Building Data Integration Queries by Demonstration

Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock
Information Science Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
pipet@isi.edu, pszekely@isi.edu, and knoblock@isi.edu

ABSTRACT
The magnitude of data available on the web prompts the need for an easy to use query interface that enables users to integrate data from multiple web sources in an intelligent fashion. Past work in the area of databases has resulted in different query interface systems that simplify query formulation. While these approaches reduce the user's effort to compose queries, the user is still required to pick data sources to use and the interaction is not guaranteed to yield a non-empty result set. We introduce a novel approach that exploits the structure of the relational data source(s) to formulate a set of constraints. These constraints are used in conjunction with partial plans to produce an intelligent query interface that (a) does not require the user to know details about data sources or existing values (b) suggests valid inputs to the user (c) creates consistent queries that always return values.

ACM Classification Keywords
H.5.2 User Interfaces: Theory and Method

General terms: Algorithms, Design, Human Factors.

Keywords
Intelligent query interface, query by example, information integration.

INTRODUCTION
We need information to make good decisions. With the proliferation of the Internet, most information can be found on the Internet today. Examples of such information include maps, statistics about events around us, and reviews of goods and services from multiple vendors. Accurately integrating the information available on the Internet can provide valuable insights useful in decision-making. However, the information we need is usually scattered among multiple websites. It is very time consuming to access, combine, filter, and make sense of that data manually. For example, a particular restaurant might receive rave reviews from a restaurant review website, but has a 'C' rating on a government's 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.

For computer literate users who can comfortably use computers, but are not trained in programming, their choices are limited to (a) finding the information on their own by browsing web sites, or (b) relying on the data integration providers to supply web interfaces to access the integrated information. One example of a data integration provider is Zillow (http://www.zillow.com). Zillow is a website that helps home buyers/sellers by offering real estate information (i.e., property tax and historical pricing) by integrating the information from various web sites. Users can research houses on this site through an interactive map interface. However, the inherent problem in any data integration service is determining its users' needs; what a service offers might not satisfy the needs of all of its users. For example, many areas in southern California previously had oil wells, and buying a house in such areas may cause problems later. However, it would be impossible for Zillow to cover every aspect of the house buying process. In the end, a do-it-yourself integration interface that can be used by a large number of users is needed.

We envision a user interface where any computer literate user could easily build his/her own mashups, a service that integrates information from multiple data sources. Accomplishing this goal requires knowledge in multiple research areas to solve four problems:

- Data retrieval: This problem concerns data extraction from a website. While the semantic web has received a lot of attention, websites still require a wrapper, an agent that uses information extraction techniques [6,12], to convert the data from HTML into structured form.
- Data cleaning and schema matching: the data retrieved from multiple sources must be cleaned (i.e., fix misspellings and resolve format inconsistencies) and aligned. For example, schema matching techniques [10] are used to map an attribute "from" from airline websites to an attribute "city" from hotel websites because both attributes refer to the same type of entity.

IUI'07, January 28–31, 2007, Honolulu, Hawaii, USA.

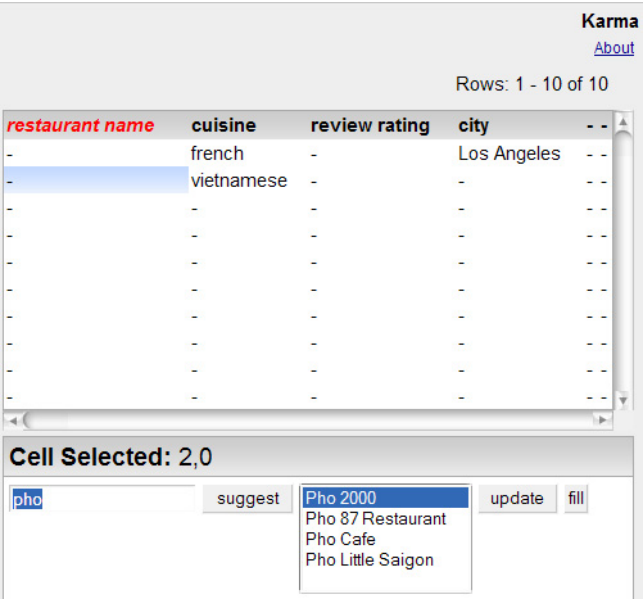


Figure 1: A snapshot of Karma. A user interacts with Karma by entering a partial value to get valid possible suggestion. Based on the selection, the next value or attribute is constrained to a limited set of values. At any time, the user can select 'fill,' and Karma will fill all the remaining slots.

- Data Integration: Assuming that the data is clean and aligned, we can treat the problem of combining the data from multiple sources similar to the way we combine the data from multiple databases.
- Filtering and visualization: once we combine the data, we need to present it in a way that is easy to digest. Depending on the type of the data, we might opt to use a single widget (i.e., a table, a map, or a graph) or a combination of widgets. Each widget works well with different filtering paradigms. For example, drawing a bounding box to select a subset of points works well for a map, but the same technique cannot be applied to a table. As a result, we consider the filtering problem to be a part of data visualization domain.

