Computational Cost of Querying for Related Entities in Different Ontologies

Chung Ming Cheung Yinuo Zhang Anand Panangadan Viktor K. Prasanna
University of Southern California
Los Angeles, CA 90089, USA
Email: {chungmin, yinuozha, anandvp, prasanna}@usc.edu

***Abstract*—The computational cost of querying for similar entities across ontologies is high since, in the worst case, every pair of entities will have to be considered. Therefore, links discovered during ontology alignment have been used to speed up querying across ontologies by following relatedness links to discover similar entities. We derive the computational complexity of querying across ontologies using the ontology alignment links discovered using the Unified Fuzzy Ontology Matching (UFOM) framework. We consider querying for related entities by following either a single alignment link or by following multiple alignment links. These methods have different computational complexity and produce different query results. We also study the impact of the specific implementation approach on query time. We consider implementations based on multiple accesses of the triplestore using a high-level procedural language and by execution of a single SPARQL graph query on the ontology server. These approaches were evaluated using ontologies derived from an enterprise-scale dataset. Experimental results show that an implementation using nested for-loops in a procedural language outperformed by nearly 2× an implementation based on a single SPARQL query.**

***Keywords*-Ontology matching; Ontology alignment; SPARQL; Semantic Web;**

I. Introduction

One of the advantages of using ontologies for storing data in enterprise-scale applications is the ability to automatically discover links across ontologies. This process is called ontology alignment and is an active area of research [1]. Ontology alignment makes it possible for end-users to query for related entities in multiple ontologies starting from a single entity. However, the computational cost of querying for similar entities can be high since, in the worst case, every pair of entities across ontologies will have to be considered. This high computational cost makes querying across ontologies infeasible in real-world enterprise-scale applications.

Recently, we proposed the Unified Fuzzy Ontology Matching (UFOM) framework for automatically detecting linkages between ontologies [2]. Unlike other ontology alignment approaches [3], [1], UFOM is designed to discover linkages of arbitrary relation types. These linkages can be exploited for efficient querying for similar entities by following relatedness links to discover similar entities instead of evaluating every entity contained in the ontologies. In this work, we derive the computational complexity of querying across ontologies using the UFOM framework. We consider querying for related entities by following only a single alignment link (*direct matching*) or by following multiple alignment links (*indirect matching*). While direct matching is computationally less intensive, indirect matching is capable of discovering entities that are not directly related to the given query entity.

After querying methods are designed to operate across ontologies, the impact of the specific implementation approaches on computation time has to be studied. Specifically, it is important to determine if the complex query should be implemented in a high-level procedural language or it should be executed as a graph query which can be optimized by an ontology server. We evaluated these different implementation approaches using ontologies from an enterprise-scale dataset (hundreds of thousands to millions of triples). Experimental results show that an implementation using nested for-loops in a procedural language outperformed an implementation based on providing a SPARQL query to an ontology server. This indicates that SPARQL query optimization strategies can be improved when operating across ontologies.

The contributions of this paper are:

1) Two algorithms for querying for similar entities across ontologies that are based on following links discovered during ontology alignment and a derivation of their computational complexity
2) Specification of the direct matching algorithm as a single SPARQL query that can be executed by an ontology server
3) Quantitative evaluation of the running times of the proposed query algorithms as implemented using a procedural language and as a graph pattern executed by an ontology server

The rest of the article is organized as follows. Section II gives an overview of the related work. Section III provide background knowledge of our work. Section IV summarizes our previous work on the UFOM ontology alignment approach which forms the basis for the query algorithms. Section V describes the data processing steps that need to be performed before similarity queries can be executed across ontologies. Section VI describes the proposed matching algorithms and their computational complexity. In Section VII, we present the experimental evaluation of the proposed query algorithms. We conclude in Section VIII.

II. Related work

There are four different fields of research that are related to this work. The first is *schema matching*. This represents the task of identifying semantically related objects across database schemas. Applications of schema matching include data integration, database warehouses and information exchange between different companies. Schema matching approaches are based on computing similarity between entities [4] such as name-based similarity, instance-based similarity, constraint-based similarity, and similarity flooding [5]. The similarity may also be computed using machine learning methods if training data is available.

