the stub head; CC3 and CC4 correspond to the top-left and bottom-right cells of the data-cell region. If the stub head consists of a single cell, as in Fig. 2, then CC1 and CC2 coincide.

From a segmented table, the row-header and column-header paths can be extracted for indexing the cells in the data region. These paths can be *factored* into canonical expressions to recover the Wang category trees of the headers [2]. With the canonical expression and table's data region indexed by the header paths, we can generate the corresponding relational table and populate it with data [10]. We can then query the

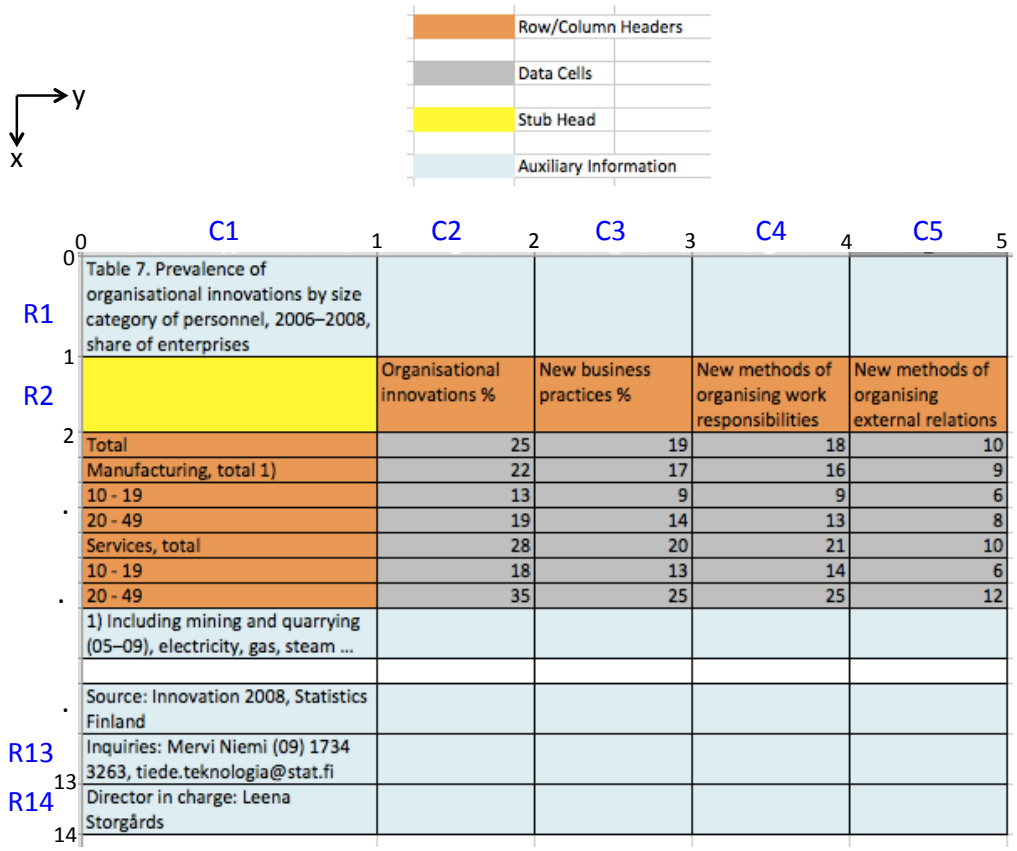

Fig. 2 The grid table, used as the running example. The color coding shows the five-part segmentation of the table into row & column headers, data cells, stub head, and auxiliary information. The stub-head and data-cell regions are sufficient to determine the segmentation. Also shown, is the coordinate system for the cells (e.g. R2C1 and x=2, y=1 for the stub head) of the grid table. The value "18" in cell R7C2 (x=7, y=2) is indexed by the header paths *Services, total 10 - 19* and *Organizational innovations*.

table with SQL and otherwise manipulate it, along with other tables, in a standard relational database.