Our work in this paper is part of a larger effort that aims to fulfill the vision above. Specifically, this paper addresses the data integration aspect by providing an intelligent query interface that allows a computer literate user to integrate and easily query multiple sources by entering examples of data they want to see. Figure 1 shows our intelligent interface called Karma where a user can enter values in a method similar to Google Suggest (http:// labs.google.com/suggest); Google Suggest suggests a list of possible words and search terms when given a partial keyword search. Once the user provides some sample data, Karma will translate partially-filled rows into queries that retrieve data from multiple sources, and fills in the table. Karma's novel approach of using constraints and partial plans allows the user to enter values without any knowledge about query language or data sources. Moreover, Karma generates consistent queries that always return data.

The rest of the paper is organized as follow: we first describe our goals and requirements, the example scenario, and the intuition behind our approach. Then we show how to formulate the approach of exploiting the table structure by utilizing partial plans. Next, we provide a quantitative evaluation comparing our system with Query by Example. We then discuss advantages and limitations of our system. Next, we cover related work in the database and data integration disciplines. Finally, we review our contribution and plans for future work.

APPROACH

It has been mentioned in multiple cognitive psychology works [14,20] that "people retrieve information from their own memory by iteratively constructing partial descriptions of the desired target item" [21]. This concept is also used in other domains, such as mix-initiative planning [13], where a user provides partial solutions to the problem and lets the planner decide how to connect all the links together. Using the same approach, we propose a paradigm where the user interactively fills in a few values, and lets the system figure out the connections between the values, using its knowledge about available data sources, and fills in the rest.

Goals and Requirements

We have two main goals. The first goal is that a computer literate user should be able to use Karma without having to write complicated queries. The second goal is that Karma should be applicable to web sources. From our two goals, we create the following requirements for our system:

- Our system will act like a black box where we will abstract away the details about the data sources and their linking relationships. All users need to do is enter values – just like when they enter keywords in a search engine.
- Our system will suggest the matching keywords to the user in a way that preserves the integrity of the end result queries; i.e. the queries translated from the interactions with the user will always return a non-empty set. Once the user enters a value in a row, our system will guarantee that each empty cell in any partially-filled rows can be filled.

The logic behind our requirements results from the differences between databases and web sources. In traditional databases, we have a limited set of tables and database users are somewhat familiar with the schema of the database. On the other hand, the number of web sources is huge and users may not know which web sources to use and what attributes are available in each source.

Example Scenario

Suppose we want to find good French and Vietnamese restaurants in Los Angeles. The additional criterion is that these restaurants must have an 'A' health rating. In reality,

restaurant review sites do not contain the health rating information. Thus, combining data from multiple sources is necessary. With these criteria in mind, we let the user create a table by interactively filling in values (both attribute names and/or values) in a table as shown in Table 1.

Restaurant name	Cuisine	Review Rating	City	Health rating
	French		Los Angeles	A
	Vietnamese		Los Angeles	A

Table 1. An example scenario where the user constructs a table filling in values that he/she knows.

From the user's point of view, Karma acts as a two-dimensional search engine that gathers the information from multiple data sources that fit the user's description. Karma differs from traditional Query by Example (QBE) [22] systems in the following ways:

(a) Starting from an empty table with no attribute specified, the user would choose a cell to work on by entering a partial keyword. As the user types in the attribute name or the value, Karma will suggest a possible set of candidates. For example, as the user type "pho" (Vietnamese noodle dish) in cell (2,0) of Figure 1, Karma might suggest restaurants that have "pho" as parts of their names.

(b) The user does not have to select the data sources or specify the join conditions before hand. The planning component will automatically determine what data sources to use and how to connect them.

(c) Karma has an invariant: in each stage of the interaction, Karma guarantees that at least one executable plan will return a result. For example, if Los Angeles does not have any Vietnamese restaurant with an A health rating, when the user tries to select the health rating on the second row, Karma will suggest a list of available health ratings for Vietnamese restaurants in Los Angeles and this list is guaranteed to be non-empty. The reason is that before allowing the user to select the attribute "health rating," Karma verifies beforehand that choosing that attribute will result in queries that return some possible values.

Once the user is satisfied, Karma will formulate queries in SQL format and retrieve the data to fill in the rest of the blank cells. We will provide more in-depth comparisons between Karma and QBE in the evaluation section.

Intuition

The intuition behind our approach is simple. Each data structure or process has its own constraints. By exploiting these constraints, we can narrow the search space of the solution. When the user selects a particular value, that value implies specific data sources and attribute names. These qualified data sources and attributes can be used to constrain the next value/attribute that the user may select.

