Building Mashups by Demonstration

RATTAPOOM TUCHINDA
National Electronics and Computer Technology Center (Thailand)
CRAIG A. KNOBLOCK
University of Southern California
and
PEDRO SZEKELY
University of Southern California

The latest generation of WWW tools and services enables Web users to generate applications that combine content from multiple sources. This type of Web application is referred to as a mashup. Many of the tools for constructing mashups rely on a widget paradigm, where users must select, customize, and connect widgets to build the desired application. While this approach does not require programming, the users must still understand programming concepts to successfully create a mashup. As a result, they are put off by the time, effort, and expertise needed to build a mashup. In this article, we describe our programming-by-demonstration approach to building mashup by example. Instead of requiring a user to select and customize a set of widgets, the user simply demonstrates the integration task by example. Our approach addresses the problems of extracting data from Web sources, cleaning and modeling the extracted data, and integrating the data across sources. We implemented these ideas in a system called Karma, and evaluated Karma on a set of 23 users. The results show that compared to other mashup construction tools, Karma allows more of the users to successfully build mashups and makes it possible to build these mashups significantly faster compared to using a widget-based approach.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*User-centered design*; *Evaluation/methodology*

General Terms: Algorithms, Design, Human Factors, Performance

Additional Key Words and Phrases: Mashups, Information Integration, Programming by Demonstration

1. INTRODUCTION

We need information to make good decisions. In the past, access to needed information was limited to traditional printed media or word of mouth. The Internet, however, has changed the information landscape. Nowadays information can

---

This article is an extended version of [Tuchinda et al. 2008], which appears in the *Proceedings of the 13th international conference on Intelligent user interfaces*, 139-148. Parts of this article also incorporate material from [Tuchinda et al. 2007], which appears in the *Proceedings of the 12th international conference on Intelligent user interfaces*, 170-179.

Author's address: R. Tuchinda, National Electronics and Computer Technology Center, 112 Thailand Science Park, Phahonyothin Road, Klong 1, Klong Luang, Pathumthani 12120, Thailand.

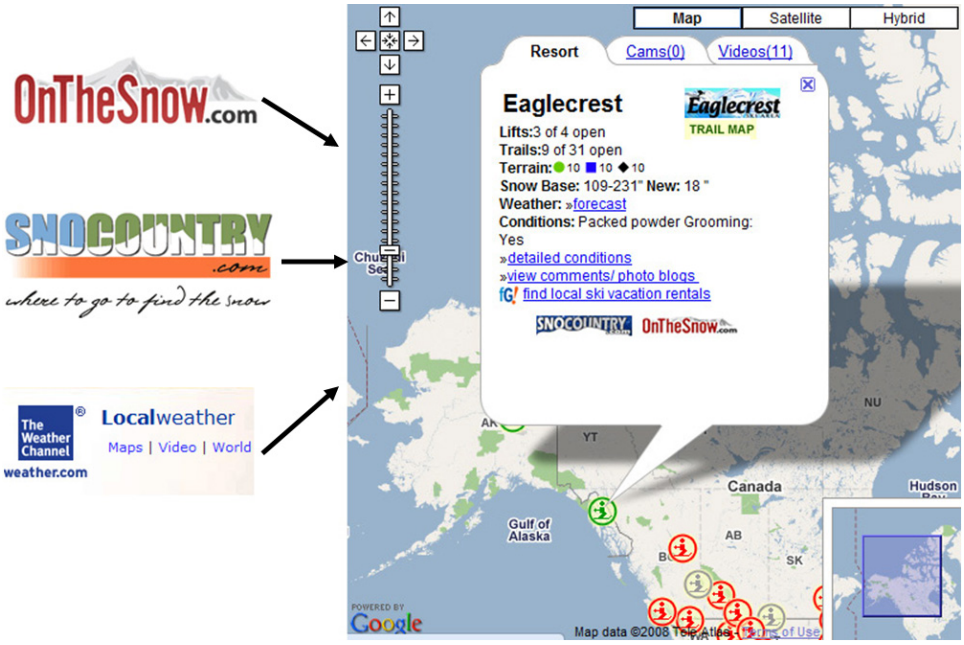

Fig. 1. An example Mashup called Skibonk. Skibonk helps users decide where to go skiing by combining resort listing Web sources with snow report sources

be accessed with a click of a mouse. Examples of such information include pricing/reviews of products from multiple vendors, maps, and statistics. Accurately integrating the information available on the Internet can provide valuable insights useful in decision-making. However, the information one needs is usually scattered among multiple Web sites. It can be time-consuming to access, clean, combine, and try to make sense of the available data manually.

A Mashup is a term used to describe a Web application that integrates information from multiple data sources. Figure 1 shows an example Mashup called Skibonk (http://www.skibonk.com). Skibonk integrates resort listings and snow reports together to help users decide where to go skiing. A green icon means that a ski resort at that location is open.

To create Mashups, integration systems must help users solve five separate problems:

(1) **Data retrieval:** This problem concerns data extraction from Web sites. While the concept of the Semantic Web has received significant attention recently, most Web sites still require a wrapper, which uses extraction techniques to convert the data from HTML into a structured form.

(2) **Source modeling:** After extracting data, we need to determine the mapping between a new data source and an existing data sources. The mapping can be done by assigning an existing attribute name to a new data column when the semantics are the same.

(3) **Data cleaning:** Data from multiple sources must be normalized to have the

same format. For example, the extracted data, *37 MAIN STREET* might need to be transformed into *37 Main St.* to match the naming convention of existing data sources.

(4) **Data integration:** Assuming that the data is normalized, we can treat the problem of combining the data from multiple sources similar to the way we combine the data from multiple databases. This process is accomplished by formulating queries by example to access and integrate data across sources.

(5) **Data display:** Once we combine the data, we need to present it in a way that is easy to understand. Depending on the type of the data, we might opt to use a table or display the information on a map using a tool such as Google Maps.

While there exist attempts to facilitate the process of building information integration applications, none is sufficiently easy to use to enable a Web user to build an end-to-end information integration application. As a result, a casual user is put off by the time, effort, and expertise needed to build a Mashup.

Most existing Mashup tools use what we define as the widget paradigm. Figure 2 shows Yahoo's Pipes, a Mashup tool that incorporates this paradigm. In the widget paradigm, a user is presented with an array of widgets to choose from, where each one represents a specific programming operation. To build a Mashup, the user needs to drag widgets onto a canvas, customize each widget, and connect them to form a work flow that generates the output data. This output data can then be exported as XML, published as an RSS feed, or displayed on a map.

There are three inherent problems with the widget paradigm. First, locating the appropriate widget to perform a specific operation can be time consuming. For example, Yahoo's Pipes has 43 widgets, while Microsoft's Popfly has around 300 widgets. Second, while there is no code to write when using widgets, the fundamental approach behind this paradigm is to abstract a particular programming operation into a widget. As a result, customizing some widgets may require knowledge of programming. The **regex** widget in figure 2 is one such example. Third, most Mashup tools only address some Mashup building problems while ignoring others. For example, Dapper (www.dapper.net) mainly addresses data retrieval, while Yahoo's Pipes completely ignores data retrieval, but focuses on data cleaning and data integration; walking their users through all five Mashup building problems seems to be too difficult and time consuming. One of the reasons might be the lack of a paradigm that can encapsulate all these problems into one simple interaction framework. As a result, building Mashups is complicated and the range of Mashups that can be built by naive users is limited.

There also exist many advanced systems that provide in-depth solutions to each separate Mashup building problem. For example, Potter's Wheel [Raman and Hellerstein 2001] and Spider [Koudas et al. 2005] support sophisticated cleaning and schema matching operations. However, these systems excel at one thing and may not integrate well with other systems to provide an *end-to-end* information integration solution.

The goal of our work is to demonstrate a conceptual framework that allows any Web user without programming knowledge to build Mashups. We use and build upon existing research, where applicable. However, the difference between Karma and other research work is that it provides users with a uniform interaction

paradigm to tackle each mashups building subproblems in an intuitive way. We created the Karma framework based on the following three key ideas:

(1) **Focus on the data, not the operation:** For a user with no programming knowledge, interacting with data is more natural. For example, the user might not know which operations/widgets to use to clean *Jones, Norah*, but the user can specify the end result that he/she wants to see in the data form (i.e., *Norah Jones*); the operation is then induced indirectly based on what the user supplies during the interaction by using data and some predefined transformations.

(2) **Leverage existing databases:** Instead of trying to solve every problem from scratch, which requires feedback and expertise from the user, we can use existing databases to help simplify the problem. For example, if a data point extracted from a Web page is misspelled (e.g., "wahington" instead of "washington"), it would be time consuming for the user to manually locate the misspelled data point. We can simplify this problem by comparing new data points with existing data points and alert the user when we find any irregularities with a list of suggestions for replacements.

(3) **Consolidate rather than divide-and-conquer:** Divide-and-conquer is often regarded as the golden rule to solve many computer science problems. However, many Mashup building issues are interrelated. It is possible to exploit the structure such that solving a problem in one area can help simplify the process of solving a problem in another area. In addition, all the interactions can occur on a spreadsheet platform where the user solves all the problems by manipulating data in a single table instead of trying to create a work flow by connecting widgets.

The rest of this article is organized as follows. Section 2 discusses Mashup types supported by our approach. Section 3 introduces an example scenario. Section 4

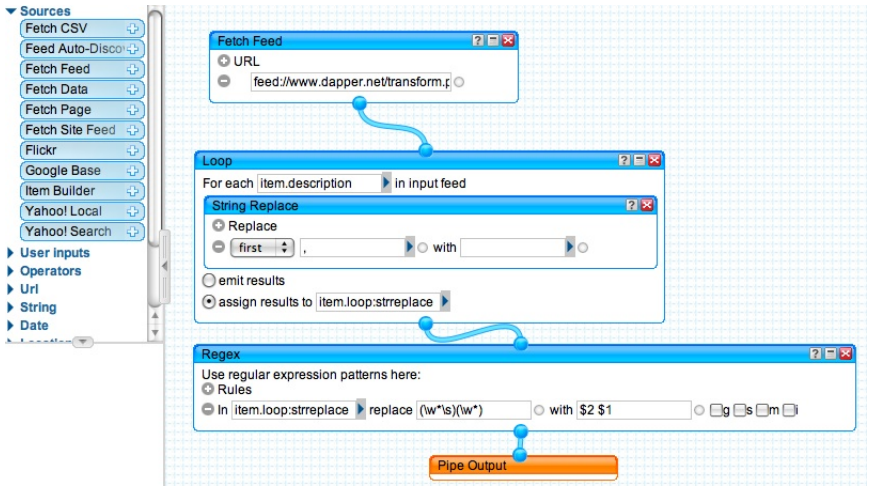

Fig. 2. An example of the widget approach to building Mashups. The user needs to locate, customize, and connect widgets to create a Mashup.

through section 7 show how to combine and solve each Mashup building problem (i.e., data retrieval, source modeling, data integration, and data display) respectively under a unified table paradigm. Section 8 reviews the related work in each problem area. Section 9 describes our evaluation and result. Finally, section 10 summarizes our contributions and lists possible directions for future work.

2. CATEGORIZING MASHUPS

One way to categorize Mashups is by functionality. Figure 3 categorizes Mashups based on functionality.

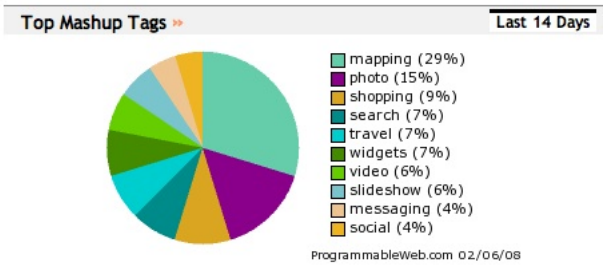

Fig. 3. The composition of all Mashups displayed by functionality from Programmableweb

In this article, we categorize Mashups by their work flow structures ranging from the easiest to the hardest to implement. To generate this categorization, we look at the top 50 popular Mashups on the Programmableweb.com site and divides them into four types of Mashups. While 50 Mashups is a small number, the distribution in terms of functionality is similar to that of Figure 3. As a result, we believe this is a good approximation of the overall Mashup population. The Mashup categorization based on the structure is shown below:

(1) **One simple source:** This Mashup type is constructed by extracting data from a single source and placing them on a map. There is minimal data cleaning and no source modeling or data integration. An example for this mashup type is Berkeley Crime Log (http://berkeleyca.crimelog.org/), a mashup that shows the various types of crimes on Google Maps.

(2) **Combining data points from two or more separate sources:** While this Mashup type incorporates two or more sources, the data does not need to be modeled, cleaned, or integrated. A map is used as a bucket to show the data from each different data source. An example for this mashup type is Hotspotr (hotspotr.com/), a mashup that shows wi-fi hotspots.

(3) **One source with a form:** To extract data the source requires a user to supply the information through a HTTP form before the data can be retrieved. An example for this mashup type is LA Housing Map (http://www.housingmaps.com/), a mashup that extracts data from Craigslist and presents it in a searchable form on Google Maps.

(4) **Combining two or more sources using a database join:** The data from one source needs to be joined with data from another source. The data cleaning and source modeling are more important in this case. An example for this mashup type is Skibonk as shown in Figure 1 (www.skibonk.com/), a mashup that combines snow condition with resorts to help skiers find a vacation spot.

Among these four Mashup types, some of them require specialized operations and knowledge, such as a customized display or complex user interactions. These specialized Mashups cannot be implemented by casual users and account for about 53 percent of the total Mashup population. Our goal is to allow casual users to build the nonspecialized Mashups (i.e., the other 47 percent) easily. With Karma, users will be able to build Mashups that have similar functions to real-world Mashups coded by programmers. For example, simpler versions of each real world Mashup mentioned above could be implemented using Karma.

3. AN EXAMPLE SCENARIO

This section shows how a user would interact with Karma to build a useful mashup. For example, a particular restaurant might receive rave reviews from a restaurant review website, but has a **C** rating on a government health inspection website. A health conscious person would require information from both websites to make a sound decision on whether to dine at this restaurant.

To build this Mashup, the user needs to combine data from two Web sources: LA City Guide (http://losangeles-.citysearch.com) and the LA department of public health (http://ph.locountry.gov/r-ating), and then display the final result on a map. Note that this mashup is a type 4 mashup where a) two data sources are used and b) a database join is required. To simplify the building step, we will assume that the data from the LA department of public health is already extracted, modeled, cleaned, and saved into the database.

We will break the Mashup building process into four steps: data retrieval, source modeling, data cleaning, and data integration. In practice, however, the user may switch freely back and forth between each step. Also, the user can preview the Mashup display (i.e., map) at any time. Details of the inner workings will be elaborated in the next section.

Figure 4 shows the interface for Karma. The left area is an embedded Web browser, where the user can navigate through Web pages. The upper right area is a blank table where the data is populated based on the user's interaction with Karma. The lower right area shows multiple modes and their options from which the user can select.

3.1 Data Retrieval

First the user will extract the data from the sushi page on the left into a table on the right side, as shown in Figure 4. The end result will look like the table in Figure 6, which contains restaurant names, addresses, descriptions, and number of reviews. Karma's goal is to let the user do this by providing only a small set of examples.

Once the user navigates to the best sushi restaurants page, he extracts the data by highlighting a segment of the text *Japon Bistro* on the page and then drags and

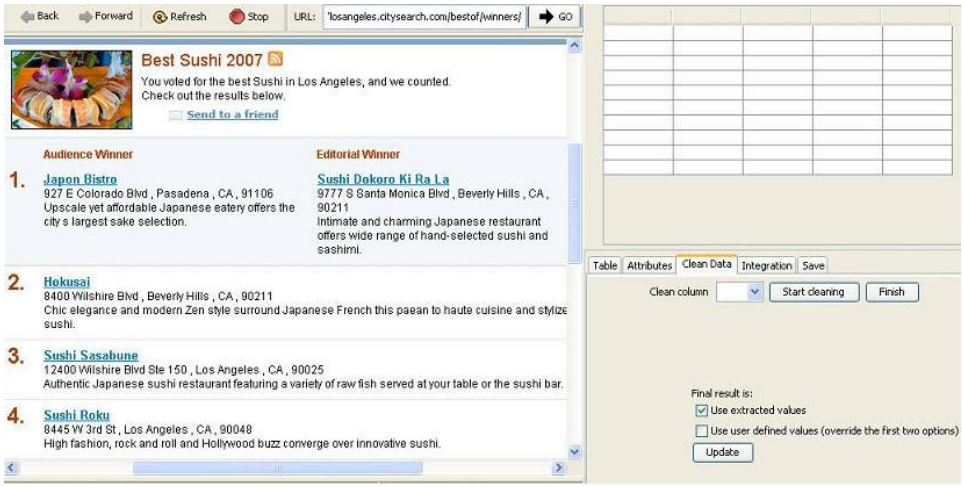

Fig. 4. The interface of Karma

select one			
Japon Bistro			
Sushi Dokor..			
Hokusai			
Sushi Sasab..			
Sushi Roku			
Hide Sushi			
Fat Fish			
Sushi Katsu-ya			
Gindi Thai /..			
Katana			
Echigo			

Fig. 5. By dragging *Japon Bistro* into the first row, Karma automatically fills the rest of the column

select one	address	select one	select one
Japon Bistro	927 E Color..	Upscale yet...	31 Reviews
Sushi Dokor..	9777 S Sant..	Intimate an...	3 Reviews
Hokusai	8400 Wilshir...	Chic eleganc...	30 Reviews
Sushi Sasab..	12400 Wilshi...	Authentic Ja...	66 Reviews
Sushi Roku	8445 W 3rd...	High fashion...	62 Reviews
Hide Sushi	2040 Sawtel...	No fuss, jus...	25 Reviews
Fat Fish	616 N Rober...	Inventive ro...	38 Reviews
Sushi Katsu-ya	11680 Vent...	The MOCA o...	49 Reviews
Gindi Thai /..	4017 W Riv...	Burbank res...	29 Reviews
Katana	8439 W Sun...	Rustic Japa...	96 Reviews
Echigo	11217 Sant...	Stellar sushi...	49 Reviews

Fig. 6. The user extracts the whole list by dragging only four values into the first row of the table

drops the highlighted text into a cell in the table on the right. Recognizing that the data element is a list on the Web page, Karma proceeds to extract the rest of the restaurants from the page and fills the first column of the table in Figure 5.

