RJ10499 (ALM1206-001) June 1, 2012
Computer Science

IBM Research Report

HIL: A High-Level Scripting Language for Entity Integration

Mauricio Hernández, Georgia Koutrika,
Rajasekar Krishnamurthy, Lucian Popa
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Ryan Wisnesky
Harvard University

HIL: A High-Level Scripting Language for Entity Integration

Mauricio Hernández
IBM Research – Almaden
mauricio@almaden.ibm.com

Georgia Koutrika
IBM Research – Almaden
gkoutri@us.ibm.com

Rajasekar Krishnamurthy
IBM Research – Almaden
rajase@us.ibm.com

Lucian Popa
IBM Research – Almaden
lucian@almaden.ibm.com

Ryan Wisnesky
Harvard University
ryan@cs.harvard.edu

ABSTRACT

We introduce HIL, a high-level scripting language for entity resolution and integration. HIL aims at providing the core logic for complex data processing flows that aggregate facts from large collections of structured or unstructured data into a set of clean, unified entities. Such complex flows typically include many stages of data processing that start from the outcome of information extraction and continue with entity resolution, mapping and fusion. A HIL program captures the overall integration flow through a combination of SQL-like rules that link, map, fuse and aggregate entities.

HIL differs from previous tool-driven schema mapping systems in that it is a programming framework that (1) allows for more flexible specification of the integration rules, (2) incorporates entity resolution, (3) allows user-defined functions for customized cleansing, normalization and matching of values, and (4) uses a notion of logical indexes in its data model to facilitate the modular construction and aggregation of entities.

As a result, HIL can accurately express complex integration tasks, while still being high-level and focused on the logical entities (rather than the physical operations). Compilation algorithms translate the HIL specification into efficient run-time queries that execute on Hadoop. We show how our framework is applied to a real-world integration of entities in the financial domain, based on public filings archived by the U.S. Securities and Exchange Commission (SEC).

1. INTRODUCTION

In recent years, data integration has largely moved outside the enterprise. There is now a plethora of publicly available sources of data that can provide valuable information. Examples 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, especially in the industry. 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).

To enable systematic analysis of such data at the aggregated-level, one needs to build an entity or concept-centric view of the domain (aka "web of concepts" [12]), where the important entities and their relationships are extracted and integrated from the underlying documents. We refer to the process of extracting data from individual documents, integrating the information, and then building domain-specific entities, as *entity integration*. Enabling such integration in practice is a challenge and requires tremendous effort; there is a big need for tools and languages that are *high-level* but still *expressive enough* to facilitate the end-to-end development and maintenance of complex integration flows.

There are several techniques that are relevant, at various levels, for entity integration: *information extraction* [14], *schema matching* [31], *schema mapping* [17], *entity resolution* [16], *data fusion* [6]. These techniques have received significant attention in the literature, although most often they have been treated separately. In many complex scenarios, all of these techniques have to be used in cooperation (in a flow), since data poses various challenges. Concretely, the data can be unstructured (hence, it requires extraction to produce structured records), it has variations in the format and the accompanying attributes (hence, it requires repeated mapping and transformation), and has variations in naming of entities (hence, it requires entity resolution, that is, the identification of the same real-world entity across different records). Moreover, fusion (conceptually related to aggregation) is needed to merge all the facts about the same real-world entity into one integrated, clean object.

The HIL Language. In this paper, we introduce HIL (High-level Integration Language), a programming (scripting) language to specify the *structured part* of complex integration flows. HIL captures in one framework the mapping, fusion, and entity resolution types of operations. High-level languages for information extraction already exist (e.g., AQL [10]) and are complementary to HIL. The main design goal for HIL is to provide the precise logic of a structured integration flow while leaving out the execution details that may be particular to a run-time engine. The target users for HIL are developers that perform complex, industrial-strength entity integration and analysis. Our goal is to offer a more focused, more uniform and higher-level alternative than programming in general purpose languages (e.g., Java, Perl, Scala), using ETL tools, or using general data manipulation languages (e.g., XQuery, XSLT).

HIL exposes a data model and constructs that are specific for the various tasks in entity integration flows. First, HIL defines the main *entity types*, which are the logical objects that a user intends to create and manipulate. Each entity type represents a collection of entities, possibly *indexed* by certain attributes. Indexes are logical structures that form an essential part of the design of HIL; they facilitate the hierarchical, modular construction of entities from the ground up. The philosophy of HIL is that entities are built or aggregated from simpler, lower-level entities. A key feature of HIL is the use of record polymorphism and type inference [30], allowing schemas to be partially specified. In turn, this enables incremental development where entity types evolve and increase in complexity.

HIL consists of two main types of rules that use a SQL-like syntax. *Entity population rules* express the mapping and transformation of data from one type into another, as well as fusion and aggregation of data. *Entity resolution rules* express the matching and linking of entities, by capturing all possible ways of matching entities, and by using constraints to filter out undesired matches.

HIL vs. Schema Matching/Mapping. Schema matching and mapping tools also address the mapping and transformation aspects of data integration. However, the focus there is on *generating* the data transformation code by matching schema elements (automatically or in a GUI). The result of matching is compiled into an intermediate, internal mapping representation (e.g., s-t tgds [18]), which is then translated into lower-level languages (e.g., SQL, XQuery, XSLT) [17, 21]. A practical issue in this three-level architecture (schema matching, internal mapping, low-level transformation) is that, more often than not, the generated transformation will not fully achieve the intended semantics of the user. As a result, the user has to modify or customize the transformation; however, tools give a limited flexibility on how to do that, since they are based on visual interfaces and have a limited number of options.

