High-Level Rules for Integration and Analysis of Data: New Challenges

Bogdan Alexe[1], Douglas Burdick[1], Mauricio A. Hernández[1], Georgia Koutrika[2], Rajasekar Krishnamurthy[1], Lucian Popa[1], Ioana R. Stanoi[1], and Ryan Wisnesky[3]

[1] IBM Almaden Research Center
{balexe,drburdic,mahernan,rajase,lpopa,irs}@us.ibm.com
[2] HP Labs
koutrika@hp.com
[3] Harvard University
School of Engineering and Applied Sciences
ryan@cs.harvard.edu

1 Introduction and Motivation

Data integration remains a perenially difficult task. The need to access, integrate and make sense of large amounts of data has, in fact, accentuated in recent years. There are now many *publicly* available sources of data that can provide valuable information in various domains. Concrete examples of public data sources include: bibliographic repositories (DBLP, Cora, Citeseer), online movie databases (IMDB), knowledge bases (Wikipedia, DBpedia, Freebase), social media data (Facebook and Twitter, blogs). Additionally, a number of more specialized public data repositories are starting to play an increasingly important role. These repositories include, for example, the U.S. federal government data, congress and census data, as well as financial reports archived by the U.S. Securities and Exchange Commission (SEC).

However, in all of these cases, the data has become increasingly more heterogeneous and less structured. Even within one source (e.g., SEC or DBpedia), bits and pieces of data about the same real-world entity (such as a person, a company or a product) are often buried in text, html, XML, or other formats, and spread over many documents. In order to make sense of all this data at the aggregated level, it is necessary to build an entity or concept-centric view [10] of the domain, where clean and rich entities, together with their relationships, are aggregated from the myriad of unstructured or semi-structured pieces of data. It is these entities and relationships that will provide the real value to a human user or to the subsequent applications that need to consume information. In fact, many companies (so called data aggregators) have started to emerge in this space, aiming to create integrated value on top of the underlying raw data.

However, achieving the level of integration that is required in such practical scenarios is a challenge. There are many types of techniques that need to be put together in a complex data processing flow. These techniques include: *information extraction* [11] (to produce structured records from text or semi-structured data), *cleansing and normalization* (to be able to even compare string values of the same type, such as a dollar amount or a job title), *entity resolution* [13] (to link records that correspond to the same real-world entity or that are related via some other type of semantic relationship), *mapping* [14] (to bring the extracted and linked records to a uniform schematic representation), and *data fusion* [6] (to merge all the related facts into one integrated, clean object). In practice, these steps are often implemented in general purpose languages (e.g., Java, Perl), using ETL tools, or using general data manipulation languages (e.g., XSLT, Pig Latin). Often, the emphasis is on the low-level operations (sort, pipe, duplicate elimination, join, string matching, etc.) without a high-level view of the data integration steps. Most of the time, there is no explicit entity or object view, but rather tuples, arrays, key/value pairs.

In this paper, we advocate the need for a *high-level language or framework* to describe the main logical operations of data integration (e.g., entity extraction, entity resolution, mapping, fusion) and analysis (e.g., aggregation, view creation, temporal analysis). We emphasize the logical specification aspects rather than the physical implementation. In addition to ease of specification or programmability, such a framework would also enable better readability, better reuse and better customization of data integration and analysis (to other domains, other tasks, other views). The target users of such framework are developers that need to perform complex, industrial-strength data integration tasks.

We will illustrate the paper with an end-to-end scenario of integration that is focused on people and company entities. This scenario is drawn from our own experience, as part of the Midas project [3, 7] at IBM, with integrating data from DBpedia and especially SEC, which we have used extensively as a source for integration in the financial domain. Similar challenges or technologies will apply to other scenarios of integration from public data sources. We will focus our discussion on the high-level rules and declarations that are needed to accomplish the various integration steps. For each of the important tasks, the rules are shown in a candidate syntax that takes inspiration from existing formalisms, languages and tools for information extraction, entity resolution and schema mapping. However, rather than fixing on a concrete language, the goal is to illustrate the features that need to be supported in such a language, as well as the challenges. Coming up with an actual integrated language that combines all these features together is a separate challenge in itself with many design choices.

This is mostly a vision paper, with the goal of raising the attention of interested researchers towards this area.

Note Some of the ideas and desiderata described in this article have subsequently led to the development at IBM of a high-level integration language called HIL [22].[4] This language includes declarative constructs for entity resolution and for mapping and fusion of data, and is now extensively used within IBM for large-scale integration over structured and unstructured data (e.g., social media, news articles, financial disclosures, enterprise data, etc.). The exact

[4] Thus, from a timeline point of view, this book chapter describes work that precedes the development of HIL.

language design choices and primitives of HIL, as well as its compilation and execution, are described in [22]. While HIL answers some of the research challenges outlined in this article, several important problems remain largely open, such as the need for tools or systems to support large-scale data exploration or to assist users with the actual development of a *good* set of data analysis rules.

1.1 Overview of the Paper

We start in Section 2 by describing some of the features of the data in DBpedia, as well as the challenges involved in data exploration, which is a phase that *precedes* the actual writing of the rules. We then illustrate some concrete rules for extracting facts from DBpedia. Here, the output of an extraction rule has a relatively simple structure (or schema), but the input is semi-structured and largely heterogeneous. Extraction from completely unstructured data (i.e., text) [11] is highly related in this context; however, in this paper, we focus our attention specifically on extraction from semi-structured data (e.g., RDF, or XML, or JSON). We also note that extraction from text, technically, is of a different nature and is discussed extensively elsewhere (e.g., [8]).

In addition to giving examples of extraction rules, we also include a discussion of the need for automatic or semi-automatic extraction of structured records that is based on data examples. Such technology, while non-trivial, would be particularly useful when the developer is in the exploration phase and does not know enough about the data and its peculiarities. Based on a few examples that are representative of the type of entities that the developer is interested to extract, the system must first be able to derive all the other entries that are "similar" to the given examples. More challenging, the system should come up with a set of extraction rules that would result in such entries. While existing work on query discovery based on data instances [18, 27] or on schema mapping design based on examples [1, 21] may provide a starting point here, new types of algorithms will have to be developed to account for highly heterogeneous data with "less" schema (such as DBpedia).

