Automatically Incorporating New Sources in Keyword Search-Based Data Integration

Partha Pratim Talukdar
University of Pennsylvania
Philadelphia, PA, USA
partha@cis.upenn.edu

Zachary G. Ives
University of Pennsylvania
Philadelphia, PA, USA
zives@cis.upenn.edu

Fernando Pereira
Google, Inc.
Mountain View, CA, USA
pereira@google.com

ABSTRACT

Scientific data offers some of the most interesting challenges in data integration today. Scientific fields evolve rapidly and accumulate masses of observational and experimental data that needs to be annotated, revised, interlinked, and made available to other scientists. From the perspective of the user, this can be a major headache as the data they seek may initially be spread across many databases in need of integration. Worse, even if users are given a solution that integrates the current state of the source databases, *new* data sources appear with new data items of interest to the user.

Here we build upon recent ideas for creating integrated views over data sources using keyword search techniques, ranked answers, and user feedback [32] to investigate how to *automatically discover* when a new data source has content relevant to a user's view — in essence, performing *automatic data integration* for incoming data sets. The new architecture accommodates a variety of methods to discover related attributes, including *label propagation* algorithms from the machine learning community [2] and existing schema matchers [11]. The user may provide *feedback* on the suggested new results, helping the system *repair* any bad alignments or *increase the cost* of including a new source that is not useful. We evaluate our approach on actual bioinformatics schemas and data, using state-of-the-art schema matchers as components. We also discuss how our architecture can be adapted to more traditional settings with a mediated schema.

Categories and Subject Descriptors

H.2.5 [**Database Management**]: Heterogeneous Databases—*Data translation—schema alignment*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Relevance feedback*

General Terms

Algorithms, Human Factors, Performance

Keywords

Machine learning, user feedback, schema matching, schema alignment, keyword search, data integration

1. INTRODUCTION

Data integration remains one of the most difficult challenges in information technology, largely due to the ambiguities involved in trying to semantically merge different data sources. In an ideal world, the data needs of science, medicine, and policy would be met by discovering new data sets and databases the moment they are published, and automatically conveying their contents to users with related information needs, in the form relevant to those users. Instead, we live in a world where both discovery and semantic conversion are for the most part time-consuming, manual processes, causing a great deal of relevant information to be simply ignored. To address these difficulties, some research communities have attempted to define a consensus global schema (*mediated schema*) for their field so that individual sources can be mapped into a common representation. Researchers in machine learning, databases, and the semantic web have made significant progress in recent years on partially automating these mapping and alignment tasks [29].

However, the global-schema approach is poorly suited to automating the process of source discovery and integration in a dynamic scientific community. It is difficult to develop a consensus mediated schema that captures the diverse needs of a large user base and keeps up with new concepts, methods, and types of experimental result. Few mechanisms exist for *discovering* relevant sources as they are first published, and for having their data *automatically put into use*. Finally, schema alignment tools rarely *scale* to large numbers of schemas and relations, and it can be difficult to determine when they have produced the *right mappings*.

Our work on the Q system [32] develops an *information need-driven* paradigm for data integration, which addresses the above problems. Q is initially given a set of databases that contain known cross-references, links, and correspondence or cross-reference tables; it does not require a global mediated schema or full schema mappings. A user specifies an *information need* through a keyword query. Leveraging ideas from keyword search in databases [4, 5, 17, 20], Q defines a *ranked view* consisting of a union of conjunctive queries over different combinations of the sources. This view is made persistent and refined through user feedback.

In this paper we build upon Q's information need-driven integration model by addressing the challenge of **automatically adding new data sources** and relating them to the existing ones. As a user (or a Web crawler) registers a new database, that source's *relevance* to existing ranked views is considered, using information about data-value overlap as well as schema alignment costs from existing schema matchers. Going beyond our previous work [32], Q can now *combine* the weighted outputs from different schema matchers. If the source is found to be highly relevant to a ranked view, then query results are refreshed as appropriate. Now the users of the view may provide *feedback* on its contents: certain new results may be valuable, or possibly erroneous. As the system gets feedback about erroneous results, it adjusts the costs it has assigned to specific mappings or alignments so that associations responsi-

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

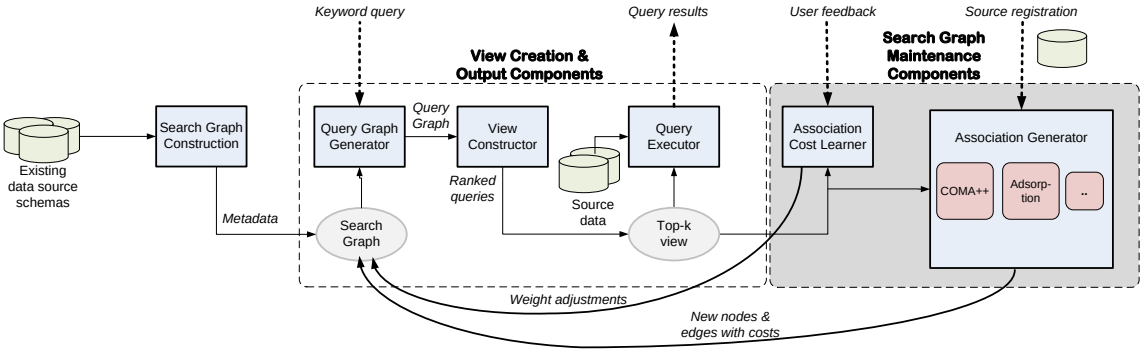

Figure 1: Basic architecture of Q. The initial search graph comes from the sources known at startup. At query time this is expanded into a query graph, from which queries and ultimately results are generated. The *search graph maintenance* modules, the focus of this paper, handle user feedback and accept new source registrations, in order to update the search graph with new alignments — triggering recomputation of the query graph and query results in response.

ble for the errors are avoided. Q can adjust weights for individual alignments, including how much to favor the outputs from different schema matchers.

Our work distinguishes itself from prior efforts in interactive, user-driven integration (e.g., dataspaces [14] and best-effort integration [30]) by *automatically* discovering semantic links among data sources, and using a data-driven approach to providing feedback to the system. Any form of automatic schema alignment is likely to make errors, especially at scale; the challenge is to determine when and where there are mistakes. Simply "eyeballing" the output mapping is unlikely to help identify what is correct. However, if a domain expert is looking at data from the perspective of a particular information need, he or she is (1) likely to invest some effort in ensuring the quality of results, (2) likely to recognize when results do not make sense.

We make the following contributions:

- We create a novel "pluggable" architecture that uses matching tools to create alternative potential alignments.
- We develop an automatic, *information need-driven* strategy for schema alignments that, for a given top-k keyword query and a new source, only aligns tables against the new source if there is potential to affect the top-k query results.
- We develop a unified representation for data values and attribute labels, using edge costs to measure relatedness; this facilitates both ranked querying and learning.
- We incorporate state-of-the-art alignment components from the database [11] and machine learning [33] literature, and show how to combine their outputs.
- We propose the use of a random-walk-inspired algorithm called *Modified Adsorption (MAD)* [31] to detect schema alignments, and study its effectiveness instead of, and in combination with, the COMA++ tool [11].
- We apply a machine learning algorithm called MIRA [10], to learn not only correct attribute alignments, but also how to combine information from multiple matching tools. Unlike the learning techniques applied in schema matching tools, our techniques are based on *feedback over answers.*

We experimentally evaluate our techniques over bioinformatics databases, demonstrating effectiveness of the proposed methods.

In Section 2 we review the data integration model of the Q system and describe our basic problem setup. Section 3 then presents our solution to the problem of determining when a *new source* is relevant to an existing view, through the use of focused schema alignment tasks. Section 4 describes how we learn to adjust the

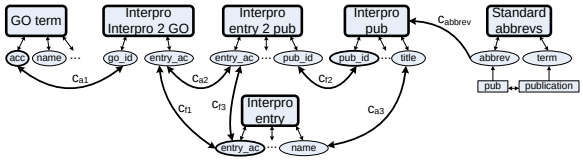

Figure 2: Search graph with weighted associations, indicated by bidirectional edges with cost terms c_i. Note the association between the table `pub`, the abbreviation `pub`, and the term `publication`, specified in the `abbrevs` table.

alignments among attributes, and their weights, from user feedback. We experimentally analyze our system's effectiveness in Section 5. We discuss related work in Section 6, before concluding and describing future work in Section 7.