Our current approach is a radical departure from our earlier work on segmentation of the grid table. Correct segmentation is critical as it impacts the performance of all the downstream steps. We presented *Vericlick,* an interactive program for segmentation in [9] and later incorporated it into the partially automated *CC Recognizer* [8]. The latter is based on seven heuristic appearance features of each cell and applies statistical pattern recognition methods to maximize the posterior probability of classifying each cell in the table. In contrast, the solution demonstrated here is essentially algorithmic and based on a fundamental property of all tables. As such, it is independent of the language, contents, and layout details. Vericlick can, of course, still be applied for interactive correction of residual segmentation errors.

The rest of the paper is organized as follows. After presenting the background in Section 2 on indexing tables by tuples of strings in the row and column headers, in Section 3 we describe the algorithm for finding the minimum index point and the post processing necessary to complete the segmentation of a table. The test dataset characteristics and segmentation results appear in Section 4. Section 5 summarizes our contribution.

II. INDEXING OF TABLES

We will find it convenient to refer to both individual cells and individual points in a grid table by using the two related coordinate systems illustrated in Fig. 2. In either case, we assume a counter-clockwise coordinate system with *x* pointing down from the origin in the top-left corner. However, to distinguish the two, we refer to the top-left *cell*, in Excel style, as R1C1 and the origin *point* as (0, 0).

The content of each cell is *cell string*. An empty cell has the *null-string* as its value. The cell-string function is extended to a group of contiguous cells in a column or a row by the ordered list (*tuple*) of cell strings from top to bottom in a column or from left to right in a row.

Consider the four-way partitioning of the table grid by point (*x, y*), as shown in Fig. 3. (x,y) is a *column-index* point of the table if no two column tuples in part C are the same. In other words, point (x,y) *indexes columns* if the top x rows in columns y+1 through ymax index the columns below them. The row indexing is defined similarly in terms of the row

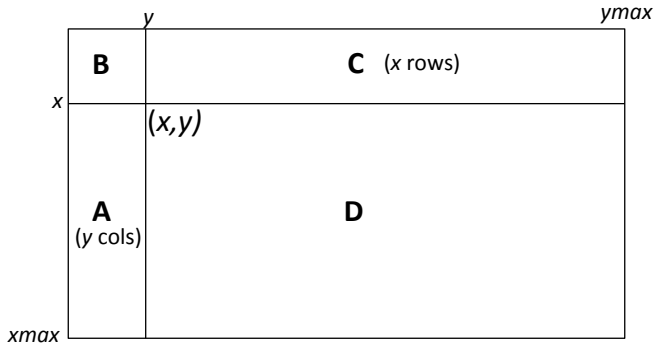

Fig. 3. Four-way partitioning defined by point (x, y) used in explaining the indexing of tables by rows in part A and columns in part C.

tuples in part A. If point (x,y) indexes both the rows and columns, it is called an *index point.*

It is easily verified that no index point exists in a table with two identical rows or columns. We call such tables non-indexable. Non-indexable tables are rare: in our experiments with 200 tables, only two related tables from the same source were non-indexable (see [4] for an example).

We note that row and column indexability of points may only increase (i.e., change from non-indexing to indexing) with increasing values of x or y. This monotonicity arises because as x increases while y is kept fixed, more rows are available to column index a *smaller* region of the table. At the same time, the same number columns may be able to row index the smaller region. Hence, if (x, y) is an index point, so are all points (x',y'), with x' > x or y' > y. However, this monotonicity does not guarantee the existence of a unique *minimum index point* (*MIP*) for an indexable table because, for example, it does not rule out (x', y'), with x' < x and y' > y, as another MIP. For segmentation we are interested in a MIP nearest to the origin, which is found to be unique for real tables and represents the critical cell CC2.