The next integration component that we address is entity resolution, in Section 3. Rather than looking at specific algorithms or implementations that match records based on various similarity measures on their fields, we take a higher-level approach where the goal is to provide the *specification* framework for entity resolution. We advocate a framework that is based on logical constraints that are similar, in spirit, to the dependencies used in data exchange [15]. However, different from data exchange where the dependencies are source-to-target, our entity resolution constraints are target-to-source: they define *declaratively* all the desired properties of the target (i.e., of the links) in terms of the sources. Furthermore, these constraints incorporate disjunction (of the alternative matching rules that may apply), rely on user-defined functions for computing similarity of values, and can include cardinality constraints (e.g., to express many-to-one type of links). We include a discussion to illustrate the differences between this framework and previous approaches such as the Dedupalog language [2].

One of the main research problems that we outline, as part of declarative entity resolution, is the compilation of the declarative constraints into an execution plan that produces a good instantiation of the links. An important related question is formulating the semantics of the declarative constraints, which then needs to be implemented by the execution plan. Finally, a major challenge for entity resolution, which goes beyond the design of the specification language, is the development of methods and tools to help users interactively resolve the inherent ambiguities in their specification. These tools can help users refine the declarative constraints, based on the actual data sets that need to be linked, to ultimately achieve a high quality specification for entity resolution.

We discuss mapping and transformation, as well as data fusion and aggregation aspects in Section 4. While there is work on schema mapping tools [14], data exchange semantics [15], and data fusion methods [6], our goal is to develop an expressive scripting language that allows developers to combine non-trivial mapping, fusion and aggregation tasks (e.g., that are often not possible within a schema mapping tool paradigm) with the declarative entity resolution and extraction operations discussed earlier. At the same time, we emphasize simplicity and ease of programming as important requirements for the language design.

We discuss several other related papers and systems in Section 5 and conclude the paper in Section 6, where we reiterate the need for a single, unified framework that incorporates all the aspects outlined in the previous sections.

2 Data Exploration and Extraction

The first step before the actual writing of extraction and integration rules is the *exploration* phase, where a human user needs to understand what is in the source data and what can be extracted. This step is usually expensive; any help that a system or tool can provide in assisting the human user can be valuable. Even if the user has an idea of what concepts need to be extracted, the form in which these concepts manifest in the actual data source can vary significantly. Hence, heterogeneity is a challenge.

We start with an example from DBpedia to illustrate the issues. We focus on financial companies (e.g., Bank of America, Citigroup); the goal here will be to extract structured records that are relevant for such financial companies and that are deemed useful towards building the final integrated view. First, we assume that the DBpedia data set is given as a set of JSON records, each corresponding to one entity. A record has a subject field (which is also the identifier of that entity), and then all the various properties recorded for that entity. This JSON representation can be easily obtained from the RDF version of Dbpedia, which records RDF triples of the form (subject, property, value).[5] The conversion from RDF to JSON is already a step towards a more unified view of the data, since it yields full objects rather triples. However, the format of these objects is wildly heterogeneous, even for the same "type" of entity, as we shall see shortly. A large

[5] See the Ontology Infobox Properties data set at http://wiki.dbpedia.org/Downloads.

```
{
  "assets": “US$ 2.264 trillion",
  "foundation": “1904",
  "homepage": [ "http://www.bankofamerica.com",
                "http://www.bofa.com” ],
  "industry": [ "Banking", “Financial services” ]
  "keyPeople": [
    “Bryan Moynihan",
    “(President and CEO)",
    “Charles Holliday",
    “(Chairman)"
  ],
  "location": [
    "Charlotte,_North_Carolina",
    "United_States",
    "North_Carolina"
  ],
  "name": "Bank of America Corporation",
  "numEmployees": "288000",
  "slogan": "Bank of Opportunity",
  "subject": "Bank_of_America",
  "type": "Public_company",
  "wikiPageUsesTemplate": "Template:infobox_company"
},
```

```
{
  "areaServed": "Worldwide",
  "assets": "$ 1.119 trillion (2007)",
  "companyName": "Goldman_Sachs",
  "companySlogan": "Our clients\' interests always come first",
  "companyType": "Public_company",
  "foundation": "1869",
  "founder": ["Marcus_Goldman“, “Samuel Sachs”],
  "homepage": "http://www.gs.com/",
  "industry": "Finance_and_insurance",
  "keyPeople": [
    "Lloyd_Blankfein",
    (Chairman & CEO)”,
    "Gary_Cohn",
    “(President & COO)”,
    “David Viniar“,
    “(Executive VP & CFO)”
  ],
  "location": [ "United_States", "New_York_City" ],
  "marketCap": "$ 65.91 billion (2007)",
  "numEmployees": "30,522 (2007)",
  "products": [
    "Financial_services",
    "Investment_bank"
  ],
  "revenue": "$ 87.968 billion (2007)",
  "subject": "Goldman_Sachs",
  "wikiPageUsesTemplate": "Template:infobox_company"
},
```

Fig. 1. Sample DBpedia records.

part of the subsequent processing will be devoted to extracting the relevant parts of the objects of interest, bringing the extracted parts to a uniform format, and then linking and integrating them with data from other sources (e.g., SEC).