Given a set of data sources in the relational form, we can "index" the data very much like search engines index web pages. We index all the values to construct a map $v \rightarrow \{(a,s)\}$ that maps the value (v) to the source (s) and the attribute (a) in that source where the value appear. A single value can be associated with multiple attributes names and data sources. For example, the mapping

"Los Angeles" → {(city, Zagat), (city, LA_health_rating), (hotel name, Orbitz)}

implies that a keyword value "Los Angeles" exists in a) the data source Zagat under the attribute "city" b) the data source LA_health_rating under the attribute "city" and c) the data source Orbitz under the attribute "hotel name." We also index all the attribute names to construct a map of $a \rightarrow \{s\}$; for each attribute name, we identify a list of data sources that contain such an attribute name. Once we index all the attributes and values, as the user types in a partial keyword, we can retrieve and suggest the possible set of candidates using SQL Boolean queries. Each attribute or value selected and filled by the user introduce global constraints that further limit the next attribute and value that the user can select. For example, when the user enters "Los Angeles" as one of the values, the number of data sources for that particular attribute are narrowed down to only three sources, and the number of possible distinct attributes are narrowed down to only two.

PLANNING AND CONSTRAINT FORMULATION

In this section, we first formalize the concept that we described in the intuition section.

Indexing tables and data source definition

Let:
S: a set of all available web sources .
A: a lookup hashtable with its key and value being $a \rightarrow \{s\}$ where, 1) $\{s\} \subseteq S$ and 2) $\forall s \subset S : a \in att(s)$
V: a lookup hashtable with its key and value being $v \rightarrow \{(a,s)\}$ where $\forall (a,s): v \in val(a,s) \wedge a \in att(s)$
$att(s)$: a procedure that returns the set of attributes from the source s
$val(a,s)$: a procedure that returns the set of values associated with the attribute a in the source s.

By indexing attributes and values into A and V hashtables, Karma can deduce the information about possible values and attributes to suggest to the user. We will walk through the example scenario and show how we use constraints and partial plans to formulate complex queries. We will start from a simple task of building a one column table and then move on to more complicated cases of building a multi-column table. We will assume that we have the following data sources in our scenario:

Zagat(*$restaurant name, $cuisine, $address, $city, $state, $zipcode, review rating*)
Asian_food_review(*$restaurant name, $cuisine, $price, $address, $city, $state, $zipcode, review rating*)
LA_health_rating(*$restaurant name, $address, $city,*

$state, $zipcode, inspection date, health rating)
EU_country_info*($country name, language, population, gdp, date, location)*

Zagat and Asian Food Review are both restaurant review sources. However, Asian Food Review only contains Asian food. EU Country Info is a data source that contains information about European Union countries. Note that EU Country Info may not be related to food. However, we introduce this data source to show how the values from different attributes and data sources can be overlapped, which often occurs in a repository of many sources.

Each data source has a set of attributes associated with it. For example, Zagat is a table with 7 attributes. The "$" sign determines the primary key constraint, which we will explain later. Note that we made an assumption earlier that all the data sources are schematically aligned. As a result, the attribute names defined here are uniform (e.g., we use *"$address"* in all the definitions rather than the possibly different attribute names that the sources may have had before alignment).

Building a single column table

With a blank single column table, the user can choose to enter an attribute or a value. Let us assume that the user

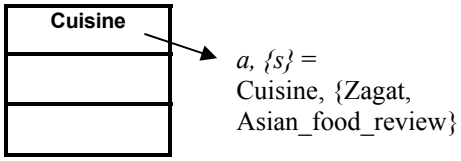


Figure 2: Building a single column table starting by entering an attribute first

chooses to enter an attribute first as shown in Figure 2. Since we do not have any constraints to start with, the attributes that can be suggested are all keys $\{a\}$ from A. Assume that the user chooses Cuisine and its associated sources are *Zagat* and *Asian_food_review*. At this point, if the user wants to enter a value, a possible list of values is constrained by the attribute selected and its associated data sources. As a result, the set of possible values that Karma can use to suggest the user is:

$$\{v\} = \text{val}(a,s) \text{ where } s \subset \{s\}$$

This mathematical expression can be translated into a query as:

(SELECT Cuisine FROM Zagat) UNION
(SELECT Cuisine FROM Asian_food_review)

Karma also allows the user to start by entering a value without having to specify any attribute as shown in Figure 3. Since we do not have any constraint for the first value, the possible suggest candidate set is simply all $\{v\}$ from V. Let us assume that the first value that the user selects is "French" and that its associated $\{(a,s)\}$ is {(Cuisine, Zagat), (Language, EU_country_info)}; this means that the value "French" exists in two data sources and can be associated to two attribute names – French can either be a Cuisine or a Language.

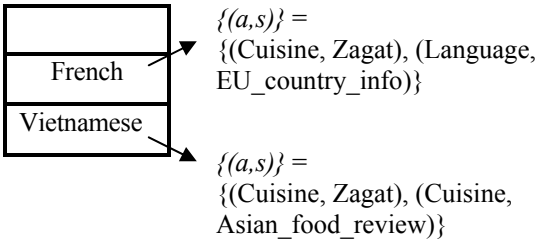


Figure 3: Building a single column table starting by entering a value first.

The value entered creates a constraint that limits the set of possible attributes and the set of values in the next move. For the sake of simplicity, we will postpone describing what the constraint is and assume that a second value "Vietnamese" satisfies the constraint.

