Searching Web 2.0 Data through Entity-Based Aggregation

Ekaterini Ioannou[1] and Yannis Velegrakis[2]

[1] Technical University of Crete, ioannou@softnet.tuc.gr
[2] University of Trento, velgias@disi.unitn.eu

Abstract. Entity-based searching has been introduced as a way of allowing users and applications to retrieve information about a specific real world object such as a person, an event, or a location. Recent advances in crawling, information extraction, and data exchange technologies have brought a new era in data management, typically referred to through the term Web 2.0. Entity searching over Web 2.0 data facilitates the retrieval of relevant information from the plethora of data available in semantic and social web applications.
Effective entity searching over a variety of sources requires the integration of the different pieces of information that refer to the same real world entity. Entity-based aggregation of Web 2.0 data is an effective mechanism towards this direction. Adopting the suggestions of the Linked Data movement, aggregators are able to efficiently match and merge the data that refer to the same real world object.

Keywords: semantic web, data integration, semantic data management.

1 Introduction

1.1 Challenges

Implementing entity-based aggregation to support entity search, poses a number of challenges due to the peculiarities of the modern web data, and specifically of that of Web 2.0 [3]. In particular, integration support needs to provide some *coherence* guarantees, i.e., to ensure that it can detect whether difference pieces of information in different sources representing the same real world object, are actually linked. It is not rare the case in which different sources contain quite different information about the same entity.

To successfully provide the above functionality, the aggregator needs first to cope with *heterogeneity*. Web 2.0 applications have typically a large amount of user-generated data, e.g., text, messages, tags, that are highly heterogeneous either by nature, or by design. For instance, DBPedia is based on RDF data and utilizes its own ontology, whereas Wikipedia adopts a more loose schema binding. The aggregator should be able to deal with a wide variety of entity descriptions ranging from keyword style entity requests to more structured and detailed descriptions.

Furthermore, the aggregator also has to be able to deal with a *discrepancy* in the knowledge about an entity available in different sources (*knowledge heterogeneity*): Due to different reasons (e.g., perspective, targeted applications), two sources might know more, less or different things about an entity. Exactly this makes their integration promising, but also challenging.

The aggregator should also be able to cope with the different *data quality levels* of the sources. User-generated datasets, or datasets generated by integrations of heterogeneous sources, are typically high in noise and missing values [2] which impedes entity identification. Finally, data brought together from multiple independently developed data sources that partially overlap, introduces *redundancy* and *conflicts* that need to be resolved in an efficient and consistent manner.

1.2 Approach & Contributions

To support entity-based searching over integrated entity descriptions of Web 2.0 data we propose an infrastructure for entity-based aggregation. Our main focus on the part of the infrastructure that matches queries to entity profiles. This requires extending search and aggregation technologies (e.g., semantic search engines, mashups, or portals) with functionallities that will allow them to maintain and exploit information about entities and their characteristics.

The current version of our system uses the Entity Name System (ENS) [23], which is an entity repository service. Our goal is to enable entity-based aggregation for the particular repository. This requires matching capabilities in order to ensure that each entity in the repository has a unique identifier. We achieve this through the search functionality. In short, when the repository is queried (i.e., entity search) we detect and return the entities from the repository that contain the characteristics found in the requested query. If no entity match is found in the repository, then a new entry is created in order to be used for future requests. If a partially matching entity is found, the stored entity may be enhanced with the additional characteristics from the source data, improving the chances of successful identification in future requests.

The introduced Matching Framework has several contributions regarding matching and entity searching. First, it brings data matching from its static and design-time nature into a dynamic process that starts at design-time with some initial knowledge and continues to evolve throughout its run-time use. An additional contribution is that it is able to employee an extensible set of matching methodologies, each based on a different matching technique. This allows us to deal with the inability of a single matching algorithm to address the matching problem. Also, matching is performed as a series of steps that include selection of the most appropriate matching algorithm and combination of results from more than one modules. Note that a short description of the Matching Framework was included in [19]. In this journal we provide the details of our work.

The remaining paper is structured as follows. Section 2 presents a motivating example. Section 3 introduces the Matching Framework and discusses the components composing it. Section 4 explains how existing matching methodologies can be incorporated in the proposed Matching Framework and also provides a

couple of concrete examples. Section 5 describes various requests we used for evaluating our infrastructure and reports quality and execution time. Finally, Section 6 provides conclusions and discusses possible future directions.