A closely related field to schema matching is *ontology matching*. An *ontology* is the conceptualization of things and phenomena. Ontology matching is the matching of similar concepts that could be modeled in different ways within different ontologies. Ontology matching can also incorporate datasets that are not purely textual data such as spatial-temporal data. There are many approaches to ontology matching [1], including those based on machine learning, name-based similarity, instance-based similarity [6], anchor flooding [7], and mapping to upper ontologies [8]. Some of the research into ontology matching is also domain-specific [9]. In this work, we use the ontology matching framework UFOM [2] to integrate ontologies and investigate the best way to optimize query execution on this framework.

Record Linkage is a field that has been studied extensively. It is the task of finding equivalent records between databases. Some of the results of this research can be applied to schema matching and ontology matching. For example, we apply some of the techniques developed to pre-process datasets for the UFOM matching process. [10] uses Bayesian inference to formulate probability equations to model the Record Linkage problem. Then, Gibbs sampling is used to estimate the joint probability of records. The records that are most likely to belong to a pair based on this modeling are output from the probability simulation step.

Fourth, the field of *constraint discovery* in databases is related to our work. A schema may not always be available with a full specification; even the creator of the schema may not be aware of all foreign key relationships. Constraint discovery methods have been applied to automatically detect primary keys and foreign keys. Foreign key discovery is defined as the task of discovering inclusion dependencies. Some methods to approach constraint discovery are to sort and filter two lists of instances [11], by machine learning [12], and by distribution [13]. Such constraint relationships on schemas help with both the matching process and query execution process in our work.

III. Background

An ontology is a particular organization of things and phenomena in a domain. An ontology organizes similar concepts into a *class* with *properties* defining semantic relationships between classes. Objects in the domain are *instances* of a class in the ontology. Since data in many real-world applications is still stored in relational databases, we draw parallels between relational database and Semantic Web ontology concepts. A *table* in a relational database may be considered to correspond to an ontology class and the *fields* (columns) of the table correspond to properties. Then, the entries in a column are instances of a class in the corresponding ontology. Throughout this paper, the term *field pair* is used to describe a pair of related properties or fields.

A single domain may be modeled differently leading to different ontologies. Finding correspondences of classes and their properties between such ontologies, the problem of *ontology matching*, is useful in many applications. Common approaches for matching includes name-based, instance-based and structural-based methods. Our method for ontology matching, called UFOM, is described in Section IV.

In this paper, we focus on analyzing the computational cost of executing inter-ontology queries using the UFOM framework. Inter-ontology queries are related to joining of tables, i.e., finding and combining matching pairs of instances in the Cartesian product of instances of the two tables. In Note that the queries only return related entities – we do not attempt to solve the harder problem of harmonizing two ontologies. Section VI, we describe in detail this problem and algorithms for executing these queries.

IV. Unified Fuzzy Ontology Matching (UFOM)

In this section, we summarize the UFOM ontology alignment framework that was introduced in [2]. The proposed query process uses the fuzzy ontology alignments derived from the UFOM framework. UFOM computes such alignments based on both a *similarity score* and a *confidence level* for every possible correspondence in the ontologies. In order to provide an extensible framework, every relation score is computed from a set of pre-defined similarity functions. The design of the UFOM framework is illustrated in Figure 1.

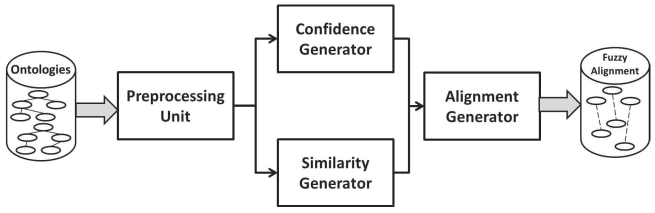


Figure 1. Components of the UFOM system for computing fuzzy ontology alignment

UFOM takes two ontologies as its input and outputs a fuzzy alignment between them. UFOM consists of four components: Preprocessing Unit (PU), Confidence Generator (CG), Similarity Generator (SG), and Alignment Generator (AG).

PU identifies the type of each entity in the ontology and classifies the entities based on their types. Different computation strategies are adopted for matching the entities with the most appropriate type. Specifically, an entity is classified as one of the following types: Class, ObjectProperty, String DatatypeProperty, Datetime DatatypeProperty, and Numerical DatatypeProperty.