Figure 1 illustrates two sample input records, in JSON, corresponding to the DBpedia entries for Bank of America and Goldman Sachs. Even though both of these records represent entities of a similar type (i.e., financial institutions), there is significant variation in the structure of the records (i.e., the attributes that are present, their types), in the naming of the attributes, and in the values and format of the values that populate the attributes. For example, Goldman Sachs has attributes such as “founder” and “marketCap”, while Bank of America does not include these attributes. Goldman Sachs has a “companyName” attribute, while the equivalent attribute for Bank of America is “name”. The “homepage” attribute for Goldman Sachs is a single string, while the similar attribute for Bank of America is an array of strings. Finally, the values themselves are not always clean or cleanly organized. For example, Bank of America includes “Banking” and “Financial services” under the “industry” attribute; the corresponding information for Goldman Sachs is actually distributed over two attributes (“industry” and “products”). Furthermore, the entries under the “keyPeople” attribute, in both records, are a mixture of person names and positions (titles), without an explicit tagging of the data.

```
FinancialCompany =
        for (r in DBpedia)
        let industryTerms = extractIndustries (r.industry),
            compName = extractCompanyName (r)
        where  contains (compName, "Bank|Insurance|Investment") or
                (some (i in industryTerms) satisfies
                     contains (i, "bank|banking|insurance|finance|financial"))
        return {company_id: r.subject,
                name: compName,
                foundation: r.foundation,
                industry: industryTerms,
                revenue: cleanDollarAmount (r.revenue)
                }
```

Fig. 2. Extraction rule for financial companies.

After exploring several more representative DBPedia entries for financial companies, the user may decide on a set of important *concepts* to be extracted from this collection of heterogeneous records. Each concept is based on a subset of attributes and, hence, it is a piece of a schema. In our scenario, the user may be interested in the following three concepts.

FinancialCompany (company_id, name, foundation, industry, revenue, ...)
CompanyAddress (company_id, street1, street2, zipcode, city, state, country)
KeyPeople (person_name, titles, company_name, age, biography, ...)

Note that, in general, the schema for these concepts must be *open* (see the above ... notation) to account for possibly other attributes of interest that may be added later. The high-level integration language will have to be flexible and account for such open schema by either not requiring the user to explicitly having to define the schemas of the concepts, or by using advanced programming language features such as record polymorphism to represent extensible record types [24, 25, 28].

Finally, other concepts can be defined later from either the same source (DBPedia) or from other sources (e.g., SEC, as we will see later). All of these extracted concepts will then be processed together, in the subsequent integration flow, to generate clean target entities with richer structure.

We focus next on how to extract the data to populate such concepts from the underlying collection of heterogeneous records.

2.1 Extraction Rules: Examples

Figure 2 gives a first example of a rule that extracts data for financial companies from DBpedia. This rule populates into the FinancialCompany concept. There may be other rules to further populate into this same concept (and possibly add new attributes). Thus, the actual instance of a concept will be given by a union of extraction rules.

The rule uses an XQuery-like syntax (although other types of syntax could also be used) to express the search for DBPedia records that match the characteristics of a financial company and also to express the extraction of the relevant attributes. Note the complex predicate that is used in the where clause to recognize a financial company. This predicate includes multiple string matching conditions that are based on financial keywords. Note also the extensive presence of user-defined functions (UDFs) that are used for various purposes:

– to *clean* the data in the individual attributes. For example, cleanDollarAmount is a function that transforms various heterogeneous string values that represent dollar amounts into a standardized form. Concretely, strings such as "$ 87.968 billion (2007)" and "US$ 2.264 trillion" could be transformed into "$87.96 billion" and "$2.26 trillion", respectively.
– to *extract* certain expected strings from an input record or value (e.g., extractCompanyName from r and extractIndustries from r.industry).
– more generally, to account for the heterogeneity in the input data or structure. For example, extractIndustries must account for the fact that the input r.industry could be a string such as "Finance_and_insurance" or an array such as ["Banking", "Financial services"]. The function must uniformly generate an array of terms identifying the various relevant industries (i.e., ["finance", "insurance"] from the first input and ["banking", "financial services"] from the second input).
  As another example, extractCompanyName has to account for the fact that the company name can appear under various attributes in the input record r (e.g., sometime name, and sometime companyName). Furthermore, the value itself must be normalized (e.g., "Goldman_Sachs" must be transformed to "Goldman Sachs").
  Note that the extracted and normalized industry terms and company name are used both in the predicate in the where clause that identifies a financial company and in the output of the rule.

In Figure 3, we show another example of an extraction rule from DBPedia, to produce records for the key people that are associated with the financial companies. As before, the rule makes use of UDFs to restrict to financial companies. An additional UDF extractNameTitles is used to convert an array of strings into a set of structured records with explicit name and titles fields. For example, the array of uninterpreted strings that is the value of the keyPeople field in the "Goldman Sachs" record in Figure 1 is converted into a set of three records:

```
{ name: "Lloyd Blankfein", titles: ["Chairman", "CEO"] }
{ name: "Gary Cohn", titles: ["President", "CEO"] }
{ name: "David Viniar", titles: ["Executive VP", "CFO"] }
```

Note that the above UDF must employ a name recognizer as well as a title recognizer. Also, it must take into account the sequence in which the names and the titles appear in the input string. In particular, the function must detect that the titles of a person follow the actual person name, and also it must be able to handle the absence of title information (e.g., two consecutive names).

```
KeyPeople =
        for (r in Dbpedia)
        let industryTerms = extractIndustries (r.industry),
            compName = extractCompanyName (r),
            peopleTitles = extractNameTitles (r.keyPeople)
        for (p in peopleTitles)
        where  contains (compName, "Bank|Insurance|Investment") or
                (some (i in industryTerms) satisfies
                     contains (i, "bank|banking|insurance|finance|financial"))
        return {person_name: p.name,
                titles: p.titles,
                company_name: compName,
                age: null,
                biography: null
               }
```

Fig. 3. Extraction rule for key people.

2.2 Challenges in Data Extraction

In general, extraction rules can be fairly complex and the development time can be extensive. On the one hand, they can be seen as a form of mapping rules that require many UDFs. On the other hand, however, they differ from traditional schema mappings in that the source schema, here, is very loose or non-existent. This makes it harder to benefit from schema mapping tools [14], which assume that the source schema and the target schema are both manageable and matched within a user interface, which is then used to drive the generation of the mapping rules. Generating a meaningful schema for DBpedia, even for a small portion of it, would mean generating a large number of union or choice types to account for the variation in the structure (even for the same type of entity). The ability to load, use and manage such schema within a mapping tool is a research challenge in itself.