2. SEARCH-BASED INTEGRATION

This paper adopts a *keyword search* query model [4, 17, 20, 32] in which keywords are matched against elements in one or more relations in different data sources. The system attempts to find links between the relations matching the given keywords. Such links are proposed by different kinds of *associations* such as foreign key relationships, value overlaps or global identifiers, similarity predicates, or hyperlinks. In general, there may be multiple relations matching a search keyword, and multiple attribute pairs may align between relations, suggesting many possible ways to join relations in order to answer the query.

Figure 1 shows the basic architecture of our Q system. We start with an initial *search graph* generated from existing data source relations and the associations among them. During the *view creation and output* stage, a keyword search is posed against this search graph, and results in a *top-k view* containing answers believed to be relevant to the user. The definition and contents of this view are maintained continuously: both the top-scoring queries and their results may need to be updated in response to changes to the underlying search graph made (1) directly by the user, who may provide feedback that changes the costs of certain queries and thus query answers; (2) by the system, as new data sources are discovered, and their attributes are found to *align* with the existing relations in the search graph, in a way that results in new top-k answers for the user's view. We refer to the process of updating the schema graph's nodes and associations as *search graph maintenance*. In fact, there is interplay between the two graph maintenance mechanisms and the view creation and output stage, as the system may propose an alignment, the view's contents may be updated, the user may provide feedback on these results, and the view output may be

updated once again. All of this is focused around alignments that are relevant to the user's ongoing information need.

2.1 Initial Search Graph Construction

Before any queries are processed, an initial *search graph* is created (leftmost module in Figure 1) to represent the relations and potential join links that we already know about. Q first scans the metadata in each data source, determining all attribute and relation names, foreign keys, external links, common identifiers, and other auxiliary information. The basic search graph (see Figure 2 for an example) consists of two types of nodes: relations, represented by rounded rectangles, and attributes, represented by ellipses. We add undirected edges between attributes and the relations that contain them (with zero-cost, indicated as thin lines with no annotations), and between tables connected by a key-foreign-key relationship (bold lines with costs $c_{f1}, \ldots, c_{f3}$) initialized to a default foreign key cost c_d.

The graph is extended with bidirectional *association* edges drawn from the results of hand-coded schema alignments (or possibly the results of schema matching tools, such as the ones we consider in this study, which are a *label propagation* algorithm and the COMA++ schema matcher). Such associations may be within the same database (such as those added between `InterPro2GO` and `entry2pub`, or `entry.name` and `pub.title`) or across databases. Each of these associations receives a cost ($c_{a1}, \ldots, c_{a3}$ in Figure 2) based on the alignment confidence level.

Each tuple in each of the tables is a virtual node of the search graph, linked by zero-cost edges to its attribute nodes. However, for efficiency reasons we will add tuples nodes as needed for query interpretation. Once the search graph has been fully constructed, Q is ready for querying, and ready to learn adjustments to the costs c_{ci}, c_{aj}, and c_{fk} or to have new association edges added.

2.2 Views from Keyword Queries

Given a keyword query $Q = \{K_1, \ldots, K_m\}$, we dynamically expand the search graph into a query graph as follows. For each $K_i \in Q$, we use a keyword similarity metric (by default tf-idf, although other metrics such as edit distance or n-grams could be used) to match the keyword against all schema elements and all pre-indexed data values in the data sources. We add a node representing K_i to the graph (see Figure 3, where keyword nodes are represented as boldfaced italicized words). We then add an edge from K_i to each graph node (approximately) matching it. Each such edge is assigned a set of costs, including mismatch cost (e.g., s_2 in the figure) that is lower for closer matches, and costs related to the relevance of the relations connected by the edge. The edge also has an adjustable weight (for instance w_2) that appropriately scales the edge cost to yield an overall edge cost (for instance c_2). Additionally, we "lazily" bring in data values as necessary. For each *database tuple* matching the keyword, we add a node for each value in the tuple, with a similarity edge between the value and the K_i node (e.g., $w_{c3}s_3$ to `plasma membrane`, where s_3 is the mismatch cost and w_{c3} represents the starting weight for that edge). To complete the graph, we add zero-cost edges between tuple value nodes and their corresponding attribute nodes.

From this query graph, each tree with leaf nodes $K_1 \ldots K_m$ represents a possible join query (each relation node in the tree, or connected to a node in the tree by a zero-cost edge, represents a query atom, and each non-zero-cost edge represents a join or selection condition). As described in [32], Q runs a *top-k Steiner tree algorithm* (using an exact algorithm at small scales, and an approximation algorithm [32] at larger scales; STAR [21] could also be used) to find the k lowest-cost Steiner trees.

From each such tree Q, we generate a conjunctive SQL query that constructs a list of items for the SQL `select`, `from`, and `where` clauses, and an associated cost expression for the particular query. For efficiency reasons, we only incorporate value-based similarity predicates in matching keywords to data or metadata, not in joining one item with another; hence the cost of each query is independent of the tuples being processed. (In ongoing work we are incorporating similarity joins and other operations that vary in cost from one tuple to the next.)

The individual SQL statements must be unioned together in increasing order of associated cost. This actually requires a disjoint or "outer" union: each query may output different attributes, and we want a single unified table for output. However, we would like to place conceptually "compatible" output attributes from different queries into the same column.

We start by defining the query output schema Q_A to match the output schema of the first query's select-list L_A. Then, for each successive query, we iterate over each attribute a in its select-list. Let n_a be the node in the query graph with label a. Suppose there exists some similarity edge $(n_a, n_{a'})$ with cost below a threshold t, and $label(n_{a'})$ appears in Q_A. If the current query is not already outputting an attribute corresponding to $label(n_{a'})$, then we rename attribute a to $label(n_{a'})$ in the output. Otherwise, we simply add a as a new attribute to Q_A. Then we create a multiway disjoint union SQL query, in which each "branch" represents one of the queries produced from a query tree. Each "branch" also outputs a cost (its e term). Finally, we execute the queries and return answers in ranked order, annotated with *provenance* information about their originating queries.

2.3 Search Graph Maintenance

The novel aspect of our system is its ability to *maintain* the search graph and adjust the results of existing user queries accordingly, as highlighted on the right side of Figure 1. We assume that a user's query has described an ongoing *information need* for that user, and that he or she will make future as well as current use of the query results. Hence we save the results of the query as a view, and we focus on enabling the user to *refine* the view by giving feedback and adjusting the weights given to various associations, and on *incorporating new data sources* if good associations can be found with the existing relations in the search graph, and the contents of these new sources affect the contents of the top-k tuples in the user's view.

The core capabilities for user feedback were addressed in our previous work [32], so we concentrate here on discovering new associations (alignments) with relevant sources (Section 3), and on using feedback to refine and repair such associations (Section 4).

3. ADDING NEW DATA SOURCES

Once a keyword search-based view has been defined as in the previous section, Q switches into *search graph maintenance mode*. One crucial maintenance process, discussed in this section, decides if and how to incorporate new sources into the current view as the system is notified of their availability.

Q includes a *registration* service for new tables and data sources: this mechanism can be manually activated by the user (who may give a URL to a remote JDBC source), or could ultimately be triggered directly by a Web crawler that looks for and extracts tables from the Web [7] or the deep Web [24, 35] .

3.1 Basic Approach

When a new source is registered, the first step is to incorporate each of its underlying tables into the search graph. The search graph is in effect the data model queried by Q. It contains both metadata (relation and attribute nodes) and data (tuple values), re-

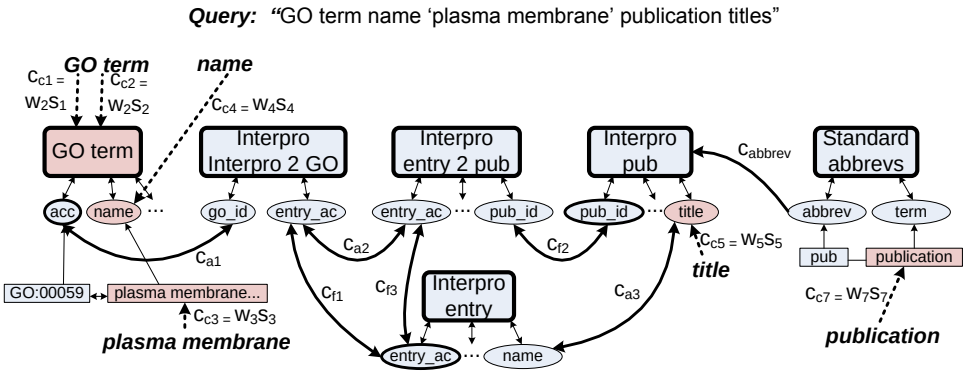

Figure 3: Query graph, given a series of keyword search terms. In general, each keyword may match a node with a *similarity score* s_{ci}, for which a *weight coefficient* w_{ci} is to be assigned by the system.

lated by edges that specify possible ways of constructing a query. The lower the cost of an edge, the more likely that the edge will be relevant to answering queries involving one of the nodes it links.