Such a MIP cannot always be found by pure indexing because table designers may use label alignment (e.g. indentation) or font attributes (e.g. size, style, or color) to denote header hierarchy in the source table. This is the case for column C1 in Fig. 2, where the headers "Manufacturing, total 1)" and "Services, total" were in boldface in the source (HTML) table because they are meant to qualify the numerical ranges underneath them.

Same-row and same-column hierarchies require prefixing duplicate labels with single labels in the same row or column, as explained in Section III. This makes all the row headers unique and yields the MIP (2, 1) for the table. A naïve approach, without prefixing, would select (2, 2) as the MIP for this table, mistakenly including the first column of data to construct unique row headers. About 12% of the tables in our collection require prefixing.

III. INDEXING AND SEGMENTATION ALGORITHM

Incorporating the minimum index point as the basis for table segmentation is the hallmark of our approach. We present a high-level description of the algorithm to find the critical cells CC1–CC4 of a table in Fig. 4. The algorithm first eliminates any empty rows and columns beyond the table. Next, it finds CC4 at the bottom of the data region based on the property that data rows rarely have empty cells at *both* ends (cf. Fig. 6). The *RowCheck* test that verifies this condition is also used later in identifying CC3. Bi_Indexer is then called to determine CC2 and CC1. Bi-indexer distinguishes repeated labels from non-repeated labels in the first column and prefixes each repeated label with the nearest non-repeated label above it, if one exists. The rows of the table are similarly prefixed to disambiguate hierarchical relations in the column header that were often marked in the original table by appearance features like indentation and font weight.

Find Critical Cells

1. Remove empty rows below and empty columns to the right of the table.

2. Find bottom of data region*:* Scan from bottom to top, exclude rows by applying the *RowCheck* condition:

> At least one of the last two, or the third or fourth cells (if the table has more than three columns), is empty.

This yields the row (x-) coordinate x4 of CC4 = (x4, y4), where y4 is the y-coordinate of the last column of the table.

3. Call Bi_Indexer to determine critical cells CC1 and CC2.

4. Determine CC3: Skip over anomalous rows satisfying *RowCheck* below current header region, to find the row (x-) coordinate x3 of CC3 = (x3, y3), where y3 is y2+1.

Bi_Indexer

Prefix first column
Prefix rows
x = 0; y = 1
while((x,y) does not index columns): x = x+1
while((x,y) does not index both rows and columns): y = y+1
while((x,y) indexes both rows and columns): x = x-1
CC2 = (x2,y2) = MIP = (x+1,y)
chr = reverse of the first R2 rows of the table; x = 1
while((x,y) does not column index chr): x = x+1
CC1 = (x1,y1) = (x2-x+1, 1)

Prefix

If the row (or column) has a repeated cell label and has to the left (above) a non-repeated cell label, prefix the row (column) label with the nearest non-repeated label.

Fig. 4. An High-level description of the algorithm to find the critical cells CC1–CC4 of a table.

The search for the MIP is carried out in three successive while loops. Starting near the origin, the first while loop moves the search point down the first column until there are enough

rows to index the columns. The second while loops switches the direction of search to the right, until both the rows and columns are indexed, i.e., an index point is reached. This index point may not, however, be minimal, because, although the second loop assures a minimal row index, it may over-index the columns. The third while loop remedies this by moving the search point up until both rows and columns are no longer indexed. The search point reached just before this condition is met is the MIP, which is also CC2.

Although CC2 corresponds to the MIP, the column header defined above often includes redundant rows at the top like the title row. Bi_indexer eliminates such redundant rows by traversing the column header in reverse until it reaches a row unnecessary for the column index. CC1, corresponding to the minimum-height column header, is then easily determined. Finally, RowCheck is applied to skip over any anomalous rows below the column header that do not belong to the data region. The x-coordinate of CC3 is the first row of the data region and its y-coordinate is one more than y2, the y-coordinate of CC2.

With the exceptions noted below, indexing uses the tuples of row and column strings as the keys.

