A Framework for Semantic Link Discovery over Relational Data

Oktie Hassanzadeh[*†]
University of Toronto
oktie@cs.toronto.edu

Anastasios Kementsietsidis
IBM T.J. Watson Research Center
akement@us.ibm.com

Lipyeow Lim[*]
University of Hawaii at Manoa
lipyeow@hawaii.edu

Renée J. Miller[†]
University of Toronto
miller@cs.toronto.edu

Min Wang
IBM T.J. Watson Research Center
min@us.ibm.com

ABSTRACT

Discovering links between different data items in a single data source or across different data sources is a challenging problem faced by many information systems today. In particular, the recent Linking Open Data (LOD) community project has highlighted the paramount importance of establishing semantic links among web data sources. Currently, LOD sources provide billions of RDF triples, but only millions of links between data sources. Many of these data sources are published using tools that operate over relational data stored in a standard RDBMS. In this paper, we present a framework for discovery of semantic links from relational data. Our framework is based on declarative specification of linkage requirements by a user. We illustrate the use of our framework using several link discovery algorithms on a real world scenario. Our framework allows data publishers to easily find and publish high-quality links to other data sources, and therefore could significantly enhance the value of the data in the next generation of web.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

Semantic link discovery, record matching, linked data, data management

1. INTRODUCTION

From small research groups to large organizations, there has been tremendous effort in the last few years in publishing data online so that it is accessible to users. These efforts have been widely successful across a number of domains and have resulted in a proliferation of online sources. In the field of biology, there were 1078 major molecular databases at the beginning of 2008, 110 more than a year earlier. In the field of medicine almost every major hospital now has its own database of patient visits and clinical trials. In the Linking Open Data (LOD) community project at W3C, the number of published RDF triples has grown from 500 million in May 2007 to over 4.7 billion triples in May 2009 [6].

While publishing data is now easier than ever, attempts to establish semantic relationships between different published data sources has been less successful. So, from a user's perspective, online sources resemble *islands of data* (or *data silos*), where each island may contain only part of the data necessary to satisfy his or her information needs. Penetrating these silos to both understand their contents and understand potential semantic connections is a daunting task. Consider a biologist interested in a specific gene. It is not enough for the biologist to search for the gene, even using a robust search that accommodates for aliases and errors in the representation of data which are very common in web repositories. Both errors and aliases are domain specific so the biologist may have to try several *approximate search* methods to find one best for her domain. Furthermore, she may also want to find information about proteins or genetic disorders that are known to be related to this gene. Again, the search for this semantically related information must be tolerant of aliasing and errors, and yet must be tailored to the specific semantic relationships the user wishes to find.

What users need is automated support for creating referential links between data that reside in different sources and that are semantically related. Such links would provide a biologist with the ability to start from a gene and directly navigate to its protein and related genetic diseases, even through sources with no direct connection and which may use different naming conventions and different representations for information. Of course, discovering such links requires the use of both approximate matching (to overcome syntactic representational differences and errors) and semantic matching (to find specific semantic relationships). These two types of matching must be used in concert to accommodate for the tremendous heterogeneity found in web repositories.

In spite of their importance, research in discovering such semantic links has mainly focused on a more restricted version of the problem, namely, on *entity resolution*, i.e., the identification of entities in disparate sources that represent the same *real-world* entity. Yet, the general problem investigated here considers links between entities that are not necessarily identical, although they are semantically related (e.g., genes related with their corresponding proteins, medical treatments related with their corresponding clinical trials, etc.) The importance of discovering such links is also highlighted by the recent efforts of the LOD project, where a number of tools and frameworks have been developed that allow the generation and publication of linked data from relational databases. Examples of

[*]Work done while at IBM T.J. Watson Research Center.
[†]Partially supported by NSERC.

CIKM'09, November 2–6, 2009, Hong Kong, China.

such frameworks include D2RQ [7], Triplify [2] and OpenLink's Virtuoso[1]. Although these frameworks simplify the process of publishing linked data, the number of links between existing LOD data sources is two orders of magnitude less than the number of base data triples. The majority of the links are either a result of existing links in the relational data (e.g., links between two data items that are both derived from a Wikipedia page, both having the same URL), the result of (manual or automated) semantic annotation [9], or a laborious implementation of a semi-automatic and domain-specific linkage algorithm.

For typical users, this means they must experiment with a myriad of different link discovery methods to find one that suits their needs. In this paper, we seek to develop a generic and extensible framework for integrating link discovery methods. Our goal is to facilitate experimentation and help users find and combine the link discovery methods that will work best for their application domain. To ease experimentation, we use a declarative framework that permits the interleaving of standard data manipulation operators with link discovery.

Our framework permits the discovery of links within and between relational sources. We introduce LinQL, an extension of SQL that integrates querying with link discovery methods. Our implementation includes a variety of native link discovery methods and is extensible to additional methods, written in SQL or as user-defined functions (UDF). This permits users to interleave declarative queries with interesting combinations of link discovery requests. The link discovery methods may be syntactic (approximate match or similarity functions), semantic (using ontologies or dictionaries to find specific semantic relationships), or a combination of both.

Our first contribution is to show that by integrating *ad hoc* querying and a rich collection of link discovery methods, our framework supports rapid prototyping, testing and comparison of link discovery methods. A common way to use our framework would be to declaratively specify a portion of the data of interest (over which accuracy can be assessed) and to invoke one or more link discovery methods. The results can be evaluated by a user or automated technique, and the specification of the link method interactively refined to produce better results.

Often, link discovery algorithms are implemented using programming languages like Java or C by third-party developers, and are automatically invoked with arguments defined through the declarative specification. For data publishers these programs act as *black-boxes* that sit outside the data publishing framework and whose modification requires the help of these developers. Our second contribution addresses these shortcomings by leveraging native SQL implementations for a number of link discovery algorithms [12, 13]. Our approach has several advantages including the ability to (a) easily implement this framework on existing relational data sources with minimum effort and without any need for externally written program code; (b) take advantage of the underlying DBMS optimizations in the query engine while evaluating the SQL implementations of the link finding algorithms; and (c) use specific efficiency and functionality enhancements to improve the efficiency of these algorithms.

Our third contribution is to show how our declarative invocation of link methods permits users to tune link methods and their performance. The native support for methods permits customization where domain knowledge is available. We give examples where domain knowledge can be specified in the database and used to greatly enhance the performance of the discovery process.

[1] `http://www.openlinksw.com/virtuoso/`

Our final contribution is the implementation of our framework along with several link discovery algorithms on a commercial database engine. We describe a case study of how our framework can be used to discover links over real clinical trial data draw from a number of disparate web sources.