When a new source is encountered, the first step is to determine potential *alignments* between the new source's attributes and those in existing tables: these alignments will suggest (1) potential joins to be used in query answering, and (2) potential alignments of attributes in query output, such that the same column in the query answers contains results from different sources. We note that in both cases, it is desirable that aligned attributes come from the same domains (since, in the case of joins, no results would be produced unless there are shared data values among the attributes).

Of course, this task requires a set of *alignment primitives* (schema matching algorithms) used to match attributes, which we describe in Section 3.2. But there are additional architectural challenges that must be faced at the overall system level. As the search graph grows in size, the cost of adding new associations becomes increasingly expensive: regardless of the specific primitives used, the cost of alignment tends to be at least quadratic in the number of compatible attributes. We must find ways of reducing the space of possible alignments considered. Moreover, not all of these proposed alignments may be good ones: most schema matching or alignment algorithms produce false positives.

We exploit the fact that a bad alignment will become apparent when (and *only when*) it affects the top-k results of a user query whose results are closely inspected. We develop an *information need-driven* strategy where we consider only alignments that have the potential to affect existing user queries (Section 3.3). As we later show in Section 5, this restricts the space of potential alignments to a small subset of the search graph, which grows at a much lower rate than the search graph itself. We then develop techniques for correcting bad alignments through user feedback on the results of their queries (Section 4).

3.2 Alignment Primitives

Since we focus here on system architecture and learning methods, our goal with Q is to develop an architecture and learning methods that are agnostic as to the specific schema matching or attribute alignment techniques used, such that we can benefit from existing methods in databases and machine learning.

To demonstrate the architecture's ability to accommodate different schema matching algorithms, we incorporate two complementary types of matchers in Q. The first type consists of typical similarity-based schema matchers from the database community that rely on pairwise matches between source and target relations, primarily looking at schema rather than instance-level features, and which we aim to plug into our architecture as "black boxes". The second kind are matchers that globally aggregate the compatibilities between data instances. To that end, we develop a new schema matching technique that looks at "type compatibility" in a way that considers *transitivity*: if attribute A has 50% overlap in values with attribute B, and attribute B has 50% overlap in values with source C, all three attributes likely come from the same domain even if A and C do not share many values. Here we adapt a technique from the machine learning and Web community called *label propagation* that exploits transitivity and data properties, which has not previously been applied to schema matching. We briefly review both kinds of matchers, then describe how we incorporate them into Q.

3.2.1 Alignment with Metadata Matcher

Prior work on schema matching has shown that it is useful to consider *multiple* kinds of features, both at the data and metadata level, when determining alignments. Many different schema matchers that incorporate multiple features have been proposed in recent years [29], with one of the most sophisticated being COMA++ [11]. The creators of the COMA++ schema matching tool graciously provided a copy of their system, so our specific implementation incorporates COMA++ through its Java API. This system is described in detail elsewhere [11]. Briefly, we used COMA++'s default structural relationship and substring matchers over metadata to produce proposed alignments[1].

3.2.2 Alignment with Label Propagation

Our second matcher focuses on which attributes are type-compatible at the instance level. The notion of *label propagation* has been used in recent machine learning work for finding associated metadata based on weighted transitive relationships across many sources. Informally, this work represents a generalization of some of the ideas in similarity flooding [26] or the Cupid algorithm [23], but at a larger scale. In label propagation, we are given a graph $G = (V, E, W)$ with nodes V, directed edges E, and a weight function $W : E \rightarrow \mathbb{R}$ that assigns a weight (higher is better) to each edge. Assume some of the nodes $i \in V$ initially are given labels l_i. Labels are propagated from each node along its out-edges to its neighboring nodes with a probability proportional to edge weight, eventually yielding a label probability distribution L_i for each node. Intuitively, this model is similar to PageRank [6], except that it computes how likely a "random surfer" *starting at an initial node with a particular label* will end up at some other node, based on a Markovian (memory- or history-free) behavioral

[1]COMA++ also optionally includes instance-level matching capabilities, but despite our best efforts and those of the authors, we were only able to get the metadata matching capabilities of COMA++ to work through its Java API.

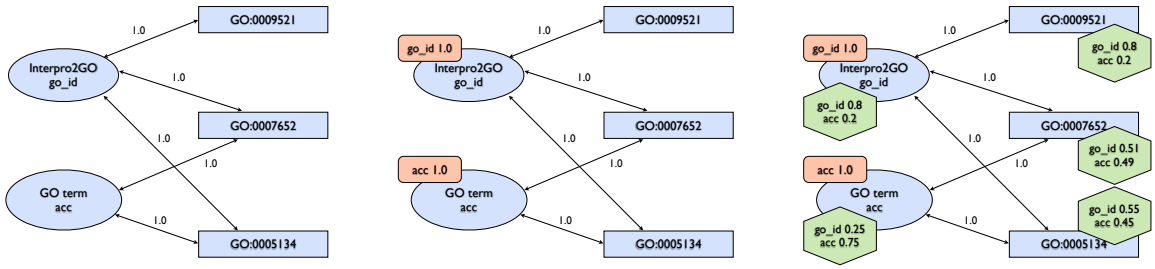

Figure 4: Propagation of labels in a column-value graph, using the Modified Adsorption (MAD) algorithm (Section 3.2.2). From left to right: the original graph with two column nodes and three value nodes; each column node injected with its own label (labels inside the rectangle); after two iterations of label propagation with estimated labels shown inside hexagons.

assumption. In this work, we use the Modified Adsorption (MAD) [31] label propagation algorithm.

MAD is one of a family of related label propagation algorithms used in several areas [36]. While these algorithms can be explained in several ways [2], for simplicity we will rely here on the *random walk* interpretation of MAD.

Let $G_{\mathrm{r}} = (V, E_{\mathrm{r}}, W_{\mathrm{r}})$ be the edge-reversed version of the original graph $G = (V, E, W)$, where $(a, b) \in E_{\mathrm{r}}$ iff $(b, a) \in E$, and $W_{\mathrm{r}}(a, b) = W(b, a)$. Now, choose a node of interest $q \in V$. To estimate L_q for $q \in V$, we perform a random walk on G_{r} starting from q to generate samples for a random label variable L. After reaching a node i during the walk, we have three choices:

1. With probability p_i^{cont}, continue the random walk to a neighbor of i.
2. With probability p_i^{abnd}, abandon the random walk. This abandonment probability makes the random walk stay relatively close to its source when the graph has high-degree nodes. When the random walk passes through such a node, it is likely that further transitions will be into regions of the graph unrelated to the source. The abandonment probability mitigates that effect.
3. With probability p_i^{inj}, stop the random walk and emit either L_i if i is one of the initially labeled nodes.

L_q will converge to the distribution over labels L emitted from random walks initiated from node q. In practice, we use an equivalent iterative fixpoint view of MAD [31], shown in Algorithm 1. In this algorithm, I_v is the injected label distribution that a node is seeded with; R_v is a label distribution with a single peak corresponding to a separate "none of the above" label $\top$. This dummy label allows the algorithm to give low probability to all labels at a node if the evidence is insufficient.

3.2.3 Combining Matchers in Q

We now describe how we fit each type of matcher into Q, starting with the "black box" interface to COMA++. Later in the paper, we discuss how we can combine the outputs of multiple matchers, using user feedback to determine how to weigh each one.

COMA++ as a black-box matcher. An off-the-shelf "black box" schema matcher typically does *pairwise* schema matching, meaning that each new source attribute gets aligned with only a *single* attribute in the existing set of data sources (rather than, e.g., an attribute in each of the existing data sources). Moreover, matchers tend to only output their *top* alignment, even when other potential alignments are considered. Our goal in Q is to determine the top-Y (where Y is typically 2 or 3) candidate alignments for each attribute, unless the top alignment has very high confidence: this way we can later use user feedback to "suppress" a bad alignment and see the results of an alternative.

To get alignments between the new source's attributes and all sources, we do a pairwise schema alignment between the new source and each existing source. We thus obtain what COMA++ assumes to be the top attribute alignments between each relation pair.