2 Motivating Example

We are currently seeing a plethora of systems that use data from Web 2.0 applications and sources. For being able to use such data, the systems need to integrate the collected/received Web 2.0 data. Our matching framework focuses on providing this task, i.e., performing the integration of data coming from Web 2.0 applications and sources, and in particular on matching and merging together the data describing the same real world objects. Consider a portal system, such as Thoora[3], or Daylife[4], that crawls the web collecting, analysing, and categorizing news articles and blog posts. The system employs special similarity search techniques that find whether different articles talk about the same topic. These techniques are mainly based on textual analysis which try to identify entities mentioned in each article/post. Consider an article that talks about a person called Barack Obama living in Washington DC and a second one talking about a Nobel Prize winner also called Barack Obama who has been a senator in Illinois (i.e., entities labeled e_1 and e_2 in Figure 1 respectively). The fact that both articles talk about a person called Barack Obama, is some indication that they may refer to the same person. However, it is not sure that they actually do, since the rest of the information they have is different. For instance, the second article, in contrast to the first, does not mention anything about the residence of the person, but instead mentions some origin place that is different from the Washington DC area.

A linked-data could link the two entities if there was a way to refer to them, for instance through the existence of some identifiers in the data, if such identifiers exist. Since the data is coming from text articles and not from database systems, this information may not be available at all, or may not be useful since these identifiers may have been locally assigned at run time with no persistence guarantees. The identification needs to be based mainly on their characteristics that are mentioned in the articles. Nevertheless, the portal developer, having the knowledge that Barack Obama is one of the Nobel Prize recipients can provide an add-hoc solution that links these two entities.

Consider now a blog post that talks about a person called Barack Obama, who is the recipient of a Nobel Prize and lives in Washington DC (shown as entity e_3 in Figure 1). Clearly no document from those seen so far had an entity with these three characteristics. There are strong indications, however, that this person is the same as the one in the first article, since they agree on the name and the residence place. It similarly agrees with the entity in the second article. Nevertheless, the portal developer, that knows that the entities in the first two articles are actually the same person, that has the combined set of their

[3] http://www.thoora.com/
[4] http://www.daylife.com/

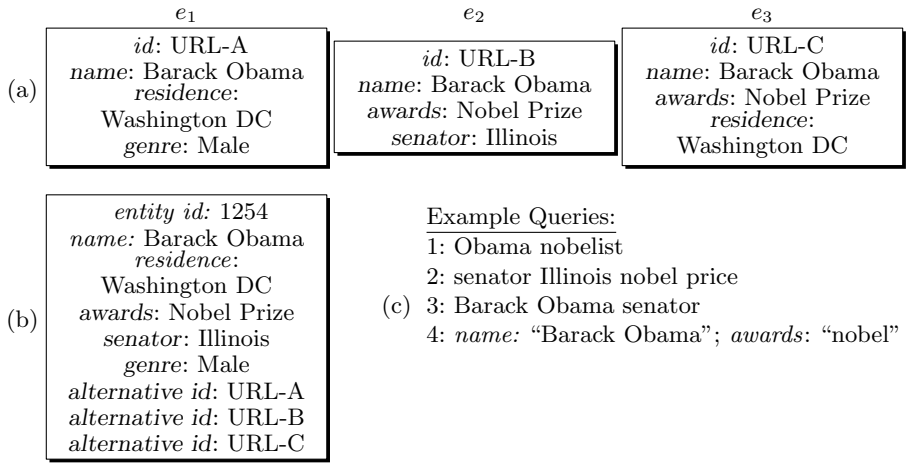

Fig. 1. (a) Theree entities found in news articles and blog posts. (b) The corresponding entity entry created through entity-based aggregation. (c) A few query examples.

characteristics, can more confidently conclude that the entity mentioned in the blog post is the one in the first two. This kind of knowledge, however, resides in the mind of the portal developer alone with no systematic way of representing and using it. Linked-data, when available, may be exploited to reach similar conclusions, but it will require global knowledge, advanced reasoning and is not guaranteed to work in all cases.

To overcome this problem, during integration, the portal can utilize a repository of entities that contains discriminative knowledge about a large number of already known real world objects. These entities could have been collected from a number of existing social and semantic applications, such as Wikipedia, DBPedia, and Freebase, as well as articles that the portal crawler has already crawled and analyzed. The repository need not be a knowledge base, since this is neither realistic, nor practical. Instead, it should contain a collection of entities alongside their characteristics that allow their identification. Each entity, should also have a unique identifier that distinguishes it from all the other entities. The creation and use of such a repository requires solutions to a number of challenging issues. These issues include among others the efficient and effective entity search, population, storage, and maintenance of entities.