The rest of the paper is organized as follows. Section 2 introduces our running example, while Section 3 describes how links between data sources can be specified declaratively. Section 4 presents the algorithms for translating the link specifications into SQL queries. Our experimental study is described in Section 5. Section 6 highlights related work and we conclude in Section 7.

2. MOTIVATING EXAMPLE

Through out the paper (and in our case study in Section 5), we use an example from the health care domain drawn from a set of real-world data sources. One of our sources is a clinical trials database which includes the sample relation in Figure 1(a). For each trial, the *CT* relation stores its identifier *trial*, the *cond*ition considered, the suggested *inter*vention, as well as the *loc*ation, *city*, and related *pub*lication. Another source stores patient electronic medical records (EMR) and includes a patient visit relation *PV* (Figure 1(b)), which stores for each patient visit its identifier *visitid*, the *diag*nosis, recommended *prescr*iption, and the *location* of the visit. Finally, we consider a web source extracted from DBpedia (or Wikipedia) which stores information about drugs and diseases and includes the *DBPD* and *DBPG* relations (Figure 1(c) and (d)) that store the *name* of diseases and drugs in DBpedia, respectively.

We now describe briefly some types of links data publishers may like to discover between these sources. For the *CT* and *PV* relations, we note that the *cond*ition column in the *CT* relation is semantically related and can be linked to the *diag*nosis column in the *PV* relation. Such links may be useful to clinicians since they associate a patient's condition with related clinical trials, and might be used to suggest alternative drugs or interventions. In Figure 1, patient visit "VID770" with diagnosis "Thalassaemia" in the *PV* relation should be linked to the trial "NCT00579111" with condition "Hematologic Diseases" since "Thalass*a*emia" is a different representation of "Thalassemia" and according to the NCI medical thesaurus "Thalassemia" *is a type of* "Hematologic Diseases". As this example illustrates, a clinician may be interested in not only *same-as* relationships, but also hyponym relationships such as *type-of*. Similarly, note that the *inter*vention column in the *CT* relation can be linked to the *prescr*iption column in the *PV* relation. Such links can provide evidence for the relevance and effectiveness of a drug for a particular condition. For example, both patient visits in Figure 1(b) should link to trial "NCT00336362" (Figure 1(a)) based on the fact that "hydroxycarbamide", "Hydroxyura" and "Hydroxyurea" all refer to the same drug.

Additional links are possible if one considers the existence of links between the locations of patients and the presence of clinical trials in these locations. As an example, "Westchester Med. Ctr" from visit "VID777" could link to "Columbia University" based on the geographical information that both locations are in the New York state. Another interesting link discovery scenario arises when a user who is interested in a particular trial, wants to find other related trials based on certain criteria, e.g., the similarity of the title and authors of the trials' corresponding publications. Obviously, to be effective, links should be tolerant of errors and differences in the data, such as typos or abbreviation differences.

Data publishers often build online web-accessible views of their data. In such settings, they often want to provide links between their data and those in other online web sources. As an example, a web source of clinical trials requires links to other web sources

trial	*cond*	*inter*	*loc*	*city*	*pub*
NCT00336362	Beta-Thalassemia	Hydroxyurea	Columbia University	New York	14988152
NCT00579111	Hematologic Diseases	Campath	Texas Children's Hospital	Austin	3058228

(a) Clinical trials (*CT*)

visitid	*diag*	*prescr*	*location*
VID770	Thalassaemia	Hydroxyura	Texas Hospital
VID777	PCV	Hydroxycarbamide	Westchester Med. Ctr

(b) Patient visit (*PV*)

name
Thalassemia
Blood_Disorders

(c) DBpedia Disease (*DBPD*)

name
Alemtuzumab
Hydroxyurea

(d) DBpedia Drug (*DBPG*)

Figure 1: Sample relations

```
linkspec_stmt:= CREATE LINKSPEC linkspec_name
                AS link_method opt_args opt_limit;

link_method:= native_link | link_clause_expr | UDF;

native_link:= synonym | hyponym | stringMatch;

link_clause_expr:= link_clause AND link_clause_expr
                 | link_clause OR link_clause_expr
                 | link_clause;

link_clause:= LINK source WITH target
              USING link_terminal opt_limit;

link_terminal:= native_link | UDF | linkspec_name

opt_limit:= LINKLIMIT number;
```

Figure 2: The LinQL grammar

related to the trials like, say, the DBpedia or YAGO sources. In our sample relations, the above example translates into finding links between the *cond* and *inter* columns of *CT* and the *name* column of the *DBPD* and *DBPG* relations, respectively. The online trials data source can link the condition "Hematologic Diseases" to DBpedia resource (or Wikipedia page) on "Blood_Disorders", and link the intervention "Campath" to DBpedia resource "Alemtuzumab" using the semantic knowledge that "Campath" is the brand name for the chemical name "Alemtuzumab".

3. THE LINQL LANGUAGE

In this section, we introduce the LinQL declarative link specification language. We discuss the characteristics of the link finding primitives required by a flexible framework that is capable of effective link discovery in many real world scenarios, and show how LinQL supports such primitives. Although the linkage specification elements we discuss below are expressible in a number of different languages and notations (e.g., RDF/XML, N3, NTriples), in our framework (and implementation) we chose an SQL-like syntax.

A *link specification*, or *linkspec* for short, defines the conditions that two given values must satisfy before a link can be established between them. In more detail, a linkspec is defined using the grammar of Figure 2 (the full grammar is omitted due to space constraints). As shown in the figure, a `CREATE LINKSPEC` statement defines a new linkspec and accepts as parameters the name of the linkspec and the names of the relation columns whose values need to be linked. To create the links, our framework provides several *native* (or *built-in*) methods including synonym, hyponym, and a variety of string similarity functions (see more details on the native methods in the following sections). Such native methods can be used as such or they can be customized by setting their parameters.

3.1 Examples

EXAMPLE 1. *A common string similarity measure that has been shown to have good accuracy and efficiency is the weighted-Jaccard measure [13]. A user can create a link specification using this measure by setting the parameters used in the similarity computation. For example, she may set the threshold parameter to* 0.5*, the length of the q-gram to* 2*, and the maximum string length to* 50*, creating the following link specification*

```
CREATE LINKSPEC myJaccard1
AS weightedJaccard (0.5, 2, 50).
```

Now this link specification can be used as a join predicate in queries by any user. Notice that this specification does not indicate processing constraints.

A link specification can also be defined in terms of *link clause expressions*. Link clause expressions are boolean combinations of link clauses, where each link clause is semantically a boolean condition on two columns and is specified using either (a) a native method; (b) a user-defined function (UDF); or (c) a previously defined linkspec.