While we do not do this in our experiments, it is feasible (if expensive) to go beyond this, to force COMA++ to reveal its top-Y overall alignments. Between each pair of schemas, we can first compute the top alignment. Next, for each alignment pair (A, B) that does not have a high confidence level, remove attribute A and re-run the alignment, determining what the "next best" alignment with B would be (if any). Next re-insert A and remove B, and repeat the process. If there are additional schema matching constraints (e.g., no two source attributes may map to the same target attribute), we can again iterate over each alignment pair (A, B). Now remove *all* attributes from A's schema that are "type compatible" with A, except for A itself; and run the alignment. Then replace those attributes, and repeat the process removing attributes type-compatible with B other than B itself.

Ultimately, we will have obtained from the matcher a set of associations (equivalent here to the alignments) and their confidence levels. Depending on the matcher used, the confidence scores may need to be normalized to a value between 0 and 1; in the case of COMA++, its output already falls within this range. These confidence scores will be used in forming a new edge cost (Section 3.4).

MAD to discover compatible datatypes. We developed a matcher module (parallelizable for Hadoop MapReduce), which performs MAD across schemas, using techniques described in [31]. While this matcher implementation is in some sense a part of Q, it is implemented in a way that does not provide any special interfaces, i.e., from Q's perspective it remains a black box. This matcher first creates an internal label propagation graph that incorporates both metadata and data. From the search graph, we take all relation attributes from all sources, and create a node in the label propagation graph for each attribute, labeled with its canonical name. We also take all data values and create a label propagation graph node for each unique value. We add to the graph an edge between a value node and each node representing an attribute in which the value appears. Now we *annotate* or label each attribute node with its name. A sample graph is shown in the left portion of Figure 4; for simplicity, all the edges have weight 1.0.

We run the MAD algorithm over this graph, propagating sets of annotations from node to node. The algorithm runs until the label distribution on each node ceases to change beyond some tolerance value. Alternatively, the algorithm can be run for a fixed number of iterations. Each value node ultimately receives a distribution describing how strongly it "belongs" to a given schema attribute, and each attribute node receives a distribution describing how closely it matches other attribute nodes.

Algorithm 1 Modified Adsorption (MAD) Algorithm

Input: **Graph:** $G = (V, E, W)$, **Seed labeling:** $I_v \in \mathbb{R}^{m+1}$ for $v \in V$, **Probabilities:** $p_v^{\text{inj}}, p_v^{\text{cont}}, p_v^{\text{abnd}}$ for $v \in V$, **Label priors:** $R_v \in \mathbb{R}^{m+1}$ for $v \in V$, **Output**: **Label Scores:** L_v for $v \in V$

$L_v \leftarrow I_v$ for $v \in V$ {Initialization}
$\mathbf{M}_{vv} \leftarrow \mu_1 \times p_v^{\text{inj}} + \mu_2 \times p_v^{\text{cont}} \times \sum_u \mathbf{W}_{vu} + \mu_3$
repeat
  $D_v \leftarrow \sum_i \left(p_v^{\text{cont}} \times \mathbf{W}_{vi} + p_i^{\text{cont}} \times \mathbf{W}_{iv}\right) \times I_i$
  for all $v \in V$ **do**
    $L_v \leftarrow \frac{1}{\mathbf{M}_{vv}} \times \left(\mu_1 \times p_v^{\text{inj}} \times I_v + \mu_2 \times D_v + \mu_3 \times p_v^{\text{abnd}} \times R_v\right)$
  end for
until convergence

Algorithm 2 VIEWBASEDALIGNER(G, G', K, C, α). **Input**: Search graph G, new source G', keywords (K) associated with current view, cost function C, cost threshold α. **Output**: Augmented schema graph G'', with alignments between G and G'.

$G'' \leftarrow \text{G} \cup G'$
$S \leftarrow \emptyset$
for $k \in K$ **do**
  $S = S \cup$ GETCOSTNEIGHBORHOOD(G, C, α, k)
end for
for $v \in S$ **do**
  A = BASEMATCHER (G', v)
  $\text{E}(G'') \leftarrow \text{E}(G'') \cup A$
end for
Return G''

In the graph in the second column in the Figure 4, we see that the attribute nodes are annotated with labels matching their names, each with probability 1. These labels are propagated to the neighboring nodes and multiple iterations are run until convergence is reached (shown in the rightmost graph). At the end, we see that all data values are annotated with *both* `go_id` and `acc` since there is significant value overlap between the two attributes. Note that MAD does not require direct pairwise comparison of sources. This is very desirable as such pairwise comparisons can be expensive when many sources are involved.

We use the label distributions generated by MAD to generate uncertainty levels from which edge costs will be derived for Q's search graph. For each node n in the MAD graph, we select the *top-Y* attributes from its label distribution, and we add an edge in the search graph between the attribute node for l and the attribute node for n. The confidence level for each such edge will be $L_n(l)$. Section 3.4 describes how this level is combined with other weighted parameters to form an edge cost.

3.3 Searching for Associations

We just saw how to harness individual schema matchers to find alignments of sources, and hence association edges between existing and new source relations. However, we need to ensure that the alignment algorithms can be applied scalably, as we increase the number of data sources that we have discovered.

Of course, the simplest (though least scalable) approach is to simply perform exhaustive matching: upon the registration of a new data source, we iterate over all existing data sources in turn, and run our alignment algorithm(s). We term this approach EXHAUSTIVE, and note that it will scale quadratically in the number of attributes in each source. As we shall see in Section 5, even for small numbers

Algorithm 3 PREFERENTIALALIGNER(G, G', P). **Input**: Search graph G, new source G', vertex cost function P. **Output**: Augmented schema graph G'', with alignments between G and G'.

$G'' \leftarrow \text{G} \cup G'$
$V_s = \text{SORT}(V(G), P)$
for $i = 1$ to $\mathbf{V}_s$.length **do**
  r = GETRELATIONNODE $(V_s[i])$
  A = BASEMATCHER (G', r)
  $\text{E}(G'') \leftarrow \text{E}(G'') \cup A$
end for
Return G''

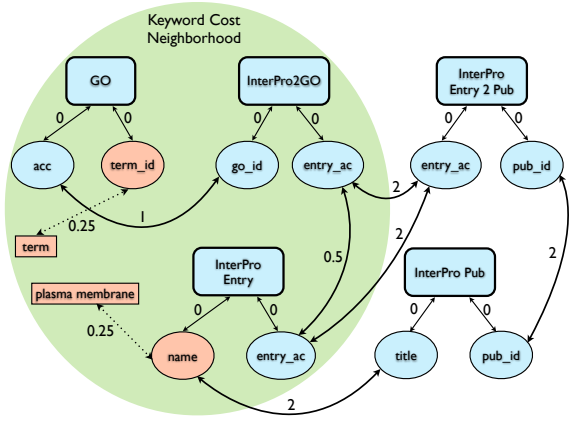

Figure 5: A schema graph for the keywords *term* and *plasma membrane*. Edges are annotated with costs. The shaded region is the α-cost neighborhood (α = 2) of the two keywords, i.e. all nodes reachable with cost $\leq$ 2 from a keyword.

of attributes schema alignment takes time, and with large numbers of sources it may be costly to find new associations.

As was previously noted, we can exploit the fact that new associations are only "visible" to users if they appear in any queries returning top-k results. Hence we exploit existing user views, and the existing scores of top-k results, to restrict the search space of alignments. As new queries are materialized within the system, we would incrementally consider further alignments that might affect the results of those queries.

Algorithm 2 shows code for VIEWBASEDALIGNER, which reduces the number of schema alignment comparisons (calls to BASEMATCHER) through a pruning strategy that is guaranteed to provide the same top-k answer set for a query as EXHAUSTIVE. Given an existing schema graph $G = (V, E, C)$ where C is a non-negative real-valued cost function for each edge (discussed in the next section), a set of keyword nodes K, and the cost α of the kth top-scoring result for the user view, VIEWBASEDALIGNER considers alignments between the new source's schema graph G', and the projection of the graph that is within α of any keyword node. To affect the view output, any new node from G' must be a member of a Steiner tree with cost $\leq \alpha$; given that edge costs are always non-negative, our pruning heuristic guarantees that we have considered all possible alignments that could lead to this condition.

We illustrate this with an example in Figure 5, where we assume two keywords have matched and the kth best score $\alpha = 2$. Here, VIEWBASEDALIGNER only considers alignments between a new source and those nodes within the shaded cost neighborhood. This yields savings in comparison with EXHAUSTIVE, which would additionally need to compare the new source against the two sources

outside of the region. Of course, in a real search graph many more sources are likely to be *outside* the region than inside it.