The user can proceed to extract the address and the restaurant description of Japon Bistro, and Karma will automatically fill in the rest of the table. Note that the user can also click the http link of Japon Bistro to go to its separate detail page and extract the number of reviews. Recognizing that the detail page belongs to Japon Bistro, which in turn is a part of the restaurant list in the original page, Karma then iterates through each restaurant in the list and extracts the corresponding data from each detail page.

Figure 6 shows the result table where the user has extracted the restaurant name, address, description, and the number of reviews. Note that the user only has to drag in the four values in the first row to populate the entire table.

3.2 Source Modeling

In the source modeling mode, Karma will help the user assign the right attribute name to each data column. While the user is busy extracting data from the page, Karma compares extracted data with existing data in its repository to identify possible attribute names. For a column where Karma is confident, it fills in the attribute name automatically (i.e., address in Figure 6). For a column that Karma cannot identify or for which it is not confident, the attribute name is entered as *select one*, as shown in Figure 6. The user can select the source modeling mode by clicking the *Attributes* tab in Figure 4 and specifying the correct attribute by entering his own or searching from the list of existing attributes in the data repository. In the example, we will assume that the following attributes are assigned (by Karma and the user) to the table: restaurant name, address, description, and number of reviews.

3.3 Data Cleaning

Frequently, the extracted data needs to be cleaned because of misspellings and/or formatting. Karma lets the user clean the data by specifying the end result of what the clean data should look like. In this case, the user wants to remove the string *Reviews* in the fourth column of Figure 6.

	description	number of r...	suggest	user defined	final
.	Upscale yet...	31 Reviews		31	
..	Intimate an...	3 Reviews			
...	Chic eleganc...	30 Reviews			
...	Authentic Ja...	66 Reviews			
..	High fashion...	62 Reviews			
...	No fuss, jus...	25 Reviews			
...	Inventive ro...	38 Reviews			
..	The MOCA o...	49 Reviews			
..	Burbank res...	29 Reviews			
...	Rustic Japa...	96 Reviews			
..	Stellar sushi...	49 Reviews			

Fig. 7. Karma in the cleaning mode. The user can specify the cleaned result and Karma will try to induce the cleaning transformation.

To enter the cleaning mode, the user selects the *Clean data* tab in Figure 4. The user can then select the column to be cleaned from the menu under the tab. Let us assume that the user selects the column *Number of reviews*. The table will be in the cleaning mode as shown in Figure 7. In the cleaning mode, three extra columns (suggested, user-defined, and final) will be populated next to the column that the user wants to clean. The *suggest* column shows suggestions for replacement by Karma if available. The replacement comes from comparing new data with existing data in the database for possible misspelling. The *user-defined* column allows the user to enter the end result, and Karma will try to deduce the cleaning transformation from the user's examples. For example, if the user enters *31* in the first row, Karma will deduce the transformation between *31 reviews* and *31*, and apply the same transformation to the rest of the data under the same column.

3.4 Data Integration

In the data integration mode, Karma will analyze attributes and data in the table to determine possible joins between the data in the table and the data in the repository. Based on the analysis, Karma can suggest existing data sources in the repository that can be linked to the new data in the table. For example, let us assume that the LA Health Rating source has been extracted and stored in the repository through a similar process, perhaps by a different user.

Based on the restaurant data in the user's table, Karma might suggest *Health Rating* as a new attribute that can be added to expand the table. If the user chooses *Health Rating* as the attribute for the new column (shown in Figure 8), Karma will generate a query to retrieve the health rating data from the repository and fill the new *Health Rating* column (shown in Figure 9).

The final result is the data table that contains restaurant data integrated with health rating information. During the process, Karma helped automate some of the subtasks to help the user build this Mashup in a short time frame. While, Karma did not offer explanations of its automated choices, the user can immediately see the results of each process and can evaluate the quality of the resulting data.

Although Karma does not focus on the data visualization problem, Karma still provides a basic Google Map display if the table contains address information. The user can display the final restaurant Mashup on a Google Map by selecting a map option from the save tab in Figure 4. The map display is shown in Figure 10. While this example is about restaurants, the structure of the problem (i.e., extracting a list from a page, cleaning and integrating with other sources) is the same in general Mashup building tasks.

In the next four sections, we will show the technical details, organized by each problem area underlying the interaction between the user and Karma. In addition, we will also point out how we use each of the three key ideas mentioned in Section 1 in our framework.

4. DATA RETRIEVAL

A large body of research considers data retrieval as a problem of extracting data from a Web page. However, in practice, data retrieval is composed of two subproblems: data extraction and page navigation. The problem of data extraction focuses on figuring out how to extract data out of HTML from a specific page or set of

restaurant ...	address	description	number of r...	health rating
Japon Bistro	927 E Color..	Upscale yet...	31	
Sushi Dokor..	9777 S Sant..	Intimate an...	3	
Hokusai	8400 Wilshir...	Chic eleganc...	30	
Sushi Sasab..	12400 Wilshi...	Authentic Ja...	66	
Sushi Roku	8445 W 3rd...	High fashion...	62	
Hide Sushi	2040 Sawtel...	No fuss, jus...	25	
Fat Fish	616 N Rober...	Inventive ro...	38	
Sushi Katsu-ya	11680 Vent...	The MOCA o...	49	
Gindi Thai /..	4017 W Riv...	Burbank res...	29	
Katana	8439 W Sun...	Rustic Japa...	96	
Echigo	11217 Sant...	Stellar sushi...	49	

Fig. 8. The user selects the attribute *health rating* from the list suggested by Karma

restaurant ...	address	description	number of r...	health rating
Japon Bistro	927 E Color..	Upscale yet...	31	91
Sushi Dokor..	9777 S Sant..	Intimate an...	3	92
Hokusai	8400 Wilshir...	Chic eleganc...	30	90
Sushi Sasab..	12400 Wilshi...	Authentic Ja...	66	97
Sushi Roku	8445 W 3rd...	High fashion...	62	93
Hide Sushi	2040 Sawtel...	No fuss, jus...	25	96
Fat Fish	616 N Rober...	Inventive ro...	38	96
Sushi Katsu-ya	11680 Vent...	The MOCA o...	49	91
Gindi Thai /..	4017 W Riv...	Burbank res...	29	94
Katana	8439 W Sun...	Rustic Japa...	96	93
Echigo	11217 Sant...	Stellar sushi...	49	91

Fig. 9. Karma automatically computes query and fills blank cells with values

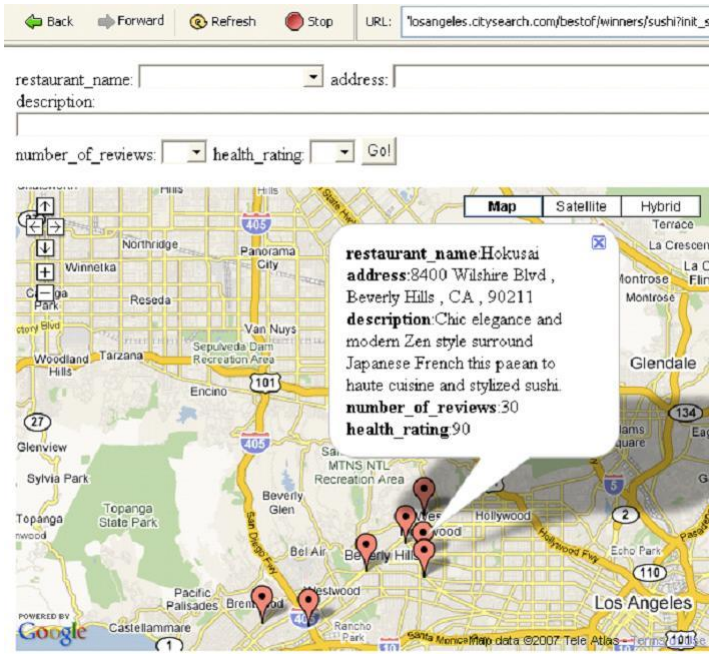

Fig. 10. A map display of the Mashup

similar pages. Page navigation deals with cases where the data we want to extract requires us to go through multiple pages to get to it. Page navigation is rarely addressed in research because it is considered more or less an engineering issue from the information retrieval point of view. However, from the Mashup building perspective, if we want to make building Mashups easy, we need an approach to do page navigation easily and effectively. In this section, we first show how Karma extracts data from a Web page. Then, we describe how Karma handles the issue of page navigation. Finally, we explain how Karma lets users reinvoke all extractions steps on a new similarly structured page by exploiting the table structure.

4.1 Extracting Data using DOM

In Karma, we use a Document Object Model (DOM) tree as the basic structure for the extraction process. The DOM tree is constructed based on the organization of HTML tags in the Web page. Figure 11 shows the simplified DOM tree of the best sushi restaurant page from our scenario.

DOM trees are used to identify lists. For example, when the user drags the value *Japon Bistro* into the table (Figure 5), we can (a) identify an XPath from the root to that value, and (b) compute parallel Xpaths in different branches to extract other nodes that store restaurant names. An XPath is an expression language that is used to access and manipulate information in XML documents. For example, the XPath for *Japon Bistro* (i.e., /tbody/tr[1]/td[2]/a) specifies the following path: **tbody**, the first **tr** tag, the second **td** tag, and retrieve all the **a** tag nodes. To find parallel paths, we can generalize the path by discarding the node number and opting for

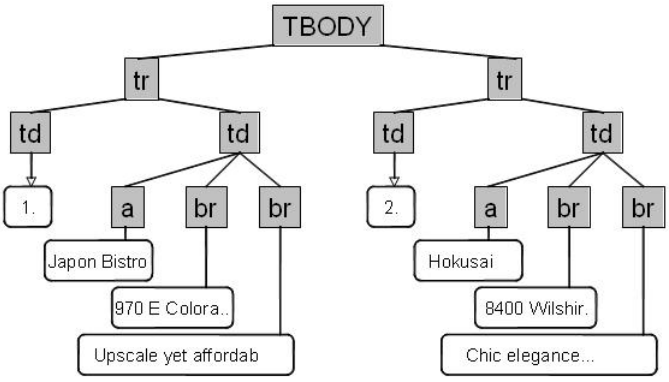

Fig. 11. A simplified DOM tree that represents the restaurant page in the motivating example. The shaded nodes represent the HTML tags, while the unshaded nodes represent the data embedded within those tags.

wild cards. For example /tbody/tr*/td*/a will return any node where its XPath matches such a structure without taking the branch number into account. In this case, the relaxation using wild cards will return two nodes: /tbody/tr[1]/td[2]/a and /tbody/tr[2]/td[2]/a.

After extracting the first column of data, Karma handles extraction in other columns based on the position of the nodes in the first column. For example, suppose the user wants to extract the address of the restaurants. To do so, he/she drags the text *970 E Colora ...* from the web page to an empty cell next to the cell containing the restaurant name *Japon Bistro*. We call the first extracted data (the restaurant name) item the marker, as it serves as a reference point for interpreting the second example (the restaurant address), which we call the value. Karma uses the DOM tree to generalize the examples, and to compose a general procedure for extracting the values to fill in the cells next to the previously extracted marker cells. In our example, this procedure extracts from the DOM tree the addresses of all the restaurants.

The basic idea is to identify the relationship between the marker and value nodes in the DOM tree. Using this relationship, Karma can compute the value nodes corresponding to a collection of previously extracted marker nodes.

Let $p_1^0 = (p_{1,1}^0, \ldots, p_{1,m}^0)$ be the path of nodes from the root of the DOM tree to the node containing the example marker node, and let $p_2^0 = (p_{2,1}^0, \ldots, p_{2,n}^0)$ be the path of nodes from the root to the example value node (see table I, rows 1 and 2). We use the superscript 0 to designate the examples the user provides. In our scenario, p_1^0 designates the path to the *Japon Bistro* restaurant, and p_2^0 designates the path to its address. The extraction algorithm produces a set of pairs (p_1^i, p_2^i) that specify how to fill in the values of the other rows by mapping each of the extracted marker nodes to a corresponding value node. In our example, the result consists of pairs of paths that identify a restaurant name and its corresponding address.

The algorithm first computes the common path c between p_1^0 and p_2^0 (table I, row

3) defined as

$$c = (c_1, \ldots, c_k)$$

where $c_i = p^0_{1,i}$, and k is the largest value for which $p^0_{1,i} = p^0_{2,i}$ for $i = 1, \ldots, k$

The next step is to generalize the common path to identify similar structures in the DOM tree. This is done by substituting the order number of the nodes by wildcards (e.g., substitute td[2] with td* as shown in table I, row 4). We denote the generalized common path as $c^* = (c^*_1, \ldots, c^*_k)$.

The output of the algorithm consists of pairs of paths (p^i_1, p^i_2):

$$p^i_1 = (c^i_1, \ldots, c^i_k, p^0_{1,k+1}, \ldots, p^0_{1,m})$$
$$p^i_2 = (c^i_1, \ldots, c^i_k, p^0_{2,k+1}, \ldots, p^0_{2,n})$$

where $c^i_1, \ldots, c^i_k$ are the DOM tree paths that match $(c^*_1, \ldots, c^*_k, p^0_{1,k+1}, \ldots, p^0_{1,m})$.

In our example DOM tree, the algorithm extracts the address of the *Hokusai* restaurant (table I, row 5).

Table I. Formalization of the restaurant example.

1.	$p^0_1 =$	(tbody, tr[1], td[2], a)	marker path identifying *Japon Bistro*
2.	$p^0_2 =$	(tbody, tr[1], td[2], br[1])	value path identifying *970 E Colora ...*
3.	$c =$	(tbody, tr[1], td[2])	common path
4.	$c^* =$	(tbody, tr*, td*)	generalized path
5.	$p^1_1 =$	(tbody, tr[2], td[2], a)	output: marker path identifying *Hokusai*
	$p^1_2 =$	(tbody, tr[2], td[2], br[1])	output: value path identifying *Hokusai*'s address

It is crucial to append the path suffixes that identify the marker and value nodes relative to the common path. For example, unless the marker suffix is appended to the common path, the algorithm would match spurious nodes such as the leftmost td node in Figure 11; the correct td node is the second one, which is identified by its child a node. Similarly, the suffix on the value path helps address ambiguities caused by nested lists (list of br nodes nested in a list of td nodes). In this case, the suffix br[1] correctly identifies the restaurant address, which is in the first br child.

4.2 Page Navigation

In our example each restaurant has a link to its detailed page, which contains more information about the restaurant (e.g., number of reviews). This kind of web page structure is quite common in the deep web, where pages are generated from databases based on HTML form input values. The first page contains a list of results, where each result has a link to its detailed page. Note that the result page and the detailed page usually have different page structures. Examples of this structure is often seen in product search result pages (e.g., Amazon and eBay).

Often times, we want to extract data from both the result page and each detailed product page. However, the extraction process cannot occur separately because detailed pages are reached through the result page. One way to do the extraction is to extract the result page first, then navigate from the result page to the detail

pages, and finally join them together. For naive users, understanding and specifying database joins can be confusing. Karma supports this process in a natural way by inducing join operations from user interactions, so the user does not need to explicitly specify a database join operation.

In the example, when the user extracts *Japon Bistro*, Karma can already induce that the first column is a list. Next, when the user navigates to the *Japon Bistro* detail page and drags the number of reviews from the detailed page into the first row of the table (Figure 12), the user indirectly specifies: (a) that a particular detail page is linked to the data of the first three columns in the first row, (b) the extraction rule (XPath) for this new data element, and (c) where the new data element from a new page should be in the table with respect to the data from the first page.

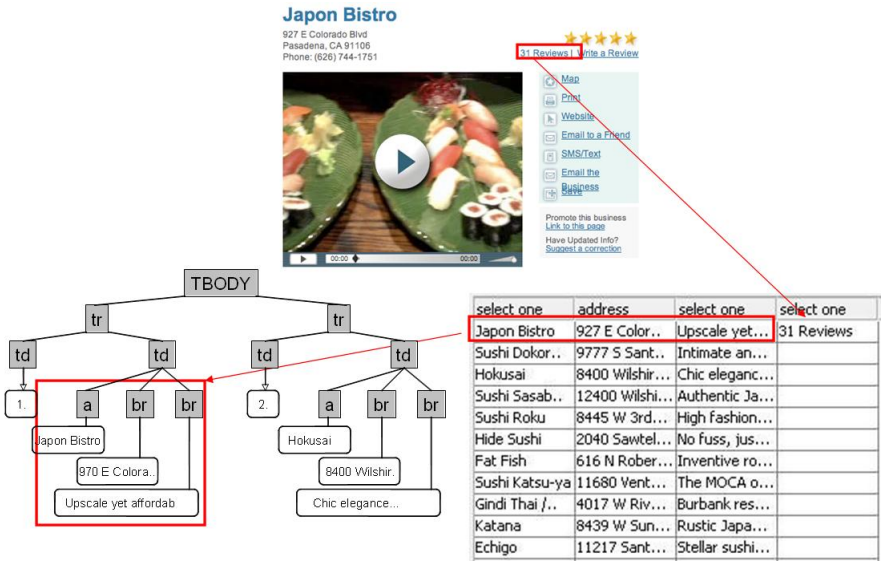

Fig. 12. When the user navigates to the *Japon Bistro* detail page and extracts the number of reviews into the table, Karma deduces the relationship between the new data element and existing data elements in the same row to help extract data for other rows.