EXAMPLE 2. *Consider a setting in which a link between two values is established if a semantic relationship (e.g., synonym or hyponym) exists between these values in an ontology. This scenario commonly occurs in a number of domains, including healthcare, where sources are free to use their own local vocabularies (e.g., diagnosis and drug names) as long as these vocabularies can eventually be matched through a commonly accepted ontology (e.g., the NCI thesaurus). Assume that the ontology is stored in table* `ont` *with concept IDs in column* `cid` *and the terms in column* `term`*.*

The following linkspec illustrates the power of link clauses by creating a link between two values if their corresponding values in an ontology are synonyms of each other. In the linkspec, the weightedJaccard native linkspec is used to match individual values to corresponding values in the ontology, while the synonym native linkspec is used to test for the synonymy of the ontology terms.

```
CREATE LINKSPEC mixmatch
AS LINK src WITH tgt
   USING synonym(ont,cid,term) LINKLIMIT 10
   AND
   LINK src WITH ont.term
   USING myJaccard1 LINKLIMIT 10
   AND
   LINK ont.term WITH tgt
   USING myJaccard2 LINKLIMIT 10;
```

Clearly, the links between values are not necessarily one-to-one and, in general, a value from one relation can be linked to more than one values from a second relation. For example, it is common for drugs to have more than one name. Therefore, while a drug appears as *"aspirin"* in one relation it might appear as *"acetylsalicylic acid"* or *"ASA"* in another. When multiple such links are possible, users often want to limit the number of such links and only consider k results, or the *top-k* where ordering is possible. The `LINKLIMIT` essentially specifies the value of this *k* parameter.

EXAMPLE 3. *In the previous example, while defining the mixmatch linkspec,* `LINKLIMIT` *is set equal to 10, for all three link clauses. Hence, only the top 10 links are considered in each method.*

The previous examples consider the *local* version of the `LINKLIMIT` construct which is associated with a particular clause. The LinQL grammar also includes a *global* `LINKLIMIT` construct which is associated with the whole `CREATE LINKSPEC` statement which can be thought of as a post-processing filter of the links returned by the linkspec methods used in the statement.

We conclude the presentation of LinQL by introducing Boolean valued user-defined functions (UDFs). The primary difference between using a UDF versus a native linkspec method is that the UDF allows only simple substitution-based rewriting of the query, whereas the native linkspec uses a non-trivial rewriting of the query into SQL (see next section on the translation of LinQL to SQL). The ability to use UDFs in link specification is provided for extensibility to non-SQL-based linking functions. Of course, native implementations have numerous advantages. Using native implementations, the query optimizer can optimize the processing of link specifications yielding very efficient execution times. Furthermore, declarative implementations of the methods within a relational engine permit great extensibility as described in the following section.

EXAMPLE 4. *Suppose a user writes a UDF that implements his own similarity function. This UDF,* `myLinkUDF(thr, delC, insC, subC)` *returns true only if the edit similarity the values on which it is applied is above the threshold value* `thr` *(where* `delC`, `insC` *and* `subC` *are the costs of delete, insert and substitute operations, respectively). Then, the following linkspec can use that UDF as follows*

```
CREATE LINKSPEC myLink
AS myLinkUDF(0.5, 1, 1, 1).
```

In the previous paragraphs, we looked at how linkspecs are defined. We now show how linkspecs are used inside queries.

EXAMPLE 5. *Suppose we want to find the tuples in the PV relation for which there is a link with the condition in the CT relation, using the native weightedJaccard linkspec with default parameter values. Then, the following query can be used.*

```
SELECT PV.*, CT.*
FROM   visit PV, trial CT
WHERE  PV.visitid = 1234 AND
       CT.city='New York' AND
       PV.diag = CT.cond
       LINK PV.diag WITH CT.cond
       USING weightedJaccard LINKLIMIT 10
```

Notice that the linkspec here is essentially defined inline. For more complex linkspecs, or for situations where the same linkspec is used multiple times by one or more users, the query can refer to a previously defined linkspec in a similar fashion. This is a way for a DBA to provide a set of specifications for methods using the best parameter settings for different domains, making these methods more accessible to less expert users who may not know how to set the parameters. For example, in the query above we can use the `mixmatch` *linkspec instead of the weightedJaccard, as follows:*

```
SELECT PV.*, CT.*
FROM   visit PV, trial CT
WHERE  PV.visitid = 1234 AND
       CT.city='New York' AND
       PV.diag = CT.cond
       LINK PV.diag WITH CT.cond
       USING mixmatch LINKLIMIT 10
```

3.2 Native Link Methods

In what follows, we present the currently supported native methods, including our reasons for including them in the initial implementation.

3.2.1 Approximate String Matching Specification

String data is prone to several types of inconsistencies and errors including typos, spelling mistakes, use of abbreviations or different conventions. Therefore, finding similar strings, or approximate string matching (or aprioxmate join), is an important feature of an (online) link discovery framework. Approximate string matching is performed based on a similarity function $sim()$ that quantifies the amount of *closeness* (as opposed to *distance*) between two strings. A similarity threshold θ is set by the user to specify that there is a link from the base record to the target record if their similarity score, returned by function $sim()$, is above θ. The right value of the threshold depends on the characteristics of the dataset, the similarity function, and the application. The user can find the optimal value of the threshold for each application by trying different thresholds and manually evaluating the results.

There exists a variety of similarity functions for string data in the literature. The performance of a similarity function usually depends on the characteristics of data, such as length of the strings, and the type errors and inconsistencies present in the data. As stated earlier, in our framework we are interested in algorithms that are fully expressible in SQL (the benefits of which are well-known [12]). There are additional benefits of this choice for our application. Specifically, the use of SQL as an implementation language for our methods permits on-the-fly calculations of the similarity scores that can be enhanced dynamically to increase the functionality of the matching algorithm by relying on characteristics of the domain of the source and target relations.

A popular class of string similarity functions is based on tokenization of the strings into q-grams, i.e., substrings of length q of the strings. By using q-gram tokens, we can treat strings as sets of tokens and use a set similarity measure as the measure of similarity between the two strings. Furthermore, q-gram generation, storage and set similarity computation can all be done in SQL. This makes the following class of functions suitable for our framework.

• Size of the *intersection* of the two sets of q-grams, i.e., the number of common q-grams in the two strings.

• *Jaccard similarity* of the two sets of q-grams which is the size of the intersection set over the size of the union, i.e., the percentage of common q-gram tokens between the two strings.

• Weighted version of the above measures. The weight of each q-gram is associated with its *commonality* in the base (or target or both) data sources. The higher the weight of a q-gram, the more important the q-gram is. For example, when matching diagnosis across medical sources, q-grams for commonly occurring strings like "Disorder" or "Cancer" should have low weights so that the value of the similarity function for the strings "Coagulation Disorder" and "Phonation Disorder" is small, compared to that for the strings "Coagulation Disorder" and "Coagulation Disease".