At this point, we can describe the constraint that dictates the possible candidate set of the attribute name. If we only have "French" in the table, the possible candidate set for the attribute name is {Cuisine, Language}. However, once we have both values ("French" and "Vietnamese") in the table, the possible candidate set for the attribute name must be attributes shared by both values. We formalize this idea as the set intersection constraint.

The set intersection constraint

We compute the attribute candidate set using the set intersection. The set intersection of the possible attribute name consistent with each row in Figure 3 is simply: Cuisine.

At any point in time, if the attribute name is not defined, the possible candidate set for attribute names is:

$$\{x\} = \text{Set intersection}(\{a\}) \text{ over all the value rows.}$$

Now, we can revisit the constraint that we ignored when the user wants to enter a second value after having entered the first value "French". To enter a value, the possible value set is:

$$\{v\} = \text{val}(a,s) \text{ where } a \in \{x\} \wedge$$
$$s \text{ is any source where } att(s) \cap \{x\} \neq \{\}$$

To enter a value, we first compute a set of the set intersection (which can be more than one attribute) between all value rows. Then we retrieve a candidate set from *any* data source that contains any attribute in the set intersection. The query to retrieve the candidate set for the second value ("Vietnamese") is:

(SELECT Cuisine FROM Zagat) UNION
(SELECT Language FROM EU_country_info)

After the user chooses "Vietnamese" as a second value, if the user wants the enter the third value, the query to retrieve the candidate set will be:

(SELECT Cuisine FROM Zagat) UNION
(SELECT Cuisine FROM Asian_food_review)

Note that

1.) The query for the third value does not include the attribute Language from the source EU_country_info because the attribute Language does not belong to the updated set intersection.

2.) As the user enters the attribute or values, the possible choices are becoming more limited.

3.) Once the choices are narrowed down to only one value (i.e., the set intersection of the attribute between "French" and "Vietnamese" = Cuisine), Karma will fill that attribute/value in the table automatically. In our case, once the user selects "Vietnamese," Karma will also update the unfilled attribute name to be "Cuisine."

Building a multiple-column table

The multiple-column case is more complex, because a value entered in one column can affect how Karma suggests attributes and values in other columns. First, we will introduce the concept of "reachable" attributes. Then we will describe partial plans. We will show different examples to explain each concept.

Computing 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 4 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, salary, name, address, phone number, latitude, and longitude.

We define a "reachable" attribute as an attribute that can be reached from a particular data source (i.e., longitude is reachable from S1).

If we have a well-defined database like the one shown in Figure 4, 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

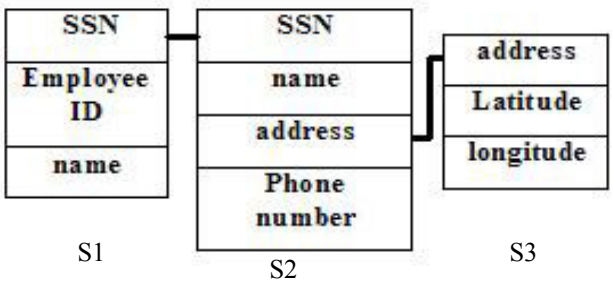
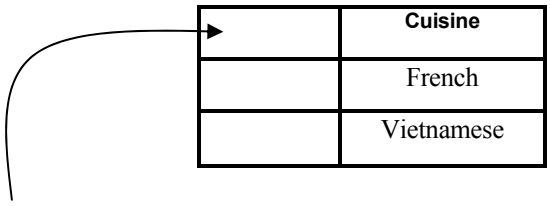


Figure 4: Joining condition through foreign keys in traditional database

with name "John Smith" in S1 and S2 who are completely different people with different SSNs. In Karma, we use a primary key constraint. The set of attributes with the *$* in the data source model acts as primary keys. For example, the primary key constraint for the LA_health_rating source is: *$restaurant name, $address, $city, $state,* and *$zipcode*

The constraint means is that if we want to retrieve *inspection date,* and *health rating*, the join condition must be over *restaurant name*, *address*, *city*, *state*, and *zipcode*.

In many websites, to retrieve the information, we need to fill out a web form. For example, to get a health rating for a particular restaurant, we need to fill out a form (i.e., restaurant name, address). For a wrapper to retrieve the data from these sites, it will also need this information, hence the input requirement. In Karma, we retain this input requirement information and use it as the primary key constraint. The reachable attribute definition allows us to handle the case when the user wants to enter a new attribute in a multi-column table.



Set intersection: { *restaurant name, cuisine, address, city, state, zipcode, review rating*, *inspection date, health rating* }

Figure 5: Entering an attribute in the new column require a computation of the set intersection of "reachable" attributes between each row.

Continuing from the single column case (Figure 3), if the user wants to enter the new attribute as shown in Figure 5, the possible set of valid attributes must be reachable from each row in the table (i.e., both the "French" row and the "Vietnamese" row). For example, the attribute price is only reachable from the "Vietnamese" row, because "price" is the attribute that belongs to the Asian_food_review data source. If we allow price to be used as the new attribute, we will not be able to suggest any value for the "French" row, because price is not reachable from that row (since the first row does not contain Asian_food_review).