The general problem of extracting information from detail pages can be modeled as a three step process. First, the user has already extracted a list of items (e.g., the restaurant names) from a results page. Using the same notation as in the previous section, we call this list $p_1^0, p_1^1, \ldots, p_1^n$. The subscript 1 indicates that this is a list of markers; p_1^0 represents the path to the example from which the user will navigate to the detail page; the p_1^i for $i > 0$ represent the paths to the other elements in the list for which Karma needs to extract information from the corresponding detail pages (e.g., all the other restaurants in Figure 12).

Second, the user clicks on a link in the results page to navigate to the detail page. Let p_2^0 represent the path in the DOM tree from the root to the element containing the link. This configuration is identical to the marker/value configuration described

on section 4.1, so the same procedure can be used to compute $p_2^1, \ldots, p_2^n$, the paths to each of the link elements. In our example, the p_2^i are the DOM paths to the links to the detail restaurant pages of all the restaurants. Note that Karma does not insert these links in the table, but uses them internally to retrieve all the relevant detail pages.

Third, once the detail page loads, the user drags a new piece of information, into the Karma workspace and drops it on an empty cell, in the same row as the marker. In our example, the user dragged the number of reviews element from the details page and dropped it next to the restaurant name in the Karma workspace. Let $d = d_1, \ldots, d_m$ be the DOM path from the root of the *details page* to the element the user extracted. We assume that the desired information in the details page is in the same location in all the detail pages, so the same path d can be used to extract the detail information from each of the details pages. This is a reasonable assumption given that these detailed pages are usually generated from databases using the same template format.

In summary, the procedure has three steps:

(1) Compute the DOM paths $(p_2^1, \ldots, p_2^n)$ of the links to the detail pages from the paths of the markers $(p_1^1, \ldots, p_1^n)$, using the algorithm described in section 4.1.
(2) Retrieve the detail pages using the links identified by the p_2^i paths.
(3) In the retrieved detailed pages, use the path d to retrieve the desired information from the detailed page, and store this information in the corresponding cells in the Karma workspace.

Using this procedure, Karma allows users to navigate deep into multiple levels of detail pages on the same subnet and extract data while retaining the whole view of the overall extracted data in one table.

4.3 Reinvoking a Wrapper

After extracting *restaurant name, address, description, and number of reviews*, Karma can reuse existing extraction rules to extract pages with the same structure. For example, if the user navigates to a new page with similar structure (e.g., Best Thai Food from the same Web site) as shown in Figure 13, she can drag and drop another restaurant (i.e., *Palms Thai*) into a new empty row. When the user drops a data element from a new page into an empty row, Karma checks the XPath of this new data element. If (a) the XPath is similar to previous XPaths from the same column and (b) the URL of the new XPath is different, then Karma proceeds to repeat all the extraction steps including page navigation extraction from just one example provided by the user. The end result is effectively a database union between the best sushi restaurant data and the best Thai restaurant data with just a single drag and drop.

The data retrieval approach shown in this section follows the Karma framework by primarily focusing on the data, not the operations. Instead of writing complex extraction and navigation rules, the user retrieves data by moving data from Web pages in the embedded browser to the table. The user is able to extract data from multiple levels of Web pages and to combine them by indirectly giving positional context. The data retrieval process is based on enhancing the DOM and exploiting positional context in the table.

Fig. 13. The user can extract pages with similar structures by dragging a data element into an empty row

Each table has two obvious but often neglected constraints, which define the positional context technique. The first constraint is the horizontal constraint. When a user puts a new data element in a new column on a particular row, the user indirectly associates the new element with other elements in the same row. It means the new element has different semantics because it is in a different column, but all elements in that row belong to the same tuple. The second constraint is the vertical constraint. When the user puts a new data element in a particular column, the user indirectly associates the new element with other elements in the same column. It means the new element has the same semantic as other elements in that column. The horizontal constraint helps Karma identify markers in order to connect and combine data from multiple page levels, while the vertical constraint signals Karma that the user wants to invoke the rules and steps learned previously on a new, but similar source. The idea of positional context will be revisited in section 7.

Karma can also extract data from Web forms. Karma captures the HTTP form parameters and values as the user fills out a Web form to get to the result page. When the user starts dragging the data from the result page onto the table, Karma also populates form parameters (as column attributes) and values alongside the data extracted. The user could modify or enter new form values in the table and Karma will query a form, extract the new data using the same XPath rule, and show the updated result in the table. Basically, Karma relies on users to supply input for each instance of form extraction. A more advanced technique in form extraction can be found in HiWe [Raghavan and Garcia-Molina 2001], where user input is used to further build a form's internal representation model to extract more data.

The data extraction techniques used by Karma will work on well-structured websites containing lists and tables. While our extraction technique will not work on ill-formed web pages, most deep web pages are well-structured because they are typically generated from relational databases. Besides Karma, Dapper and Yahoo's

Pipes also have similar problems with data extraction on ill-formed web pages. Even with a limited pool of extractable web pages, it is still possible to create many interesting data integration applications as shown on Dapper and Yahoo's Pipes pages.

Many of the data fields extracted from the Web are unstructured and ungrammatical. Further processing of this data can yield important information (i.e., locations, names, etc.). Systems like Phoebus [Michelson and Knoblock 2007b] and Gate [Cunningham et al. 2002] support this kind of extraction. Karma does not address this problem, but could be extended to support unstructured text extraction.

5. SOURCE MODELING

In Karma, we keep a repository of data that can be used for bootstrapping processes in source modeling, data cleaning, and data integration.

As soon as the user puts the data into the table in the data retrieval step, the new data elements are pipelined into the source modeling step. Karma analyzes those new entries by comparing them with data elements that exist in databases to generate a possible candidate set for each column's attribute.

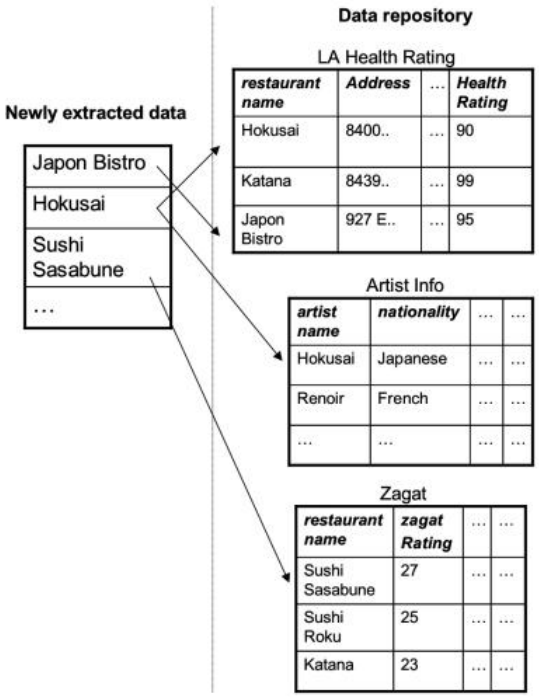

Fig. 14. A view of the overlap between newly extracted data and existing data in the repository

Figure 14 shows the mapping according to the constraint formulated for the first data column that contains restaurant names in Figure 6. After the user extracts the

first value and Karma fills the rest of the column, Karma then uses all the values in that column as a starting set to determine possible attribute mappings. For each value in the starting set, Karma queries the repository to determine whether that value exists in any table. If it exists, Karma extracts the attribute to which a value corresponds. For example, there exist *Sushi Sasabune* and *Japon Bistro* under the attribute *restaurant name*. However, *Hokusai* can be associated with multiple attributes: *restaurant name, artist name*.

In order to formally specify how Karma computes this candidate set, we will first introduce a formalization of sources. Let S be the set of all sources, A the set of all attributes (column headings) in all sources, V the set of all values that can appear in any source, and N the set of natural numbers, which identify the rows. The contents of sources are defined using the functions shown in Table II. We use the shorthand $\nu(s,a) = \bigcup_{i \in N} \nu(s,a,i)$ to represent all the values defined for attribute a in source s (all the values in a column).

Table II. Formal definition of sources

$\alpha:$	S	$\rightarrow$	A	
	s	$\rightarrow$	$\alpha(s)$	the attributes defined in source s
$\kappa:$	S	$\rightarrow$	2^A	
	s	$\rightarrow$	$\kappa(s)$	the attributes that define a key for source s
$\nu:$	$S \times A \times N$	$\rightarrow$	V	
	(s,a,i)	$\rightarrow$	$\nu(s,a,i)$	the value in source s identified by attribute a and row i

To compute a candidate set to suggest the attributes for a column with no attribute defined, we define a value overlap function ω as follows:

$$\begin{array}{lll} \omega: & 2^V \times S & \rightarrow & 2^{A \times N} \\ & (X,s) & \rightarrow & \omega(X,s) = \{(a,n) : n = |\nu(s,a) \cap X|, \nu(s,a) \cap X \neq \emptyset\} \end{array}$$

The input to ω is a set of values X and a source s. The output is a set of pairs. The first element in the pair is an attribute in source s whose range overlaps with X; the second element in the pair records the number of values in the range of the attribute that overlap with X.

To produce the menu of suggested attributes, Karma computes $\bigcup_{s \in S} \omega(X,s)$, where X is the collection of values that appear in the column. It replaces sets of pairs (a_i, n_i) that have the same a_i with $(a_i, \sum_i n_i)$, thus removing duplicate appearances of an attribute, and recording the total amount of overlap across all sources. Then it sorts the remaining set by the amount of overlap, and shows the attributes and their overlap in a menu.

In the example shown in Figure 14, X is the set of newly extracted restaurant names. Karma computes ω for each source, producing a singleton set for each of the sources shown, given that in this example, the elements in X only overlap one attribute in each source. The resulting menu will contain two elements, corresponding to **restaurant name** and **artist name**. The overlap for **restaurant name** is

3 given that the overlap with source **LA Health Rating** is 2 and the overlap for **Zagat** is only 1; the overlap for **artist name** is 1.

In the case where all new values can be associated to only one attribute, Karma sets the attribute name of that particular column in the user table automatically. When there is an ambiguity, Karma sets the attribute name for that column to *select one*. Then, the user can select the attribute from a ranked candidate list.

Our method assumes that there is an overlap between newly extracted data and existing data in the repository. If there is no overlap, then Karma will output "*select one*" as the attribute name for that column (Figure 6), and let the user select from the list of existing attribute names from the repository, or allow him to specify the attribute name himself. The seed data in our repository is obtained from a) importing data tables from existing databases, and b) storing and reusing data from previous Mashup building tasks; the more users create Mashups using Karma, the better Karma performs in the source modeling step.

6. DATA CLEANING

Data cleaning is tedious and time consuming because it involves (a) detecting discrepancies in the data, (b) figuring out the transformation to fix them, and (c) finally applying the transformation to the dataset [Raman and Hellerstein 2001]. In Karma, we support two kinds of data cleaning: misspelling detection and transformation by example.

6.1 Misspelling Detection

When the user enters the data cleaning mode, if Karma detects any possible misspelling in the new data, Karma will suggest possible replacements, if any, in the *suggest* column in Figure 7. For example, let us assume that there is a possible misspelling for data in the *restaurant name* column. Figure 15 shows intuitively how Karma figures out the possible replacement.

As soon as the source modeling problem is solved (e.g., the attribute is *restaurant name*), Karma locates data in the database in any table (e.g., **LA Health Rating** and **Zagat**) that is associated with the attribute *restaurant name*. Then, Karma executes pairwise comparisons between newly extracted data and existing data in the database to determine if there are any close matches. The algorithm to suggest spelling corrections is defined by the following function σ that takes as arguments an arbitrary value v and an attribute name a, and returns a set of suggested corrections paired with their similarity score.

$$\begin{array}{lccl} \sigma: & V \times A & \rightarrow & 2^{V \times R} \\ & (v, a) & \rightarrow & \sigma(v,a) = \{(x,y) : x \in \bigcup_{s \in S} \nu(s,a), f(v,x) > 0.6, y = f(v,x)\} \end{array}$$

Function f is a string similarity metric. In Karma, we use the unsmoothed Jacard similarity [Cohen et al. 2003] for its fast computation and performance. The threshold is chosen empirically based on experimental results in [Michelson and Knoblock 2007a]. The function σ considers as candidates values in any source that appear under the specified attribute.

If $v \in \sigma(v, a)$, i.e., the value exists in a the database under the desired attribute, then no suggestions are given. Otherwise, Karma colors the value red in the table.

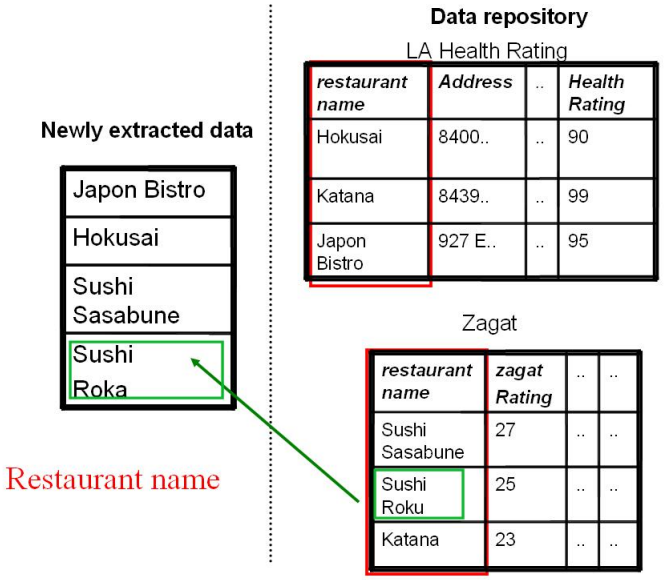

Fig. 15. The diagram showing how Karma uses the database to help identify possible misspelling replacements

When the user chooses to clean the data, Karma presents the user with suggestions for replacement for each red element. If $|\sigma(v, a)| = 1$, the single suggestion is shown to the user. If $|\sigma(v, a)| > 1$, the user can click on the suggestion cell to see the drop down list of available suggestions.

The performance of this approach is tied to how well the source modeling problem is addressed. If the user selects an incorrect attribute type for a newly extracted data column, then Karma will not be able to use the correct set of data to compute the suggestion set for replacement. However, even if the user selects the wrong attribute, Karma does no worse than other Mashup tools as they do not provide any support to detect misspellings in the extracted data.

6.2 Transformation by Example

Transformation by example lets users specify the format of the cleaned data by example. Karma then uses the example to induce the cleaning transformation rules. Figure 16 shows what happens when the user types in an cleaning example in Figure 7. The original data element "*31 reviews*" is sent to a set of predefined transformations to generate the output. If the output generated by a transformation is equal to the user defined input (i.e., *31*), then that transformation rule is selected. The selected transformation rule is then used to apply to the rest of the data in the original column (*number of reviews*).

By allowing the user to specify the end result, the user does not need to know how to specify the transformation rule, which can be challenging. For example, the regex widget in Figure 2 specifies the regular expression (i.e., replace (\w*\s)(\w*) with $2 $1) that transform "*Lastname Firstname*" to "*Firstname Lastname*". While our approach does not require a user to understand programming, it does have three

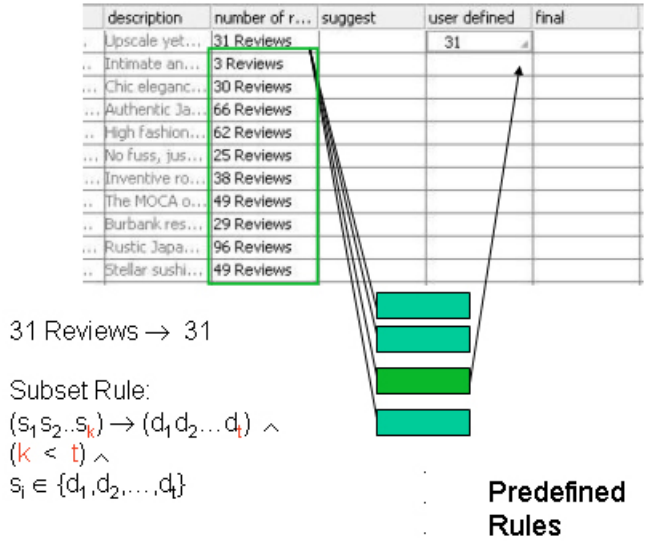

Fig. 16. Karma induces a transformation rule by instantiating predefined transformations in its library

limitations:

(1) *There may be no predefined transformation match*: The user would need to type in the cleaned values manually. Table III lists an initial preliminary set of transformations supported by Karma. Each transformation is implemented as a JAVA class and can be created and added as a modular unit to cover more sophisticated cleaning operations.

(2) *More than one rule could be applied*: Currently, Karma selects the first rule that matches. In the future work, the user can use more than one example to help Karma determine the correct rule.

(3) *More than one transformation may be needed*: There may be some cases where a data column contains data in a mixed format. For example, the first row might contain the data "*12/03/2008*", while the second row might contain the data "*March 15, 2006*". To clean a column of mixed format data, more than one transformation rule is required. However, the current version of Karma, while it can be extended to support multiple transformation rules, only supports one transformation rule for each column.

Our two data cleaning approaches fit the Karma framework by using all three key ideas mentioned in section 1. First, the user focuses on the data. By specifying the cleaning result that the user wants to see, she indirectly induces the cleaning transformation function. Second, Karma leverages existing databases to detect misspellings, so the user does not have to spend time locating these errors. Third, as soon as the source modeling problem is solved (e.g., attribute is specified), Karma uses the information about that attribute to help determine which existing data set should be used to detect misspellings.