A somewhat different research question is the following: *Can we generate or learn extraction rules directly from the data and/or from examples?* The starting points for such generation would be: the input source data (e.g., DBpedia), an existing library of UDFs (for normalization, cleansing, etc.), and a set of representative examples of the intended output data. Existing work on query discovery based on data instances [18, 27] or on schema mapping design and refinement based on examples [1, 21] may provide some foundations towards solving this problem. However, most of the existing work on query or mapping discovery has been restricted to the case of fixed, strictly relational, schemas; it is not clear to what extent their methods or ideas generalize to a highly heterogeneous environment.

The Lixto [20] system, aimed at extracting data from heterogeneous web documents, takes a different approach where a visual tool can be used to specify the various patterns that navigate a tree-like structure and select the relevant subsets of nodes. Although it uses example documents as a starting point, this framework is closer in spirit to the paradigm of visual query builders. One downside of Lixto is that, in a highly heterogeneous environment (like DBpedia), a user may end up having to specify a large number of navigation and selection patterns to account for all the variations in the structure (or instance values) of the objects to be extracted. Being able to further automate the process and to reduce the amount of user interaction is left as an open question.

Coming back to data examples, a related and possibly simpler research question than that of generating the extraction rules is the following: Given the input source data, and a set of representative examples of the output data, *is there a procedure that directly extracts all output records that are similar to the given examples?* In other words, instead of generating rules to extract data, one could employ a procedure that performs the extraction starting from the given examples. In more concrete terms, a developer manually extracts records for, say, “Bank of America”, “Goldman Sachs”, “American Express” and “Visa”, and then asks the procedure to extract all other “similar” such records from the input. Of course, defining what similar means is one of the challenges here.

3 Entity Resolution

To illustrate the problem of entity resolution, assume now that another extraction process uses SEC (rather than DBpedia) as a data source and extracts facts about key executives of public companies. The relation SecPerson, shown below, associates with each person a set of employment records that span, possibly, multiple companies over many years.

SecPerson (name, $\underline{\text{cik}}$, employment: (company, position, date), ...)

Note that the relation is nested in that the employment attribute is itself a relation (i.e., a set of records with attributes for company, position and date). In general, the support for a nested data model is a pre-requisite for any system or language that aims at integrating richly structured entities from heterogeneous data sources.

Specific to SEC data, each person is associated with a unique key (cik) that is globally identifies a person across multiple SEC filings. In contrast, such key does not always exist for DBpedia. Hence, before we can merge the information about people extracted from the two data sources (SEC and DBpedia), we need to be able to link or relate corresponding records in the two data sources that refer to the same person. This problem is widely known as *entity resolution.* Let us assume that we add a record id field (rid) to each KeyPeople record. Then, in an abstract sense, the problem of entity resolution becomes one of creating links of the form (rid, cik). Note that we use cik on the right side, since we know that cik is a key that identifies a person entity in SecPerson. However, on the left side, we use the entire record id, since we do not have a key of a person there. Essentially, we need to link multiple records, in general, in KeyPeople to exactly one person entity in SecPerson, by exploiting information such as name and also

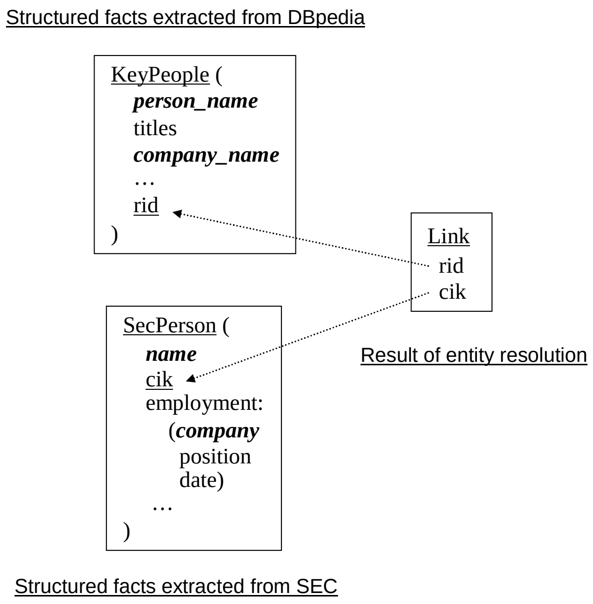

Fig. 4. Entity resolution diagram.

other contextual information such as employment. Figure 4 depicts schematically the concrete entity resolution scenario that we are considering.

3.1 Declarative Constraints for Entity Resolution

We now illustrate the logic that is needed to express the above entity resolution problem. We advocate a declarative formalism where one specifies the properties or *constraints* that the outcome of entity resolution (i.e., the link table) must satisfy, without having to specify a concrete procedure or implementation for computing this outcome. It will be the role of the underlying system to materialize a good solution (i.e., a set of links) that satisfies the specified constraints in the best possible way.

For our entity resolution example, we show in Figure 5 a set of declarative constraints that can be used to specify the desired properties of the link table. We believe that such constraints (and their extensions) should form the basic ingredients of any language that attempts to specify entity resolution at a high-level.[6] We explain the constraints first and then discuss the issues involved in building a language and system that implements such specification.

First, we have provenance or identification constraints that specify the attributes or combinations of attributes that identify the source objects to be

[6] However, the syntax of the actual language does not have to have follow the logical notation we use here. Furthermore, some of these constraints may be implicit in the semantics of the language.