If we need even more aggressive pruning, we can adapt ideas from network formation in social networks [3], and assume the existence of an alignment prior (P) over vertices of the existing search graph G, specifying a preference ordering for associations with the existing nodes. This can capture, e.g., that we might want to align with highly authoritative or popular relations. Algorithm 3 shows pseudocode for such a PREFERENTIALALIGNER. A new source, G', is compared against the existing nodes in G in the order of the ranking imposed by the prior P. The prior might itself have been estimated from user feedback over answers of keyword queries, using techniques similar to those of the next section, or it might be computed using alternate methods such as link analysis [1].

3.4 Measuring Schema Graph Edge Quality

As we take the output of the aligner and use it to create an association in the search graph, we would like to set the edge cost in a principled way: ideally the value is not simply a hard-coded "default cost," nor just the confidence value of the aligner, but rather it should take into account a number of factors. For instance, the edge cost might take into account costs associated with the relations being joined, derived from their authoritativeness or relevance; and when we are using multiple matchers to create an alignment, we might want to perform a weighted sum of their confidence scores.

We use a cost function for each edge that considers a combination of multiple weighted components, some of which may be shared across edges, and others of which may be exclusive to a specific edge. We formalize this by describing the cost of an edge as a sum of weights times *feature values* (also called *scores*). The weights will be *learned* by Q (Section 4), whereas the features are the base cost components whose value does not change. For instance, to incorporate the uncertainty score from a black-box schema matcher, we capture it as a feature, whose associated weight we will learn and maintain. In some cases, we consider features to be Boolean-valued: for instance, if we want to learn a different weight for each edge, then we will create a feature for that edge whose value is 1 for that edge (and 0 elsewhere).

Let the set of predefined features across the search graph be $F = \{f_1, \ldots, f_M\}$. Formally, a feature maps edges to real values. For each edge (i, j), we denote by $\mathbf{f}(i, j)$ the *feature vector* that specifies the values of all the features of the edge. Each feature f_m has a corresponding weight w_m. Informally, lower feature weights indicate stronger preference for the edges that have those features. Edge costs are then defined as follows:

$$C((i,j), \mathbf{w}) = \sum_m w_m \times f_m(i,j) = \mathbf{w} \cdot \mathbf{f}(i,j) \qquad (1)$$

where m ranges over the feature indices.

When we add a new association edge based on an alignment, we set its cost based on the following weighted features:

- A *default feature* shared with all edges and set to 1, whose weight thus comprises a default cost added to all edges.
- A feature for the confidence value of each schema matcher, whose weight represents how heavily we (dis)favor the schema matcher's confidence scores relative to the other cost components.
- A feature for each relation R connected by the association, whose value is 1 for this relation$R$, and whose weight represents the negated logarithm of the R's *authoritativeness*.
- A feature that uniquely identifies the edge itself, whose value is 1, and whose weight comprises a cost added to the edge.

Together, the weighted features form an edge cost that is initialized based not only on the alignment confidence levels, but also on information shared with other nodes and edges.

4. USER FEEDBACK & CORRECTIONS

When the user sees a set of results, he or she may notice a few results that seem either clearly correct or clearly implausible. In Q the user may provide feedback by optionally annotating each query answer to indicate a valid result, invalid result, or a ranking constraint (tuple t_x should be scored higher than t_y). Q first *generalizes* this feedback by taking each tuple, and, by looking at its provenance, replacing it with the *query tree that produced it*, using a scheme similar to [32]. Recall that our model is one of tuple and edge *costs* so a lower cost results in higher ranking.

The *association cost learner* converts each tuple annotation into a constraint as follows:

- A query that produces correct results is constrained to have a cost at least as low as the top-ranked query result.
- A query Q_x that should be ranked above some other query Q_y is constrained to have a cost that is lower than Q_y's cost.

These constraints are fed into an algorithm called MIRA [10], which has previously been shown to be effective in learning edge costs from user feedback on query results [32]. We briefly summarize the key ideas of MIRA here, and explain how we are using it in a less restricted way here, learning over *real-valued* features, as opposed to the Boolean features in the previous work [32].

Relationship between Edge Costs and Features. Recall from Section 3.4 that each edge is initialized with a cost composed of multiple weighted features: the product of the weight and the feature value comprise a default cost given to every edge, a weighted confidence score from each schema alignment algorithm, the authoritativeness of the two relations connected by the edge, and an additional cost for the edge itself. Q's association cost learner takes the constraints from user feedback and determines a weight assignment for each feature — thus assigning a cost to every edge.

Learning Algorithm. The learning algorithm (Algorithm 4) reads training examples sequentially and updates its weights after receiving each of the examples based on how well the example is classified by the current weight vector. The algorithm, which was first used in [32], is a variant of the Margin Infused Ranking Algorithm (MIRA) [10]. We previously showed in [32] that MIRA effectively learning top-scoring queries from user feedback; however, in that work only binary features were used, while here we need to include real-valued features from similarity costs. Using real-valued features directly in the algorithm can cause poor learning because of the different ranges of different real-valued and binary features. Therefore, as described above, we *bin* the real-valued features into empirically determined bins; the real-valued features are then replaced by features indicating bin membership.

The weights are all zero as Algorithm 4 starts. After receiving feedback from the user on the r^{th} query S_r about a top answer, the algorithm retrieves the list B of the k lowest-cost Steiner trees using the current weights. The user feedback for interaction r is represented by the keyword nodes S_r and the *target tree* T_r that yielded the query answers most favored by the user. The algorithm then updates the weights so that the cost of each tree $T \in B$ is worse than the target tree T_r by a margin equal to the mismatch or *loss* $L(T_r, T)$ between the trees. If $T_r \in B$, because $L(T_r, T_r) = 0$, the corresponding constraint in the weight update is trivially satisfied. The update also requires that the cost of each edge be positive, since non-positive edge costs will result in non-meaningful Steiner tree computations. To accomplish this,

Algorithm 4 ONLINELEARNER(G, U, k). **Input**: Search graph G, user feedback stream U, required number of query trees k, zero-cost constraint edges A. **Output**: Updated costs of edges in G.

$\mathbf{w}^{(0)} \leftarrow \mathbf{0}$
$r = 0$
while U is not exhausted **do**
  $r = r + 1$
  $(S_r, T_r) = U.\text{NEXT}()$
  $C_{r-1}(i,j) = \mathbf{w}^{(r-1)} \cdot \mathbf{f}_{ij} \;\; \forall (i,j) \in E(G)$
  $B = \text{KBESTSTEINER}(G, S_r, C_{r-1}, K)$
  $\mathbf{w}^{(r)} = \arg\min_{\mathbf{w}} \left\| \mathbf{w} - \mathbf{w}^{(r-1)} \right\|$
  s.t. $C(T, \mathbf{w}) - C(T_r, \mathbf{w}) \geq L(T_r, T), \;\; \forall T \in B$
    $\mathbf{w} \cdot \mathbf{f}_{ij} = 0 \; \forall (i,j) \in A$
    $\mathbf{w} \cdot \mathbf{f}_{ij} > 0 \; \forall (i,j) \in E(G) \setminus A$
end while
Let $C(i,j) = \mathbf{w}^{(r)} \cdot \mathbf{f}_{ij} \; \forall (i,j) \in E(G)$
Return C

we include the *default feature* listed above, whose weight serves as a uniform cost offset to all edge weights in the graph, which keeps the edge costs positive. Some edges in the query graph are constrained to have a fixed edge cost, irrespective of learning. For example, attribute-relation edges have a cost of zero that should always be maintained. We achieve this by adding such constraints to the MIRA algorithm. Our implementation requires a modification of MIRA (shown in Algorithm 4) that takes as input a set A specifying edges with zero cost constraints.

An example loss function, used in our experiments, is the *symmetric loss* with respect to the edges E present in each tree:

$$L(T, T') = |E(T) \setminus E(T')| + |E(T') \setminus E(T)| \qquad (2)$$

The learning process proceeds in response to continued user feedback, and finally returns the resulting edge cost function.

5. EXPERIMENTAL ANALYSIS

In this section, we use Q as a platform to validate our strategy of performing schema alignment in a query-guided manner (Section 5.1), as well as our techniques for using user feedback over data to correct bad alignments (Section 5.2). The search graph maintenance modules in Q comprise approximately 4000 lines of Java code, and all experiments were run on a Dell PowerEdge 1950 computer running RedHat Enterprise Linux 5.1 with 8GB RAM. We used the COMA++ 2008 API, and a Java-based implementation of our MAD-based schema matcher.