As a result, the first constraint for entering a new attribute in the table is:

1.) The set intersection of "reachable" attributes of all partially filled rows.

However, simply matching the primary key constraint between data sources does not guarantee that all the attributes in the set intersection will create a join condition that does not return the empty value. The second constraint that Karma imposes is:

2.) Each attribute in the set intersection of the "reachable" attributes set must produce a non-empty suggested value set.

This can be accomplished by executing multiple queries through partial plans as if Karma wants to suggest possible values for each attribute from (1) in each partially-filled row, and eliminate the ones that produce an empty set from the list of "reachable" attribute list. This constraint allows Karma to guarantee that for each empty cell in a non-empty row, Karma can fill those cells.

Partial Plans

Assuming that the user selects "restaurant name" for the new attribute in Figure 5, the user might want to (a) have Karma fill the blank cells in the table. (i.e., find all the restaurants with Cuisine = French or Cuisine = Vietnamese) or (b) fill in some cells under the column "restaurant name" by letting Karma suggest what are the available choices (as in Figure 1). In both instances, Karma uses partial plans to determine what data should be used to either fill the table or suggest values to the user.

Each individual row in the Karma table contains a partial plan in the form of a tree that keeps track of selected values and attributes. Each row can contain more than one partial plan depending on the number of data sources. For example, the second row in Figure 3 would contain two partial plans (one plan using Zagat, and the other using Asian_food_review).

The partial plan tree can easily be translated into queries to retrieve the candidate values to suggest to users. First, we will show the final tree (Figure 6) that Karma constructs for the first row from the table in Table 1. Then, we will explain (1) node types and parent types (2) how we use the tree to evaluate the set of possible suggestions that can be used to suggest a value for a cell or to fill the whole table, and (3) how we construct such a tree.

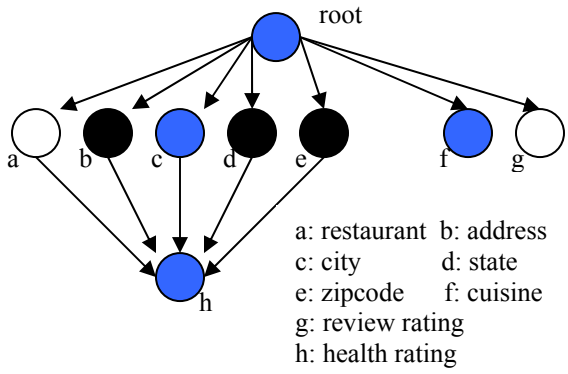


Figure 6: the partial plan for the first row of the table in Table 1 that includes joining between two data sources according to the primary key constraint.

Node types and parent types

Each node represents a cell in a row, except for a root node, which is used as a starting point. There are three types of nodes: a value node (blue), a place holder node (white), a hidden node (black). The value node is the node where the user has already specified a value in that cell (i.e., "French", "Vietnamese"). The place holder node is a node where the user has not selected any value yet, but the attribute for that column is already specified. For example, an empty cell on the first row under the attribute "restaurant name" in Table 1 corresponds to a place holder node. The hidden node is a node that is required as a part of the joining condition through the primary key constraint, but is not selected into the table by the user (nodes b,d,e). Each node will contain a triplet of attribute, source, and value. For example, node f will contain f(*a,s,v*) = (cuisine, Zagat, French). While node a (a place holder node) will contain a(*a,s,v*) = (restaurant name, Zagat, _PLACE_HOLDER). When the user first enters a value, we designate its corresponding data source as a starting source. Any node with its *s* equal to the starting source has the root as its parent, while a node that is the result of a joining data source will have multiple parents. Those parent nodes are essentially nodes with primary key attributes.

Tree evaluation

The partial plan tree can be used to suggest a particular set of available values by translating it into a query for a particular node. This process can also be used to determine whether a particular attribute will produce a non-empty suggest value set or not. To translate a tree into a query, we use the following rules:

1. A value node implies a value equal "=" condition.

2. A place holder node can only be included in the SELECT part of the query

3. A node with multiple parents implies a join condition over its parents.

For example, a possible candidate set for the attribute "restaurant name" (a place holder node *'a'* in Figure 6) in the first row is:

```
SELECT DISTINCT Zagat.'restaurant name'
FROM Zagat, LA_health_rating as L
WHERE Zagat.city = "Los Angeles" AND
Zagat.cuisine = "French" AND
L.'health rating' = "A" AND
Zagat.'restaurant name'=L.'restaurant name' AND
Zagat.'address' = L.'address' AND
Zagat.'city' = L.'city' AND
Zagat.'state' = L.'state' AND
Zagat.'zipcode' = L.'zipcode';
```

The first three conditions are the constraints imposed by value nodes (rule 1), while the rest are the constraints imposed by joining conditions (rule 3).

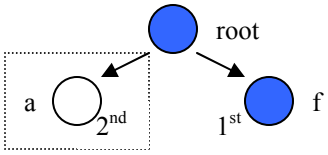
Note that we can also create a query that fills the whole table by retrieving the possible set by including: city, cuisine, review rating, and health rating in the SELECT part of the query above.