At run time, when the portal analyses an article and detects an entity, it collects its characteristics, and generated entity search queries using these characteristics. The repository receives this query and retrieves the identifier of the corresponding matched entity. Naturally, it may be the case that no entity with all these characteristics exists, or the case that there are more than one such entities. Thus, the repository responds to such a query request by providing a small list of candidates.

The richer the repository is, the more the confidence for the returned entities of a search request will be. If no entities are returned, then a new entry may be

created in the repository for that new entity with the characteristics mentioned in the query, for example as shown in Figure 1(b). This will enable its future identification in other articles or posts. If, instead, the entity is found in the repository through some partial match, then the characteristics of the found entity may be enhanced with the additional characteristics mentioned in the query to improve the chances and confidence of its identification in future articles.

3 Matching Framework

Consider again the challenges of integrating Web 2.0 data. Such data do not have a fixed schema and may consist of name-value attributes, attributes with only values, or a mix of both. Furthermore, the data given for integration may contain only one entity as well as a collection of entities which are somehow related to each other, such as entity repositories from Social applications. Despite the many existing results in entity matching, no solution has been found to work for all situations.

Our matching framework focuses on performing integration of data coming from Web 2.0 applications and sources, and in particular on matching and merging together the data describing the same real world objects. As such, it is responsible for receiving the queries requesting entities, controlling the execution flow, which primarily invokes the matching process, and returning the result set. Thus, our infrastructure can be useful for applications and systems that incorporate Web 2.0 data, for example the one described in the motivating example (i.e., Section 2).

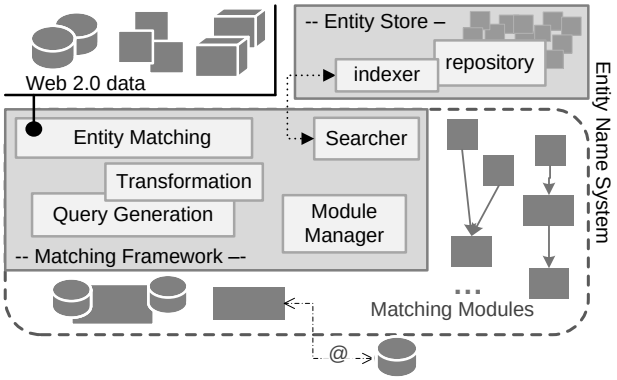

Fig. 2. An illustration of the Matching Framework within the ENS service.

Entity Name Service (ENS). The main components of the Matching Framework are illustrated in Figure 2. Note that our framework is part of the Entity Name Service (ENS) [23]; an entity repository service that maintains a collection of entities as well as a number of discriminative properties that are

used for entity identification and for assigning a unique identifier to each of the entities.

To achieve its goals, ENS addresses a set of challenges. The first challenge is to introduce a generic methodology for matching that incorporates an extendable set of matching modules, each focusing on addressing a specific matching situation. Another related challenge involves the selection of either the most promising module for a given query, or the combination of different modules. Additionally, the ENS service need to efficiently store and retrieve entities from the potentially very large entity collection and this is performed through the Entity Store.

Entity Store. A matching algorithm typically performs a lot of query-entity comparisons, which means that existing techniques either fail at Web scale or have performance that is prohibited for run-time. To minimize the time required for comparisons, we try to reduce the number of entities given to the matching algorithms for processing. The entity store provides a repository of entities along with an index for efficient entity retrieval. The requirement for the entity store is therefore to provide the set of entities that contain some of the information from the given query. By further processing these entities we can then identify the one requested by the query. We refer to the set of entities returned by the store as the *candidate set*.

In our current implementation we used NECESSITY system [20] as the entity store. NECESSITY is composed of two main part, which are shown in Figure 2. The first part is the *repository*, implemented as a key-value Voldemort repository[5] and able to maintain a large number of entities. Scalability in Voldemort is obtained by simply adding more servers when more entities are added to the repository, which means that the remaining components of the ENS are left untouched. Moreover, this key-value repository supports linear scalability since it is designed from scratch to scale by simply adding new nodes. The second part of is the inverted *index*, implemented as a Solr Brocker[6], which uses Lucene[7] for full-text indexing and search. When a new query arrives, the Solr broker will assign the query to some of its shards, collect their *top-k* results, and then aggregate them to identify the best *top-k* entities, i.e., the candidate set that is then given to the matching framework. Please not that the details with respect to the entity store are not included in the journal since these are available in the publication describing the particular system [20].