1. Our algorithm does not assume that visual cues such as boldface type, indentation, color, or larger font size are preserved in Source Table to Grid Table (ST2GT) conversion. Instead, row indexing seeks a unique key consisting either of the contents of a single cell, or of the combination of that string with a unique prefix above it. Then a key is sought in the prefixed column. The minimum number of columns that constitutes an index is reported as the width of the row header candidate. Column indexing is performed identically, but on a transposed version of the table.

2. Another situation, where tuples may be modified for indexing, is illustrated in Fig. 5: Column C1 shows State names, say, AZ in R6C1; next to and below AZ in column C2 are unique city names like Phoenix in R7C2, and Tempe in R8C2. Then, the key for row R6 will be ('AZ', ''), and the keys for R7 and R8 will be ('AZ', 'Phoenix') and ('AZ', 'Tempe'). The critical cells are CC1 = (R2, C1) and CC2 = (R2, C2).

3. If all the cells in the data region of a header row or column are blank that row or column is ignored for indexing. Also, indexing fails if all the cells in a header row or column are blank

	1	2	3	4	5
1		Maximum temperature			
2			2010	2011	2012
3	AL		109	116	115
4		BIRMINGHAM	104	108	107
5		MBILE	104	110	108
6	AZ		102	99	104
7		PHOENIX	99	97	101
8		TEMPE	95	96	98

Fig. 5. Prefixing. States names prefixed to the city names that they modfy.

Locating CC1 and CC2 is completely algorithmic. The postprocessing necessary to ensure that CC3 and CC4 delimit only the data region and exclude superfluous rows above and below it, i.e., *RowCheck*, has a heuristic component based on table publishing conventions.

IV. EXPERIMENTAL RESULTS

200 tables were randomly drawn from a set of tables collected earlier from large statistical websites in the US and abroad [11]. The geopolitical and research sources included Statistics Canada, Science Direct, The World Bank, Statistics Norway, Statistics Finland, US Department of Justice, Geohive, US Energy Information Administration, and US Census Bureau. On average the numbers of rows and columns in a table were 7 and 17; the corresponding maximum values were 20 and 64 (without counting footnote rows).

As we aim here to achieve minimum indexing, the existing ground truth for these tables was adjusted as follows. The new ground truth does not modify the critical cell CC2 as it corresponds to the MIP. CC4, corresponding to the bottom-right cell of the data region is also kept unchanged. However, CC1 and CC3 may change because the column header height is defined by only the bottom rows of the column header that are required for indexing. The original column header may include additional rows above this minimal column header. For example, a row spanning the width of the column header may include the table title or the label of the root-category.

The performance of our Python program on the 200 tables can be summarized as follows:

- 99% correct segmentation: (198/200)
- 100% correct on stub heads (correct CC1 and CC2)
- Correct identification of the two non-indexable tables
- Total execution time for 200 tables: 3 second

Examples of correct segmentation where appearance-based methods might have trouble are shown in Figs. 6 through 8 that display the relevant parts of three web tables imported into Excel. In Fig. 6, our algorithm correctly identifies the three-column row header by detecting the blank last-two row-header cells. Fig. 7 is a table with a complex row-header structure: there are repeated entries in the first two columns for Minnesota ('MN') and for several other states not shown in the figure. Unlike a casual human reader, our algorithm correctly recognizes this fact and finds the row header that extends to the third column. In Fig. 8 an appearance based method would have difficulty in including row R6 or excluding R7 from the column header.

The table in Fig. 9 and another one similar to it are incorrectly segmented because the sparse top row of the data is attributed to the ancillary region.

This method provides more than segmentation: the row and column indexes that are byproducts of this segmentation paradigm are useful for converting the contents of the table to a relational form. Our method also seems accurate enough for checking the integrity of web tables design.