CG quantifies the sufficiency of the resources used to generate a potential match between two entities. It computes a confidence score for each correspondence which reflects if there is sufficient underlying data to generate this correspondence. For correspondence between properties, their instances are the main resources. The more instances that are used for computing similarity, the more confident we can be in the matching process. In order to quantify the sufficiency of the properties, we utilize two metrics — *Volume* and *Variety*.

SG computes multiple types of similarity for every pair of entities. It generates a vector of similarities between two entities. These similarities form the basis for computing different types of relation correspondences (using their respective fuzzy membership functions). In UFOM, the vector consists of four values: Name-based Similarity, Mutual Information Similarity, Containment Similarity, and Structural Similarity (Figure 2).

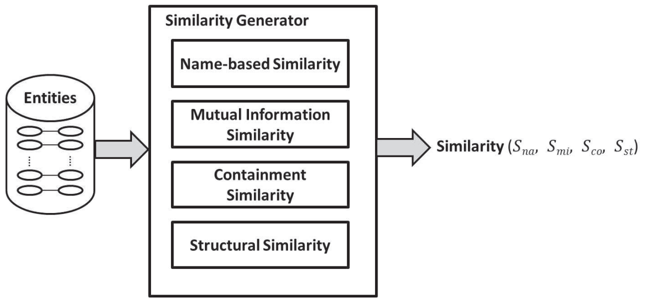


Figure 2. Components of the UFOM Similarity Generator

Name-based Similarity is calculated based on both the semantic similarity and syntactic similarity between the names of the two entities. The name denoting an entity typically captures the most distinctive characteristic of the instances. Mutual Information Similarity models the mutual information that exists between the *individuals* of one entity and the *domain* represented by the second entity. If two properties have a high proportion of instances shared between them, then this is indicative of these properties being highly related. Containment Similarity models the average level of alignment between an instance of an entity and its most similar instance in another entity. It is designed to detect pairs of entities that share a large number of common instances even if the instances themselves are misaligned. The fourth value in the vector of similarities is designed to capture the structural similarity between two properties as they are represented within their ontologies. We represent the ontology as a graph with properties as edges and classes as nodes. If two properties have similar domains and ranges (classes), then they are assigned high similarity. In turn, two classes that have similar properties should have higher similarity.

AG calculates the relation score using the fuzzy membership functions for each relation type and constructs the correspondence based on both this relation score and confidence score. The output of AG is a set of fuzzy correspondences in the form of 6-tuples: $< id, E_1, E_2, r, s, c >$. The confidence score c is obtained from CG. In order to calculate the relation score s, a set of membership functions are pre-defined in UFOM. Each such membership function corresponds to one type of relation. Once both s and c are derived, AG prunes the correspondences with s and c less than pre-defined cutoff thresholds s_δ and c_δ. Different applications will have different thresholds. For example, a recommendation system may have relatively low thresholds since false positives are tolerated, while a scientific application may have high thresholds.

V. Ontology relationship discovery

Before we present the algorithms for identifying matching entities across multiple ontologies, we describe the process of discovering relationships between attributes within an ontology and across ontologies. We call this process ontology relationship discovery. The goal is to discover relationships, including relevance and equivalence, between properties such that these relationship links can be used to prune property pairs that are unlikely to match in the query processing step. This ontology relationship discovery process is divided into three steps.

1) **Rank properties by their information value:** In an ontology, many properties typically have low utility when it comes to distinguishing records. For example, the field `Gender` in a table of People in a school only divides the records into two sets — "Male" and "Female"; while the field `Birthdate` divides the records into 366 sets, and is possibly unique among the ontology for some records. In this example, `Birthdate` has a higher information utility than `Gender` for distinguishing individual records. To represent this information value as a number, the concept of entropy is used.
2) **Discover relevant properties across different ontologies:** Relevance is a type of relationship defined in UFOM [2]. It is a fuzzy relation that quantifies the likelihood of finding equivalent entities between

instances of two properties. Relevance can be computed by applying the UFOM approach to every pair of properties across ontologies.