Our focus in Q is on supporting bioinformatics applications, and hence wherever possible, we use real biological databases and compare with *gold standard* results, i.e., reference results supplied by domain experts. This enables us to perform an experimental study without having to conduct extensive user studies.

For the first set of experiments, we use a dataset for which we have logs of actual SQL queries executed by Web forms, such that we can determine which proposed source associations are actually valid (as witnessed by having real queries use them). This dataset, GBCO[2], consists of 18 relations (which we model as separate sources) with 187 attributes.

In the second set of experiments, we used a different dataset, based on the widely used (and linked) Interpro and GO databases, where we could obtain *keyword queries* and find multiple alternative means of answering these queries. This dataset consists of 8 closely interlinked tables with 28 attributes.

[2] http://www.betacell.org/

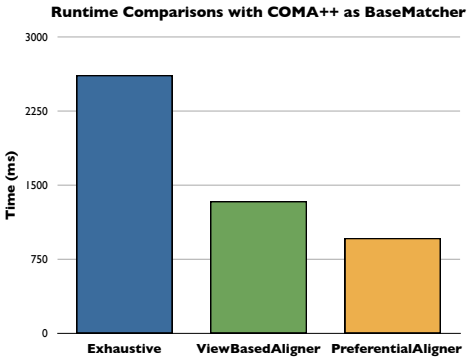

Figure 6: Running times (avgd. over intro of 40 sources) when aligning a new source to a set of existing sources (COMA++ as base matcher). VIEWBASEDALIGNER and PREFERENTIALALIGNER significantly reduce running times vs. EXHAUSTIVE.

5.1 Incorporating New Sources

We first look at the cost of adding new data sources to an existing search graph, in a way that keeps the alignment task tractable by limiting it to the "neighborhood" of an existing query. We set up the experiment, using the GBCO dataset described above, as follows.

We first scanned through the GBCO query logs for *pairs* of SQL queries, where one query represented an *expansion* of the other, *base*, query: i.e., the *expanded query* either joined or unioned additional relations with the base query. Intuitively, the expanded query tells us about new sources that would be useful to add to an existing search graph that had been capable of answering the base query. When the expanded query represents the union of the base query with a new query subexpression, then clearly adding the new data source results in new association edges that provide further data for the user's view. When the expanded query represents an additional join of the base query with new data, this also affects the contents of the existing view if the additional join represents a segment of a new top-scoring Steiner tree for the same keyword query.

For each base query, we constructed a corresponding keyword query, whose Steiner trees included the relations in the base query. Next, we initialized the search graph to include all sources *except* the ones unique to the expanded query. We initially set the weights in the search graph to default values, then provided feedback on the keyword query results, such that the SQL base query from our GBCO logs was returned as the top query. For all experiments in this section, the edge costs learned in the process were used as the value of the function C in the VIEWBASEDALIGNER algorithm. The vertex cost function P in PREFERENTIALALIGNER was similarly estimated from the weights of features corresponding to source identities.

5.1.1 Cost of Alignment

Our first experiment measures the cost of performing alignments between the new source and a schema graph containing all of the other sources — using our EXHAUSTIVE, VIEWBASEDALIGNER, and PREFERENTIALALIGNER search strategies, with the COMA++ matcher. Figure 6 compares the running times of these strategies. Figure 7 shows the number of pairwise attribute comparisons necessary, under two different sets of assumptions. The *Value Overlap Filter* case assumes we have a content index available on the attributes in the existing set of sources and in the new source; we only make compare attributes that have shared values (hence can join). More representative is likely to be the *No Additional Filter* case, which has only metadata to work from.

We observe that, regardless of whether a value overlap filter is available, limiting the search to the neighborhood of the existing

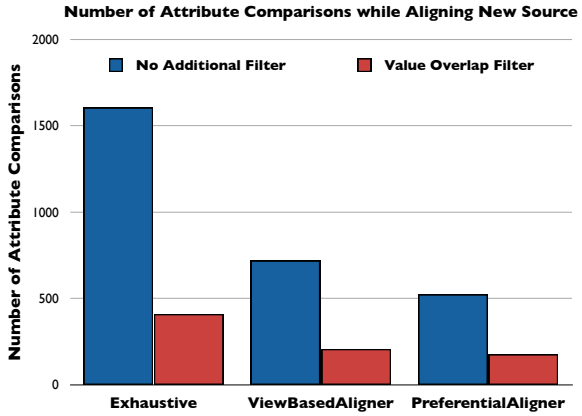

Figure 7: Pairwise attribute comparisons performed in aligning new source(s) to existing sources (avgd. over intro of 40 sources in 16 trials, where each trial introduces one or more new sources). VIEWBASEDALIGNER and PREFERENTIALALIGNER significantly reduce comparisons vs. EXHAUSTIVE.

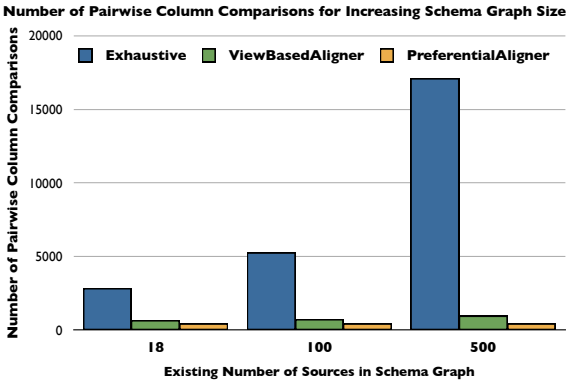

Figure 8: Number of pairwise attribute comparisons as the size of the search graph is increased (avgd. over introduction of 40 sources). VIEWBASEDALIGNER and PREFERENTIALALIGNER are hardly affected by graph size.

query (i.e., our information need-driven pruning strategy) provides significant speedups (about 60%) versus doing an exhaustive set of comparisons, even on a search graph that is not huge. Recall that VIEWBASEDALIGNER will provide the *exact same* updates to a user view as the exhaustive algorithm. PREFERENTIALALIGNER does not have this guarantee, and instead focuses on the alignments specified in the prior, but gives even lower costs.

The differences in costs results from the fact that the number of comparisons in EXHAUSTIVE depends on the number of source relations in the schema graph, whereas the number of comparisons in the other cases is only dependent on the number of nodes in the local neighborhood of the query.

5.1.2 *Scaling to Large Number of Sources*

We next study how the cost of operations scales with respect to the search graph size. Since it is difficult to find large numbers of interlinked tables "in the wild," for this experiment we generated additional synthetic relations and associations for our graph. We started with the real search graph, and built upon it as follows. We initialized the original schema graph of 18 sources with default costs on all edges. Then we took our set of keyword queries and executed each in sequence, providing feedback on the output such that the base query was the top-scoring one. At this point, the costs on the edges were calibrated to provide meaningful results. Now

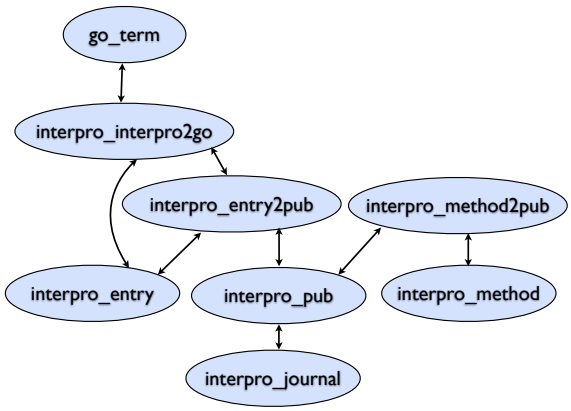

Figure 9: Schema graph used in the experiments of Section 5.2 (attributes are not shown).

we randomly generated new sources with two attributes, and then connected them to two random nodes in the search graph. We set the edge costs to the average cost in the calibrated original graph.

Once the schema graph of desired size was created, the three alignment methods were used to align the new sources in the expanded graph. Since our mostly-synthetic expanded search graph does not contain realistic node labels and attributes, we do not directly run COMA++ on the results, but instead focus on the number of column comparisons that must be performed. The results appear in Figure 8. Recall that Figure 6 shows that COMA++'s running times grow at a rate approximately proportional to the number of column comparisons. From Figure 8, we observe that the number of pairwise column comparisons needed by VIEWBASEDALIGNER and PREFERENTIALALIGNER remained virtually unchanged as the number of sources increased from 18 to 500, whereas EXHAUSTIVE grew quite quickly.