Tree Construction

In our scenario, the user starts in the single column table with an empty table and selects "French" in a cell. This is when

we add the first value node f into the tree. On the other hand, if the user starts by selecting an attribute first, we will add a place holder node instead, because the value for that cell has not been specified by the user yet.

To add the first node, the set intersection constraint must be satisfied. Note that it is possible that (a) the size of the set intersection is more than one (the value selected corresponds to more than one attribute name), or (b) the corresponding data sources are more than one. In such cases, we permute to create multiple partial plan trees for each row. As the user enters more values Karma eliminates attributes/data sources according to the set intersection constraints, and removes the partial plan trees that refer to the eliminated attributes/data sources. For example, (Language, EU_country_info) is eliminated when we compute the set intersection constraint in Figure 3. From the second node on, we are dealing with a multiple-column table case. For example, to add a second node (node *a*) in Figure 7:



a(*a,s,v*) = (restaurant name, Zagat, _PLACE_HOLDER)
f(*a,s,v*) = (cuisine, Zagat, French)

Figure 7: The 1st row partial plan generated by Karma as the user enters attributes and values.

1. Compute the set intersection of the reachable attributes of all the rows. Then, keep only reachable attributes that return non-empty suggest value sets.

2. At this point, we have a set of attributes that Karma can suggest to the user. If the user selects the attribute (as in Figure 5), create a place holder node (node a in Figure 7). If the attribute is in the same data source as the starting source, add the new node as the child of the root. However, if the attribute is in a different source, create necessary hidden nodes according to the primary key constraints and set the new node as the child of those hidden nodes.

3. On the other hand, if the user decides to enter a value instead of selecting the attribute in Figure 5, permute the partial plan for the set intersection attribute set in (1). At this point, we will have multiple partial plans similar to that of a plan in Figure 7, although the attribute of the node *a* will be permuted over all set of reachable attributes. We can evaluate each plan to retrieve the possible values from node *a* of all partial plans. The union of the result will be used as a suggest set for the user. Once the attribute can be determined, remove the partial plans that do not correspond to that attribute.

We have demonstrated how the set intersection and the reachable constraints can be applied to partial plans. This approach allows the user to select any cell in the Karma table, get a list of suggestions, and once satisfied tell the system to fill the remaining part of the table with the data that satisfies all the constraints.

EVALUATION

In this section, we perform analysis of the effort that it would take a user to do a typical query. Our baseline comparison tool is Microsoft Access, which integrates the QBE approach into its query design view.

Claim and Hypothesis

Throughout the paper, we have made qualitative claims that our approach does not require the user to know about the query formulation, the data sources, or the schema of those data sources. Based on this claim, we formulate the hypothesis that any user can finish the task of integrating data from multiple data sources faster using Karma.

Scenarios

In our experiments, we measure the performance of Karma using three scenarios:

a.) *Retrieving the restaurant health rating*: This scenario only retrieves data from one data source (LA_health_rating).

b.) *Retrieving the restaurant information with reviews and health ratings*: This scenario includes the join between two data sources, but limit the number of sources so there would be no union operation.

c.) *Retrieving the restaurant information with reviews and health ratings*: This is the scenario example that includes union and join as discussed in the previous section.

Experimental Setup

We record the number of optimal mouse clicks and keystrokes required to complete the task in each of the systems. Moreover, we also define the cost for each of the operations required to complete the task below:

Typing in a value or Selecting a value = 1t unit

Selecting a data source to use = 1d unit

Selecting an attribute = 1a unit

Results reported

	Clicks (c) and Key Strokes (k)	Cost
QBE A	28c+16k	4a+2t
Karma A	17c+4k	3a+2t
QBE B	39c+28k	5a+3t+2d
Karma B	25c+7k	3a+3t
QBE C	78c+54k	2*(5a+6t+2d)
Karma C	37c+14k	3a+6t

Table 2: the data collected from the experiment. A and B designate different scenarios as defined earlier

From Table 2, the number of clicks in Scenario A and B is comparable in performance between the two systems. In general, the clicks are for selecting attributes and specifying condition in those attributes.

In scenario C, the number of clicks in QBE is high because the QBE grid cannot represent three kinds of queries: union query, pass through query, and data definition query[1]. In scenario C, we have the overlapping data from both Zagat and Asian_food_review. Therefore, the union operation is required. As a result, we need to repeat the whole process twice to finish the task in MS Access, resulting in roughly doubling the clicks required. However, even in tasks that do not require union, the number of clicks in Karma is still less than that of QBE.

In terms of the number of keystrokes, Karma suggests possible values when at least three characters are specified. On the other hand, QBE requires a full value to be typed in. The more value selection criteria, the more Karma saves in terms of the number of keystrokes.

In terms of the cost, if we only want to retrieve data from a single data source (scenario A), the performance of Karma and QBE are comparable. However, in the case that involves data integration from multiple data sources, we can clearly see that Karma costs less in terms of operations to perform the task. First, a user does not have to perform the same task twice with Karma when the task involves the union operation (Scenario C). Secondly, Karma suggests and fills some attributes automatically. Finally, using Karma does not incur the cost of selecting the data source (Scenario B-C). The last advantage is very important. Imagine a database containing 1,000 tables, some of which may be overlapping; it would take even an expert a lot of time to locate the data sources he/she needs.