3) **Discover equivalent properties within an ontology:** Equivalence is the second type of relationship whose links will be used for speeding up querying. Equivalence is different from relevance in that two properties are equivalent if their corresponding instances refer to the same set of entities, while two properties can be relevant even if instances of one property includes references to entities of the other property but do not represent the entities themselves. An example to illustrate the difference between relevance and equivalence is provided. Consider a database of courses in a university. Attributes `CourseID` and `CourseCode` are equivalent but simply named differently. However, if the `Description` field of `Professors` mentions courses they are teaching, then it is relevant to `CourseID` but is not equivalent to it. As with Relevance, Equivalence can also be computed by applying the UFOM approach to every pair of properties among each ontology class.

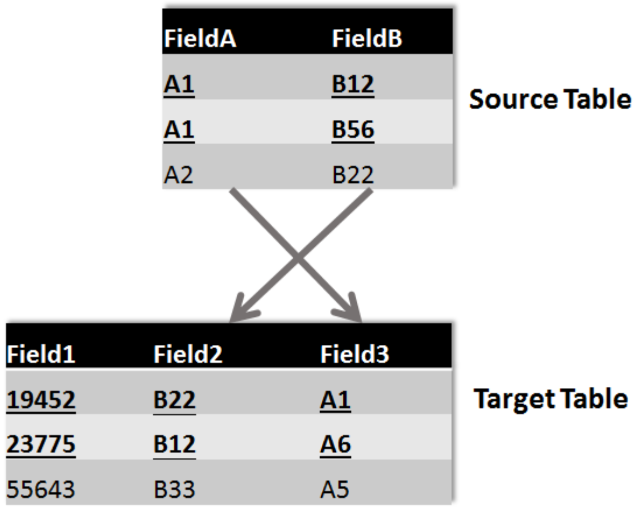
VI. Querying algorithms

We now describe how the relationship links in integrated ontologies can be used for executing inter-ontology queries. We consider queries involving a source and target table. The user specifies a value in the source table. The output is all instances in the target table that are related to instances with this value in the source table. The matching process consists of two methods — *direct matching* and *indirect matching*.

A. Direct matching

Direct matching finds instances among the source and target tables that are relevant based on a single relationship link discovered during the previously described relationship discovery procedure. The steps are described in Algorithm 1. First, fields with low information value (smaller than threshold ϕ) are pruned. All instances in the source table with the user-specified value are retrieved. We then consider the mapping of fields from source table to target table. We keep only those fields with relevance score and confidence score above a pre-defined threshold θ_r and θ_c, respectively. For each source table $\rightarrow$ target table field mapping pair, all values of the obtained instances are compared to that of the instances in the target table, and the matched instances are output. There can be many definitions of what is considered a pair of matching instances. In our case, two instances match if one *contains* the other. Other possible definitions are exact matches, or similarity measures like edit distance.

For example, in Figure 3, suppose the user desires to retrieve records related to instances with `FieldA`="A1." Then the third record with"A2" is pruned. The arrows indicate that `FieldA` is relevant to `Field3` and `FieldB`



FieldA	FieldB
A1	B12
A1	B56
A2	B22

Field1	Field2	Field3
19452	B22	A1
23775	B12	A6
55643	B33	A5

Figure 3. Example of Direct Matching

is relevant to `Field2`. Therefore, values in these two pairs of fields are compared. The resulting output is then records "19452" and "23775".

Algorithm 1 directMatch (target field f, target value v, source table S, target table T, thresholds ϕ, θ_r, θ_c)

```
1: Prune all fields in $S$ with information value smaller than $\phi$;
2: $P \leftarrow$ Query for all $(S \rightarrow T)$ fields pairs with relevance $\geq \theta_r$ and confidence $\geq \theta_c$;
3: $E \leftarrow$ Query for all entries in S with value v in field f;
4: for each $(S \rightarrow T)$ pair $(field1, field2)$ in $P$ do
5:     for each entry $e \in E$ do
6:         $M \leftarrow$ Query for all values in $field2$ that matches with the value of $e.field1$;
7:         $results \leftarrow results \cup M$;
8:     end for
9: end for
10: return $results$
```

B. Indirect matching

Direct matching only finds instances in the target table that have matching values in one of the fields it shares with the source table. However, not all information regarding a certain entity of an instance is stored in one table. It is possible that all fields regarding an entity are categorized and stored in separate tables. Thus, some matching instances may be missed by direct matching if the critical matching field is located in a separate tables.