There are several other string similarity measures including but not limited to Edit-Similarity, Jaro, Jaro-Winkler, SoftTFIDF, Generalized Edit similarity, and methods derived from relevance score functions for documents in information retrieval, namely Cosine with tf-idf, Okapi-BM25, Language Modeling and Hidden Markov Models. Some of these functions can be implemented in SQL and some others can only be implemented using a UDF. However, we focus only on the above q-gram based measures based on their better accuracy and efficiency and also their flexibility for functionality and efficiency enhancements, as discussed below.

Token Weight Assignment: To assign weights to q-gram tokens, we use an approach inspired by the Inverse Document Frequency (IDF) metric in information retrieval. IDF weights reflect the commonality of tokens in documents with tokens that occur more frequently in the documents having less weight. So, by analyzing (offline) the q-gram token frequency, we assign less weight to common tokens like "Disorder" or "Cancer" in a medical source. As a functionality enhancement, we also let the user manually specify in a user-defined table the weights of some tokens. These weights override the automatically assigned (IDF-based) weights for these tokens. Manual weight-assignment is useful in applications where the user has prior knowledge about the importance of some tokens. For example, when matching diagnosis across sources, the user knows that often the use of numbers plays a more important role in the diagnosis than the name of the disease itself. So, by assigning a very low (or negative) weight to numbers, wrong matches between highly similar strings like "Type 1 Diabetes" and "Type 2 Diabetes" can be avoided. Similarly, when matching conditions (e.g., "Diabetes", "Cancer") from an online source such as WebMD to their corresponding entries in, say, Wikipedia, the conditions in Wikipedia might include the term "(disease)" to disambiguate the disease from other terms with the same name (e.g., "Cancer" has close to seven entries in Wikipedia, in addition to the one for the disease, including one for astrology and one for the constellation). Knowing this, the user can adjust the weight of the otherwise common token "disease" to increase the likelihood of a correct link.

Scalability Although, we do not address explicitly the efficiency of the string matching implementation, our similarity predicates can be used along with several existing scalable indexing and hashing techniques. Examples of such techniques include the indexing algorithms of [3], the *Weighted Enumeration* (WTENUM) signature generation algorithm [1] and *Locality Sensitive Hashing* [14].

3.2.2 Semantic Matching Specification

Link discovery between values often requires the use of domain knowledge. In a number of domains, there are existing, commonly accepted, semantic knowledge bases that can be used to this end. In domains where such semantic knowledge is not available, users often manually define and maintain their own knowledge bases.

A common type of such semantic knowledge is an ontology. In the health care domain, well-known ontologies such as the NCI thesaurus are widely used and encapsulate a number of diverse relationship types between their recorded medical terms, including, synonymy, hyponymy/hypernymy, etc. Such relationship types can be conveniently represented in the relational model and (recursive) SQL queries can be used to test whether two values are associated with a relationship of a certain type [15]. Therefore, semantic knowledge in the form of ontologies can be seamlessly incorporated in our framework and used for the discovery of links. So, while considering links between two sources, semantic knowledge can be used to link a diagnosis on "Pineoblastoma" to one on "PNET of the Pineal Gland", since the two terms are synonyms of each other. Similarly, a diagnosis on "Brain Neoplasm" can be potentially linked with both of the previous diagnoses, since the latter term is a hypernym of the former terms. No level of sophistication in string matching can result in links such as the ones described earlier and therefore semantic matching complements the string matching techniques described in the previous section.

4. FROM LINQL TO SQL

In what follows, we describe the algorithm for translating a LinQL query to an SQL query, and then describe the algorithm for implementing each native link specification. Finally, we outline a number of efficient strategies to combine some of the specifications.

input : LinQL query L
output: SQL query Q

1 $lce \leftarrow$ extract link clause expr. from L;
2 $Q_{base} \leftarrow L - lce$;
3 $Q \leftarrow$ LINKCLAUSEEXPR2SQL(Q_{base}, lce);
4 **return** Q;

Figure 3: The LINQL2SQL Algorithm

input : An SQL base query Q_{base}, a link clause expression lce
output: SQL query Q

1 **if** $lce \neq \emptyset$ **then**
2 $l \leftarrow$ next link clause in lce;
3 $Q \leftarrow$ LINKCLAUSE2SQL(Q_{base}, l);
4 **if** *operator = AND* **then**
5 $Q \leftarrow$ LINKCLAUSEEXPR2SQL($Q, lce - l$);
6 **else if** *operator = OR* **then**
7 $Q \leftarrow Q +$ 'UNION' + LINKCLAUSEEXPR2SQL($Q_{base}, lce - l$);
8 **return** Q;

Figure 4: The LINKCLAUSEEXPR2SQL Algorithm

Algorithm LINQL2SQL translates a LinQL query to a SQL query by first splitting the former query into the base query (which is in SQL) and the link clause expression. The latter is translated into SQL by LINKCLAUSEEXPR2SQL by iterating through the boolean combination of link clauses and generating a separate SQL query for each clause. Then, the boolean combination of link clauses is translated into a set of intersections/unions between the generated SQL queries.

LINKCLAUSE2SQL parses a link clause to determine what type of link terminal is used. If the link terminal is a UDF, we simply add an invocation of the UDF in the where clause of the SQL base query. If the link terminal is a native link, we rewrite the SQL base query using the rewrite rules associated with that particular native link. If the link terminal is a reference to a named linkspec, we retrieve the associated linkspec statement and parse the associated link method. The link method can be a UDF, native link or link clause expression. UDFs and native links are translated as described previously. Link clause expressions are translated by a recursive call to the LINKCLAUSEEXPR2SQL sub-routine. The recursion stops when either a UDF or a native link is encountered.

The main translation logic is in the rewriting rules associated with the native links. A native link's rewriting rules are specified in two parts: view definitions and link conditions. For example, the rewriting rules for the *weightedJaccard* native link on *tab1.col1* and *tab2.col2* consists of the view definitions for *tab1col1weights, tab1col1sumweights, tab1col1tokenweights, tab2col2tokens, scores, scores2*, and the link conditions *scores.tid1=col1 AND scores.tid2=col2 AND s.tid1=s2.tid1 AND scores2.mx=s.score*.