$\text{Link [rid]} \subseteq \text{KeyPeople [rid]}$

$\text{Link [cik]} \subseteq \text{SecPerson [cik]}$

$\text{Link : rid} \rightarrow \text{cik}$

(m) every Link
satisfies
KeyPeople.person_name = SecPerson.name
or
(KeyPeople. person_name $\sim_{\text{name}}$ SecPerson.name
and
KeyPeople.company_name in SecPerson.employment [company]
)

Fig. 5. Declarative constraints for entity resolution.

linked. In this example, the two inclusion dependencies from Link to the sources specify that the projection of Link on rid must be a subset of the projection of KeyPeople on rid and, similarly, the projection of Link on cik must be a subset of the projection of SecPerson on cik. Thus, the intention behind Link is to be a subset of all the pairs of rid and cik values that appear in the two sources. In general, it is up to the user to define what constitutes the identifier of an object of interest for entity resolution. The framework we suggest is independent of what makes the identifier of an object. As a result, we can naturally capture most types of entity resolution described in the literature, from record linkage and deduplication [17, 23] to reference reconciliation [12] and to more general, semantic type of linkage among entities (e.g., the relationship between companies and subsidiaries). To follow some of the terminology in the literature, in our example, the first type of object that participates in Link can be viewed as an entity reference (since it refers indirectly to an actual person, via person name and other non-identifying attributes), while the second type of object can be viewed as an entity (since it identifies a person in SEC).

The next constraint in the specification is a functional dependency (on the Link table) to specify that an rid from the first source must be linked to a unique cik in the second source. Note that, in this example, it is is ok to have multiple rid's linked to the same person cik. Thus, by using a functional dependency, we encode an N:1 type of entity resolution (where multiple objects of interest in one source must be linked to a single object in another source). For 1:1 type of entity resolution, we would write a functional dependency in the other direction as well. For an N:M type of entity resolution, we do not need to specify any functional dependencies.

The final constraint in this example, probably the most important, is used to declare a disjunction of all the valid reasons for why two objects can match. Essentially this constraint specifies that a link can exist only if at least one of several matching conditions holds. The matching conditions are formulated with respect to the source tuples that are related via the link. In the example, we can have a match because of exact equality of person names, or because of similarity of person names (via a user-defined similarity predicate) and, moreover, because the company_name in the KeyPeople record appears in the employer set in the SecPerson record. Note that the second matching condition relaxes the equality on person names, when compared to the first matching rule, but at the same adds a strenghtening condition that is based on employment information. Note that the employment-based condition, although a strengthening, may apply to less tuples (those that have a non-empty employment set in SecPerson). In practice, one will have to formulate multiple matching conditions, in order to improve the recall of entity resolution. Furthermore, each matching condition has to be strong enough to prevent the generation of accidental links.

Other types of constraints that appear in practice are structural type of constraints requiring properties such as transitivity of matching or variations of it. Such constraints are needed to specify clustering behavior or to specify the linking of two objects in two sources due to another object in a third source that links to them.

A slight extension to this basic framework of constraints allows us to express *collective entity resolution* [5], where the task is to create multiple, inter-related types of links (rather than to create a single type of link). For example, assume that we have the following two source relations:

```
Paper (pid, title, venue, year, ...)
Venue (venue, conferenceOrJournal, sponsor, ...)
```

In this context, we may want to specify links between papers *and* links between venues. Assume that the first type of link is represented as a binary relation PaperLink(pid1, pid2), while the second type of link is represented as a binary relation VenueLink(venue1, venue2). Then, the matching rules for one type of link may depend on the other type of link. For example, we can declare the matching conditions for VenueLink as follows:

```
every VenueLink satisfies
  ... (some similarity condition on venue names) ...
or
  ... (other condition) ...
or
  exists (p1 in Paper, p2 in Paper)
    p1.venue = VenueLink.venue1 and p2.venue = VenueLink.venue2 and
    PaperLink (p1.pid, p2.pid)
```

In particular, the last condition says that a possible reason for a venue link is that there exist two papers that are linked via PaperLink and whose venues are the two venues related by the link.

Note that in the framework we suggest, we do not *force* the generation of links, but rather define them *implicitly* through a declaration of the possible matching rules. For example, satisfying the last matching condition in the above constraint does not mean that a VenueLink tuple will necessarily be created, since the existence of such tuple may be prevented due to other constraints. In fact, creating such link may be the wrong choice sometimes (e.g., a conference version and a journal version of a paper may be linked via PaperLink, but that does not mean that the conference and the journal represent the same venue). The disjunction allows us to enumerate, declaratively, all the possible reasons for why a link may exist without forcing the link generation. It is then the job of the underlying system to take into account all the constraints to reach a good set of links, as we discuss in the next section.

Other frameworks aimed at declarative entity resolution exist. Perhaps, the most comprehensive one is the Dedupalog [2] language which allows the use of constraints, expressed in a Datalog style of syntax, to drive the identification of duplicate entities. Several remarks are in order here. First, Dedupalog limits itself to links that are equivalence relations, thus focusing strictly on deduplication. In contrast, we require a more flexible framework for links that represent more general semantic relationships, going beyond the "same-as" type of relationship. Furthermore, Dedupalog rules are not entirely declarative. Generally speaking, rules in Dedupalog are a guideline for the implementation, and the intention of a rule is to populate links based on conditions on the sources or other links. Since forcing links may create inconsistencies in the result, Dedupalog compensates by allowing some rules to be soft: for such rules, links are "likely" to be generated. The system then figures out to what extent to satisfy these rules (e.g., by attempting to minimize the overall number of constraint violations). As a consequence, an important downside is that the result of Dedupalog evaluation does not satisfy, in a precise first-order logic sense, the Dedupalog rules that were given as a specification. Furthermore, it may not be easy for a user of the system to understand the properties of the final result.

In contrast, the matching constraints that we envision have a purely declarative flavor, where we specify all the desired properties on the target links, without worrying about how to actually generate the links. This achieves a better separation between specification and execution. Furthermore, we require all the declarative constraints to be satisfied, in a precise first-order logic sense, by any solution that implements the specification. Ultimately, we believe that such framework forms a better foundation for entity resolution that is transparent and high-quality while at the same time high-level.

3.2 From Declarative Constraints to Execution: Challenges

There are many foundational and architectural challenges that need to be solved, in order to achieve a functional framework for declarative entity resolution. The main research questions here will be to define precisely the language that captures all of the above types of constraints, to formulate its semantics, and to investigate the expressive power and computational aspects of the language. We outline some of the issues here, and leave further details, solutions or algorithms for future work.