The Indirect matching algorithm finds instances in other tables that are within the same ontology as the target table and that match with instances in the source table. We denote these tables that exist in the same dataset as the target table

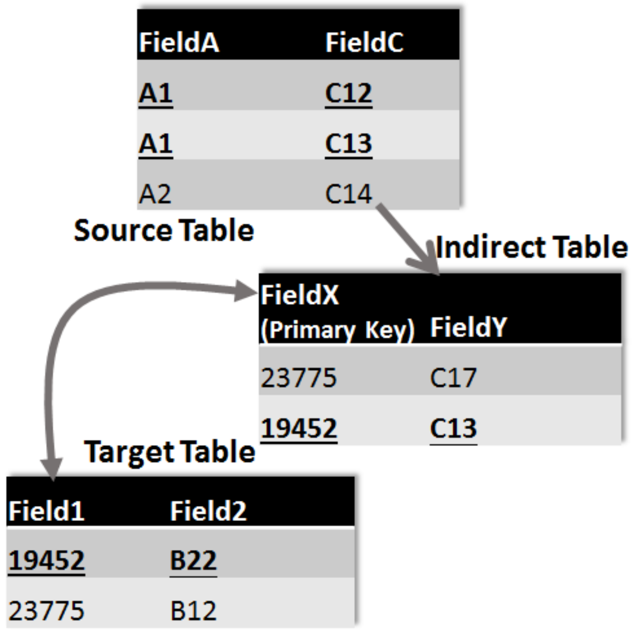


Figure 4. Example of indirect matching

as *indirect tables*. The algorithm then searches for equivalent instances of the matched entities in the target table. The steps are described in Algorithm 2. The initial steps are similar to direct matching. For each indirect table, we perform direct matching to match instances from the source table to the indirect table. Then, we consider the mapping of fields from indirect tables to target table. This step considers fields that have an equivalence score higher than threshold θ_e. Every equivalent property pair that is identified and every instance in an indirect table that is relevant to the specified value is then compared to instances in the target table to identify equivalent ones. All matched equivalent instances are then output.

In the example shown in Figure 4, the source table and target table have no relevant field pairs. However, a relevant record can still be found which is related to the source table via information in an indirect table. Here, `FieldC` is relevant to `FieldY` and record "19452" is identified to be relevant to a specified instance in the source table. This primary key is stored. `FieldX` and `Field1` are equivalent. Since "19452" is in the target table, it is output as the final result.

C. Complexity analysis

We first consider the computational complexity of direct matching. In this analysis, we assume that the cost for comparing a single pair of entities is constant. Let the target table have $|T|$ records. Let $|P|$ be the size of a set of relevant field pairs P. Let n be the number of records in the source table that has the specified value in the given field. Then, the computational complexity is given by $O(|P| \times n \times |T|)$.

Algorithm 2 indirectMatch (target field f, target value v, source table S, target table T, thresholds ϕ, θ_r, θ_c, θ_e)

```
1: Prune all fields in $S$ with information value smaller than $\phi$;
2: $E \leftarrow$ Query for all entries in S with value v in field f;
3: for each indirect table $T_i$ do
4:     $P_i \leftarrow$ Query for all $(S \rightarrow T_i)$ fields pairs with relevance $\geq \theta_r$ and confidence $\geq \theta_c$;
5:     for each $(S \rightarrow T_i)$ pair $(field1, field2)$ in $P_i$ do
6:         for each entry $e \in E$ do
7:             $K' \leftarrow$ Query for primary keys of all entries $e_i$ in $T_i$ such that $e_i.field2 = e.field1$;
8:             $K \leftarrow K \cup K'$;
9:         end for
10:    end for
11:    $Q_i \leftarrow$ Query for all $(T_i- > T)$ fields pairs with equivalence $\geq \theta_e$;
12:    for each value $k \in K$ do
13:        for each $(T_i \rightarrow T)$ pair $(field1, field2)$ in $Q_i$ do
14:            $results \leftarrow$ Query for all entries $e_i$ in $T_i$ such that $(e_i.primaryKey = k) \wedge (e_i.field2 = e.field1)$;
15:            $results \leftarrow results \cup M$;
16:        end for
17:    end for
18: end for
19: return results
```