The use of the view definition syntax is purely for readability. In practice, the SQL queries associated with the view definitions are inlined into the actual query itself (resulting in a possibly hard to read query). The WITH statement supported by some DBMS (e.g. IBM DB2) is another means for inlining some of the view definitions. Depending on the application, one may choose to materialize all or part of these views using *link index* statements in order to speed up the query time. We present the SQL queries based on view definitions in this section and discuss briefly the cost of materializing these views in the experimental results. The rest of this section describes the rewriting rules used to implement some of our native link methods.

input : A SQL base query Q_{base}, a link clause l
output: SQL query Q

```
1  Q ← Q_base ;
2  if link_terminal(l) = UDF then
3      add UDF invocation to where clause in Q;
4  else if link_terminal(l) = native_link then
5      rewrite Q using native_link's rewriting rules;
6  else if link_terminal(l) = link_spec_name then
7      get link_method from associated link_spec_stmt ;
8      if link_method = UDF then
9          add UDF invocation to where clause in Q;
10     else if link_method = native_link then
11         rewrite Q using native_link's rewriting rules;
12     else if link_method = link_clause_expr then
13         lce ← get associated link_clause_expr;
14         Q ← LinkClauseExpr2SQL(Q_base, lce);
15 return Q;
```

Figure 5: The LINKCLAUSE2SQL Algorithm

4.1 Approximate Matching Implementation

The rewriting of the approximate string matching native link specification into SQL consists of three steps, namely, (a) the creation of tables containing the tokenization of the strings into q-grams or word tokens; (b) the gathering of statistics and calculation of token weights from the token tables; and (c) the calculation of link scores based on the weights. In more detail:

Step 1: This step can be done fully in SQL using standard string functions present in almost every DBMS. Assume a table `integers` exists that stores integers 1 to N (maximum allowable length of a string). The main idea is to use basic string functions `SUBSTR` and `LENGTH` along with the sequence of integers in table `integers` to create substrings of length q from the string column `col1` in table `table1`. The following SQL code shows this idea for $q = 3$:

```
SELECT tid, SUBSTR(col1,integers.i,3) as token
FROM   integers INNER JOIN table1
       ON integers.i <= LENGTH(col1) + 2
```

In practice, the string `col1` is used along with `UPPER()` (or `LOWER()`) functions to make the search case insensitive. Also, the string is padded with $q - 1$ occurrences of a special character not in any word (e.g. '$') at the beginning and end using the `CONCAT()` function. Similarly the spaces in the string are replaced by $q - 1$ special characters. In case of tokenization using word tokens a similar SQL-based approach can be used. At the end of this process, the token generation queries are declared as views, or are materialized in tables like `table1_col1_tokens` and `table2_col2_tokens`.

Steps 2 and 3: These steps are partly native-link specific and are more easily presentable through an example. In what follows, we use the weighted Jaccard specification as an example.

EXAMPLE 6 (WEIGHTEDJACCARD NATIVE LINKSPEC). *Consider the following* LinQL *specification:*

```
SELECT PV.visitid, CT.trial
FROM   visit AS PV, trial AS CT
WHERE  PV.visitid = 1234 AND CT.city='NEW YORK' AND
       PV.diag = CT.cond
       LINK PV.diag WITH CT.cond
       USING weightedJaccard
```

This specification is translated into the following SQL queries. Initially, two queries calculate the IDF weights for the tokens and the auxiliary views/tables needed for the final score calculation:

```
CREATE VIEW visit_diagnosis_weights AS
SELECT token, LOG(size - df + 0.5) - LOG(df+0.5) as weight
FROM   ( SELECT token, count(*) as df
       FROM     (SELECT * FROM visit_diagnosis_tokens
                GROUP BY tid, token) f
       GROUP BY token ) D,
       ( SELECT count(*) as size
       FROM visit_diagnosis_tokens ) S

CREATE VIEW visit_diagnosis_sumweights AS
SELECT tid, B.token, weight
FROM   visit_diagnosis_tokenweights idf,
       (SELECT DISTINCT tid, token
       FROM visit_diagnosis_tokens B) B
WHERE  B.token = idf.token

CREATE VIEW visit_diagnosis_tokenweights AS
SELECT      tid, sum(weight) as sumw
FROM        visit_diagnosis_weights
GROUP BY    tid
```

Then, the next query returns the links along with their final scores:

```
WITH   scores(tid1, tid2, score) AS (
       SELECT tid1, tid2,
              (SI.sinter/(BSUMW.sumw+QSUMW.sumw-SI.sinter))
              AS score
       FROM   (SELECT  BTW.tid AS tid1,QT.tid AS tid2,
                       SUM(BTW.weight) AS sinter
              FROM     (SELECT * FROM visit_diagnosis_weights
                       WHERE id = 1234) AS BTW,
                       trials_condition_tokens AS QT
              WHERE    BTW.token = QT.token
              GROUP BY BTW.tid, QT.tid) AS SI,
              (SELECT  *
              FROM     visit_diagnosis_sumweights
              WHERE    id = 1234 ) AS BSUMW,
              (SELECT  Q.tid, SUM(BTW.weight) AS sumw
              FROM     trials_condition_tokens Q,
                       visit_diagnosis_tokenweights AS BTW
              WHERE    Q.token = BTW.token
              GROUP BY Q.tid ) AS QSUMW
       WHERE  BSUMW.tid=SI.tid1 and SI.tid2 = QSUMW.tid )
SELECT PV.visitid, CT.trial
FROM   scores AS s, visit AS v, trials AS t,
WHERE  PV.visitid = 1234 AND CT.city='NEW YORK' AND
       s.tid1=PV.visitid AND s.tid2=t.trial
```

4.2 Semantic Matching Implementation

Assume that the synonym and hyponym data are stored in two tables `synonym` and `hyponym` with columns `src` and `tgt`. The column `src` contains *concept IDs* of the terms, and the column `tgt` contains the terms. This is a common approach in storing semantic knowledge, used in NCI thesaurus and Wordnet's synsets for example. Alternatively, this data could be stored in a table `thesaurus` with an additional column `rel` that stores the type of the relationship, or it could even be stored in XML. In the case of XML, `synonym` and `hyponym` can be views defined in a hybrid XML relational DBMS such as DB2. For brevity, we limit our discussion in this paper to semantic knowledge stored as relational data, although our framework is easily extensible to other formats. We show the details of the SQL implementation of the synonym and hyponym native link specifications in the following two examples.