Table III. Predefined transformations supported by Karma

Name	Function
Symbol Substitution	Allow a single symbol replacement i.e., 12-30-2008 → 12/30/2008
Name Reverse	Lastname, Firstname i.e., Obama, Barack → Barack Obama
Name Reverse2	Firstname, Lastname i.e., Barack Obama → Obama, Barack
All Uppercase	Change every character to uppercase i.e., Hillary → HILLARY
All Lowercase	Change every character to lowercase i.e., Hillary → hillary
Word Capital	Capitalize the first character of each token i.e., barack obama → Barack Obama
Substitution	Allow a single token replacement i.e., 3 Ames St. → 3 Ames Street
SubstringEqLength	Similar to Java substring i.e., 28 reviews → 28

7. DATA INTEGRATION

Our goal in data integration is to find an easy way to combine a new data source (that we extract, model, and clean) with existing data sources. The general problems are (a) locating the related sources from the repository that can be combined with a new source, and (b) figuring out the query to combine the new source and existing valid sources. Karma solves these problems by utilizing table constraints with programming by demonstration. The user fills an empty cell in the table by picking values or attributes from a suggestion list, provided by Karma. Once the user picks a value, Karma calculates the constraint that narrows down the set of sources and data and uses that constraint to fill in the other cells.

7.1 Intuition

To demonstrate how Karma handles data integration, let us assume that the user first extracts the list of restaurant names and addresses and then invokes the data integration mode. We will assume that our data repository only contains the three data sources from Figure 14.

Figure 17a shows a table with the newly extracted data, where the empty cells that can be expanded are labeled with numbers (1-7). Based on the existing data repository, there is a limited set of values that can fill each cell. For example, the value set that Karma will suggest to the user for cell 1 would be {*Katana*, *Sushi Roku*}. The reason is that to preserve the integrity of this column, each suggestion for cell 1 must be associated with the attribute *Restaurant name*; the values under the same column must be associated with the same attribute name. Currently, there are only two sources with column *Restaurant name*, so Karma formulates the query based on this constraint to generate the suggestion list. In Figure 17b, we assume that the user picks *Katana* to fill cell 1. To fill cell 7, we need to ensure that the values Karma suggests (a) must come from a row in the source that has the value *Katana* associated with *Restaurant Name*, and (b) must be values under the attribute *Address*. These two constraints narrow down possible choices to only

one value. If only one choice exists, Karma will fill it in automatically (as shown in Figure 17c).

To fill cell 6 in Figure 17b, we need to find one or more sources that (a) can be linked with the data cell *Katana*, and (b) contain new attributes (i.e., *health rating* and *zagat rating*) not already in the table. As a result, the possible values that can be suggested in cell 6 would be {99, 23}. The reason is that since Katana is a restaurant, there are only two valid rows that have Katana as a restaurant in the repository as shown in Figure 18 (row 2 from the LA Health Rating source and row 3 from the Zagat source).

Likewise, cell 2 is also limited to two attributes (as shown by the shaded examples in the attribute rows in Figure 18) since these attributes come from sources that have *Restaurant name* and *Address* in their list of attributes. If the user picks cell 2 to be *Health rating* in Figure 17c, Karma can narrow down the choices through constraints and automatically fill the rest of the column (cell 3,4,5,6) with the health rating value for each restaurant.

By choosing to fill an empty cell from values suggested by Karma, the user (a) does not need to search for data sources to integrate, (b) picks values that are guaranteed to exist in the repository, yielding a query that will return results, (c) indirectly formulates a query through Karma, so the user does not need to know about complicated database operations, and (d) narrows possible choices in other empty cells.

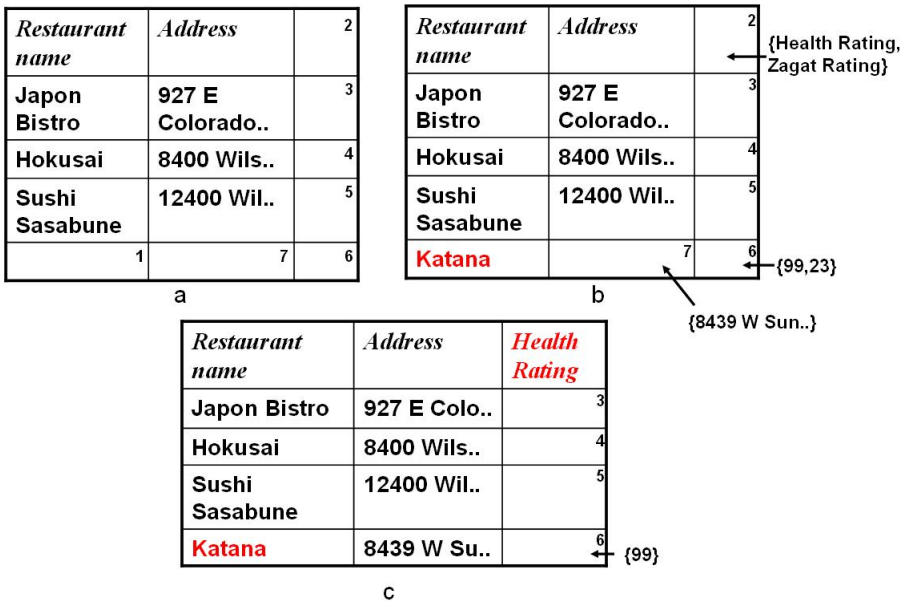

Fig. 17. Illustration of how the user can integrate new data with existing data through examples. When the user selects more examples, the table becomes more constrained. The value 1-7 designate empty cells.

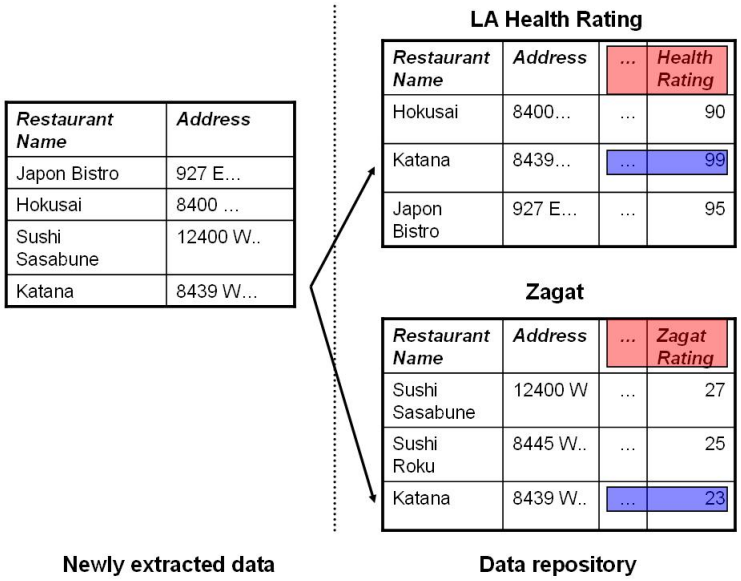

Fig. 18. Selecting Katana in cell 1 limits the choices in other cells, such as cell 6 and cell 2, through the horizontal constraint

7.2 Formalization

We formalize the constraints described in the previous subsection, using the formal definition of sources given in Table II. First, we introduce the concept of *reachable attributes*. In traditional databases, we can link different tables together using the join operation. Depending on the join condition, it is possible to create a successive chain of tables. Figure 19 shows an example of how tables in a database can be linked together. Given an employee ID, we could retrieve the following attributes using join conditions through foreign keys: SSN, name, address, phone number, latitude, and longitude. We define a *reachable* attribute as an attribute that can be reached from a particular data source through a sequence of joins via foreign keys (i.e., longitude is reachable from S1).

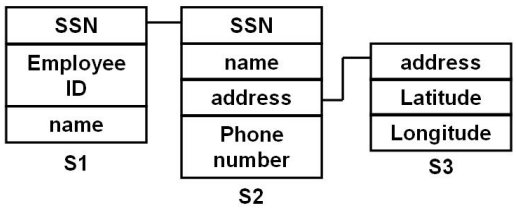

Fig. 19. Join paths through foreign keys in a traditional database

If we have a well-defined database like the one shown in Figure 19, join conditions between tables can be composed over foreign keys. Joining two tables using non-

foreign key attributes (i.e., name) is possible, but the result generated may not make sense. For example, there might be a record with name *John Smith* in S1 and S2 who are completely different people with different SSNs.

We use a similar concept for Web sources. In many Web sites, to retrieve the information, we need to fill out a Web form. This Web form requires some key input to produce one or more output. Specifically, the output results make sense only in the context of specific key input values; there is a functional relationship between the input and the output in a Web form. Karma exploits this relationship and uses the Web form input requirement as a special primary key constraint. This primary key constraint allows Karma to link two tables over attributes, so that the join result makes sense. For example, the primary key constraint for the **LA health rating** source might be: *restaurant name* and *address*. The constraint means is that if we want to retrieve the rest of the attributes (e.g., *health rating, inspection date*), the join condition must be over *restaurant name*, and *address*.

Suppose we have two sources $s, t \in S$. Source t is reachable in one step from source s if $\kappa(t) \subseteq \alpha(s)$, i.e., if s and t have a key/foreign key relationship. We define the function ρ_n to represent the set of sources that are reachable from a source s as follows:

$$\begin{array}{lccll} \rho_n: & S & \rightarrow & 2^{S^1 \cup \ldots \cup S^n} & \text{set of sequences of sources of length up to n} \\ & s & \rightarrow & \rho_n(s) & \text{the sources reachable from source } s \end{array}$$

The function ρ_n records the complete sequence of sources used to reach every reachable source. We define two sequences of sources $(s_1, \ldots, s_n)$ and $(t_1, \ldots, t_m)$ to be compatible if they have the same length and for each $i : \kappa(s_i) = \kappa(t_i)$, i.e., the keys are the same. This is important because we do not want to put in the same column values from incompatible sources. For example, consider the *rating* attribute used in the example in Figure 20. The rating is a number between 0 and 100, and both restaurants and artists have ratings, but we do not want to mix ratings of restaurants and artists in the same column. A column can only contain values from compatible sources according to our definition. In our example, it is valid to include in the same column ratings from the two restaurant sources, as they may have complementary data. Suppose Karma had access to a source that provides information about the restaurants where artists may be seen. In this case, the artist *rating* would be reachable from the restaurant source, but using the compatibility definition, Karma would prevent including the two types of rating in the same column: the two *rating* attributes are incompatible because one uses the new restaurant-to-artist source that the other does not use.

The reachable attributes from a source s are the attributes $\alpha(s_i)$ of the reachable sources s_i. Similarly to the definition of reachable sources, it is important to record the full path of sources so that Karma can reason about the compatibility of values to be included in a column. Even though our examples suggest that the attributes in the columns are simply their names, internally, Karma stores the full paths so that it can apply the compatibility tests.

Karma can compute the n-step reachable attributes for any n, but we found that for $n > 2$ the results become less intuitive and computation of the suggested values for cells becomes a bottleneck.

Our algorithms for computing suggested values for a cell fall into four cases as

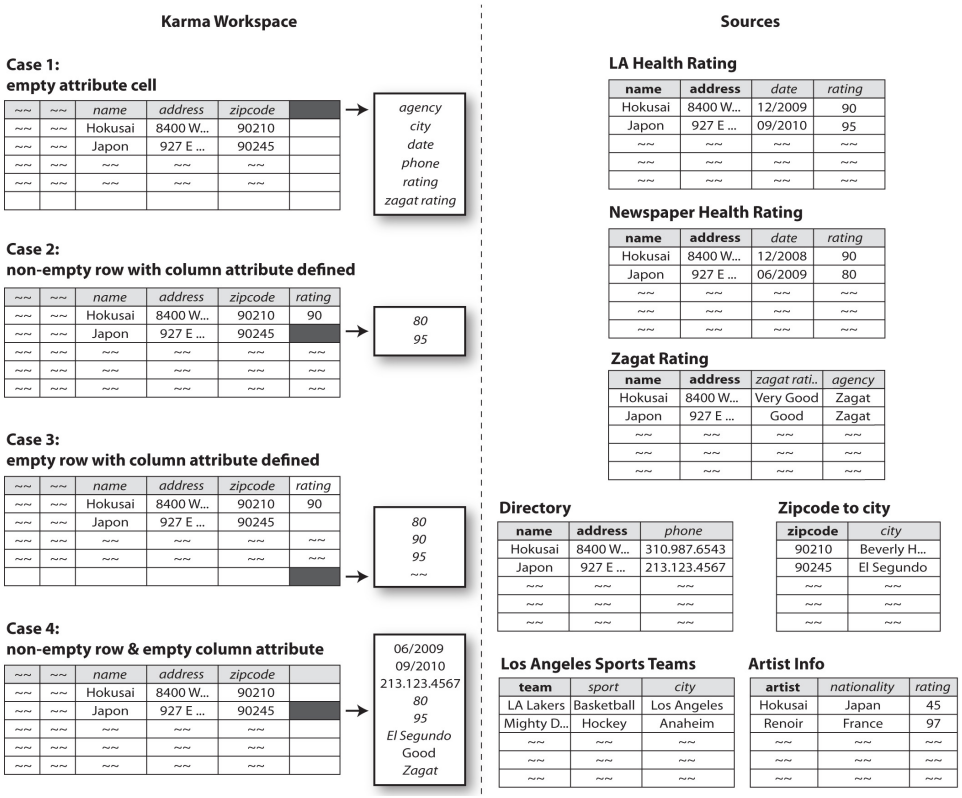

Fig. 20. Examples of computation of suggested values of cells

illustrated in Figure 20. The cell highlighted in dark grey represents the cell for which users request suggestions. The menus next to the Karma tables show the suggested values drawn from the sources shown in the figure.

Case 1: empty attribute cell.. In this case, the user selected the column heading for a column where no attribute has been specified ($a = \emptyset$).

If all the rows in that column are empty ($\forall i : v(s, a, i) = \emptyset$), then the suggested attributes are all the attributes of the reachable sources $\rho_n(s)$. If there are cells in that column with values, the set of attributes is trimmed to include only those with compatible values. Formally, if row i contains value x, then the set of attributes are trimmed to include only those that satisfy $\nu(t_{m_j}, a_j, i) = x$, where each suggested attribute a_j is defined by a sequence of sources $(t_1, \ldots, t_{m_j})$.

The menus only show the attribute names even if there are multiple source paths that lead to attributes with the same name. When users select such an ambiguously defined attribute, Karma records the set of ambiguous definitions. Karma will automatically trim this set when users provide examples in the value cells below the attribute.

In the example in Figure 20, the menu includes all the attributes reachable via

name and **address** (shown in bold to indicate they are the keys). In this example, there are two reachable definitions of the *rating* attribute, one via **Newspaper Health Rating**, and the other via **LA Health Rating**. However, the *rating* attribute only appears once in the menu.

The *rating* attribute from **Artist Info** is not included in the ambiguous definitions because it is not reachable. If it was reachable, it would be included, and either it or the restaurant rating attributes would get trimmed once users fill in values for the cells

As soon as the user chooses a value from the menu, Karma will attempt to automatically populate the value cells. It will do so for all cells where the set of suggested values includes only one value. The next case specifies how these suggestions are computed.

Case 2: non-empty row with column attribute defined. In this case, the user selected an empty cell in a column where the attribute has already been defined. Note that the attribute could be ambiguous, as described in case 1. Furthermore, there are values in other cells in the row. Formally, $a \neq \emptyset, \exists x \in \alpha(s) \colon \nu(s, x, i) \neq \emptyset$, where i designates a row in the table.

The computation of the suggested values simply involves doing a sequence of joins along the sequence of sources that lead to the attribute. If the attribute definition is ambiguous, the result is the union of the set of values computed from each of the ambiguous definitions. Suppose the attribute is defined by a sequence of sources $(t_1, \ldots, t_m)$. In the first step, Karma performs a join between the Karma table s and t_1 using the keys $\kappa(t_1)$ with the constraint that for each $k \in \kappa(t_1)$: $\nu(t_1, k, j) = \nu(s, k, i)$ for some row j in t_1. If $\nu(s, k, i) = \emptyset$ then no join is attempted and no values are suggested. This join produces a set of values V_1. The procedure is repeated for the rest of the t_h sources in the sequence. The constraint is generalized so that for each $k \in \kappa(t_h) : \nu(t_h, k, j) = x$ for some row j in t_1 and some $x \in V_{h-1}$.

Karma automatically applies the case 2 computation on every empty cell where the column attribute or any value in the same row are defined. If the set of suggested values contains a single value then Karma automatically fills in the cell. In the example in Figure 20 we show a case where Karma had automatically entered the values for all the *rating* rows except the one where the user clicks. The menu shows two choices because the values from the different health rating sources are different. Had they been the same, Karma would have automatically filled in the cell.

When the user selects a value, Karma uses the path that led to it to trim the possibly ambiguous set of definitions of the column attribute to include only those that are compatible with the sequence of sources that led to the value. This is how eventually, Karma would automatically disambiguate the incompatible definitions of *rating*. In some cases the value provided may be consistent with multiple incompatible definitions of the attribute. In that case the ambiguous set of attribute definitions is not trimmed.

The automatic fill-in of cells with unambiguous values is an important part of the user interaction. Users are directed to focus on the empty cells where a user decision is required, either to resolve a discrepancy between compatible sources, or to enter a value when no suggestions are available.

Case 3: empty row with column attribute defined.. In this case the user selected an empty cell in the location defined by attribute a and row i in source s. The other cells in the row are still empty, but the user has already assigned an attribute to the column. Formally, $a \neq \emptyset, \forall x \in \alpha(s) \colon \nu(s, x, i) = \emptyset$.

This case arises when users want to append new rows corresponding to new sources containing relevant information. For example, suppose that in our scenario users had not yet extracted data for the **Newspaper Health Rating**, and had already integrated the **LA Health Rating** with the other sources shown in Figure 20. After extracting the data for **Newspaper Health Rating**, users may want to integrate data from this new source.