One of the main problems for declarative entity resolution is the ability to execute or compile the declarative constraints that specify the desired properties of entity resolution into a more procedural plan that implements the specification. But what do we want this implementation to actually compute? Ultimately, we need one instance for Link that is a good solution, satisfying all the constraints. But there may be many such good solutions. This is similar, in some aspects, to data exchange semantics [15], where we can also have multiple solutions. For our example in Figure 5, we could have an instance (Solution 1) with one link satisfying the first disjunct in constraint (m), and another instance (Solution 2) that is exactly identical but replaces that one link with a new link satisfying both disjuncts in constraint (m). Intuitively, Solution 2 is a better solution, since it contains a stronger link (a link for which there is a stronger matching evidence).

While in the previous example, Solution 1 is dominated by Solution 2 and could be replaced by it, it is easy to come up with “good” instances for Link that are incomparable. For example, there could be multiple candidate links, satisfying the same disjuncts of constraint (m), each linking a KeyPeople record to a different cik. Since all of these links cannot co-exist together due to the functional dependency rid $\rightarrow$ cik, each of these links will be in a different good solution. The presence of incomparable “good” solutions is a more challenging situation than in data exchange, where universal solutions (i.e., the “good” solutions in data exchange) are all equivalent, and furthermore there is always a unique core universal solution. Thus, the entity resolution problem is inherently more ambiguous than the data exchange problem.

One of the more challenging aspects is therefore to design an *interactive* sytem for entity resolution that brings the human user in the loop in order to resolve ambiguity. Conceptually, the interactive system must take the initial specification (i.e., the constraints) and then enumerate through multiple good solutions for Link. In particular, the differences between these solutions must be pinpointed to the user, which can then decide how to further resolve these differences (for example, by adding stronger matching clauses to (m)). An essential part of the problem is being able to compactly represent and efficiently navigate through the space of all different solutions. This problem of efficient, interactive enumeration of a space of solutions, is similar in spirit to the problem addressed in [9] in the context of schema integration. There, multiple solutions for the schema integration problem are defined implicitly via a set of constraints (of a simpler nature than here), and the question is how to interactively explore and refine the space of solutions, in order to reach one final integrated schema. While similar in spirit, the problem of navigating through solutions for entity resolution is likely more challenging, especially due to the fact that the size of the data, in general, is much larger than the size of schemas.

4 Mapping and Fusion

We illustrate next how mapping and fusion operations can be used to put all the extracted facts together into rich entities, by also making use of the result of

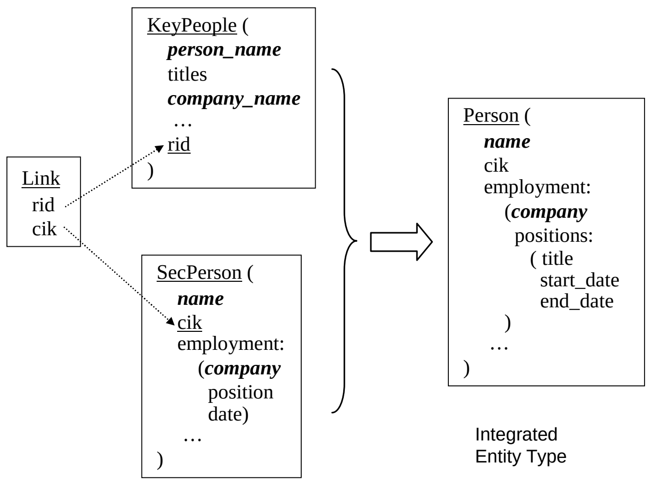

Fig. 6. From extracted facts and links to integrated entities.

entity resolution. While there is extensive work on schema mapping tools [14], data exchange semantics [15], and data fusion methods [6], there is not much work towards developing an actual scripting language that allows developers to combine all the necessary ingredients (mapping, fusion, aggregation, entity resolution, schema definition), while still maintaining simplicity and ease of use. An important aspect behind such desired language is the ability to express non-trivial ways of fusion and aggregation of data that are often not possible in a typical schema mapping tool, but are essential for developing industrial-strength data integration flows.

4.1 An Example of Transformation

To illustrate the issues, consider the (simplified) scenario shown in Figure 6 where the goal is to take the extracted facts (i.e., KeyPeople and SecPerson) as well as all the links generated so far, and create unified entities that conform to a target Person type or schema. The desired target entity type contains, in general, a union of many of the attributes from the sources. However, the structure is generally richer than in the sources, with various nesting levels to better aggregate and organize information. Furthermore, it is often the case that a target attribute represents a non-trivial aggregation over a set of source values. In this example, the employment history of Person has a two-level nesting where, for each company, we want a listing of all known positions with the given company, together with the start/end dates (as best as they can be inferred from the sources.) Part of the task here is to construct the nested structure, where we list the unique companies for which a person works, the unique positions the person held, and also to compute the start/end dates from the input facts.

```
Person =
  for (s in SecPerson)
  return {name: s.name,
          cik: s.cik,
          employment: for (e in s.employment)
                      group by comp = normalizeCompanyName (e.company)
                      return { company: comp,
                               positions: for (g in Group)  // Group is the group of all (company, position, date)
                                                             // records having the same normalized company value
                                          group by pos = normalizeTitle (g.position)
                                          return { title : pos,
                                                   start_date: minDate (Group),
                                                           // Group is now the group of all (company, position, date)
                                                           // records having the same normalized company and position
                                                   end_date: maxDate (Group)
                                                 }
                             }
         }
```

Fig. 7. Transformation from SecPerson to Person.

Computing the start/end date for a position is an example of *temporal aggregation*. These values that must be aggregated from all the input evidence (i.e., input dates) for a person working for a given company in a given position. Concretely, the fact that person X worked for a company C in some position P may be appear in multiple extracted records (possibly from many documents, each with a different date). This is especially true for SEC, which is a temporal archive that keeps track of past history, and where information must be periodically filed by the companies and their executives (even if nothing changed). Thus, in order to infer the start date for position P, we must look globally across all the sources and all the extracted records that mention person X as working for company C in position P and return the earliest known date.