We next consider the computational cost of indirect matching. Let I be the set of indirect tables and m be the number of indirect tables. Each indirect table I_i has $|I_i|$ number of records. Let $|P_i|$ be the size of set of relevant field pairs P_i between the source table and I_i. Suppose K_i is the final set of primary keys of relevant records identified in T_i, with size $|K_i|$. Let $|Q_i|$ be the size of set of equivalent field pairs Q_i between the target table and I_i. The computational complexity of indirect matching is then given by $O(\sum_{i=1}^{m}(|I_i| \times |P_i| \times n + |Q_i| \times |K_i| \times |T|))$

The organization of data in the indirect tables affects the computational cost of indirect matching. We study the effect of number and size of indirect tables on the computational cost in one special case. Consider there is only one indirect table, if this table is to be normalized and split into m tables, how will the computational cost of indirect matching change? To simplify the analysis, we assume that after the split, the number of field pairs are evenly distributed into each split table. We also assume that $|K_i|$ is a fraction f of $|I_i|$, as the more records there are, the more likely a match is found. We ignore the impact of duplicated fields in the normalization process. The computational cost then becomes $m \times (|I_i| \times \frac{|P_i|}{m} \times n + \frac{|Q_i|}{m} \times \frac{f \times |I_i|}{m} \times |T|)) = |I_i| \times |P_i| \times n + |Q_i| \times \frac{f \times |I_i|}{m} \times |T|$ for any i since the table

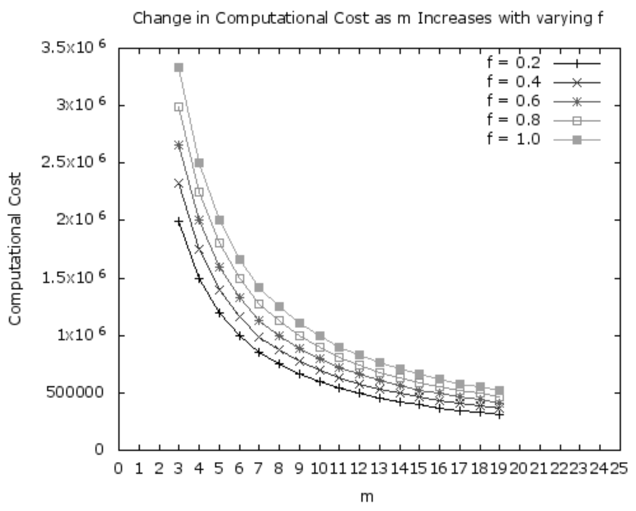


Figure 5. Computational cost plotted against m

sizes are the same.

Table I
PARAMETERS USED

Parameter	Value
n	50
T	50
$\sum_i P_i$	100
$\sum_i Q_i$	100
$\|I_i\|$	1000

Figure 5 shows the change in cost as m (the number of indirect tables) increases and the impact of f, the fraction of matched pairs. The other parameters set as shown in Table I. We can see that the cost decreases as m increases. As the table is split up, less comparisons are made for each record in K_i since there are fewer fields. As a result, we may miss some records that should have matched with records in K_i. There is thus is a trade-off between speed and accuracy of the query.

VII. EXPERIMENTAL EVALUATION

A. Implementation strategies

In this section, we evaluate the impact of specific implementation strategies of matching algorithms on the query execution time. We consider two different approaches of implementing the direct matching algorithm (Algorithm 1).

The first approach is to implement the direct matching algorithms shown in section VI using a procedural language. We denote this the *nested-loop* implementation due to the multiple for-loops required to perform queries for every pair of fields, and every pair of instances.