This case is similar to case 2 except that all the cells in the row are empty, so the joins that would be performed in case 2 would fail, producing no suggestions. Producing no suggestions is undesirable, so we use a different algorithm.

If the attribute is defined unambiguously by a sequence of sources $(t_1, \ldots, t_m)$, then the set of suggested values is $\nu(t_m, a)$, i.e., all the values that the attribute can take in the last source of the chain. If the attribute definition is ambiguous, the result is the union of $\nu(t_{m_i}, a)$ for each of the source sequences in the definition of the attribute. In Figure 20, these are all the values for the *rating* attribute in sources **Newspaper Health Rating** and **LA Health Rating**. The suggestions do not include the values of the *rating* attribute in source **Artist Info** because this source is not reachable from the Karma table that the user is working on.

When users provide examples in columns that belong to the key of an integrated source, then Karma uses the algorithms defined for case 2 to propagate values to other cells in the same row. However, when users provide an example in other columns, no propagation is done, as it is computationally expensive to infer the keys that produce the given example via a sequence of joins.

Case 4: non-empty row and empty column attribute.. In this case, the user wants to enter examples of values that he/she wants included in the table, but perhaps has not been able to identify the column attribute yet, and is expecting that Karma will identify the column attribute from the examples. Formally, $a = \emptyset, \exists x \in \alpha(s) \colon \nu(s, x, i) \neq \emptyset$.

The computation of suggested values is similar to the one used in case 2 where the attribute value is defined. The set of suggested values is the union of the suggested values produced in case 2 for all reachable attributes.

In the example in Figure 20, the menu includes a mixture of values including the inspection dates, the restaurant phone number, the health and zagat ratings, and the name of the city where the restaurant is located.

Once the user chooses a value, Karma trims the set of suggestions for the attribute to include only those attributes that could have yielded the selected value. No additional computation is needed because the value suggestions were computed from the reachable attributes in the first place, so it is only necessary to record them when the suggestion set is computed. If the set of suggested attributes is reduced to attributes with the same name, Karma automatically fills in the column heading.

Our data integration approach is based on the idea that every problem has a structure that dictates the constraints. Once we find the constraints, we use them

to limit the search space of the solution. Our work illustrates how past approaches [Zloof 1975] underutilized the information from the structure of the problem that, once exploited, can reduce the user's time and knowledge requirement to perform a task.

8. RELATED WORK

This section surveys previous work related to Mashups. We divide the related work into two parts. In the first part, we discuss existing Mashup building tools from both academia and industry. We selected and compared tools that have a target audiences similar to Karma – naive users with no programming background who might build Mashups casually. Then, we discuss related research fields including data extraction, source modeling, data cleaning, and data integration.

8.1 Mashup building tools

We divide Mashup building tools into two categories: the widget approach and other approaches. We also show how Mashup tools that aim to support naive users address each Mashup building issue.

8.1.1 *Widget Approach.* The Mashup tools in this category use a widget paradigm as their basis. This approach probably originates from a merger between visual programming language [Burnett and Baker 1994] and dataflow programming [Sutherland 1966]. In the widget paradigm, a user selects a widget, drops a widget onto a canvas, customizes the widget, and specifies how to connect that widget to other widgets, creating a connected graph.

The following systems use the widget paradigm: Yahoo's Pipes, Microsoft's Popfly, Marmite [Wong and Hong 2007], JackBe (http://www.jackbe.com), Bungee Labs (http://www.bungeelabs.com), Proto Software (http://www.protosw.com), and IBM QED Wiki (http://www.ibm.com). Yahoo's Pipes, Microsoft's Popfly, and CMU's Marmite [Wong and Hong 2007] are similar structurally in terms of their approach. They rely on the widget paradigm where users select a widget, drop a widget onto a canvas, customize the widget, and specify how to connect widgets. The difference between each system is the number of widgets (e.g., 43 for Pipes and around 300 for Popfly), the type of widgets supported, and the ease of use. For example, Marmite and Popfly will suggest possible widgets that can be connected to existing ones on the canvas, while Pipes will rely on users to select the correct widgets. Compared to these systems, Karma uses a unified paradigm that does not require users to locate widgets or understand how each widget works.

Bungee Labs (www.bungeelabs.com), IBM's QED wiki (www.ibm.com), and Proto Software (www.protosw.com) are example Mashup tools for enterprise applications. These tools also use widgets to support most Mashup building functionality. As a result, experts are required to use them because configuring these widgets to handle industry strength applications is difficult.

While it would be interesting to report detailed comparison between each system that uses the widget approach and Karma, it is impossible to obtain and test each software since a) some are commercial applications that require purchasing and b) some are research prototypes that cannot be obtained. However, we can compare Karma with these systems based on their approach paradigm. Karma differs from

the widget paradigm in two different aspects:

—Interface: In the widget paradigm, a user builds a Mashup by composing a connected graph of widgets on a canvas. Except for Marmite, users of these systems have to re-execute all the widgets while debugging a Mashup to see any intermediate result. Also, as the Mashup becomes more complicated, the user has to navigate through a complex graph to debug or add new widgets. In Karma, the user builds a Mashup by trying to populate one table with data. By letting the user sees the result in every step and interact in a familiar workspace (e.g., spreadsheet), Karma is less confusing and easier to use compared to the widget based approach.

—Interaction: In the widget paradigm, the user needs to customize a widget and specifies how to connect widgets. This can be challenging and confusing to naive users [Wong and Hong 2007], and it requires the user to understand how each widget works. In Karma, the user specifies an example in the form of data and then Karma indirectly infers operations from the sample data. As a result, the user learning curve is low and the time required to build Mashups is smaller.

8.1.2 *Other approaches.* Besides the widget approach, there are other approaches to building Mashups. These approaches are covered below:

Simile [Huynh et al. 2007], the earliest system developed at MIT, focuses mainly on retrieving the data from Web pages using a DOM tree. Simile works as a Firefox plugin where users can click on a particular text on a Web site and save it in the repository. If that data element belongs to a list, the rest of the elements are also highlighted and extracted. After extracting data from Web pages, users can also tag sources with keywords that can be searched and published later. Dapper improves over Simile by providing an end-to-end system to build a Mashup. In Dapper, users are led step by step linearly through an interaction screen, where they address data extraction, source modeling, and data integration problems. However, the users still have to do most of the work manually. Dapper also provides only one cleaning operation that enables users to extract a segment of text (similar to Java's substring). Compared to Simile and Dapper, Karma extends the DOM tree approach to support more data structures and extraction from detail pages.

Potluck [Huynh et al. 2008] is a recent Mashup tool developed at MIT. Potluck assumes that Web sources already have RDF descriptions that enable easy extraction and provide infrastructure that addresses source modeling, data cleaning, and data integration. One could use Simile (e.g., for data extraction) in conjunction with Potluck to build Mashups. Potluck does provide suggestions to users during data cleaning. Compared to Karma, however, Potluck lacks many automated features that enable users to address Mashup building problems quickly. For example, Potluck's users need to manually map attributes between sources and specify data integration without any system suggestions.

Intel's MashMaker [Ennals and Gay 2007] takes a different approach where its platform supports multiple levels of users. In MashMaker, expert users do all the work in each area. A normal user would use the system by browsing a page (e.g., Craigslist's apartment), and MashMaker will suggest data from other sources that can be retrieved and combined (e.g., movie theaters nearby) with data on the user's current page. Note that MashMaker supports only Web pages that are already

extracted through Dapper. Compared to Karma, MashMaker limits choices for normal users to pages that exist in Dapper and data integration plans that have already been specified by experts.

Mario [Riabov et al. 2008] casts the problem of Mashup building as search and refinement in the planning domain. The search and refinement are done through a tag, a short keyword describing an operation or a previously built Mashup. There are two types of users: naive and expert. To build a Mashup, a naive user selects and combines tags to refine the output of the Mashup. The task of defining tags is left to expert users. Mario models their operators after Yahoo's Pipes. As a result, its coverage is limited to Pipe's widgets and existing Pipes. Mario's approach has two limitation compared to Karma. First, naive users are limited to tags already defined by experts. Using a tag might not be descriptive enough to convey existing complex Mashups. Second, while naive users are able to set input parameters for a tag representing a Mashup, they are not able to fine tune widgets residing in that Mashup. For example, it would not be possible for users to customize regular expression (like in Figure 2) to change the way the data is cleaned.

Cards [Dontcheva et al. 2007] views a Mashup as a collection of *cards*; a different way to define a tuple of data. Its users extract data from a Web site and store it in a card. Users can model relationships between sources and indirectly specify a way to integrate them by drawing a link between attributes, between cards, or between a card and a Web site. To build a Mashup in Cards, its users would have to search and manually connect different cards together. Karma, however, provides a spreadsheet platform where a) users can work on an individual data cell which offers finer-grained manipulation than Cards, and b) Karma automatically picks sources and deduces join conditions based on existing data in the table and databases.

D.Mix [Hartmann et al. 2007] and OpenKapow (openkapow.com) allow users to cut and paste data from Web pages to be used later. However, both systems assume some level of expertise in programming in HTML and Javascript. In contrast, Karma does not require users to understand any programming language.

Google MyMaps allows users to create and import map points from limited sources. However, the process of adding a map point is often done manually. Google also has its own Mashup Editor (editor.googleMashups.com). However, it is aimed at programmers.

Agent Wizard [Tuchinda and Knoblock 2004] lets users create a Mashup by answering a series of questions. A users builds a Mashup incrementally in a bottom up manner. As a user answers more questions, more operations are added to an overall plan that extracts, filters, and integrates data. However, agent wizard has two disadvantages compared to Karma. First, as a Mashup gets more complicated, Agent Wizard's users need to answer many more questions. Second, they also need to understand how to specify filter and join conditions. In Karma, those steps are done indirectly through data samples given by users.

The systems app2you [Kowalczykowski et al. 2009] and AppForge [Yang et al. 2008] share similar goals of allowing users to create a complex form-based application without programming. By analyzing the user-directed placement of forms and attributes, app2you and AppForge can construct database schema automatically. The similarity between these two works and Karma is that all systems try to learn

and create a database schema through user interactions. However, Karma differs from these systems in two aspects. First, app2you and AppForge are form-driven, an application in these systems revolves around creating web forms. On the other hand, Karma is data-driven, an application in Karma is created by combining data from multiple web sources and databases. Second, compared to Karma, app2you and AppForge lack supports for data extraction and data cleaning.

CopyCat [Ives et al. 2009] uses Karma's paradigm as a basis to explore a best-effort data integration tool that provides an explanation behind each choice presented. It lets users integrate data by copying data from different sources and pasting it into a spreadsheet-like workspace to answer an ad-hoc question. The system deduces user actions and provides them with auto-completion choices, each with an explanation in the form of provenance [Cui 2001]. The user can give feedback on these suggestions and the system learns from this feedback to improve future choices.

8.1.3 *Mashup Building Problem Coverage.* Table IV shows a comparison, based on their coverages, of only the Mashup tools that a) can extract data from a Web page, and b) aim to support casual users, and c) integrate data from multiple sources. The terminology of how each tool handles each problem area is shown below in Table V

Table IV. Approach comparison between different Mashup tools segmented by problem areas

System	Data Retrieval	Source Modeling	Data Cleaning	Data Integration	Mashup Type Supported
MIT's Simile	DOM	Manual	N/A	N/A	1
MIT's Pot Luck	RDF	Manual	PBD	Manual	1,3,4
Dapper	DOM	Manual	Manual	Join only	1,2,4
Yahoo's Pipes	Widgets	Manual	Widgets	Union only	1,2,3
Mario	Tag	Tag	Tag	Union only	1,2,3
MS's Popfly	Widgets	Manual	Widgets	Widgets	1,2,4
CMU's Marmite	Widgets	Manual	Widgets	Widgets	1,2,4
Intel's Mashmaker	Dapper	Manual	Widgets	Expert	1,2,3,4
Google MyMap	Widgets	Manual	N/A	Union only	1,2
Agent Wizard	Q/A	Q/A	Q/A	Q/A	1,3,4
Cards	DOM	Manual	N/A	Manual	1,2,4
CopyCat	DOM	Database	PBD	PBD	1,2,3,4
Karma	DOM	Database	PBD	PBD	1,2,3,4

Note that the only Mashup building tools that support all four types of Mashups beside Karma and CopyCat (which is based on Karma) is Intel Mashmaker. However, Mashmaker requires an expert user to customize at least Mashups of type 3 and type 4. Most systems support up to three types of Mashups, where type 3 seems to be ignored because of the complexity of capturing HTML forms.

In conclusion, Karma serves its target group better than similar systems by having *all* of the following advantages:

—an end-to-end approach: Karma allows users to build a more complete Mashup by letting users tackle important Mashup building subproblems, when needed.

Table V. Definition of terminologies used

Term	Explanation
Database	Databases are leveraged to help generate suggestions to assist users
DOM	A document object model approach is used to handle extraction. However, there is also varying degrees of how each system utilizes the DOM. For example, Simile might only uses the DOM as is. However, Karma might build on DOM to enhance data extraction.
Experts	An expert is required to solve that specific problem area
N/A	The specific problem area is not addressed or supported or the information about it cannot be found.
Join only	Only database join is supported in the data integration step.
Manual	The specific problem area is supported, but a user needs to do it manually. For example, a manual approach in source modeling means that the user has to specify the relationship between data sources.
PBD	The Programming by Demonstration approach is used.
Q/A	The user has to answer one or more questions to solve a specific problem area. The answer might also involve specifying a join condition between sources.
RDF	The data extraction step assumes that a Web source has an RDF representation which allows easy retrieval of data.
Tag	The user selects the operation abstractly by choosing from a tag cloud. Fine grain customization of selected operations may not be available.
Widgets	A widget must be selected and customized to tackle that specific problem area.
Union only	Only database union is supported in the data integration step.

—a consistent paradigm: Karma exploits user familiarity with the tabular platform and provides an easy to understand, learn, and use interaction platform.

—wide coverage: Karma lets users build all four Mashup types.

8.2 Related Research Fields

For the data retrieval problem, earlier work focuses on a) automatic extraction of lists and table [Crescenzi and Mecca 2004; Lerman et al. 2004; Gatterbauer et al. 2007] or b) using AI techniques (i.e., machine learning) to capture the extraction rules or models from user's labeled examples [Muslea et al. 2003; Cohen et al. 2002; Raghavan and Garcia-Molina 2001]. Automatic extraction only works when it is possible to identify lists and tables, and machine learning techniques require users to provide more examples as the structure of Web sources are getting more complicated. Simile [Huynh et al. 2007], Dapper, PLOW [Allen et al. 2007], and Cards [Dontcheva et al. 2007] employ the DOM approach, which requires less labeling. While this approach makes data retrieval easier, the DOM alone does not provide a mechanism to handle Web pages with multiple embedded lists or detail page extraction. Karma fills these gaps by extending the DOM approach with the use of marker and table constraints.

Extracting data from unstructured text can yield more information. A survey paper of this field can be found at [Reeve and Han 2005]. Phoebus [Michelson and Knoblock 2007b; 2007a] extracts this kind of information in an unsupervised manner using reference sets. Gate [Cunningham et al. 2002] provides the framework for writing an application that supports natural language extraction. The current version of Karma did not allow users to extract information from unstructured text.

However, Karma could be extended to support extraction of unstructured data in the form of suggestions what is shown in section 6.

In the schema matching domain, there are several good surveys [Rahm and Bernstein 2001; Halevy et al. 2006]. Initially, researchers focused their efforts on 1:1 matching (i.e., matching one attribute to another attribute). Early work such as TranScm [Milo and Zohar 1998] and Artemis [Bergamaschi et al. 2001], use a rule-based approach to determine how to map attributes together. Later on, Semint [Li et al. 2000] and ILA [Perkowitz and Etzioni 1995] employed machine learning techniques to learn matching attributes from training samples. LSD [Doan et al. 2000] provides the framework to support multiple learners to achieve better matching accuracy. Currently, the n:m matching problem, where one or more attribute can be mapped to multiple attributes, is addressed in [Xu and Embley 2003; Dhamankar et al. 2004]. Spider [Koudas et al. 2005] addresses the schema matching in a transparent manner, where users have access to and can customize matching criteria. We do not present a new technique to solve the problems of source modeling and schema matching. Karma uses simple techniques, such as string similarity comparison, and relies on users to interactively narrow the candidate matches.

In the data cleaning domain, most commercial tools [Chaudhuri and Dayal 1997] focus on the process of Extract-Transform-Load (ETL) through a scripting language; only trained experts can use these tools. Potter's Wheel [Raman and Hellerstein 2001] takes an interactive approach to data cleaning by letting users specify the clean data and then inducing transformation language adaptation techniques described in [Abiteboul et al. 1999; Chen et al. 1993; Lakshmanan et al. 1996]. Karma's cleaning by example approach is based on Potter's wheel. However, Karma also suggests cleaning values based on the overlapping of new data and existing data in the databases.

The goal of the data integration research is to allow casual users to access, locate, and integrate data using a uniform query interface. A general survey of the data integration field in the past twenty years can be found in [Halevy et al. 2006]. Our data integration framework is a combination of programming by demonstration [Cypher et al. 1993; Lau 2001; Lieberman 2001] and query by example (QBE) [Zloof 1975]. In programming by demonstration, methods and procedures are induced from users' examples and interaction. This approach can be effective in various domains [Sugiura and Koseki 1998; Lau et al. 2004; Gibson et al. 2007] where users understand and know how to do such tasks. In Karma, however, users may not know how to formulate queries and only interact with the system through data. The interaction is in a table similar to QBE. However, QBE requires users to manually select data sources. On the other hand, Karma induces the sources to use automatically and guides users to fill in only valid values. As a result, users do not need to know about data sources.