Figure 7 shows an example of transformation that achieves the intended result for Person when considering the SecPerson in isolation (thus, ignoring KeyPeople and Link). The transformation is written in an pseudo-query language that abstracts features from query languages such as XQuery and Jaql [4]. The transformation consists of multiple levels of for statements that construct the structure of the target. To start with, the top-level part populates the name and the cik fields in Person. The rest of the transformation then makes essential use of the group by operation to put the target data into the desired form and also to perform aggregation. First, the employment records under SecPerson are grouped by the company name. Notably, the company name must be normalized to account for name variations for the same company. As a result of normalization and grouping, we obtain a set of unique company entries, each with an associated group containing all the records that share the same normalized company name. The group itself can then be further accessed by using the reserved word Group. A second level of grouping, this time by normalized position, produces the listing of unique positions. Finally, start_date can now be computed by taking the minDate function over the current group of records. A symmetric computation takes place for maxDate.

4.2 Mapping and Fusion: Making it Easier

Even though it achieves the intended result, given SecPerson alone, the transformation in Figure 7 is neither declarative nor easy to write. The programmer has to be quite familiar with the semantics of group by and has to understand the implicit collections over which aggregation needs to be performed. Furthermore, things become a lot more complex when additional data (e.g., KeyPeople from DBpedia, or other extracted records from other types of filings in SEC) need also to be fused into the Person entity. In such case, the above transformation has to be either rewritten to account for the new sources (and links), or its result must be integrated with the result of similar transformations from the other sources. However, the integration itself is low-level and complex, since the target components in Person, at various levels in the hierarchy, must be merged with the new data, and the values for start/end dates must be re-aggregated to account for the new data.

So, how do we make all this easier? The solution that has been tried in the past is to use graphical schema mapping tools [14] to help generate or re-generate the transformations. However, the process becomes clumsy when the transformations are complex and require a lot of aggregation and, ultimately, customization that is beyond the realm of the tool. Hence, we still need a language-level solution, but one that is more declarative and easier to use than writing raw transformations such as the one above.

The alternative that we are investigating is a rule language that allows for decorrelation of complex transformations via a mechanism that is similar to Skolem functions. As an example, the earlier transformation in Figure 7 can be rewritten as a simple rule where the value of employment is given by an explicit function call Employment(s.cik) that replaces the entire query block in the box. In other words, we would write:

```
Person = for (s in SecPerson)
         return {
                 name: s.name,
                 cik: s.cik,
                 employment: Employment (s.cik)
           }
```

Of course, explicit rules have to be written to define the value of the Employment function. The advantage is that the rule to populate the top-level part of Person does not need to know about how Employment is defined. The actual definition of Employment as a function parameterized by cik is delegated to separate rules that use possibly different data sources and that could rely themselves on other similar Skolem functions. Hence, we achieve a separation of concerns that can make the entire specification process more scalable and easier to evolve.

Another advantage of the decorrelation approach is that the Skolem functions themselves become first-class objects in the language, and can be used to express important parts of the integration logic that otherwise would be implicit. For example, the aggregation start_date: minDate(Group) can be rewritten as:

start_date: minDate (EmploymentProvenance (cik, comp, pos))

where EmploymentProvenance is now an explicit function that associates a triplet (cik, company, position) to the set of all source records that mention the fact that the person given by cik worked for company in the given position. As before, separate rules have to be written out to explicitly define EmploymentProvenance. But, again, the rule to aggregate and compute start_date need not know about how the provenance function is defined. Hence, we achieve the same separation of concerns.

Fleshing out the concrete details for this language, such as the type system, the allowed constructs, the efficient support for the functions that decorrelate the rules, as well as the integration with declarative entity resolution and extraction operations, falls outside the scope of this paper. Here, we outlined the issues as well as some of the motivation for why there is, still, a need for a good programmable language to address mapping and fusion in the context of the larger data integration.

5 Further Related Work

We have already discussed some of the relevant and recent work in the areas of entity resolution, schema mapping, data exchange and data fusion. We mention now a few other related research papers and systems. Ajax [19] is an early data cleaning framework. However, it was focused on matching and clustering and less on mapping and fusion. In particular, Ajax had no high-level constructs to support complex fusion and temporal aggregation, and had no notion of logical entities. On the other end of the spectrum, iFuice [26] combines mapping with fusion of data. However, iFuice includes no entity resolution (it assumes instead that the links are given), and fusion is focused narrowly on individual atomic attributes rather than applying on richer entity types.

More recently, the work on the interaction between matching dependencies and data repairs [16] combines record matching and data repairing for better data quality. As part of the high-level specification, matching dependencies (MDs) are used to identify or equate components of tuples in different data sets, while conditional functional dependencies (CFDs) are used to specify certain equalities of values within a given relation. In order to achieve a clean data set, cleaning rules implement the collection of MDs and CFDs by following certain pre-defined strategies (e.g., by using master data) to actually force the correction of the data. However, like in Dedupalog, matching dependencies only look at equivalence (same-as) type of linkage. Moreover, the notion of an entity (or entity link) is only implicit with matching dependencies. Furthermore, there is no notion of mapping or transformation from one entity type to another. In contrast, we are interested in a framework where entities have rich types and their properties (including the links) are first-class citizens. Additionally, we emphasize the programmability and customization aspect behind the cleansing, merging, transformation and aggregation of complex entities from the input data and the links.

6 Concluding Remarks

In summary, we outlined a vision of a high-level framework that covers multiple important steps in data integration. We exemplified rules and UDFs for extraction from semi-structured, heterogeneous data, which is complementary to text extraction. We outlined the need for and the challenges involved in learning or generating the extraction rules from examples. We illustrated the use of constraints as a foundation for declarative entity resolution, and outlined the challenges involved in defining the semantics and the compilation methodology for the declarative constraints. We further illustrated the types of rules for mapping and fusion that are needed to generate clean, unified entities.