EXAMPLE 7 (SYNONYM NATIVE LINKSPEC). *Consider the following query written using* LinQL.

```
SELECT PV.visitid, CT.trial
FROM   visit AS PV, trial AS CT
WHERE  PV.visitid = 1234 AND CT.city='NEW YORK' AND
       PV.diag = CT.cond
       LINK PV.diag WITH CT.cond
       USING synonym
```

This query is rewritten to:

```
SELECT DISTINCT PV.visitid, CT.trial
FROM   trials AS CT, visit AS PV, synonym AS syn
WHERE  PV.visitid = 1234 AND CT.city='NEW YORK' AND
       (src in (SELECT src
```

```
              FROM synonym s
              WHERE s.tgt = CT.cond))
       AND PV.diag = syn.tgt
UNION
SELECT PV.visitid, CT.trial
FROM   trials AS CT, visit AS PV
WHERE  PV.visitid = 1234 AND CT.city='NEW YORK' AND
       CT.cond = PV.diag
```

EXAMPLE 8 (HYPONYM NATIVE LINKSPEC). *Consider the following query written using* LinQL.

```
SELECT PV.visitid, CT.trial
FROM   visit AS PV, trial AS CT
WHERE  PV.visitid = 1234 AND CT.city ='NEW YORK' AND
       PV.diag = CT.cond
       LINK PV.diag WITH CT.cond
       USING hyponym
```

This query is rewritten to:

```
WITH   traversed(src, tgt, depth) AS (
       (SELECT src,tgt,1
       FROM    hyponym AS ths
       UNION ALL
       (SELECT ch.src, pr.tgt, pr.depth+1
       FROM    hyponym AS ch, traversed AS pr
       WHERE   pr.src=ch.tgt AND
               pr.depth<2 AND ch.src!='root_node'))
SELECT distinct PV.visitid, CT.trial
FROM   trials AS CT, visit AS PV, hyponym AS ths
WHERE  PV.id = 1234 AND CT.city ='NEW YORK' AND
       (src in (SELECT distinct src
               FROM traversed tr
               WHERE tr.tgt = CT.cond)) AND
       PV.diag = ths.tgt
```

Note that the hyponym depth is by default set to 2, which could be customized to any other value.

5. CASE STUDY

The main goal of this section is twofold. First, we illustrate the flexibility of our framework by applying it in a variety of linkage scenarios. Second, we use these scenarios to justify our choices in terms of functionality for the various components in our framework. We build our scenarios around an online database of clinical trials published on ClinicalTrials.gov. This database is a registry of federally and privately supported clinical trials conducted in research centers all around the world. It contains detailed information about the trials, including information about the conditions associated with the trials, their eligibility criteria and locations.

5.1 Datasets

The clinical trials database used in our experiments contains approximately 61,920 trials. Originally, the database was in XML format. Using the functionality of DB2 as a hybrid relational-XML DBMS, we stored all the data in relational tables. Other datasets that we used in our experiments for linkage include a database of patient visits or Electronic Medical Records (EMR) and DBpedia (Wikipedia) entries about diseases and drugs. We also used the National Cancer Institute (NCI)'s thesaurus as a source of semantic information about medical terms. Detailed statistics on these datasets is shown in Table 1.

Due to privacy issues associated with EMR records, our patient visits database is synthetic, generated using a data generator that resembles real EMR records in a hospital. The diagnosis and prescription values are randomly picked by the data generator from NCI terms. The data generator also creates an additional column with a small random string error in the diagnosis field. The error injected in the string resembles real errors and typos occurring in string databases, e.g., replacing a character with an adjacent character on a keyboard, or swapping two characters or word tokens.

Dataset	Entity	Count
Clinical Trials (CT)	Trial	61,920
	Condition	14,055
	Intervention	42,333
	Drug (Intervention)	21,396
	Publications	45,138
Patient Visits (PV)	Visit	10,000
	Diagnosis (clean)	9,151
	Diagnosis (dirty)	9,319
	Prescription (drug)	8,290
	Therapy	5,114
DBpedia (Wikipedia)	Disease	5,486
	drug	2,235
NCI Thesaurus	Concept IDs	63,924
	Terms	159,291
	'hasSynonym' Relationships	168,932
	'hasHyponym' Relationships	72,486

Table 1: Dataset Statistics

5.2 Effectiveness and accuracy results

In what follows, we describe several link discovery scenarios involving clinical trials. While the first scenario is described in more detail (including its intermediate steps and corresponding linkage specifications), for the other scenarios we only show the final results and only mention changes to preceding LinQL statements.

Case 1 (Linking patient visits to trial conditions) The objective here is to discover links to clinical trials that are related to the conditions of certain patients. For this study, we consider 1,000 random patients from table PV, where column DIAGNOSIS stores the condition associated with a patient's visit. The CT table stores the trial condition in its column CONDITION. The records matched by a simple exact matching are obtained by the SQL query:

```
SELECT v.*, c.*
FROM   PV v, CT c
WHERE  v.DIAGNOSIS = c.CONDITION
```

The query returns only 33 matches, linking only 2 out of 1,000 patient visit records to matching clinical trials. This is due to the string errors in DIAGNOSIS values. As a next step, we try an approximate string matching predicate with a low similarity threshold using the following linkspec and query:

```
CREATE LINKSPEC weightedJaccard04
AS weightedJaccard (0.4, 2, 50).
```

```
SELECT v.*, c.*
FROM   PV v, CT c
WHERE  v.DIAGNOSIS = c.CONDITION
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING weightedJaccard04
```

Since matching a condition to the *right* trial is imperative here, we are rather strict (conservative) in the application of approximate matching. For example, links from "Alpa Thalassemia" (misspelled record of "Alpha Thalassemia") to "Alpha Thalassemia", "α-Thalassemia" and "Thalassemia" are considered correct and we would like to find them. However, "Beta Thalassemia" is considered an incorrect link. The following accuracy results were obtained by investigating 100 random queries using different thresholds:

Threshold	Number of Links	Accuracy
0.70	22	91%
0.65	36	86%
0.60	63	84%
0.55	104	77%
0.50	182	66%
0.45	303	54%
0.40	579	40%

Therefore, by choosing a high threshold 0.70, 22 links are returned out of which 20 (91%) are correct. However, by choosing threshold 0.4, 579 links are returned (more than 5 links per each patient visit), but only 231 (40%) of them are correct. Given these observations, a user can choose the appropriate threshold that works best for the specific linkage needs. For example, we choose threshold 0.55 that returns on average almost one link per visit, and has a reasonable accuracy. As a result we will have 1,102 links to clinical trials from 335 (out of 1,000) distinct patient visits.