Overview of Query Answering. The processing flow is controlled by the *Entity Matching* component, which also receives queries from users. When this component receives a query, it first gives it to the *Query Generation* and *Transformation* components for generating initial request commands for the Entity Store, and also to the *Module Manager* component for identifying the most suitable matching module or modules that could process this query. The initial request commands are then revised by the selected matching module(s) and then

[5] http://project-voldemort.com/
[6] http://lucene.apache.org/solr/
[7] http://lucene.apache.org/

given to the *Searcher* component for pass it on to the Entity Store. The store processes the request commands and returns the candidate set. The entities from the candidate set are then given to the module(s) for performing matching and identifying the entity that corresponds to the given query. Through the *Entity Matching* component, the users receives the results for the processed queries.

The following paragraphs, we discuss the components composing the Matching Framework. We also give the details of the processing performing in each of these components.

3.1 Query Generation & Transformation

The Matching Framework needs to generate the request commands for the Entity Store. Our current system incorporates the NECESSITY entity store, thus, we need to generate a Lucene query. Since the Entity Store offers very efficient but restricted search functionality, this step might also require the generation of more that one queries, with the final candidate set being the merging of the results returned by the entity store for all generated queries.

We already explained, the query can be enhanced and refined by the matching modules according to their needs. This might involve query transformations on the schema level, for example to adapt from attributes used by the user/applications to attributes available in the repository, to include attribute alternatives, or to relax the query to the most frequent naming variants.

3.2 Matching Modules

Individual matching modules implement their own method for matching queries with candidates, i.e., entities returned from the store. Naturally, the algorithm of each module will focus on a specific matching task. For example, we can have matching modules providing algorithms for entities that do not contain attribute names, or for entities that contain inner-relationships. As shown in Figure 2, modules may also use a local database for storing their internal data, or even communicate with external sources for retrieving information useful for their matching algorithm.

In addition to the individual modules, the Matching Framework can also contain modules that do not compute matches directly, but by combining the results of other modules. The current version of our system can handle the following two types of combination modules: (i) sequential and (ii) parallel processing. In sequential processing, the matching module invokes other modules in a sequence, and each module receives the results of previously invoked module. Therefore, each module refines the entity matches they receive and the resulted entity matches are the ones returned by the last module. The parallel processing invokes a set of matching modules at the same time. Each module returns its entity matches, and thus the combination module needs to combine their results.

3.3 Module Manager

This component is responsible for managing the modules included in the Matching Framework. To know the abilities of each module, the *Module Manager* maintains the module profiles. These profiles contain not only the module description and classification, but also information on their matching capabilities. For example, the average time required for processing queries and the query formats that they can handle.

The manager is responsible to select the best suitable matching module to perform the entity matching for the given entity. The *basic methodologies* for the performing the module selection are the following:

- The entity request explicitly defines the matching module that should be used. Such a selection can, for example, be based on previous experience or by knowing that a specialized matching module is more effective when integrating the data of a specific Social application.
- The matching module is selected based on information in the entities to be integrated. This may include requirements with respect to performance or supported entity types.
- The module is selected based on an analysis of the data in the entity to be integrated, for example existence or not of attribute names.

In addition to the *basic methodologies*, we can also have *advance methodologies*. Given the architecture of our Matching Framework we can actually perform the matching using more than one modules. This will result in a number of possible linkages, each one encoding a possible match between two entities along with the corresponding probability (indicating the belief we have for the particular match). Thus, the goal now is to maintain these linkages in the system and efficiently use them when processing incoming requests.

This is an interesting and promising direction that has not yet been studied deeply by the community. With respect to our Matching Framework, we have investigated how to perform only the required merges at run-time by following the possible world semantics given the linkages as well as effectively taking into consideration the entity specifications included in given requests [17]. In addition, we have also studied the possibility of executing complex queries instead of just answering entity requests. More specifically, we proposed the usage of an entity-join operator that allows expressing complex aggregation and iceberg/top-k queries over joins between such linkages with other data (e.g., relational tables) [16]. This includes a novel indexing structure that allows us to efficiently access the entity resolution information and novel techniques for efficiently evaluating complex probabilistic queries that retrieve analytical and summarized information over a (potentially, huge) collection of possible worlds.

4 Matching Modules

In this section we explain how the suggested system can incorporate and use existing matching techniques. For this task, it is important to understand the

methodologies followed by the existing techniques. Thus, we consider the existing techniques categorized according to their methodology and explain how each category can be incorporated in our system.