The second approach is to implement the algorithm with a single SPARQL query that can be executed by an ontology server which then returns the results as a set of triples. The

```
SPARQL SELECT ?p ?e ?c ?RMISprop
where {

graph<Relationships> { ?analysis
prop:#confidence ?SourceConf;
prop:#name ?SourceProp.
FILTER(?SourceConf > 0.5)}

graph<OntologyMatchingResults> {
?m mapping:#property2 ?SourceProp;
mapping#property1 ?TargetProp;
mapping:#hasProbability ?prob;
mapping:#hasConfidence ?conf . FILTER
(?prob > 0.5 && ?conf > 0.8)}

graph <sourceTable>{ ?a <field>
"value"; ?SourceProp ?c. }

graph<targetTable> { ?d ?TargetProp
?e; <PrimaryKeys"> ?p. FILTER
regex(?e, ?c, "i")} };
```

Figure 6. Graph pattern implementation of Direct Matching

query pattern would operate on the graphs that represent the ontology and pattern matches comprise the result. We denote this implementation approach as the *graph pattern* implementation. The corresponding graph pattern query for direct matching is shown in Figure 6. The `prop:` prefix is used for terms describing a property. The `mapping:` prefix is used for terms describing a matching between two properties across tables.

B. Performance evaluation

To evaluate these two implementation strategies, the direct matching algorithm was implemented in C# (nested-loop implementation) and with the SPARQL code shown in Figure 6 (the graph pattern implementation). We executed each implementation on a triple store server. The ontology server was executed on an AMD Processor (4228 HE) at 2.8 GHz with 32 GB main memory. Two ontologies containing instances were used for these experiments. We used an enterprise-scale dataset to provide the source and target tables. Ontology O_1 was derived from two tables $T_{1,1}$ and $T_{1,2}$. Ontology O_2 is derived from four tables $T_{2,1}$, $T_{2,}2$, $T_{2,3}$, and $T_{2,4}$. The number of triples in each of these tables is shown in Table II.

When executing similarity queries, we used different tables as the source and target table. For each pair of source and target tables, different input value were used in our evaluation. These values result in output sets of different sizes as shown in Table III.

Table II
NUMBER OF TRIPLES IN ONTOLOGIES

Ontology	Number of triples
$T_{1,1}$	125,865
$T_{1,2}$	98,325
$T_{2,1}$	398,352
$T_{2,2}$	3,933,711
$T_{2,3}$	132,928
$T_{2,4}$	2,289,258

Table III
INPUT PAIRS AND NUMBER OF NON-PRUNED INPUT ENTRIES AND SIZE OF OUTPUT

(Source, Target) pair	Non-pruned inputs	Outputs
$T_{1,1}, T_{2,1}$	15	15
$T_{1,1}, T_{2,1}$	12	13
$T_{1,1}, T_{2,3}$	16	16
$T_{1,1}, T_{2,4}$	18	31

The algorithm is executed 11 times for each input and the first run is ignored. The mean and standard deviation of the remaining 10 running times for the nested-loop and graph pattern implementations are compared for the four different query inputs.

The experiment was carried out with the two client implementations running on the same server hosting the ontology server. The resulting running times and deviations for the two implementations are shown in Table IV.

In order to account for the effect of network access latency, the experiment is repeated with the client executing on a host that is different from that of the ontology server. This host is equipped with an Intel Core i7-4770 CPU at 3.40GHz with 8 GB main memory. The resulting running times and deviations for the two implementations are shown in Table V.

From these results, it can be seen that the running time of the nested-loop implementation is approximately half of the graph pattern implementation for three of the four inputs. However, as the output size increases (input $T_{1,1}, T_{2,4}$), the difference between the two methods decreases.

The standard deviation of the data from the experiment ran on the ontology server is smaller than that of the remote host in general. The maximum standard deviation value for ontology server is under 200ms, while that executing on a remote host is over 200ms in the case of the $T_{1,1}, T_{2,4}$ input. This is explained by the lower network latency when the direct matching code is ran directly on the ontology server.

Note that the graph pattern implementation based on a single SPARQL query can be optimized by the ontology server before it is executed. The observation that the nested-for-loop implementation gives better performance than the SPARQL query indicates that significant further query optimizations are possible within the triple store server.

VIII. CONCLUSIONS

We considered methods to speed up querying across ontologies by making use of inter-ontology links discovered during automatic ontology alignment. We derived the computational complexity of querying across ontologies for the specific case where the ontology alignment links were discovered using the Unified Fuzzy Ontology Matching (UFOM) framework. We analyzed two methods for querying for related entities — following only a single alignment link and by following multiple alignment links. We showed that these methods have different computational complexity and produce different query results. We also studied the impact of the specific implementation approach on query time using ontologies derived from an enterprise-scale dataset (millions of triples). We considered implementations based on multiple accesses of the triplestore with a high-level procedural language and by execution of a single SPARQL graph query on the ontology server. Experimental results show that the implementation using nested for-loops in a procedural language is nearly $2\times$ faster than an implementation based on a single SPARQL query.