There has also been recent related work on integrating large data sets using a spreadsheet paradigm in a project called Google Fusion Tables [Gonzalez et al. 2010; Gonzalez et al. 2010]. In this effort they developed an online service that allows users to integrate large datasets, annotate and collaborate on the data, and visualize the integrated results. The focus of the work is primarily on the scalability and visualization of the integrated data instead of on the problems of extracting,

modeling, cleaning, and integrating the data. The ideas in Google Fusion Tables are relevant to Karma and would apply directly to deal with the problems of scaling up to large datasets and visualizing the integrated results.

A more in depth survey of related research fields can be found in [Tuchinda 2008].

9. EVALUATION

This section presents a formal user evaluation of Karma. The overall evaluation plan follows the evaluation methodology outlined in [Tallis et al. 2001]. We have selected a combination of Dapper/Pipes as a baseline comparison. Yahoo's Pipes is a state-of-the-art Mashup building system that employs the widget approach and is readily available. However, since Pipes does not have enough capability to do data extraction, we chose Dapper to fill that role; a subject has to extract data from a Web page using Dapper, then process it using Pipes. This combination approach to building Mashups is often used by programmers who build Mashups.

9.1 Claims

In this section, we evaluate the following three claims:

(1) Users with no programming experiences can build all four Mashup types specified in section 2.
(2) Karma takes less time to complete each subtask (e.g., data extraction, source modeling, data cleaning, and data integration) and scales better as the tasks get more difficult.
(3) The user takes less time to build the same Mashup in Karma compared to Dapper/Pipes.

9.2 Users

While the approach in this article is designed for users with no programming experience, given the time required to teach users how to use each of the systems, it was not possible to find nonprogrammer subjects to commit to the full evaluation. As a trade off, there are two types of users in our evaluations: users with programming experience and users with no programming experience.

Programmer users are M.S. and Ph.D. students from a graduate-level class focusing on the problem of information integration on the Web. The total number of students who participated in the evaluation is 20. They are prime candidates as subjects for the evaluation. Since they were given one assignment on Dapper and two assignments on Pipes in the course, they spent a significant amount of time learning and practicing with Dapper/Pipes.

For non programmers, we have recruited three users with no programming experience. One of the users is an M.S. student in accounting and the others are administrative assistants. They had no prior programming experience, but they are familiar with Web technology and Excel.

9.3 Tasks

There are three tasks in the evaluation. These tasks are designed to capture the structure of the four Mashup types discussed in Section 2. The first task involves building a Mashup from one simple source. The second task is building a Mashup

that combined multiple query results using a database union from Web pages with an HTML form. Finally, the third task is building a Mashup that combines data from multiple sources using a database join.

The Web sources used in these tasks are well-structured websites (e.g., tables, lists, and forms), such as UPS, Google, and Craigslist. Our selected web sources are representative of different types of websites (e.g., corporate, shopping, and web board) that Karma should be capable of extracting data from.

Note that these tasks are designed in a hit-or-miss fashion. As a result, if they are done correctly, the quality of the resulting Mashup, whether it is done using Dapper/Pipes or Karma, should be the same.

Table VI shows how difficult it is to solve each Mashup building subtask in each task. Differentiating how hard each subtask is allows us to do fine-grained comparisons for each subtask as well as do an overall comparison.

Table VI. Difficulty breakdown for each Mashup building subtask in each task. Task 1 has no data integration subtask, while task 3 has no data cleaning subtask.

	Data Extraction	Source Modeling	Data Cleaning	Data Integration
Task 1	Moderate	Simple	Hard	N/A
Task 2	Hard	Simple	Simple	Union (Simple)
Task 3	Simple	Simple	N/A	Join (Hard)

To put things in perspective, if these tasks were to be implemented using a normal programming language like Java, it can take an hour or more depending on the expertise of a programmer because it involves writing a parser, manipulating data, and customizing the display. Mashup building tools, such as Dapper/Pipes, were created to alleviate this problem by reducing the time to implement these tasks to around 10 minutes. Karma uses Dapper/Pipes as a baseline and we compare our performance to these systems.

The users with programming experience were asked to do each task outlined twice: once using Dapper/Pipes and once using Karma. The users with no programming experience were asked to do each task once using Karma. The evaluation results for non-programmers are used to suport claim 1, while the result from the programmers are used to support claims 2 and 3.

9.4 Procedure

The experimental procedure is divided into three phrases: familiarization, practice, and test.

—**Familiarization:** The tutorials and videos for each system were sent out two days before the evaluation. On the day of the experiment, each subject was given a quick 30 minute tutorial that covered all required systems to refresh their memory.

—**Practice:** After finishing the quick tutorial, both type of subjects were given two practice tasks in Karma. Note that nonprogrammer did not have to implement Mashup tasks using Dapper/Pipes and programmers were already familiar with Dapper/Pipes through their assignments. As a result, we decided that no practice was necessary for Dapper/Pipes.

—**Test:** The test phrase lasted about one hour. We used a cross-over trial [Hills and Armitage 1979], where each subject was asked to do each task twice: using Karma and Dapper/Pipes. There are many ways to configure a cross-over trial depending on how to segment subjects into groups and how to set periods of different tests (e.g., two-period two-treatment). However, any cross-over trial has a carry-over effect; doing the same task the second time will always be easier since users gain knowledge about the task and there is no easy way to discount this knowledge. As a result, we decided to let the subject implement each task in Karma first and then do the same task using Dapper/Pipes. This setup gives an advantage to Dapper/Pipes because the subject would familiarize herself/herself with the Web sources and requirements while performing the task using Karma. This approach allows us to provide a lower bound on the use of Karma compared to Dapper/Pipes. In addition, hints and advice were given when asked. When a subject got stuck in a particular subtask (e.g., cleaning and integration) for more than 5 minutes, the cutoff time was enforced and the task would be marked as *fail*.

9.5 Data Collection

The computer screen was recorded while subjects were completing the three evaluation tasks. These video records allowed us to see how long each subject took to complete each task and what kind of choices and options each subject made. All the videos recorded were segmented into multiple time slots based on the four subtasks (i.e., data extraction, source modeling, data cleaning, and data integration) and other miscellaneous operations. To ensure fairness in the evaluation, we discarded unrelated time segments, such as page loading time, and interfacing time between Dapper and Pipes (when applicable). To enable a comparison when a subject failed on a task, we substituted the *Fail* slot with the cut off time of 5 minutes. The normalized data, segmented by each subproblem is shown in appendix A.

9.6 Results

The results will be segmented and discussed based on each of the claims made earlier. For claim 2 and claim 3, we compute statistical significance tests to support our results. Note that [Segre et al. 1991] argued that time bound experiment can bias the result of the evaluation and Etzioni [Etzioni and Etzioni 1994] extensively discussed some possible solutions. In our case, the time bound is applied to both the Dapper/Pipe and Karma results. However, every subject finished his/her task under the time limit using Karma. As a result, the scenario introduced in [Segre et al. 1991] that shows a possible bias in time bound experiments does not apply to our experimental results.

Note that the evaluation results for nonprogrammer are used to satisfy claim 1, while the result from programmers is used to satisfy claim 2 and claim 3. We believe that if programmer subjects, who are familiar with work flows and widget paradigms in general, spend more time to implement a task using Dapper/Pipes compared to Karma, then non-programmer subjects would also spend more time to implement the same task using Dapper/Pipes (if they were to learn how to use these systems) compared to Karma.

9.6.1 *Claim 1: Users with no programming experiences can build all four Mashup types.* Figure 21 shows the result measured in minutes for non programmers. On average, it takes non programmers 3:47 minutes to build a single Mashup. Each subject was able to complete all three tasks (designed to be representative of four Mashup types in section 2) without failing, which validates our claim that user with no programming experiences can build all four Mashup types.

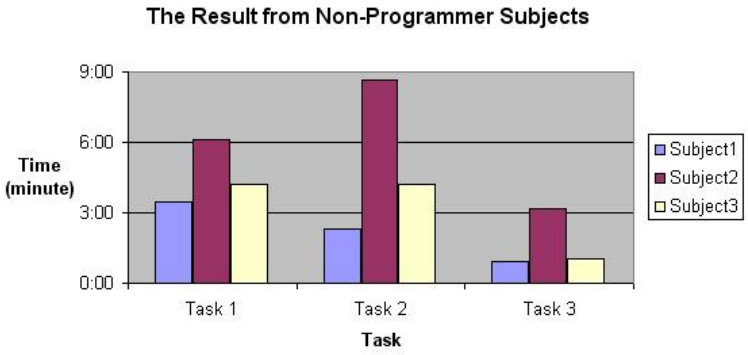

Fig. 21. The result for non programmers using Karma to build three Mashup tasks.

9.6.2 *Claim 2: Karma takes less time to complete each subtask and scales better as the tasks get more difficult.* To validate this claim, we will show the performance comparison segmented by each subtask: data extraction, source modeling, data cleaning, and data integration. Note that since there are two subjects (i.e., No.3 and No.17) who did not have time to finish task 1, the result from these two subjects will be excluded from results that involve task 1 to ensure a fair comparison.

Data Extraction

Figure 22 shows the performance measurements for Karma and Dapper/Pipes for the data extraction subtask. In each graph, the x-axis is the time spent to complete the data extraction subtask, while the y-axis is the number of subjects that fall into each time segment.

In task 3, the extraction task is very simple as it involves extracting only one field of data. The result indicates that most subjects can finish the extraction task using Karma in less than 30 seconds, while most of them finish the extraction task using Dapper in around 30 seconds to one minute. In task 1, the extraction task is of medium difficulty because of the irregular DOM structure of the Web source. The graph shows that Karma performs better as more subjects finish the subtask faster using Karma. In task 2, the extraction task is hard because it involves extracting data from a data source with an HTML form. The graph indicates that Karma performs better than Dapper/Pipes as all subjects finish the subtask using Karma in less than 3 minutes, while 9 subjects take 3 minutes and longer and 14 subject fail to finish the same subtask using Dapper/Pipes.

Table VII shows the T-test for the hypothesis that Karma is faster on average than Dapper/Pipes for data extraction in each of the tasks and the overall subtask.

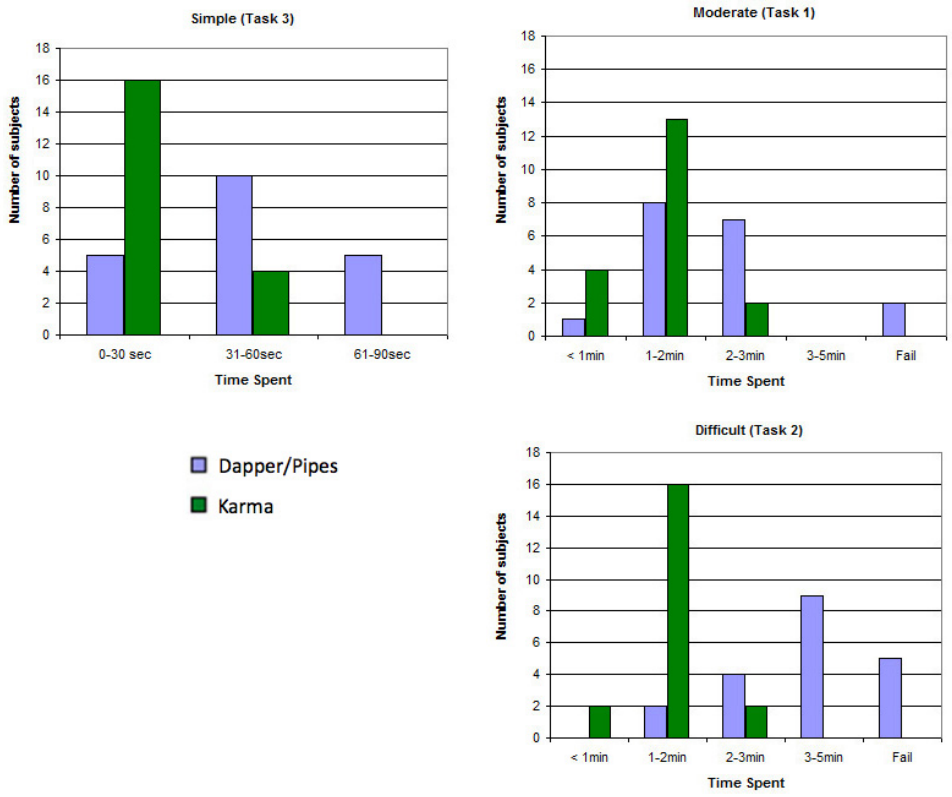

Fig. 22. The performance comparison (programmers) between Karma and Dapper/Pipes for the data extraction subtask in each task ranging from simple to difficult

Table VII. The statistical significance test result for data extraction subtask

Task No	T-test
Task 3 (Simple)	t=4.69, degree of freedom=38, and $p < 0.01$
Task 1 (Moderate)	t=3.58, degree of freedom=34, and $p < 0.01$
Task 2 (Hard)	t=7.05, degree of freedom=38, and $p < 0.01$
Overall data extraction	t=5.35, degree of freedom=114, and $p < 0.01$

Source Modeling

Figure 23 shows the performance measurements for Karma and Dapper/Pipes for the source modeling subtask. The result suggests that Dapper/Pipes is faster than Karma in task 1 and task 2. Karma does perform better in task 3 where attributes are set automatically because of value overlapping in the database. However, Karma performs worse than Dapper/Pipes overall, because of two factors:

—The table implementation of Java in Karma does not allow a user to set the attribute directly by clicking at the attribute cell. Users need to go to the attribute tab to select an attribute name for each column.

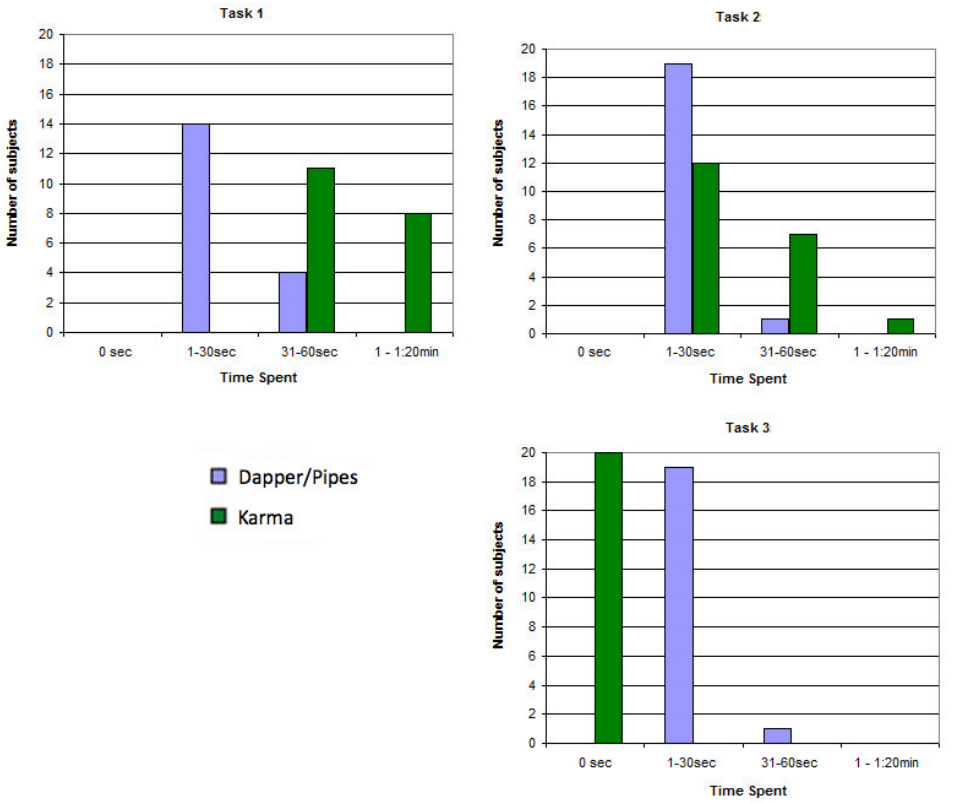

Fig. 23. The performance comparison (programmers) between Karma and Dapper/Pipes for the source modeling subtask

—When using Karma, subjects also look to see what kind of attributes are suggested by Karma, while they can just type in any attribute value for Dapper.

While Karma performs worse than Dapper/Pipes in this subtask, notice that the time spent doing source modeling compared to the overall Mashup building task is small. Furthermore, by assigning the right attribute using Karma, users save more time during the data integration subtask, which involves a database join (task 3).

Data Cleaning

Figure 24 shows the performance comparison between Karma and Dapper/Pipes for the data cleaning subtask. The performance comparison difference between Karma and Dapper/Pipes is more obvious in the data cleaning subtask compared to the earlier two subtasks. Furthermore, only about 40 percents of subjects can complete the data cleaning subtask in task 2 using Pipes, while all subjects can complete the same subtask using Karma.

By allowing a subject to enter the cleaned result and having Karma try to infer the cleaning rule, he/she does not have to spend time customizing a cleaning widget. Customizing a cleaning operation in Pipes can be difficult as it requires its users to understand regular expressions; even students who know how to program had

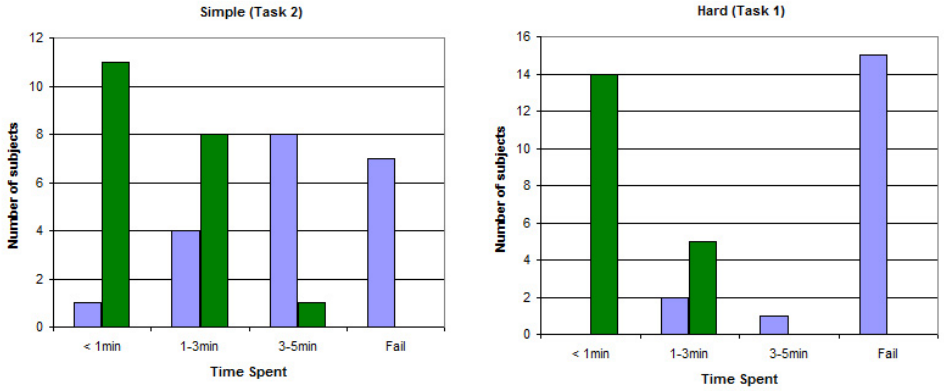

Fig. 24. The performance comparison (programmers) between Karma and Dapper/Pipes for the data cleaning subtask

difficulty trying to use it.