Please note that we discuss existing matching techniques to the extend needed for this purpose of the introduced system. A complete overview of existing matching techniques as well as related surveys is available in [9], [12], [15], [14], and [29].

4.1 Atomic Similarity Techniques

The first category of matching techniques consider that each entity is a single word or a small sequence of words. Matching issues is the entities of this category can occur from misspellings, abbreviations, acronyms, naming variants, and use of different languages (i.e., multilingualism). As examples consider the following: entity-1 is "TCCI Journal" and entity-2 is "Transactions on Computational Collective Intelligence journal". The matching methodology followed by the techniques of this category is based on detecting resemblance between the text values on the entities; more details about the followed methodology can be found in [6] and [8].

Module Example - Handing multilingualism. In the modern global environments, the data is collected from many physically distributed sources or applications that may be located in different countries. This unavoidable leads to data expressed in a diverse set of languages. Furthermore, users with different cultural backgrounds typically pose queries in their native languages, that is not necessarily the same as the one in which the data has been stored. In such a situation, it is not at all surprising that traditional string matching algorithms are dramatically failing. A popular solution to cope with the problem is to use some form of standardization. For instance, as a convention, all the information can be translated into English and then stored into the system. When a query is received, it is first translated into English, if not already, and then evaluated against the stored data.

The specific approach has unfortunately two main limitations. The first is that the data will be returned to the users in English which will be surprising if the user knew that the data had been inserted is some other language. The second limitation is that the translation from one language to another is not always perfect. Thus, if the data was original in a language, say Italian, and the query posed by the user is also in Italian, translating both of them in English and then evaluating the query against the data may loose in terms of accuracy and precision.

For the above reasons, we follow an approach in which we combine the best of both worlds. In particular, we maintain two copies of the data. One that is the original form in which the data has been inserted. The second one, called the canonical representation, is the translation of the original data into English. When a user query is received, then two evaluations are taking place in parallel. The first is the evaluation of the user query as-is over the original data. This evaluation has the advantage that it allows the high-accuracy retrieval of the data that match the provided query in the same language. At the same time, the

provided query is translated into the canonical form, i.e., English, and evaluated against the canonical forms of the data. This will allow the query to retrieve as answers, data that has been inserted originally in some completely different language. At the end, the result lists of the two approaches are merged together into one single list, by considering the combined returned score.

Another advantage of the approach we have followed, apart from the fact that it is easily extended to support more than one languages. Also, the translation mechanism allows for the utilization of additional information that guides the translation such as personalization, time, etc.

4.2 Entities as sets of data

This entity representation of this category can be seen as an extension of the previous one. More specifically, the entities are now represented as a small collection of data. The most typical situation is considering that each record of a relational table provides an entity. As examples consider the following: entity-1 is {"TCCI Journal", "2015", ...} and entity-2 is {"Transactions on Computational Collective Intelligence journal", "2015", ...}

One methodology to address this issue, it to concatenate the data composing each entity into one string and then perform matching using a technique from the previous category [7, 21]. Other techniques perform matching by detecting mappings between entities. For example, [30] detects mappings by executing *transformations*, such as abbreviation, stemming, and the use of initials. Doan et al. [10] apply *profilers* that correspond to predefined rules with knowledge about specific concepts.

In the following paragraphs we present two such modules that have been included in our system. The first implements the methodology suggested from an existing technique, whereas the second is a new technique created for targeting a specific entity type.

Module Example II - Group Linkage. This module adapts the algorithm suggested in [24], and matches an entity to a candidate when it detects a large fraction of similarities between their data. For being able to use this algorithm we consider the given query Q and the candidate C as the two entities. Their matching probability is given by:

$$MP(Q,C) = \frac{\sum\limits_{\forall p_i \in Q \forall p_j \in C} \begin{cases} sim(p_i,p_j)\ if\ \ sim > t \\ 0\ otherwise \end{cases}}{|Q| + |C| - matched\_pairs,} \quad (1)$$

where $|C|$ gives the number of attribute value pairs in the candidate, $|Q|$ the number of predicates in the query, and $matched\_pairs$ the number of $sim(p_i, p_j)$ higher that threshold t.