We conclude from the experiments in this subsection that localizing the search to the neighborhood around a query yields much better scalability. VIEWBASEDALIGNER gives the same results as the exhaustive strategy, and hence is probably the preferred choice.

5.2 Correcting Matchings

The previous section focused on the cost of running alignment algorithms, without looking at their quality. We now look at how well Q takes the suggested alignments from the individual alignment algorithms, as well as user feedback on query answers, to get the correct associations. These experiments were conducted over the InterPro-GO dataset described previously (shown visually in Figure 9), for which we were able to get a set of keyword queries based on common usage patterns suggested in the description of the GO and InterPro databases[3]. We know from the original schema specifications and documentation that there are 8 semantically meaningful join or alignment edges among these relations, but we remove this information from the metadata.

Our experimental setup is to start with a schema graph that simply contains the tables in Figure 9, and then to run the association generation step (using COMA++ and/or MAD) to generate a search graph in the Y most promising alignments (for different values of Y) are recorded for each attribute. Next we execute the set of keyword queries obtained from the databases' documentation. For each query, we generate one *feedback response*, marking one answer that only makes use of edges in the gold standard. Since the gold standard alignments are known during evaluation, this feedback response step can be simulated on behalf of a user. Our goal

[3] http://www.ebi.ac.uk/interpro/User-FAQ-InterPro.html

Y	System	Precision	Recall	F-measure
1	COMA++	62.5	62.5	62.5
	MAD	70	87.5	77.78
2	COMA++	63.64	87.5	73.68
	MAD	66.67	100	80
5	COMA++	63.64	87.5	73.68
	MAD	66.67	100	80

Table 1: Evaluation of top-Y edges (per node) induced by COMA++ and MAD for various values of Y (see Section 5.2.1). The schema graph of Figure 9 was used as the gold reference.

is to "recover" all of the links shown in Figure 9, which forms the *gold standard*.

We now present our results using precision, recall and F-measure as our evaluation metrics. We compute these metrics with respect to the search graph, as opposed to looking at query answers. For different values of Y, we compare the top Y alignment edges in the search graph (that also fall under a cost threshold) for each attribute, versus the edges in the gold standard. Clearly, if the alignment edges in the schema graph exactly match the gold standard, then they will result in correct answers.

5.2.1 *Baseline Matcher Performance*

Our first set of experiments compares the relative performance of the individual matchers over our sample databases, as we increase the number of alternate attribute alignments we request from the matcher in order to create the search graph. We briefly describe setup before discussing the results.

COMA++ setup. As described in Section 3.2.1, COMA++ [11] was applied as a pairwise aligner among the relations in Figure 9. This involved computing alignments and scores in COMA++ for attributes in each pair of relations. Using this scheme we were able to induce up to 34 alignment edges.

MAD setup. We took the relations in Figure 9 and the values contained in the tables, and constructed a MAD graph resembling Figure 4. All nodes with degree one were pruned out from the MAD graph before the matching algorithm was run, as they are unlikely to contribute to the label propagation. Also, all nodes with numeric values were removed, as they are likely to induce spurious associations between attributes. The resulting graph had 87K nodes. We used the heuristics from [31] to set the random walk probabilities.

MAD was run for 3 iterations (taking approximately 4 seconds total), with $\mu_1 = \mu_2 = 1$, and $\mu_3 = 1e{-}2$. Each unique column name (attribute) was used as a label, and so 28 labels were propagated.

Results. For each of the algorithms, we added to the search graph (up to) the top-Y-scoring alignments per attribute, for Y values ranging from 1 to 5, as shown in Table 1. Our general goal is to have the matchers produce 100% recall, even at the cost of precision: the Q learner must be able to find the correct alignment in the search graph if it is to be able to allow for mapping correction.

We conclude that our novel MAD scheme, which is purely based on data values, does very well in this bioinformatics setting, with a recall of 7 out of 8 edges even with $Y = 1$, and 100% recall with $Y = 2$. COMA++ produced good output (7 out of 8 alignments) with $Y = 2$, but we were not able to get it to detect all of the alignments even with high Y values.

Note that we compute precision under a fairly strict definition, and one might compellingly argue that some of the "wrongly" induced alignments are in fact useful in answering queries, even if they relate attributes that are not synonymous. For instance, if we look at the "incorrect" edges induced by MAD, we see one between `interpro.method.name` and `interpro.entry.name`. The data shows an overlap of 780 distinct values (out of 53,007 entries in `interpro.method.name` and 14,977 in `interpro.entry.name`). Joining these two tables according to this alignment may in fact produce useful results for exploratory queries (even if these results should be given a lower rank in the output). We hope in the future to conduct user studies to evaluate how useful biologists find Q's answers.

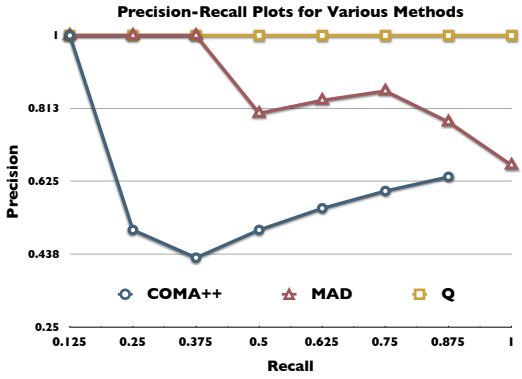

Figure 10: Precision vs. recall for COMA++, MAD and Q (which combines COMA++ and MAD). Q was trained from feedback on 10 keyword queries, replayed three times to reinforce the feedback. Precision and Recall were computed by comparing against the foreign key associations in Figure 9.

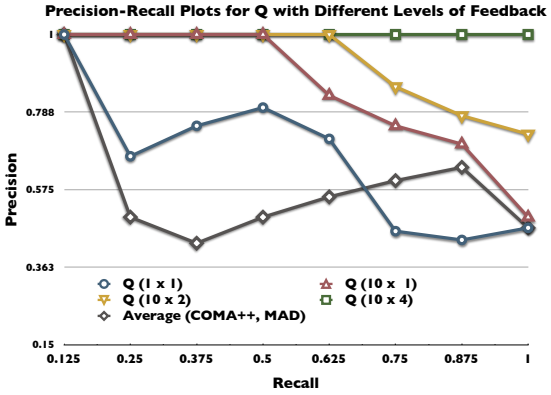

Figure 11: Precision versus recall in Q, given default weighting, then successively greater amounts of feedback.

5.2.2 *Correcting Associations*

We next study Q's performance in combining the output of the two matchers, plus processing feedback to correct alignments. This performance (measured in precision and recall) is dependent on how high a similarity (how low a cost) we require between aligned attributes. Generally, the more strict our similarity threshold, the better our precision and the lower our recall will be.

Benefits of learning. In Figure 10, we take the schema alignments from both matchers (COMA++ and MAD) when $Y = 2$ (the lowest setting where we get 100% recall, see Table 1) and combine them, then provide feedback on 10 different two-keyword queries (created as previously discussed), with $k = 5$ (see Algorithm 4). In order to ensure that weight updates are made in a way that consistently preserves all of the "good" answers, we actually apply the feedback *repeatedly* (we replay a log of the most recent feedback

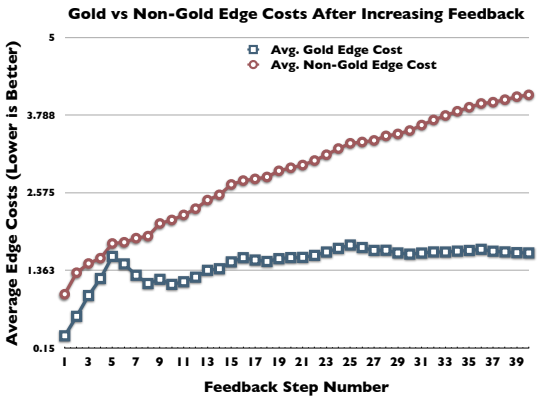

Figure 12: Average costs of gold edges (i.e., those in Figure 9) vs. non-gold edges in the search graph, as more feedback is applied. To obtain Steps 11–40 we repeat the feedback steps from 1–10 up to 3 times. Q continues to increase the gap between gold and non-gold edges' average scores.

steps, recorded as a sliding window with a size bound). Here we input the 10 feedback items to the learner four times in succession (i.e., replay them three times) to reinforce them. In order to remove the edge cost variations resulting from intermediate feedbacks, we consider the average edge cost over all feedback steps.