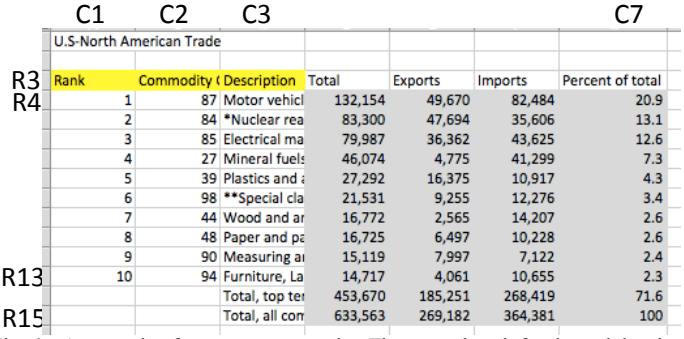

Fig. 6. A example of correct segmentation The ground truth for the stub head and data cell regions is shown by the color code.

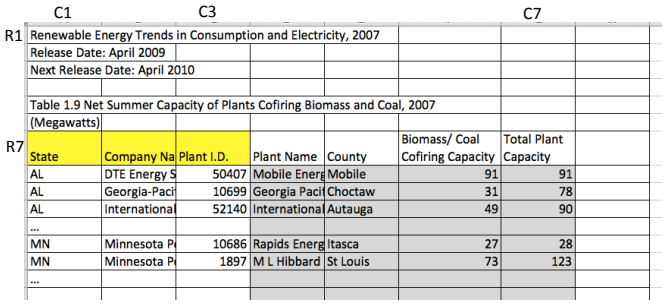

Fig. 7. Correct segmentaton in spite **o**f the unusual three-column row header and the blank data rows.

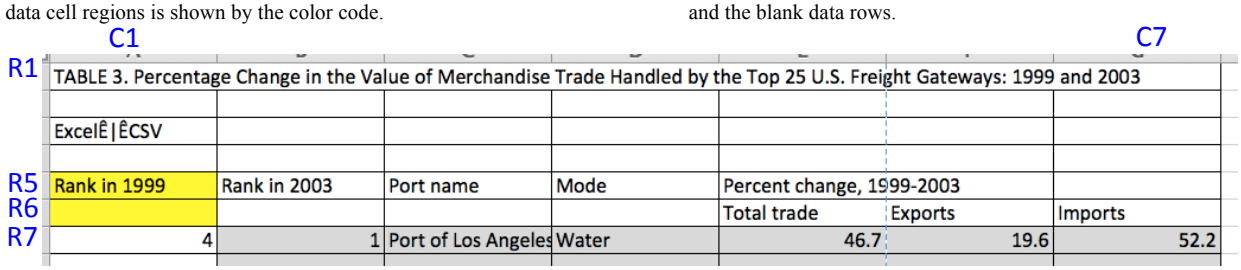

Fig. 8. Our segmentation algorithmfinds the correct segmentaton of this lopsided column header hierarchy.

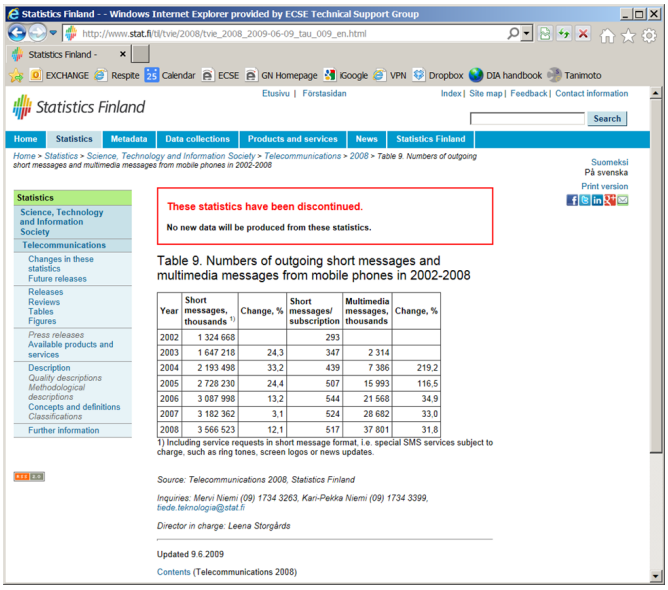

Fig. 9. Errors. The sparse top data row was assigned to the ancillary region in both incorrectly segmented tables (that were from the same source).