Module Example II - Eureka Matching. The matching methodology of the Eureka module computes the overlap of the predicates in the query with the attributes of the candidates. As an initialization step, the algorithm creates a small local inverted index as follows: Each term (i.e., word) in the values from the attribute-value pairs of the query become keys in a hash table. We then process

the information in each candidate and when we identify a candidate contains one of these values, we add the candidate's identifier with the values attribute to the list of entities of the corresponding key. The score $MP(Q, C)$ between the entity described in the query Q and candidate C is computed by:

$$\sum_{\forall p_1 \in Q,\, \forall p_2 \in C} \begin{cases} 1 \times importance(p_1.attr), \\ \quad if\ \ p_1.attr = p_2.attr \,\&\, p_1.value \in p_2.value \\ 0.5 \times importance(p_1.attr), \\ \quad if\ \ p_1.attr = null \,\&\, p_1.value \in p_2.value \end{cases} \tag{2}$$

where *importance* is a weight that reflects the importance of a specific attribute, e.g., attribute *name* is more important than attribute *residence* for the entities of Figure 1.

4.3 Collective Matching

The techniques performing collective matching are based not only on the data composing each entity (as in Sections 4.1 and 4.2) but also on available relationships and associations between the entities. For an example, consider that we are now working on addressing the entity matching problem in a collection of publications. One of the publications has authors α, β, and γ. Another publication has authors α, β, and γ'. Performing matching using one of the methodologies from the previous categories would result in the matching of authors α and β between the two publications, and a strong similarity between author γ and γ'. The former (i.e., matching of authors α and β) gives a relationship between the two publications. Combining this relationships with the similarity between γ and γ' increases our belief that these authors are actually the same, and thus, perform their match.

One approach for performing collection matching was introduced by Ananthakrishna et al. [4]. It detects fuzzy duplicates in dimensional tables by exploiting dimensional hierarchies. The hierarchies are built by following the links between the data from one table to the data from other tables. Entities are matched when the information across the generated hierarchies is found similar. The approach in [5] uses a graph structure, where nodes encode the entities and edges the inner-relationships between entities. The technique uses the edges from the graphs to cluster the nodes. The entities inside clusters are considered as matches and are then merged together. Another well-know algorithm is the *Reference Reconciliation* [11]. Matching starts by comparing the entity literals to detect possible relationships between entities. The detected matches are then propagated to the rest of the entities and is used for increasing the confident for the related entity matches. The approach introduced in [18] models the data into a Bayesian network, and uses probabilistic inference for computing the probabilities of entity matches and for propagating the information between matches.

Incorporating collective matching techniques is also possible with our system. The main required functionality for having such a module is being able to maintain the information related to the relationships between entities, including efficient update and navigation mechanism. This is a functionality that can be achieved through the entity store by storing relationships inside each entity, i.e.,

as attribute value pairs. The entity store can also implement specialized indexes for the relationships - if this is useful for the particular technique. Another functionality that would be helpful is being able to call other matching modules, and in particular modules from Section 4.1 and 4.2 for detecting possible similarities between entities. As we explained in Section 3 and also illustrate in Figure 2, this capability can by realized through the matching framework thourgh sequential or parallel processing of modules.

4.4 Matching using Schema Information

Another helpful source of knowledge related to entity matching is through the available schema information. Note that knowledge coming from schema information is typically correspondences between the entity attributes and not between the actual entities [13, 28], and thus can not directly be used for matching. However, it definitely assist the matching methodologies presented in the previous categories. For example, by knowing which schema attributes are identical, or present a high similarity, we perform a focused entity matching only on the particular attributes.

Our system can easily incorporate such methods. The most prominent mechanism is to implement them as a typical matching module. Schema information is anyway present in the entities of the Entity Store, and thus, accessible to the modules. The entity matching process that uses the results of this processing can be included in the same module, or in another module that just call it.

4.5 Blocking-based Matching

One methodology to increase the efficiency of matching is by reducing the performed entity comparisons. Blocking is focusing on this, and more specifically through the following process: entities are separated into blocks, such that the entities of the same block can result in one or more matches, whereas the entities of different blocks can not result in matches. Having such blocks means that we do not need to compare all entities between them but only the entities inside the same block, which of course reduces the comparisons.

The most common methodology for blocking is to associate each entity with a value summarizing key its data and use this to create the blocks. For example, the approach introduced in [22], builds high-dimensional overlapping blocks using string similarity metrics. Similarly, the approach in [1], uses suffixes of the entity data. One group of approaches focused on data with heterogeneous semi-structured shemata. For instance, [25, 26] introduced an attribute-agnostic mechanism for generating the blocks and explained how efficiency can be improved through scheduling the order of block processing and identifying when to stop the processing. The approach introduced in [31] processes iteratively blocks in order to use the results of one block in the processing step of another block. The idea of iteratively block processing was also studied in [27]. It provided a principled framework with message passing algorithms for generating a global solution for the entity resolution over the complete collection.