The next step is to use the semantic information in NCI to improve the matching using the LinQL query below:

```
SELECT v.*, c.*
FROM   PV v, CT c
WHERE  v.DIAGNOSIS = c.CONDITION
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING synonym
```

The semantic matching based only on synonyms results in 147 links to 104 distinct trials. From these, 69 links to 24 distinct trials could not be found using exact or string matching. Repeating the above query with semantic matching based on hyponyms of depth 2 from NCI, results in 68 additional links to 21 distinct trials. One reason for the relatively low number of matches based on synonyms and hyponyms is the string errors present in the DIAGNOSIS values of the PV table. This calls for using string matching combined with semantic matching. The LinQL code to do this is:

```
CREATE LINKSPEC mixmatch
AS LINK source WITH target
   USING synonym(ont,cid,term)
   AND
   LINK source WITH ont.term
   USING weightedJaccard
   AND
   LINK ont.term WITH target
   USING weightedJaccard;

SELECT v.*, c.*
FROM   PV v, CT c
WHERE  v.DIAGNOSIS = c.CONDITION
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING mixmatch
```

Using combined string matching and semantic matching results in 173 links to 120 distinct trials, 26 more links to 16 more distinct trials when compared with matching based on synonyms only. Depending on the results of the above steps, the user can write a single query for the linkage needs specific to the application. Here we choose to combine exact matching, string matching, semantic matching based on synonyms and hyponyms, and mixed semantic matching allowing string errors. This can all be expressed using the query below:

```
SELECT v.*, c.*
FROM   PV v, CT c
WHERE  v.DIAGNOSIS = c.CONDITION
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING weightedJaccard
       OR
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING synonym
       OR
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING hyponym
       OR
       LINK v.DIAGNOSIS WITH c.CONDITION
       USING mixmatch
```

The combined approach results in 1,255 links from 383 visit records to the related clinical trials. Overall, we have:

	Links #	Entities #
1.Exact Match	33	2
2.String Match	1,102	335
3.Synonym Match	147	104
4.Mixed Match	173	120
5.Hyponym Match	68	21
Total (Combined)	1,255	383

These results can help a user better understand both her data and the (combinations of) link methods that are suitable for her needs.

Case 2 (Linking prescriptions to trial interventions) Now consider a user who wishes to link patients who were prescribed a drug with *all* clinical trials that use that drug. To collect all these trials, the user will need the results produced from a variety of algorithms. A sample of such results is shown in the table below. These results use threshold 0.6 for weightedJaccard string matching, and depth 1 for hyponym matching. The table summarizes the results obtained for matching 1,000 random drug prescriptions:

Method	Links #	Entities #
1.Exact Match	318	88
2.String Match	806	289
3.Synonym Match	4,225	355
4.Hyponym Match	2,410	44
Total (Combined)	6,630	500

Notice that some methods do find the same links (that is, the total is less than the sum of the methods). However, the overlap is not that big. For this application, if the goal is to find as many possible matches as possible, all four of these methods add value.

Case 3 (Linking trial conditions to DBpedia diseases) and **Case 4 (Linking trial interventions to DBpedia drugs)** In these scenarios, we are seeking links from the clinical trials' condition and interventions fields to the DBpedia (or Wikipedia) disease and drug categories, respectively. Unlike the previous cases, assume here that the user only needs to link to a single DBpedia entry per each condition and drug. This makes sense since in most cases there should be a single record in DBpedia for a single disease (condition) or drug intervention in the trials data. Therefore the user uses the LINKLIMIT 1 option in the LinQL query to limit the number of matches. Then, when an exact match is found for a record, there is no need to look for approximate string or semantic matches for that record. Considering the running times reported in Section 5.4, this leads to a significant performance improvement.

The linkage specification query for matching conditions to DBpedia diseases is as follows. The query for trial interventions to DBpedia drugs is similar.

```
SELECT c.*, d.*
FROM   CT c, DBPD d
WHERE  c.CONDITION = d.NAME
       LINK c.CONDITION WITH d.NAME
       USING weightedJaccard
       OR
       LINK c.CONDITION WITH d.NAME
       USING synonym
       OR
       LINK c.CONDITION WITH d.NAME
       USING hyponym
       OR
       LINK c.CONDITION WITH d.NAME
       USING mixmatch
       LINKLIMIT 1
```

Again we choose threshold 0.6 for string matching based on investigation of the accuracy of a few random queries. The table below summarizes the results for different steps of the matchings from 1,000 condition and drug interventions:

1.Method Match	Disease Links#	Drugs Links#
2.Exact Match	16	9
3.String Match	180	33
4.Synonym Match	12	21
5.Mixed Match	141	22
Total	248	62

Notice the obvious need for allowing mixed string and semantic matching in these two cases. The trials source, NCI thesaurus and DBpedia/Wikipedia names all use different conventions and therefore there are cases where strings do not exactly match. For example, "Adenocarcinoma of Esophagus" in trials matches with "Carcinoma of Esophagus", synonym of "Esophageal Cancer" in the thesaurus which matches with "Esophageal_cancer" in DBpedia.

Case 5 (Finding related trials) To show the flexibility of our framework, we investigate its effectiveness in a rather different scenario. In this case, the goal is linking trials that are related to each other. Different attributes and measures can be used to identify trials that are related. In this experiment, we use the `pub` attribute of the trials and consider two trials related if the title and authors of their associated publications are similar. Our trials database stores publications associated with the trials in a single long text record that includes the names of the authors, title of the paper, the conference or journal and the date of the publication. Therefore, in order to find similarity we cannot use any type of semantic information about the strings. Furthermore, we are not interested in typos and different representations of the same string here. Instead the similarity function should measure the amount of co-occurrence of (important) words in the two strings. The following weightedJaccard linkspec performs matching based on word tokens:

```
CREATE LINKSPEC wordTokenJaccard
AS weightedJaccard (0.5, 0, 100)
```

Using the linkspec over 10,000 random trials results in 2,074 links, whereas exact matching results in only 11 matches.

5.3 Effectiveness of Weight Tables

In what follows, we briefly show the effectiveness of the functionality enhancement we proposed based on manual definition of a weight-adjustment table by the user. Assume that the user defines the following simple weight table:

String	Weight
'Syndrome'	0.4
'The'	0.1
'Disorder'	0.2
'Disease'	0.2

We repeat the experiment for Case 1 (linking to trial conditions from a database of patient visits) with updated weight tables based on the above input weight adjustment table. String matching with the same settings, i.e., using `weightedJaccard` similarity function with threshold 0.6 on 1,000 random base records, results in 121 additional links out of which 91 are correct (accuracy 75%) and drops 63 of the links found with no weight adjustments out of which 15 were wrong matches. This means that overall, the matching has resulted in 58 more links (5% increase) with roughly the same accuracy as the case with no adjustments.

Note that we obtained these results by choosing the weight adjustment values in the above simple table based only on our domain knowledge, and we have not varied the values to obtain the best results. What is more important is that we can use this method and leverage weight values to improve the accuracy of the link discovery, as a result of our SQL-based implementation method. The implementation of the weighted-Jaccard similarity function and the above customization of weights using a UDF, rather than a native method, could be quite complex and inefficient.