Our experimental result indicates that improved query optimization strategies would be useful for efficiently executing SPARQL queries over linked ontologies. For future work, we will develop such query optimization techniques for executing SPARQL queries across linked ontologies.

ACKNOWLEDGMENT

This work is supported by Chevron U.S.A. Inc. under the joint project, Center for Interactive Smart Oilfield Technologies (CiSoft), at the University of Southern California.

REFERENCES

[1] P. Shvaiko and J. Euzenat, "Ontology matching: state of the art and future challenges," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 25, no. 1, pp. 158–176, 2013.

[2] Y. Zhang, A. Panangadan, and V. Prasanna, "UFOM: Unified fuzzy ontology matching," in *Proc. of the* 16^{th} *IEEE International Conference on Information Reuse and Integration*, 2014.

[3] N. Choi, I.-Y. Song, and H. Han, "A survey on ontology mapping," *SIGMOD Record*, vol. 35, no. 3, pp. 34–41, 2006.

[4] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[5] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Data Engineering, 2002. Proceedings. 18th International Conference on.* IEEE, 2002, pp. 117–128.

[6] R. Nath, H. Seddiqui, and M. Aono, "Resolving scalability issue to ontology instance matching in semantic web," in *Computer and Information Technology (ICCIT), 2012 15th International Conference on.* IEEE, 2012, pp. 396–404.

Table IV
RUNNING TIME OF DIFFERENT QUERY IMPLEMENTATION STRATEGIES (CLIENT AND SERVER ON SAME HOST

Direct Matching on (Source, Target) pair	Mean (Nested-loop) (ms)	Mean (Graph pattern) (ms)	Standard Deviation (Nested-loop) (ms)	Standard Deviation (Graph pattern) (ms)
$T_{1,1}, T_{2,1}$	5271.6	10260.3	54.0395	108.8169
$T_{1,1}, T_{2,1}$	4209.7	8202.3	20.6454	69.3078
$T_{1,1}, T_{2,3}$	8311.7	14325.4	71.0634	131.1031
$T_{1,1}, T_{2,4}$	17694.5	17489.9	65.5138	174.4429

Table V
RUNNING TIME OF DIFFERENT QUERY IMPLEMENTATION STRATEGIES (CLIENT AND SERVER ON DIFFERENT HOSTS)

Direct Matching on (Source, Target) pair	Mean (Nested-loop) (ms)	Mean (Graph pattern) (ms)	Standard Deviation (Nested-loop) (ms)	Standard Deviation (Graph pattern) (ms)
$T_{1,1}, T_{2,1}$	5290.8	10122.7	38.0520	66.9777
$T_{1,1}, T_{2,1}$	4247.7	8124	19.8385	39.4518
$T_{1,1}, T_{2,3}$	8399.8	14109.7	97.0519	76.9532
$T_{1,1}, T_{2,4}$	17945.1	18652.6	242.5322	1547.8781

[7] M. H. Seddiqui and M. Aono, "Anchor-flood: results for oaei 2009," in *Proceedings of the ISWC 2009 Workshop on Ontology Matching*. Citeseer, 2009, pp. 127–134.

[8] V. Mascardi, A. Locoro, and P. Rosso, "Automatic ontology matching via upper ontologies: A systematic evaluation," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 5, pp. 609–623, 2010.

[9] T. Kirsten, A. Thor, and E. Rahm, "Instance-based matching of large life science ontologies," in *Data Integration in the Life Sciences*. Springer, 2007, pp. 172–187.

[10] M. D. Larsen, "Record linkage modeling in federal statistical databases," in *FCSM Research Conference, Washington, DC*. Citeseer, 2010.

[11] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz, "Efficiently detecting inclusion dependencies," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 1448–1450.

[12] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser, "A machine learning approach to foreign key discovery." in *WebDB*, 2009.

[13] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava, "On multi-column foreign key discovery," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 805–814, 2010.