Incorporating blocking-based matching in our system can be achieved with two ways. The first is by modifying the storage functionality, and in particular,

the method for adding them in the Entity Store. Entities should be also processed to generate the value summarizing key which is also included (and stored) in their attribute value pairs. These keys are used by the matching modules for skipping entity comparisons. The alternative incorporation of blocking is by modifying the matching framework in order to follow one specific blocking technique, i.e., entities are actually separated using the value summarizing key and are given to the modules grouped according to the block in which they belong.

5 Usage Experience

We now demonstrate and discuss the applicability of the suggested infrastructure. For this we use various collection of requests that can be used to evaluate methodologies for entity linkage as well as entity search. Each collection is taken from a different real world data source (e.g., structured and unstructured data) and this leads to requests that have different format. Note that the entity requests are accompanied with a small set of urls from the systems in which they are described (e.g., Wikipedia url, OKKAM id, etc.). The following paragraphs describe these entity collections[8] and report the required processing time as well as the quality of the returned answer set.

(A) People Information. The first collection contains 7444 entity requests from short descriptions of people's data from Wikipedia. The text describing these people was crawled and then processed using the OpenCalais extractor to extract the contained entities. Some examples are: (i) "Evelyn Dubrow" Country="US" Position="womenlabor advocate" and (ii) "Chris Harman" Position="activist" Position="journalist".

(B) News Events & Web Blog. Wikipedia contains small summaries of news events reported in various online systems, such as BBC and Reuters. We used the OpenCalais extractor to identify the entities in the events. This also resulted in entity type along a few name-value attributes for each entity. We also used the OpenCalais extractor to identify the entities described in a small set of blogs discussing political events, e.g., http://www.barackoblogger.com/. The process resulted in a collection with 1509 entity requests. Some request examples are: (i) Alex Rodriguez and (ii) name="Charles Krauthammer".

(C) Historical Records. Web pages sometimes contain local lists of entities, for example members of organizations, or historic records in online newspapers. This collection contains 183 entity requests taken from such lists, i.e., no extraction process was involved. Some examples are: (i) Don Henderson British actor and (ii) George Wald American scientist recipient of the Nobel Prize in Physiology or Medicine.

(D) DBPedia. The last entity collection contains 2299 requests from the structured DBPedia data. Some examples are: (i) name="Jessica Di Cicco" occupation="Actress Voice actress" yearsactive="1989-present" and (ii) birthname="Jessica Dereschuk" eyeColor="Green" ethnicity="WhiteCaucasian" hairColor="Blonde" years="2004".

[8] The collections can be found at http://www.softnet.tuc.gr/~ioannou/entityrequests.html

To evaluate the introduced infrastructure we used the NECESSITY entity store [20] with ∼6.8 million entities. The entities in the store were people and organizations from Wikipedia, and geographical items from GeoNames. We then used the infrastructure to retrieve the entities for 4000 requests from our four collections. For each request the infrastructure returned a list of entities matching the particular request. The requested entity was the first item in the list for 77.5%. The average time for processing the requests was 0.025 seconds.

6 Conclusions

In this work we presented an novel approach for enabling aggregators to perform entity-based integration, leading to more efficient and effective integration of Social and Semantic Web data. In particular we equipped aggregators with an Entity-Name-System, which offers storage and matching functionality for entities. The matching functionality is based on a generic framework that allows incorporation and combination of an expandable set of matching modules. The results of an extended experimental evaluation on Web data, such as Web blogs and news articles, demonstrated the efficiency and effectiveness of our approach.

Ongoing work includes the improvement of the existing functionality based on the fact that many entity integration requests may arrive at the same time to the aggregator. This requires a special handling both for reasons of efficiency and for reasons of accuracy. We refer to this task as *bulk integration*. The entities given for bulk integration may exhibit some special characteristics, such as presence of inner-relationships, which should be also considered to improve integration's performance and quality. We are also investigating the ways of improving integration by exploiting external knowledge in systems, such as WordNet. Another interesting direction, which we are currently investigating, is how to *combine* and reason over the matching results that have been generated by more than one matching modules.