DISCUSSION

We use the Google Web Toolkit to implement our intelligent query interface, Karma. This toolkit provides easy to use AJAX and client-server libraries. Also, the code, written in Java, is automatically translated to Javascript for easy web deployment.

Our approach is based on a simple idea that every problem has a structure that dictates the constraints. Once we find the constraints, we use these constraints to limit the search space of the solution. Our work is an example of how past approaches underutilize the information from the structure of the problem that, once exploited, can reduce the user's time and knowledge requirement to perform a task. We describe the advantages and limitations or our approach below.

Advantages

Consistent Query Generation: The problem in many query systems is that the user can formulate a query, but that query is not guaranteed to return a non-empty result set. Our approach uses the set intersection and reachable constraints to ensure that if we suggest a new attribute, we can fill every row in that column with non-empty values. Since we only allow the user to select attributes/values from the list of possible attributes/values, there is no way a user can make a mistake.

True Query by Example: Our approach is a true query by example. By abstracting away the data sources and their linking information, the user only needs to enter values to create a complex query.

Monotonically Decreasing Property: Once every attribute in the table is defined, the space of the possible solutions decreases every time the user enters a value. Each value entered constrains other values through the set intersection constraint, which results in the reduction of the number of partial plans and the conversion of a place holder node into a value node.

Limitations

Primary Keys Requirement: As discussed in the multiple-column cases, we need primary keys between data sources to compute reachable attributes. For web sources, we can use the wrapper input requirement as the primary key constraint. However, if we want to integrate a data source that is not a wrapper, we will need an expert to label the primary key of that particular source.

Filtering: Our approach does not allow comparison conditions (i.e., < and >) or aggregate conditions (i.e., avg and max). Without these capabilities, however, the user can still generate complex queries that retrieve and integrate data from multiple data sources. The decision not to support these operations is a strategic one. We believe that filtering and visualization problems are more closely related (as explained in the introduction section) and we will address them as a part of the overall vision in future work.

Scaling: In terms of indexing all the attributes and values into *A* and *V* lookup tables, we believe that it is feasible on the WWW scale; Google suggest is one such example. Our limitation in terms of scaling belongs to computing reachable attributes. Given enough attributes and the right set of primary key constraints, we can link data sources together in a very long chain. Since our suggest set relies on evaluating and verifying partial plans to eliminate the ones that return empty data, the size and the number of partial plans can be exponentially high. Currently we limit the length of the chain to be only two (the partial plan tree can only have a maximum depth of three).

RELATED WORK

The idea of inferring procedures or queries by example is not new. The work that is most closely related to our work is Query by Example [22], which we discuss in the evaluation section. Aside from using examples, there are a variety of

[1] http://www.fontstuff.com/access/acctut14.htm

approaches using different techniques, such as programming by demonstration, filtering through machine learning, question-answering, planning, graphical visual query languages, and retrieval by formulation.

Programming by demonstration [2,7] automatically builds a program to perform repetitive tasks by observing how the user interacts with an arbitrary system. While this approach is highly effective in multiple problem domains [8,9,16], the assumption of this approach is that the user knows what he/she is doing. On the other hand, Karma assumes that the user might not know which data sources are available and needs help from the system to suggest possible values. Furthermore, the tasks in Karma are not repetitive, because every selection requires human judgment.

Query by Browsing (QbB) [3] uses machine learning approaches like decision trees and genetic algorithms to deduce the query by letting the user provide positive/negative sample rows. The user would label some rows of data as correct or incorrect, and QbB will create a query that fit the data set. However, QbB is intended to be used as a filtering tool on one single table and is not useful when integrating the data from multiple sources.

Agent wizard [19] and Artermis [18] use the question-answering technique to build a plan that integrates multiple data sources. The user would answer a set of questions and these systems will incrementally build a plan from each answer. These systems also provide simple filtering conditions and monitoring capabilities (since the data from the web sources can change their values). However, the user has to answer many questions. In Agent Wizard, the minimum number of questions for a task that integrates two data sources is 17, including questions for selecting attributes, data sources, and filtering conditions. In Karma, the user does not have to know about the data sources available and can build the plan by entering values.

Prometheus [17] is an example of a mediator system that uses planning techniques. The user needs to formulate a query according to the domain models defined by experts. The mediator will analyze the query and automatically determine which data sources will be used to retrieve the data. This approach is powerful because it uses planning techniques to determine an optimal set of data sources with minimum data overlapping. However, extensive knowledge about the domain model is required and the system is not guaranteed to return a non-empty result from a query.

Graphical visual query languages [1,5,15] use graphs to build a query. A graph contains nodes that represent attribute names and data source names, and links that represent the relationship between nodes. The relationship can be a source condition (from), source operations (union and join), or aggregate conditions (i.e., avg, max). While in these systems the user needs to specify nodes and links in these systems, Karma induces the graph automatically from the values that the user enters.