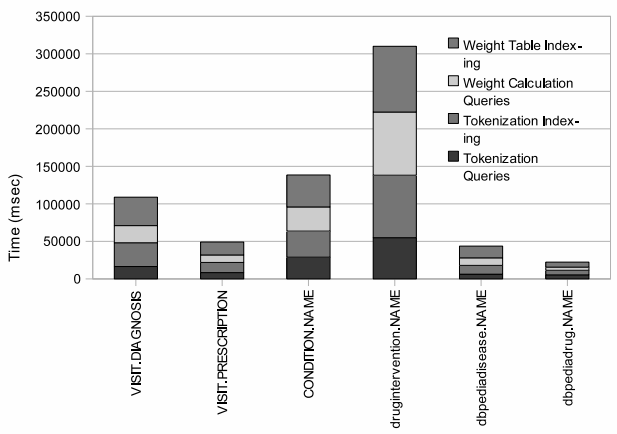

Figure 6: Preprocessing Time for the Datasets

5.4 Performance Results

As mentioned earlier, our focus in this paper is on the functionality of the framework and we do not address efficiency, although as described in Section 3.2 several hashing and indexing techniques can be applied to our framework to make it more efficient. However, we report running times of the above examples to show the performance of the system without any of these enhancements. We ran the experiments on a Pentium 4 3GHz HT CPU with 3GB of RAM, running Windows XP SP2. To obtain statistical significance, we report the average time from several runs of each experiment.

The table below shows the running time (in seconds) for 1,000 random queries for exact, synonym and hyponym matchings with depth 1 and 2 for the queries presented in our case study. Notice these queries involve approximate joins on relatively large tables.

	Case 1	Case 2	Case 3	Case 4
Exact Match	0.031	0.062	0.031	0.031
Synonym Match	0.515	0.474	0.323	0.333
Hyponym Depth 1	2.432	2.547	2.317	2.328
Hyponym Depth 2	4.443	4.594	4.317	4.276

For the string matching performance, due to the nature of our data sources it is reasonable to materialize token and weight table views in a preprocessing step and index them for better efficiency. The time required for this preprocessing of the table columns related to all the cases in our scenarios is shown in Figure 6, including the time for tokenization, weight table generation and the associated indexing times. As shown in this figure for all the cases the preprocessing time is relatively low. Using the preprocessed tables in our linkage cases, the following table shows the running time (in seconds) per each query.

	Case 1	Case 2	Case 3	Case 4
String Match	1.555	3.163	0.287	0.091
Mixed Match	60.189	63.142	59.322	30.425

6. RELATED WORK

The idea of Linked Data has recently attracted a lot of attention in the semantic web community. Linked Data is a method of publishing data on the web based on principles that significantly enhance the adaptability and usability of data, either by humans or machines. The notable growth of linked data sources as a part of the Linking Open Data Community project is in part a result of

technologies recently developed and adopted to simplify publishing such data sources. A wide variety of data publishing methodologies based on generation of RDF view over relational data are widely used in these data sources. These methodologies are often based on declarative specification of the mapping between relational tables and RDF triples. These frameworks include, but are not limited to, D2RQ and D2Rserver [7], Openlink Virtuoso and Triplify [2]. The success of these tools motivates a similarly declarative framework for link discovery such as ours so that not only the data sources can be published according to the principles of publishing linked data, but also they can be interlinked to other existing data sources, which is another important principle of linked data.

Duplicate detection, also known as entity resolution or record linkage has been the subject of extensive study in different communities. A recent survey [10] contains an overview of various techniques and algorithms used for duplicate record detection in databases. Many AI and machine learning techniques have been applied for entity resolution. The online entity resolution framework of [4] presents techniques for query-time entity resolution specifically designed for data that contains co-occurrence and relational information such as bibliographic data. Another closely related area is the work on declarative data quality and cleaning [5, 11, 13]. A distinctive feature of our framework comparing with all the existing techniques is our focus on discovering links and entity matching not necessarily for cleaning or duplicate detection purposes. Our work complements and extends work on semantic matching ([8, and others]) and semantic annotation and tagging (see for example, Dill et al. [9]). We provide a framework for fast prototyping and testing of semantic and syntactic matching which could exploit semantic annotations, if available. A key advantage of our approach is allowing string matching along with semantic matching which is crucial in many real world matching scenarios. Moreover, our specification language allows the definition of new operators, which could be a mix of several semantic and string matching operators.

7. CONCLUSION

In this paper, we presented a declarative extensible framework for link discovery from relational data. We proposed a simple specification language, LinQL, along with the details of its implementation. We adopted and extended existing string matching and semantic matching techniques, and proposed functionality enhancements specifically designed for our framework. We showed the effectiveness of our approach in several link discovery scenarios in a real world health care application. Our focus has been on developing efficient techniques that can handle large data sets, but also on usability. We showed how a user can interactively experiment with and customize different link methods to better understand what are the most effective methods for her domain. We believe that our framework can significantly enhance the process of publishing a high-quality data source with links to other data sources on the web. A user/data publisher can use our framework to easily find the appropriate linkage algorithm for the specific application, as well as the optimal value of the required parameters. Our framework combined with an existing popular declarative approach for generating linked data on the web such as [7], can lead to a quick and simple way of publishing an online data source with high-quality links. This could significantly enhance the value of the data in the next generation of web.

8. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient Exact Set-Similarity Joins. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 918–929, 2006.

[2] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: Light-Weight Linked Data Publication from Relational Databases. In *Int'l World Wide Web Conference (WWW)*, pages 621–630, 2009.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *Int'l World Wide Web Conference (WWW)*, pages 131–140, Banff, Canada, 2007.

[4] I. Bhattacharya and L. Getoor. Query-time Entity Resolution. *Journal of Artificial Intelligence Research (JAIR)*, 30:621–657, 2007.

[5] A. Bilke, J. Bleiholder, C. Böhm, K. Draba, F. Naumann, and M. Weis. Automatic Data Fusion with HumMer. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1251–1254, 2005.

[6] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data: Principles and State of the Art. In *Int'l World Wide Web Conference (WWW)*, November 2008.

[7] C. Bizer and A. Seaborne. D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs. In *Proc. of the Int'l Semantic Web Conference (ISWC)*, November 2004.

[8] S. Das, E. I. Chong, G. Eadon, and J. Srinivasan. Supporting Ontology-Based Semantic matching in RDBMS. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1054–1065, 2004.

[9] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien. SemTag and Seeker: Bootstrapping the Semantic Web via Automated Semantic Annotation. In *Int'l World Wide Web Conference (WWW)*, 2003.

[10] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[11] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 371–380, 2001.

[12] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 491–500, 2001.

[13] O. Hassanzadeh. Benchmarking Declarative Approximate Selection Predicates. Master's thesis, University of Toronto, February 2007.

[14] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-Preserving Hashing in Multidimensional Spaces. In *ACM Symp. on Theory of Computing (STOC)*, pages 618–625, 1997.

[15] A. Kementsietsidis, L. Lim, and M. Wang. Supporting Ontology-based Keyword Search over Medical Databases. In *Proceedings of the AMIA 2008 Symposium*, pages 409–13. American Medical Informatics Association, 2008.