References

1. Aizawa, A., Oyama, K.: A fast linkage detection scheme for multi-source information integration. In: WIRI. pp. 30–39 (2005)
2. Alexe, B., Tan, W.C., Velegrakis, Y.: STBenchmark: towards a benchmark for mapping systems. PVLDB 1(1), 230–244 (2008)
3. Amer-Yahia, S., Markl, V., Halevy, A.Y., Doan, A., Alonso, G., Kossmann, D., Weikum, G.: Databases and Web 2.0 panel at VLDB 2007. SIGMOD Record (2008)
4. Ananthakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In: VLDB (2002)
5. Bhattacharya, I., Getoor, L.: Deduplication and group detection using links. In: LinkKDD (2004)
6. Bilenko, M., Mooney, R., Cohen, W., Ravikumar, P., Fienberg, S.: Adaptive name matching in information integration. IEEE Intelligent Systems 18(5), 16–23 (2003)
7. Cohen, W.: Data integration using similarity joins and a word-based information representation language. TOIS 18(3), 288–321 (2000)

8. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In: IIWeb co-located with IJCAI. pp. 73–78 (2003)
9. Doan, A., Halevy, A.Y.: Semantic integration research in the database community: A brief survey. AI Magazine (2005)
10. Doan, A., Lu, Y., Lee, Y., Han, J.: Object matching for information integration: A profiler-based approach. In: IIWeb co-located with IJCAI. pp. 53–58 (2003)
11. Dong, X., Halevy, A., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD Conference. pp. 85–96 (2005)
12. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. TKDE (2007)
13. Fagin, R., Haas, L., Hernandez, M., Miller, R., Popa, L., Velegrakis, Y.: Clio: Schema mapping creation and data exchange. In: Conceptual Modeling: Foundations and Applications, pp. 198–236. Springer (2009)
14. Ferrara, A., Nikolov, A., Scharffe, F.: Data Linking for the Semantic Web. Journal of Data Semantics 7(3) (2011)
15. Getoor, L., Diehl, C.P.: Link mining: a survey. SIGKDD Explorations (2005)
16. Ioannou, E., Garofalakis, M.: Query analytics over probabilistic databases with unmerged duplicates. TKDE 27(8), 2245–2260 (2015)
17. Ioannou, E., Nejdl, W., Niederée, C., Velegrakis, Y.: On-the-fly entity-aware query processing in the presence of linkage. PVLDB 3(1), 429–438 (2010)
18. Ioannou, E., Niederée, C., Nejdl, W.: Probabilistic entity linkage for heterogeneous information spaces. In: CAiSE. pp. 556–570 (2008)
19. Ioannou, E., Niederée, C., Velegrakis, Y.: Enabling entity-based aggregators for web 2.0 data. In: WWW. pp. 1119–1120 (2010)
20. Ioannou, E., Sathe, S., Bonvin, N., Jain, A., Bondalapati, S., Skobeltsyn, G., Niederée, C., Miklos, Z.: Entity search with NECESSITY. In: WebDB (2009)
21. Koudas, N., Marathe, A., Srivastava, D.: Flexible string matching against large databases in practice. In: VLDB. pp. 1078–1086 (2004)
22. McCallum, A., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets with application to reference matching. In: KDD. pp. 169–178 (2000)
23. Miklós, Z., Bonvin, N., Bouquet, P., Catasta, M., Cordioli, D., Fankhauser, P., Gaugaz, J., Ioannou, E., Koshutanski, H., Maña, A.: From web data to entities and back. In: CAiSE. pp. 302–316 (2010)
24. On, B.W., Koudas, N., Lee, D., Srivastava, D.: Group linkage. In: ICDE (2007)
25. Papadakis, G., Ioannou, E., Niederée, C., Fankhauser, P.: Efficient entity resolution for large heterogeneous information spaces. In: WSDM. pp. 535–544 (2011)
26. Papadakis, G., Ioannou, E., Niederée, C., Palpanas, T., Nejdl, W.: Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In: WSDM. pp. 53–62 (2012)
27. Rastogi, V., Dalvi, N., Garofalakis, M.: Large-scale collective entity matching. PVLDB 4(4), 208–218 (2011)
28. Shen, W., DeRose, P., Vu, L., Doan, A., Ramakrishnan, R.: Source-aware entity matching: A compositional approach. In: ICDE. pp. 196–205 (2007)
29. Staworko, S., Ioannou, E.: Management of inconsistencies in data integration. In: Data Exchange, Integration, and Streams, pp. 217–225 (2013)
30. Tejada, S., Knoblock, C.A., Minton, S.: Learning domain-independent string transformation weights for high accuracy object identification. In: KDD (2002)
31. Whang, S., Menestrina, D., Koutrika, G., Theobald, M., Garcia-Molina, H.: Entity resolution with iterative blocking. In: SIGMOD Conference. pp. 219–232 (2009)