To see the relationship between recall and precision levels, we vary a *pruning threshold* over the schema graph: any alignment edges with cost above this threshold will be ignored in query result generation, and any below will be included. Compared to both schema matchers in isolation, with the ten feedback steps, Q does a much better job of providing both good precision and recall: we can get 100% precision with 100% recall.

Relative benefits of feedback. Next we study how performance improves with successive feedback steps. Figure 11 repeats the above experiment with increasing amounts of feedback. As a baseline, we start with the setting where the matchers' scores are simply averaged for every edge — in the absence of any feedback, we give equal weight to each matcher. Next we consider a single feedback step, designated Q (1x1), then ten feedback steps. Previously we had applied the feedback four successive times: we show here what happens if we do not repeat the feedback (10x1), if we repeat it once (10x2), and if we repeat it three times (10x4).

Looking at relative performance in Figure 11, we see that the baseline — the average of the two matchers' output — approximately follows the output of COMA++. It turns out that COMA++ gives higher confidence scores on average than MAD, and hence this simple average favors its alignments. Of course, we could adjust the default weighting accordingly — but it is far better to have the system automatically make this adjustment. We see from the graph that this happens quite effectively: after a single feedback step, we immediately see a noticeable boost in precision for most of the recall levels below 60%. Ten items of feedback with no repetitions makes a substantial difference, yielding precision of 100% for recall values all the way to 50%. However, repeating the feedback up to four times shows significant benefit.

Figure 12 shows the average costs of edges in the gold-standard (i.e., edges in Figure 9) versus non-gold edges, as we provide more feedback. Q assigns lower (better) costs on average to gold edges than to non-gold edges, and the gap increases with more feedback.

Feedback vs. precision for different recall levels. Finally, we consider the question of how much feedback is necessary to get perfect precision (hence, ultimately exact query answers) if we are willing to compromise on recall: Table 2 summarizes the results. Note that perfect precision is actually obtained with only 2 feedback steps even with 100% recall. At first glance this may seem incongruous with the results of the previous figures, but it is important to remember that each feedback step is given on a different query, and each time the online learner makes local adjustments that may counter the effects of the previous feedback steps. Hence we can see drops in precision with additional feedback steps, and it takes several more steps (plus, as we saw previously, multiple repetitions) before the overall effects begin to converge in a way that preserves all of the correct edges.

Recall Level	12.5	25	37.5	50	62.5	87.5	100
Feedback Steps	1	2	2	2	2	2	2

Table 2: Number of feedback steps required to initially get precision 1 with a certain recall level in the schema graph.

We conclude from these experiments that (1) the simple act of combining scores from different matchers is not enough to boost scores, (2) with a small number of feedback steps Q learns to favor the correct alignments, (3) particularly if a sequence of feedback steps is replayed several times, we can achieve very high precision and recall rates. Ultimately this means that we can learn to generate very high-quality answers directly using the output of existing schema matching components, plus feedback on the results.

6. RELATED WORK

In this paper, we addressed one of the shortcomings of the version of Q presented in [32], namely, that all alignments were specified in advance. Many systems supporting keyword search over databases [4, 5, 16, 17, 18, 20] use scores based on a combination of similarity between keywords and data values, length of join paths, and node authority [1]. Existing "top-k query answering" [9, 15, 22, 25] provides the highest-scoring answers for ranked queries.

Schema alignment or matching is well-studied across the database, machine learning, and Semantic Web communities [29]. General consensus is that methods that incorporate both data- and metadata-based features, and potentially custom learners and constraints, are most effective. Thus, most modern matchers combine output from multiple sub-matchers [11, 12, 26]. Our focus is not on a new method for schema matching, but rather an *architecture* for incorporating the output of a matcher in a complete iterative, end-to-end pipeline where the matches or alignments are incorporated into existing user views, and *feedback on answers* is used to correct schema matching output. Our approach requires no special support within the matcher, simply leveraging it as a "black box."

The notion of propagating "influences" across node connectivity for schema alignment is used in similarity flooding [26] and the Cupid system [23], among other schema matching studies. However, in the machine learning and Web communities, a great deal of work has been done to develop a principled family of *label propagation* algorithms [2, 36]. We incorporate this kind of matching method not only to align compatible attributes in the output, but to discover synonymous tables and transitively related items. This paper builds upon recent observations [33] showing that one could find potential labelings of tables extracted from the Web using a particular label propagation algorithm called *Modified Adsorption (MAD)*.

Our ranked data model propagates uncertainty from uncertain mappings to output results. Intuitively, this resembles the model of *probabilistic schema mappings* [13], although we do not use a probabilistic model. Our goal is to *learn* rankings based on answer feedback, and hence we need a ranking model amenable to this.

Our work is complementary to efforts on learning to construct mashups [34], in suggesting potential joins with new sources. Recent work on "pay as you go" integration has used decision theory to determine which feedback is most useful to a learner [19].

As opposed to feedback-driven query expansion and rewriting in [28], our goal here is to exploit user feedback to learn to correct schema matching errors. A method that learns to rank pairs of nodes based on their graph-walk similarity is presented in [27]. In contrast, the learning method used in this paper learns to rank *trees* derived from the query graph, and not just node pairs. The method for incorporating user feedback as presented in [8] requires developers to implement declarative user feedback rules. We do not require any such intermediate rule implementation, and instead *learn* directly from user feedback over answers.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have developed an automatic, information need-driven strategy for automatically incorporating new sources and their information in a data integration setting. Schema matches or alignments, whether good or bad, only become apparent when they are used to produce query answers seen by a user; we exploit this to make the process of finding alignments with a new source more efficient, and also to allow the user with an information need to actually *correct* bad mappings through explicit feedback (from which the system *learns* new association weights). Through experiments on real-world datasets from the bioinformatics domain, we have demonstrated that our alignment scheme scales well. We have also demonstrated that our learning strategy is highly effective in combining the outputs of "black box" schema matchers and in re-weighting bad alignments. In doing this, we have also developed a new instance-based schema matcher using the MAD algorithm.

We believe that Q represents a step towards the ultimate goal of automated data integration, at least for particular kinds of datasets. In ongoing work we are expanding our experimental study to consider a wider array of domains, including Web sources with information extraction components.

Acknowledgments

This work was supported in part by NSF IIS-0447972 and the DARPA Computer Science Study Panel. Thanks to Elisabetta Manduchi and Chris Stoeckert for sharing the GBCO dataset with us, and to the authors of COMA++ for providing us with assistance. We thank the anonymous reviewers for their valuable comments.

8. REFERENCES

[1] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, 2004.

[2] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly. Video suggestion and discovery for YouTube: taking random walks through the view graph. In *WWW*. ACM New York, NY, USA, 2008.

[3] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509, 1999.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[5] C. Botev and J. Shanmugasundaram. Context-sensitive keyword search and ranking for XML. In *WebDB*, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7), 1998.

[7] M. Cafarella, A. Halevy, D. Wang, E. Wu, and Y. Zhang. Webtables: Exploring the power of tables on the web. *VLDB*, 2008.

[8] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD*, New York, NY, USA, 2009.

[9] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, 1998.

[10] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585, 2006.

[11] H. H. Do and E. Rahm. Matching large schemas: Approaches and evaluation. *Inf. Syst.*, 32(6), 2007.

[12] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, 2001.

[13] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, 2007.

[14] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4), 2005.

[15] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *WWW*, 2003.

[16] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[17] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[18] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

[19] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, 2008.

[20] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[21] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. Star: Steiner-tree approximation in relationship graphs. In *ICDE*, 2009.

[22] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[23] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *VLDB*, 2001.

[24] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy. Google's deep web crawl. *PVLDB*, 1(2), 2008.

[25] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2), 2004.

[26] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, 2002.

[27] E. Minkov, W. W. Cohen, and A. Y. Ng. Contextual search and name disambiguation in email using graphs. In *SIGIR*, 2006.

[28] H. Pan, R. Schenkel, and G. Weikum. Fine-grained relevance feedback for XML retrieval. In *SIGIR*, 2008.

[29] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4), 2001.

[30] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan. Toward best-effort information extraction. In *SIGMOD*, 2008.

[31] P. Talukdar and K. Crammer. New Regularized Algorithms for Transductive Learning. In *ECML/PKDD (2)*, 2009.

[32] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. In *VLDB*, 2008.

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

[34] R. Tuchinda, P. Szekely, and C. A. Knoblock. Building mashups by example. In *IUI*, 2008.

[35] Z. Zhang, B. He, and K. C.-C. Chang. Understanding web query interfaces: Best-effort parsing with hidden syntax. In *SIGMOD*, 2004.

[36] X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. *ICML*, 2003.