It is important to emphasize that it is the combination of all these ingredients together (extraction, entity resolution, mapping, fusion) that gives enough expressive power to tackle complex, end-to-end data integration tasks. It is often the case that different types of rules must be interleaved together as part of the integration flow. Therefore, all the outlined components must be, ideally, part of a single framework that can be easily used by domain experts to specify and deploy sophisticated data integration flows for various scenarios. A further important factor that permeates all aspects of such framework is the need for tools that will assist users in various phases such as the data exploration or the development and refinement of the actual rules for entity resolution, for fusion, or for further analysis of the data.

References

1. Alexe, B., ten Cate, B., Kolaitis, P.G., Tan, W.C.: Designing and Refining Schema Mappings via Data Examples. In: SIGMOD. pp. 133–144 (2011)
2. Arasu, A., Ré, C., Suciu, D.: Large-Scale Deduplication with Constraints Using Dedupalog. In: ICDE. pp. 952–963 (2009)
3. Balakrishnan, S., Chu, V., Hernández, M.A., Ho, H., Krishnamurthy, R., Liu, S., Pieper, J., Pierce, J.S., Popa, L., Robson, C., Shi, L., Stanoi, I.R., Ting, E.L., Vaithyanathan, S., Yang, H.: Midas: Integrating Public Financial Data. In: SIGMOD. pp. 1187–1190 (2010)
4. Beyer, K., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.C., Ozcan, F., Shekita, E.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In: VLDB (2011)
5. Bhattacharya, I., Getoor, L.: Collective entity resolution in relational data. TKDD 1(1) (2007)
6. Bleiholder, J., Naumann, F.: Data Fusion. ACM Comput. Surv. 41(1) (2008)
7. Burdick, D., Hernández, M.A., Ho, H., Koutrika, G., Krishnamurthy, R., Popa, L., Stanoi, I.R., Vaithyanathan, S., Das, S.: Extracting, Linking and Integrating Data

from Public Sources: A Financial Case Study. IEEE Data Eng. Bull. 34(3), 60–67 (2011)
8. Chiticariu, L., Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F., Vaithyanathan., S.: SystemT: An Algebraic Approach to Declarative Information Extraction. In: ACL. pp. 128–137 (2010)
9. Chiticariu, L., Kolaitis, P.G., Popa, L.: Interactive Generation of Integrated Schemas. In: SIGMOD Conference. pp. 833–846 (2008)
10. Dalvi, N.N., Kumar, R., Pang, B., Ramakrishnan, R., Tomkins, A., Bohannon, P., Keerthi, S., Merugu, S.: A Web of Concepts. In: PODS. pp. 1–12 (2009)
11. Doan, A., Naughton, J.F., Ramakrishnan, R., Baid, A., Chai, X., Chen, F., Chen, T., Chu, E., DeRose, P., Gao, B.J., Gokhale, C., Huang, J., Shen, W., Vuong, B.Q.: Information Extraction Challenges in Managing Unstructured Data. SIGMOD Record 37(4), 14–20 (2008)
12. Dong, X., Halevy, A.Y., Madhavan, J.: Reference Reconciliation in Complex Information Spaces. In: SIGMOD Conference. pp. 85–96 (2005)
13. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate Record Detection: A Survey. IEEE TKDE 19(1), 1–16 (2007)
14. Fagin, R., Haas, L.M., Hernández, M.A., Miller, R.J., Popa, L., Velegrakis, Y.: Clio: Schema Mapping Creation and Data Exchange. In: Conceptual Modeling: Foundations and Applications. pp. 198–236. Springer (2009)
15. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data Exchange: Semantics and Query Answering. TCS 336(1), 89–124 (2005)
16. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Interaction between Record Matching and Data Repairing. In: SIGMOD Conference. pp. 469–480 (2011)
17. Fellegi, I.P., Sunter, A.B.: A Theory for Record Linkage. J. Am. Statistical Assoc. 64(328), 1183–1210 (1969)
18. Fletcher, G.H.L., Gyssens, M., Paredaens, J., Gucht, D.V.: On the Expressive Power of the Relational Algebra on Finite Sets of Relation Pairs. IEEE TKDE 21(6), 939–942 (2009)
19. Galhardas, H., Florescu, D., Shasha, D., Simon, E., Saita, C.A.: Declarative Data Cleaning: Language, Model, and Algorithms. In: VLDB. pp. 371–380 (2001)
20. Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Flesca, S.: The Lixto Data Extraction Project - Back and Forth between Theory and Practice. In: PODS. pp. 1–12 (2004)
21. Gottlob, G., Senellart, P.: Schema Mapping Discovery from Data Instances. Journal of the Association for Computing Machinery (JACM) 57(2) (2010)
22. Hernández, M.A., Koutrika, G., Krishnamurthy, R., Popa, L., Wisnesky, R.: HIL: A High-Level Scripting Language for Entity Integration. In: EDBT. pp. 549–560 (2013)
23. Hernández, M.A., Stolfo, S.J.: The Merge/Purge Problem for Large Databases. In: SIGMOD Conference. pp. 127–138 (1995)
24. Ohori, A.: A Polymorphic Record Calculus and Its Compilation. ACM Trans. Program. Lang. Syst. 17(6), 844–895 (1995)
25. Ohori, A., Buneman, P.: Type Inference in a Database Programming Language. In: LISP and Functional Programming. pp. 174–183 (1988)
26. Rahm, E., Thor, A., Aumueller, D., Do, H.H., Golovin, N., Kirsten, T.: iFuice - Information Fusion utilizing Instance Correspondences and Peer Mappings. In: WebDB. pp. 7–12 (2005)
27. Sarma, A.D., Parameswaran, A.G., Garcia-Molina, H., Widom, J.: Synthesizing View Definitions from Data. In: ICDT. pp. 89–103 (2010)
28. Wand, M.: Complete Type Inference for Simple Objects. In: LICS. pp. 37–44 (1987)