V. CONCLUSION

The main contribution of this research is an accurate and algorithmic method of table segmentation based on the one essential property of tables, rather than on heuristics like all previous document segmentation methods. Although it cannot be used for most forms, it is equally useful for tables with textual, numerical, or foreign language content.

REFERENCES

[1] Bayer, T.A. "Understanding structured text documents by a model based document analysis system." *ICDAR*. 1993. 448-453.

[2] Embley, W., M. S. Krishnamoorthy, S. Seth, and G. Nagy. "Factoring Web Tables." *IEA/AIE*. Springer-Verlag, 2011. 253-263.

[3] [3] Fang, J., P. Mitra, Z. Tang, and C. L. Giles. "Table Header Detection and Classification." *AAAI*. 2012. 599-605.

[4] Finland, Statistics. "Students in apprenticeship training number 70,000 in 2008." 2008. http://www.stat.fi/til/aop/2008/04/aop_2008_04_2009-11-03_tie_004_en.html (accessed 2009).

[5] Halevy, A., P. Norvig, and F. Pereira. "The Unreasonable Effectiveness of Data." *IEEE Intelligent Systems* 24, no. 2 (2009): 8-12.

[6] Handley, J. C. "Table Analysis for Multiline Cell Identification." *DRR VIII*. IS&T/SPIE ElectronicImaging, 2001. 34-43.

[7] Krüpl, B., M. Herzog, and W. Gatterbauer. "Using Visual Cues for Extraction of Tabular Data from Arbitrary HTML Documents." *WWW*. ACM, 2005. 1000-1001.

[8] Nagy, G. "Learning the Characteristics of Critical Cells from Web Tables." *ICPR*. 2012.

[9] Nagy, G., and M. Tamhankar. "Vericlick, An Efficient Tool for Table Format Verification." *DRR*. 2012.

[10] Nagy, G., S. C. Seth, D. Jin, D. W. Embley, S. Machado, and M. S. Krishnamoorthy. "Data Extraction from Web Tables: The Devil is in the Details." *ICDAR*. 2011. 242-246.

[11] Padmanabhan, R., R. C. Jandhyala, M. Krishnamoorthy, G. Nagy, S Seth, and W. Silversmith. "Interactive Conversion of Large Web Tables." *GREC*. 2009. 25-36.

[12] Pinto, D., A. McCallum, X Wei, and W.B. Croft. "Table extraction using conditional random fields." *SIGIR*. 2003. 235-242.

[13] Pivak, A., P. Ciamiano, Y. Sure, M. Gams, V. Rahkovic, and R. Studer. "Transforming arbitrary tables into logical form with TARTAR." *Data and Knowledge Engineering* 60, no. 3 (2007): 567-595.

[14] Tijerino, Y. A., D.W. Embley, D. W. Lonsdale, and Nagy G. "Towards Ontology Generation from Tables." *World Wide Web Journal* 6, no. 3 (2005): 261-285.

[15] Venetis, P., et al. "Recovering semantics of tables on the web." *Proceedings of the VLDB Endowment* 4, no. 9 (2011): 528-538.

[16] Wang, X. "Tabular Abstraction, Editing, and Formatting." *Doctoral Dissertation*. University of Waterloo, 1996.

[17] Wang, Y., and J. Hu. "Detecting Tables in HTML Documents." *DAS*. 2002. 249-260.

[18] Yoshida, M., and K. Torisawa. "A method to integrate tables of the World Wide Web." WDA. 2001. 31-34.