RABBIT[21] and HELGON[4] are query interface systems that use the retrieval by formulation technique. The user would supply a partial description of the data he/she wants. These systems will retrieve matching data, which the user can criticize. Based on the user criticism, the system will reformulate the query to retrieve different data that may satisfy the user's goal. While these systems can suggest values to the user like Karma, the user needs to know which concept to use to form a partial description and navigate through the hierarchy of the database to select the value needed.

CONCLUSION AND FUTURE WORK

Our contribution in this paper is an approach to data integration with the following characteristics (a) does not require the user to know details about data sources or existing values (b) suggests valid possible values to the user (c) creates consistent queries that always return values. While past work addresses the same vision of allowing the user to complete a task without programming, Karma is the only system that contains these three characteristics.

In terms of future work, we plan to address other problems in our unified vision outlined in the introduction. Our next priority is to formulate a general framework for filtering and visualization. Depending on the type of data, different visualization and filtering techniques should be used. Using the same concept that a structure implies constraints, we believe that we can use an ontology to help us simplify the framework for filtering and visualization.

ACKNOWLEDGMENTS

This research is based upon work supported in part by the National Science Foundation under Award No. IIS-0324955, in part by the Air Force Office of Scientific Research under grant number FA9550-04-1-0105, and in part by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

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

1. Benzi, F. , Maio, D., and Rizi, S.,. VisTool: a visual tool for querying relational databases. *In Proc. Of Advanced Visual Interface*, ACM Press, 1998
2. Cypher, A., Watch what I do: Programming by demonstration, 1993. MIT Press

3. Dix, A., Interactive Querying - locating and discovering information, *Second Workshop on Information Retrieval and Human Computer Interaction 1998*.
4. Fischer, O. and Nieper-Lemke, H., HELGON: Extending the retrieval by reformulation paradigm. *In Proc. of ACM CHI'89*. 1989.
5. Haw, D., Goble, H., and Rector, A., GUIDANCE: making it easy for the user to be an expert. *In Proc. of International Workshop on Interfaces to Database 1994*. Springer-Verlag. 19–43.
6. Knoblock, C.A., Lerman, K., Minton, S. and Muslea, I.. Accurately and reliably extracting data from the web: A machine learning approach, *Intelligent Exploration of the Web*. Springer-Verlag, 2003.
7. Lau, T., Programming by Demonstration: a Machine Learning Approach, PhD thesis, University of Washington, 2001.
8. Lau, T., Bergman., L., Castelli, V., Oblinger, D., Programming shell scripts by demonstration, *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI 2004*.
9. Lau, T., Bergman., L. Castelli, V., Oblinger, D., Sheepdog: Learning Procedures for Technical Support, *In Proc. IUI 2004*.
10. Lee, Y., Sayyadian, M., Doan, A., Rosenthal, A., eTuner: Tuning Schema Matching Software Using Synthetic Scenarios, *To appear in VLDB Journal Special Issue 2006.*
11. Michalowski, M., Ambite, J. L., Knoblock, C., Minton,S., Thakkar, S., and Tuchinda, R. 2004. Retrieving and semantically integrating heterogeneous data from the web. *IEEE Intelligent Systems* 19(3):72–79.
12. Michelson, M. and Knoblock, C.A., Phoebus: A System for Extracting and Integrating Data from Unstructured and Ungrammatical Sources, *In Proc AAAI-2006*.
13. Myers, K. L., Jarvis, P. Tyson, W. M., and Wolverton, M. J. 2003. A Mixed-initiative Framework for Robust Plan Sketching. *In Proc ICAPS 2003*.
14. Norman, D.A., and Bobrow, D.G. "Descriptions: An Intermediate Stage in Memory Retrieval," *Cognitive Psychology* 11 (1979).
15. Papantonakis, A. and King, P. J. H., Syntax and semantics of Gql: a graphical query language. *Journal of Visual Languages*, 6:3–25.
16. Sugiura, A., and Koseki Y. 1998. Internet Scrapbook: Automating Web browsing tasks by demonstration. *In Proceedings of UIST'98*.
17. Thakkar, S., Ambite, J.S., and Knoblock, C.A., Composing, optimizing, and executing plans for bioinformatics web services, *VLDB Journal, Special Issue on Data Management, Analysis and Mining for Life Sciences* 14,3(2005), 330-353.
18. Tuchinda, R., Thakkar, S., Yolanda, G., and Deelman, E. Artemis: Integrating Scientific Data on the Grid, *In Proc. IAAI 2004*.
19. Tuchinda, R. and Knoblock, C.A., Agent Wizard: Building Information Agents by Answering Questions, *In Proc. IUI 2004*.
20. Williams, M.D., and Hollan, J.D. "The Process of Retrieval from Very Long Term Memory," *Cognitive Science* 5 (1981), 87-119
21. Williams, M.D., and Tou, F.N., RABBIT: An interface for database access, *In Proc. ACM '82 conference* (1982), 83-87
22. Zloof, M.M. Query by example, *In Proc. National Computer Conference*, AFIPS Press. 1975