Karma's cleaning by example does have a limitation. While it is easier to use, it is less expressive than regular expressions; if Karma cannot match the user's example with its predefined rules, then the user would have to manually clean each result. However, for casual Mashup building, providing a predefined set of rules that are most used is a logical trade off compared to spending a long time writing a regular expression to clean the data.

Table VIII shows the T-test for the hypothesis that Karma is faster than Dapper/Pipes for data cleaning in each of the task and the overall subtask.

Table VIII. The statistical significance test result for data cleaning subtask

Task No	T-test
Task 2 (Simple)	t=7.68, degree of freedom=38, and $p < 0.01$
Task 1 (Hard)	t=12.76, degree of freedom=34, and $p < 0.01$
Overall data cleaning	t=13.54, degree of freedom=74, and $p < 0.01$

Data Integration

Figure 25 shows the performance comparison between Karma and Dapper/Pipes for the data integration subtask. In the database union case, Karma does not require any time to customize the database union because the spreadsheet structure allows a Karma user to indirectly specify the union by dragging the data from a new similar source into a new row. While it takes more time in Pipes to specify union, the time taken is small compared to the overall time required to build Mashups.

In the database join case shown, Karma performs much better than Dapper/Pipes; all subjects completed this subtask in less than three minutes, while the majority of the subjects cannot complete the task using Dapper/Pipes. This result shows promise for Karma because one of the main advantages in Mashups is that once built, a Mashup can be reused as a module in another Mashup. However, given the number of existing Mashups, it would be time consuming for a casual user to locate

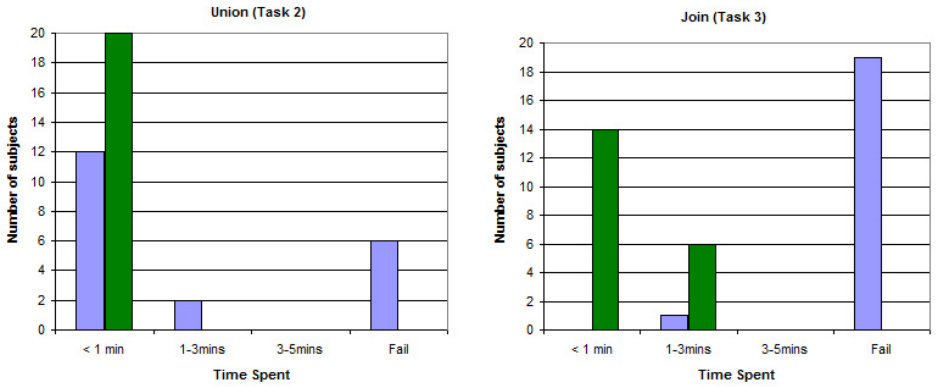

Fig. 25. The performance comparison (programmers) between Karma and Dapper/Pipes for the data integration subtask

and figure out how to integrate a Mashup built by other people on their own. This is apparent in task 3, where a Dapper/Pipes subject needs to join her own Mashup with another one created by someone else. Karma's approach to data integration allows the subject to bypass the search step and instead focuses on selecting what kind of data (suggested by Karma) to integrate into a newly extracted data source.

Table IX shows the T-test for the hypothesis that Karma is faster than Dapper/Pipes for data integration in each of the task and the overall subtask.

Table IX. The statistical significance test result for data integration subtask

Task No	T-test
Task 2 Union (Simple)	t=3.15, degree of freedom=38, and $p < 0.01$
Task 3 Join (Hard)	t=20.51, degree of freedom=38, and $p < 0.01$
Overall data integration	t=7.05, degree of freedom=78, and $p < 0.01$

We also want to show that as the subtasks gets progressively harder, Karma performs better in terms of time spent to finish the subtask. For each programmer subject, we compute the following values where i designates the subtask type (i.e., data extraction, source modeling, data cleaning, and data integration) and j designates the subject id:

(1) *DiffD(i,j)*: time spent using Dapper/Pipes in difficult $subtask_i$ - time spent using Dapper/Pipes in easy $subtask_i$

(2) *DiffK(i,j)*: time spent using Karma in difficult $subtask_i$ - time spent using Karma in easy $subtask_i$

For example, *DiffD* for the extraction subtask for subject No.1 are:

DiffD(extraction, 1) = time that subject No.1 spent using Dapper/Pipes in the extraction subtask of task 2 (difficult) - time that subject No.1 spent using Dapper/Pipes in the extraction subtask of task 3 (easy)

We repeat this computation on every subtask for each subject. Table X shows the average of *DiffD* and *DiffK* in each of the subtask and the T-test under the hypothesis that *DiffK* < *DiffD*.

Table X. The average value of *DiffD* and *DiffK* in seconds for each subtask

	Avg *DiffD*	Avg *DiffK*	T-test for $p < 0.05$
Data extraction	170	62	t=5.87 and DF=34
Data cleaning	72	-10	t=1.96 and DF=24
Data integration	261	57	t=4.90 and DF=30

As seen in table X, the average time increment for using Dapper/Pipes going from an easy subtask to a difficult subtask is higher compared to that of Karma and the result is statistically significant. For example, the subjects spent, on average, 170 seconds more to finish the difficult extraction subtask in task 2 compared to the easy extraction subtask in task 3, when using Dapper/Pipes. On the other hand, the same group spent 62 seconds, on average, more to finish the difficult extraction subtask in task 2 compared to the easy extraction subtask in task 3, when using Karma. We omit the result from the source modeling because we already know that Karma performs worse than Dapper/Pipes in the source modeling subtask. The negative number indicate for average *DiffK* in the data cleaning subtask means that users take even less time doing a more difficult cleaning subtask compared to the easy cleaning subtask.

Note that computing *DiffD* and *DiffK* for some data points generate doubly censored data [Etzioni and Etzioni 1994]. In the case of doubly censored data, where it is expensive to obtain more sample points, the standard statistical practice is to throw away such data and consider the data that has a) singly censored pairs or b) uncensored pairs [Woolson and Lachencruch 1980]. As a result, the total number of subject for each subtask comparison might vary.

Table XI shows a more quantitative comparison between Karma and Dapper/Pipes. The value is computed by averaging the time spent for each subtask over three scenario tasks from results using programmers. The overall result indicates Karma performs better in each of the subtasks (except source modeling) and overall. Among the four subtasks, Karma performs extremely well in data cleaning and data integration due to the reasons explained earlier.

Table XI. Overall comparison (programmers) between Dapper/Pipes and Karma average over three tasks

Task	Avg time for Dapper/Pipes in minutes	Avg time for Karma in minutes	Factor of improvement
Data Extraction	2:09	0:58	2.22
Source Modeling	0:19	0:28	-0.67
Data Cleaning	4:07	1:00	4.16
Data Integration	3:06	0:29	6.49
Overall	9:41	2:55	3.32

While Karma performs worse than Dapper/Pipes for source modeling, the mean difference is only 10 seconds. In addition, this shortcoming is compensated for during the data integration step where Karma took about 2.5 minutes less on average to complete the subtask.

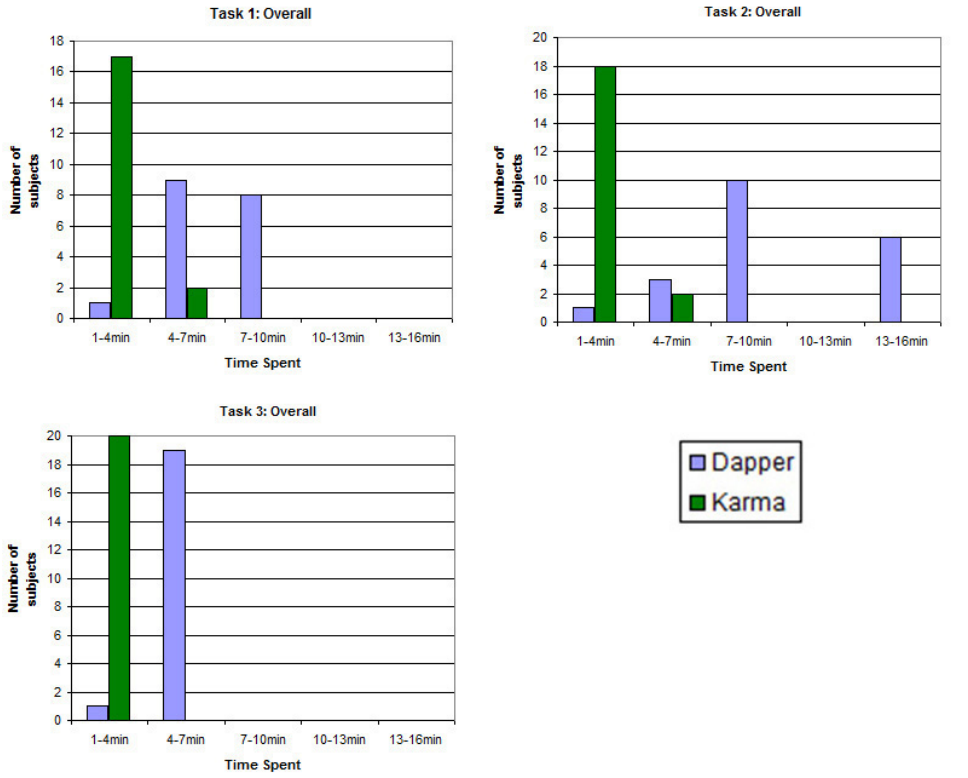

Fig. 26. The performance comparison (programmers) between Karma and Dapper/Pipes for all three tasks

9.6.3 *Claim 3: A user takes less time to build the same Mashup in Karma compared to Dapper/Pipes.* Figures 26 shows the overall comparison between Karma and Dapper/Pipes in each task. Each graph combines the time spent for each subtask (i.e., data extraction, source modeling, data cleaning, and data integration). For each task, Karma performs noticeably better than Dapper/Pipes; most subjects spent less than 4 minutes to complete each of the tasks using Karma. Also, this result is statistically significant as shown in table XII using the hypothesis that Karma is faster than Dapper/Pipes for each task.

As shown in table XI, it takes our subjects 9:41 minutes on average to build a Mashup using Dapper/Pipes, while it takes our programmer subjects only 2:55 minutes (3:47 minutes for non-programmers) to build the same Mashup using Karma. As a result, Karma performs better than Dapper/Pipes by at least a factor of 3.

Table XII. The statistical significance test result for each task

Task No	T-test
Task 1	t=9.93, degree of freedom=34, and $p < 0.01$
Task 2	t=7.37, degree of freedom=38, and $p < 0.01$
Task 3	t=23.45, degree of freedom=38, and $p < 0.01$
Overall (combining 3 tasks)	t=13.24, degree of freedom=114, and $p < 0.01$

In addition, Karma also allows users to build Mashups that they fail to build using Dapper/Pipes.

Note that we underestimate the actual time spent in Dapper/Pipes by using only 5 minute cutoff time. The actual time spent could be longer depending on how many failures and the type of failure in a task. Table XIII shows the failure rate in Dapper/Pipes. Note that both programmer and nonprogrammer subjects were able to complete all three tasks using Karma without failing. However, subjects who use Dapper/Pipe sometimes fail in a particular subtask. The *Overall* row in Table XIII shows the percentage of users who fail on least one subtask in each respective task.

Table XIII. Individual and overall failure rates in Dapper/Pipes.

Task	Task 1	Task 2	Task 3
Data Extraction	5.5%(1/18)	25%(5/20)	0%(0/20)
Source Modeling	0%(0/18)	0%(0/20)	0%(0/20)
Data Cleaning	83%(15/18)	35%(7/20)	n/a
Data Integration	n/a	30%(6/20)	95%(19/20)
Overall	83%(15/18)	45%(9/20)	95%(19/20)

Besides the difficulty of customizing widgets, there are three additional factors that contribute to failures in Dapper/Pipes. These factors are described below:

(1) Cutoff Failure: In our evaluation, we use the cutoff time of 5 minutes; users who spend more time than 5 minutes in a particular subtask is marked as failing. However, of all 53 instances of failures, there are 11 instances (i.e. 20% of all failure instances) where users were able to complete the subtask using more than 5 minutes. We did not take this extra time into the calculation to ensure a uniform result.

(2) Cascading Failure: In task 2, failing to complete the data cleaning subtask can lead to failing to complete the integration subtask. Five out of nine overall failures in task 2 can be attributed to this type of failures. However, task 2 is designed such that users could complete the integration subtask before doing the data cleaning subtask; users had cascading failures because of their decisions to solve a harder subtask first.

(3) Minimal Support Failure: In task 3, we have a failure rate of 95% for Dapper/Pipes. This failure can be attributed to the length it takes users to customize the database join operation. As mentioned in Chapter 1, most Mashup tools choose to focus on particular subtasks while ignoring others. However,

the breakdown of Mashup types show that the database join is one of the important features. Karma's support of the database join operation allows users to build Mashups that combine two sources through a join. The other Mashup tool that fully supports database join is Intel's Mashmaker [Ennals and Gay 2007]. However, Mashmaker requires experts to customize predefined join operations between sources; casual users cannot customize database join between data sources by themselves.

10. CONCLUSION AND FUTURE WORK

To address the four problems of building Mashups (i.e., data extraction, source modeling, data cleaning, and data integration), we have introduced the Karma framework for building Mashups based on three key ideas. The first idea is to focus on the data, not the operations. By using the programming-by-demonstration paradigm, Karma can learn the operation that the user wants to perform indirectly by looking at the data provided by the user. The second idea is to leverage existing databases. By comparing the value that the user enters with the data in the existing database, it is possible to deduce some relationships between new data and existing data. These relationships allow Karma to be able to assist users in the problems of source modeling, data cleaning, and data integration. The third idea is to consolidate the Mashup building problems. Since many Mashup building issues are interrelated, it is possible to exploit the structure such that solving a problem in one area can help simplify the process of solving a problem in another area.

We have demonstrated effectiveness of Karma through a user evaluation that compared the approach with the widget approach of Dapper/Pipes. The experiments show that:

—Users can finish the three tasks designed to capture the structure of the four Mashup types using Karma. In contrast, users have trouble finishing some of the tasks using Dapper/Pipes.

—Users took less time to finish the same task using Karma compared to Dapper/Pipes. The average time used in Karma to build Mashups is 2:55 minutes, while the average time used in Dapper/Pipes is 9:41 minutes.

—Overall, Karma is faster by a factor of 3.3. The key saving areas are data cleaning and data integration. In the data cleaning phrase, Karma allows users to specify the result by example and automatically deduces the cleaning operation, while Dapper/Pipes requires users to customize complicated widgets. In data integration phrase, Karma leverages databases to help users decide what data is available to add, while Dapper/Pipes requires users to manually search through existing sources to link with the newly extracted data source.

There are a number of interesting directions for future work:

—**Customize Display by Examples:** While Karma provides a means to display the data in the table on the Map, users cannot fine tune how the display should look like and how the display should behave. We believe that we can apply the ideas used in Karma framework to address the data display problem too. For example, programming by demonstration can be used to help Karma learn from examples the kind of display that users might want to specify.

—**Recovering From Errors:** One of the limitations in Karma is recovering from errors. Karma uses heuristics to capture Mashup building operations, but these heuristics can be wrong. Recovering from errors is an active area of research in the programming-by-demonstration domain. However, in existing systems, users often need to trace through a concept tree induced by PBD's heuristic to fix these errors. As a result, it will be difficult for casual users to trace and recover from errors using existing frameworks. A new interaction framework that allows casual users to browse through different result scenarios instead of searching through an error tree might be a possible approach to solve this problem.

—**Source Quality:** Not all data sources are created equal. Some data sources might be useful or pertinent to users in one task, but not in another task. Currently, Karma's seed data comes from existing databases and user-generated Mashups. There is no integrated view of this data and thus there could be a lot of duplication. In the future, we could build an internal model of data sources similar to what is done in MadWiki [DeRose et al. 2008] to ensure integrity and uniformity of Karma's internal data.

—**Support For Advanced Users:** The current version of Karma has one level of users – naive users. We plan to extend Karma to support multiple tiers of users from naives to advanced users like Intel's Mashmaker[Ennals and Gay 2007]. In an advanced mode, Karma could provide more transparency and allow users to do a complex fine-tune like Spider [Koudas et al. 2005]. We have already started experimenting with the idea of providing system transparency in CopyCat [Ives et al. 2009] by letting users access the explanation of how the system generates auto-completion choices.

—**Data Cleaning Transformations:** The current preliminary transformation supported by Karma were implemented for casual users. We could extended Karma to support more sophisticated transformations like Potter's Wheel [Raman and Hellerstein 2001] while maintaining the same interaction paradigm.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/acmtw/2011-5-3/p1-URLend`.

ACKNOWLEDGMENTS

This research is based upon work supported in part by the National Science Foundation under award number IIS-0324955, in part by the Air Force Office of Scientific Research under grant number FA9550-07-1-0416, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004.

The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

REFERENCES

Abiteboul, S., Cluet, S., Milo, T., Mogilevsky, P., Simeon, J., and Zohar, S. 1999. Tools for Data Translation and Integration. *IEEE Data Engineering Bulletin 22,* 1, 3–8.

Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., and Taysom, W. 2007. PLOW: A Collaborative Task Learning Agent. In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*. AAAI Press, 1514–1519.

Bergamaschi, S., Castano, S., Vincini, M., and Beneventano, D. 2001. Semantic integration of heterogeneous information sources. *Data & Knowledge Engineering 36,* 3, 215–249.

Burnett, M. M. and Baker, M. J. 1994. Classification System For Visual Programming Languages. *Journal of Visual Languages and Computing 5,* 3, 287–300.

Chaudhuri, S. and Dayal, U. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record 26,* 1, 65–74.

Chen, W., Kifer, M., and Warren, D. S. 1993. HILOG: a foundation for higher-order logic programming. *Journal of Logic Programming 15,* 3, 187–230.

Cohen, W. W., Hurst, M., and Jensen, L. S. 2002. A flexible learning system for wrapping tables and lists in html documents. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*. ACM, New York, NY, USA, 232–241.

Cohen, W. W., Ravikumar, P., and Fienberg, S. E. 2003. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the International Joint Conferences on Artificial Intelligence Workshop on Information Integration*. 73–78.

Crescenzi, V. and Mecca, G. 2004. Automatic information extraction from large websites. *Journal of the ACM 51,* 5, 731–779.

Cui, Y. 2001. Lineage Tracing in Data Warehouses. Ph.D. thesis, Stanford University.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. 2002. GATE: an architecture for development of robust HLT applications. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, Morristown, NJ, USA, 168–175.

Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., and Turransky, A., Eds. 1993. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA.

DeRose, P., Chai, X., Gao, B. J., Shen, W., Doan, A., Bohannon, P., and Zhu, X. 2008. Building Community Wikipedias: A Machine-Human Partnership Approach. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 646–655.

Dhamankar, R., Lee, Y., Doan, A., Halevy, A., and Domingos, P. 2004. iMAP: discovering complex semantic matches between database schemas. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 383–394.

Doan, A., Domingos, P., and Levy, A. 2000. Learning source descriptions for data integration. In *Proceedings of the International Workshop on The Web and Databases (WebDB)*. Springer-Verlag, 60–71.

Dontcheva, M., Drucker, S. M., Salesin, D., and Cohen, M. F. 2007. Relations, cards, and search templates: user-guided web data integration and layout. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 61–70.

Ennals, R. and Gay, D. 2007. User-friendly functional programming for web mashups. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*. ACM, 223–234.

Etzioni, O. and Etzioni, R. 1994. Statistical Methods for Analyzing Speedup Learning Experiments. *Machine Learning 14,* 3, 333–347.

Gatterbauer, W., Bohunsky, P., Herzog, M., Krüpl, B., and Pollak, B. 2007. Towards domain-independent information extraction from web tables. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*. ACM, New York, NY, USA, 71–80.

Gibson, A., Gamble, M., Wolstencroft, K., Oinn, T., and Goble, C. 2007. The Data Playground: An Intuitive Workflow Specification Environment. In *E-SCIENCE '07: Proceedings of*

the Third IEEE International Conference on e-Science and Grid Computing. IEEE Computer Society, 59–68.

Gonzalez, H., Halevy, A. Y., Jensen, C. S., Langen, A., Madhavan, J., Shapley, R., and Shen, W. 2010. Google fusion tables: data management, integration and collaboration in the cloud. In *Proceedings of the First Symposium on Cloud Computing, Industrial Track*. 175–180.

Gonzalez, H., Halevy, A. Y., Jensen, C. S., Langen, A., Madhavan, J., Shapley, R., Shen, W., and Goldberg-Kidon, J. 2010. Google fusion tables: web-centered data management and collaboration. In *Proceedings of SIGMOD, Industrial Track*. 1061–1066.

Halevy, A., Rajaraman, A., and Ordille, J. 2006. Data integration: the teenage years. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 9–16.

Hartmann, B., Wu, L., Collins, K., and Klemmer, S. R. 2007. Programming by a sample: rapidly creating web applications with d.mix. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 241–250.

Hills, M. and Armitage, P. 1979. The two-period cross-over clinical trial. *British Journal of Clinical Pharmacology 8*, 7–20.

Huynh, D., Mazzocchi, S., and Karger, D. 2007. Piggy Bank: Experience the Semantic Web inside your web browser. *Web Semantics 5,* 1, 16–27.

Huynh, D. F., Miller, R. C., and Karger, D. R. 2008. Potluck: Data mash-up tool for casual users. *Web Semantics 6,* 4, 274–282.

Ives, Z. G., Knoblock, C. A., Minton, S., Jacob, M., Talukdar, P. P., Tuchinda, R., Ambite, J. L., Muslea, M., and Gazen, C. 2009. Interactive data integration through smart copy & paste. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

Koudas, N., Marathe, A., and Srivastava, D. 2005. Spider: flexible matching in databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 876–878.

Kowalczykowski, K., Ong, K. W., Zhao, K. K., Deutsch, A., Papakonstantinou, Y., and Petropoulos, M. 2009. Do-It-Yourself custom forms-driven workflow applications. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

Lakshmanan, V., Safris, F., and Subramaniant, I. 1996. Schemasql: A language for intereoperability in relational multi-database systems. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*. Morgan Kaufmann, 239–250.

Lau, T. 2001. Programming by demonstration: a machine learning approach. Ph.D. thesis, University of Washington.

Lau, T., Bergman, L., Castelli, V., and Oblinger, D. 2004. Sheepdog: learning procedures for technical support. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*. ACM, 109–116.

Lerman, K., Getoor, L., Minton, S., and Knoblock, C. 2004. Using the structure of web sites for automatic segmentation of tables. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 119–130.

Li, W.-S., Clifton, C., and Liu, S.-Y. 2000. Database Integration Using Neural Networks: Implementation and Experiences. *Knowledge and Information Systems 2,* 1, 73–96.

Lieberman, H. 2001. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Michelson, M. and Knoblock, C. A. 2007a. An Automatic Approach to Semantic Annotation of Unstructured, Ungrammatical Sources: A First Look. In *Proceedings of the International Joint Conferences on Artificial Intelligence Workshop on Analytics for Noisy Unstructured Text*. 123–130.

Michelson, M. and Knoblock, C. A. 2007b. Unsupervised information extraction from unstructured, ungrammatical data sources on the world wide web. *International Journal of Document Analysis and Recognition (IJDAR), Special Issue on Noisy Text Analytics 10,* 3, 211–226.

Milo, T. and Zohar, S. 1998. Using Schema Matching to Simplify Heterogeneous Data Translation. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 122–133.

Muslea, I., Minton, S. N., and Knoblock, C. A. 2003. Active learning with strong and weak views: a case study on wrapper induction. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 415–420.

Perkowitz, M. and Etzioni, O. 1995. Category translation: learning to understand information on the internet. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 930–936.

Raghavan, S. and Garcia-Molina, H. 2001. Crawling the hidden web. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 129–138.

Rahm, E. and Bernstein, P. A. 2001. A survey of approaches to automatic schema matching. *The VLDB Journal 10,* 4, 334–350.

Raman, V. and Hellerstein, J. M. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 381–390.

Reeve, L. and Han, H. 2005. Survey of semantic annotation platforms. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. ACM, New York, NY, USA, 1634–1638.

Riabov, A. V., Bouillet, E., Feblowitz, M. D., Lui, Z., and Ranganatham, A. 2008. Wishful Search: Interactive Composition of Data Mashups. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*. ACM, New York, NY, USA, 775–784.

Segre, A., Elkan, C., and Russell, A. 1991. A Critical Look at Experimental Evaluations of EBL. *Machine Learning 6,* 2, 183–195.

Sugiura, A. and Koseki, Y. 1998. Internet scrapbook: automating Web browsing tasks by demonstration. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*. ACM, 9–18.

Sutherland, W. R. 1966. The On-Line Graphical Specification of Computer Procedures. Ph.D. thesis, Massachusetts Institute of Technology.

Tallis, M., Kim, J., and Gil, Y. 2001. User studies of knowledge acquisition tools: Methodology and lessons learned. *Journal of Experimental & Theoretical Artificial Intelligence 13,* 4 (October), 359–378.

Tuchinda, R. 2008. Building Mashups by Example. Ph.D. thesis, University of Southern California.

Tuchinda, R. and Knoblock, C. A. 2004. Agent Wizard: Building Information Agents by Answering Questions. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*. ACM, 340–342.

Tuchinda, R., Szekely, P., and Knoblock, C. A. 2007. Building Data Integration Queries by Demonstration. In *IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces*. ACM, 170–179.

Tuchinda, R., Szekely, P., and Knoblock, C. A. 2008. Building Mashups by Example. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*. ACM, New York, NY, USA, 139–148.

Wong, J. and Hong, J. I. 2007. Making mashups with marmite: towards end-user programming for the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, New York, NY, USA, 1435–1444.

Woolson, R. and Lachencruch, P. 1980. Rank tests for censored matched pairs. *Biometrika 67,* 3, 597–606.

Xu, L. and Embley, D. 2003. Using domain ontologies to discover direct and indirect matches for schema elements. In *ISWC '03: Proceedings of the 2nd International Semantic Integration Workshop*. 105–110.

Yang, F., Gupta, N., Botev, C., Churchill, E. F., Levchenko, G., and Shanmugasundaram, J. 2008. Wysiwyg development of data driven web applications. *Proceedings of the Very Large Data Bases Endowment 1,* 1, 163–175.

Zloof, M. M. 1975. Query-by-example: the invocation and definition of tables and forms. In *VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases*. ACM, 1–24.

Received October 2008; Revised October 2010; Accepted December 2010

This document is the online-only appendix to:

Building Mashups by Demonstration

RATTAPOOM TUCHINDA
National Electronics and Computer Technology Center (Thailand)
CRAIG A. KNOBLOCK
University of Southern California
and
PEDRO SZEKELY
University of Southern California

ACM Transactions on the Web, Vol. 5, No. 3, July 2011, Pages 1–50.

A. NORMALIZED DATA

After normalizing the data, the final results in terms of minutes spent, segmented by subtasks, in each task are shown in table XIV, XV, and XVI. Note that since there are two subjects (i.e., No.3 and No.17) who did not have time to finish task 1, the result from these two subjects are be excluded from results that involve task 1 to ensure a fair comparison.

Table XIV. Normalized data for task 1. E stands for data extraction, M stands for source modeling, and C stands for data cleaning. The asterisk indicates time substitution when failures happen. The data is reported in minutes. The first twenty subjects have programming background, while the last three subjects have no programming background.

Task1	Dapper/Pipes				Karma			
Subject	E	M	C	Total	E	M	C	Total
No.1	*5:00	0:20	*5:00	10:20	2:19	1:08	1:00	4:27
No.2	1:43	0:30	*5:00	7:13	1:00	0:40	0:29	2:09
No.3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
No.4	0:52	0:48	*5:00	6:40	1:12	1:00	0:50	3:02
No.5	5:00	0:35	*5:00	10:35	1:15	1:18	1:20	3:53
No.6	2:30	0:15	*5:00	7:45	1:00	0:54	0:28	2:22
No.7	1:20	0:22	*5:00	6:42	0:51	0:51	0:46	2:28
No.8	1:40	0:14	*5:00	6:54	1:04	0:41	0:33	2:19
No.9	1:26	0:16	*5:00	6:42	1:14	1:00	1:10	3:24
No.10	1:39	0:10	*5:00	6:49	0:53	0:42	0:50	2:26
No.11	2:00	0:19	*5:00	7:19	1:04	1:00	0:53	2:57
No.12	2:00	0:49	2:00	4:49	1:07	1:00	0:40	2:47
No.13	2:00	0:05	*5:00	7:05	0:58	0:50	0:56	1:44
No.14	2:46	0:15	*5:00	8:01	1:12	0:45	0:48	2:45
No.15	2:27	0:14	3:11	5:52	1:10	0:49	1:20	3:19
No.16	1:16	0:12	*5:00	6:28	0:58	0:42	0:25	2:05
No.17	n/a	n/a	n/a	n/a	2:00	1:00	0:50	3:50
No.18	2:30	0:14	*5:00	7:44	1:06	1:10	1:46	4:02
No.19	1:38	0:47	1:20	3:45	1:20	0:49	0:35	2:44
No.20	1:30	0:16	*5:00	6:46	1:04	0:44	0:35	2:23
No.21	n/a	n/a	n/a	n/a	1:11	1:17	0:59	3:27
No.22	n/a	n/a	n/a	n/a	2:58	1:46	1:24	6:08
No.23	n/a	n/a	n/a	n/a	1:19	1:40	1:14	4:13

Table XV. Normalized data for task 2. E stands for data extraction, M stands for source modeling, C stands for data cleaning, and I stands for data integration. The asterisk indicates time substitution when failures happen. The data is reported in minutes. The first twenty subjects have programming background, while the last three subjects have no programming background.

Task2	Dapper/Pipes					Karma				
Subject	E	M	C	I	Total	E	M	C	I	Total
No.1	4:38	0:22	2:45	1:15	9:00	1:26	0:43	0:43	0:00	2:52
No.2	1:35	0:12	3:30	0:12	5:29	0:50	0:57	0:57	0:00	2:44
No.3	*5:00	0:25	*5:00	*5:00	15:25	2:52	1:00	3:00	0:00	5:52
No.4	4:49	0:17	3:29	0:38	9:14	1:26	0:48	1:03	0:00	3:18
No.5	*5:00	0:29	1:44	1:16	8:29	1:43	0:45	1:20	0:00	3:48
No.6	*5:00	0:20	*5:00	*5:00	15:20	2:07	0:30	0:50	0:00	3:27
No.7	2:17	0:15	4:46	0:18	7:36	1:13	0:25	0:52	0:00	2:31
No.8	3:23	0:21	*5:00	*5:00	13:44	1:10	0:21	0:24	0:00	1:55
No.9	4:11	0:21	*5:00	*5:00	14:32	1:22	0:47	2:11	0:00	4:20
No.10	2:16	0:07	3:14	0:20	5:50	1:04	0:20	1:06	0:00	2:30
No.11	3:04	0:17	*5:00	*5:00	13:21	1:06	0:34	0:53	0:00	2:33
No.12	2:00	0:27	*5:00	0:20	7:47	1:23	0:30	0:37	0:00	2:30
No.13	*5:00	0:07	1:43	0:10	7:00	1:42	0:32	0:41	0:00	2:55
No.14	3:03	0:23	4:42	0:10	8:21	1:40	0:31	0:56	0:00	3:07
No.15	2:06	0:12	3:13	0:22	5:53	1:30	0:24	2:05	0:00	3:59
No.16	3:58	0:11	3:29	0:27	8:05	0:51	0:17	1:00	0:00	2:08
No.17	4:15	0:28	3:39	0:30	8:52	1:04	0:28	1:18	0:00	2:50
No.18	*5:00	0:23	*5:00	*5:00	15:23	1:17	0:30	1:10	0:00	2:57
No.19	4:01	0:14	2:42	0:21	7:16	1:39	0:21	0:50	0:00	2:50
No.20	1:36	0:43	0:36	0:22	3:17	1:07	0:28	0:40	0:00	2:15
No.21	n/a	n/a	n/a	n/a	n/a	1:03	0:21	0:55	0:00	2:19
No.22	n/a	n/a	n/a	n/a	n/a	3:56	1:52	2:50	0:00	8:38
No.23	n/a	n/a	n/a	n/a	n/a	2:15	0:31	1:27	0:00	4:13

Table XVI. Normalized data for task 3. E stands for data extraction, M stands for source modeling, and I stands for data integration. The asterisk indicates time substitution when failures happen. The data is reported in minutes. The first twenty subjects have programming background, while the last three subjects have no programming background.

Task3	Dapper/Pipes				Karma			
Subject	E	M	I	Total	E	M	I	Total
No.1	1:30	0:26	*5:00	6:56	0:14	0:00	2:16	2:30
No.2	0:30	0:10	*5:00	5:40	0:25	0:00	0:26	0:54
No.3	1:00	0:15	*5:00	6:15	0:15	0:00	0:44	0:59
No.4	0:40	0:16	*5:00	5:56	0:20	0:00	1:06	1:26
No.5	0:40	0:14	*5:00	5:54	0:20	0:00	0:37	0:57
No.6	0:30	0:10	*5:00	5:40	0:20	0:00	0:31	0:51
No.7	0:27	0:10	*5:00	5:37	0:14	0:00	0:50	1:04
No.8	0:29	0:20	*5:00	5:49	0:30	0:00	0:51	1:21
No.9	0:40	0:23	*5:00	6:03	0:13	0:00	0:44	0:57
No.10	0:30	0:10	*5:00	5:40	0:20	0:00	0:35	0:55
No.11	0:51	0:20	*5:00	6:11	0:16	0:00	1:05	1:21
No.12	1:05	0:18	*5:00	6:23	0:30	0:00	0:46	1:16
No.13	0:31	0:14	*5:00	5:45	0:16	0:00	0:57	1:13
No.14	0:36	0:14	*5:00	5:50	0:14	0:00	2:00	2:14
No.15	0:26	0:21	*5:00	5:47	0:30	0:00	0:45	1:15
No.16	0:27	0:13	*5:00	5:40	0:15	0:00	0:56	1:11
No.17	0:33	0:38	1:56	3:07	0:30	0:00	0:46	1:16
No.18	1:03	0:07	*5:00	6:10	0:20	0:00	1:10	1:30
No.19	0:33	0:17	*5:00	5:50	0:25	0:00	1:20	1:45
No.20	0:18	0:13	*5:00	5:31	0:12	0:00	0:44	0:56
No.21	n/a	n/a	n/a	n/a	0:15	0:00	0:39	0:54
No.22	n/a	n/a	n/a	n/a	0:21	0:00	2:50	3:11
No.23	n/a	n/a	n/a	n/a	0:12	0:00	0:51	1:03