Here, we take a different and arguably more flexible approach, where we surface a *programmable* language (HIL) that operates at the same level as the internal representations used in schema mapping tools. In HIL, the programmer has full control over specifying the mapping and fusion rules. These rules have a completely specified semantics in terms of execution, while at the same time being above the low-level execution layer. The core part of HIL borrows features from schema mapping formalisms (e.g., s-t tgds [18]) but at the same time its design is aimed at making the language easy and intuitive. As such, HIL drops features such as Skolem functions and complex quantifiers, it does not require any a priori schemas, it is polymorphic (to address heterogeneity and complexity in the input data), and includes user-defined functions that can be used for aggregation and data cleaning (e.g., normalization). Furthermore, HIL adds features such as the use of first-class indexes at the data model level, in order to model, explicitly, the important data structures in the integration flow. As an example, HIL can express a form of provenance that is based on inverted indexes. Finally, HIL includes the notion of a flow of rules, which is largely absent from schema mapping tools.

HIL vs. Entity Resolution (ER). A large fraction of entity resolution work focuses on measures of record similarity, such as edit distance [5], TF/IDF [11], Jaro [25] and complex multi-attribute measures [20]. Existing ER algorithms aim at efficiently generating pairs of similar records (e.g., [15, 27]) and clustering the similar records with respect to various constraints (e.g., [23, 33]). However, these techniques are for the most part black boxes providing users with no control over the accuracy of the ER result. Users cannot specify the ER logic, that is, the rules and constraints that determine when two entities match. Instead, this logic is hard-wired in the ER algorithm that internally determines the trade-off between the accuracy of the result and its computational cost.

Declarative approaches to entity resolution have emerged, including WHIRL [11], Dedupalog [1], and LinQL [24]. However, these are less expressive and flexible when compared to HIL, which allows the specification of complex ER rules that combine user-defined matching functions with semantic constraints that are required to hold on the result of entity resolution. WHIRL and LinQL do not support constraints at all, and WHIRL supports only TF/IDF similarity. In Dedupalog, given a set of candidate matches and constraints, the problem of generating valid ER results is framed as an optimization problem that minimizes the number of constraint violations. Since this optimization problem is intractable, efficient heuristics are used to find good solutions. While this approach reduces the burden on the user, the results may not be repeatable or easy to understand. HIL rules, on the other hand, provide explicit resolution actions on constraint violations, and this ensures that the result is deterministic.

Furthermore, it is the combination of entity resolution rules together with the rules for mapping, fusion and aggregation of data, in one framework, that gives HIL enough expressive power to achieve

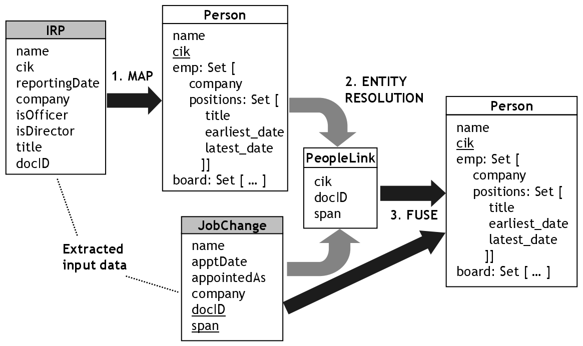


Figure 1: Example Integration Flow

complex, end-to-end integration tasks. We illustrate such task next.

1.1 Motivating Example

As a motivating scenario, we will use the financial data integration from SEC that was described as part of the Midas system [8]. In this scenario, company and people entities, together with their relationships, are extracted and integrated from regulatory SEC filings that are in semi-structured or unstructured (text) form. While SEC integration is one example application out of many, it is a good illustration of the kind of integration that is performed, in real-life, not only in Midas but also by financial data providers[1]. These providers often use a combination of manual methods (e.g., copy-and-paste then clean) and low-level coding to achieve a reasonable level of clean integrated data. In Midas, information extraction relies on SystemT [10] and its high-level language AQL. However, the subsequent, structured part of entity integration is a complex mixture of domain-specific rules for entity resolution, mapping and fusion. These rules were expressed mostly using Jaql [4], a general query language for JSON data, which in turn compiles into Map/Reduce jobs on Hadoop. As noted in [8], the integration process required a high cost of development and maintenance; the need for a high-level language to specify *what* needs to be done rather than *how* was identified as a key issue. Ideally, one would like to focus on the logical entities and the logical integration steps, declaratively, in the same way SQL is a higher-level alternative to a physical plan based on relational algebra operators.

Given the public availability of the SEC data, and its wide use by the financial data providers, analysts, and regulators, we are re-examining in this paper the SEC integration scenario, but from the perspective of using a high-level language. We will use this scenario to illustrate the core functionality that is covered by HIL, and also to evaluate HIL. Since HIL is a generic language for entity integration, other application domains can be envisioned. We start by showing a simplified portion of the SEC integration flow in Figure 1. The goal of this flow is to construct an entity type Person, representing the key people of major U.S. companies.

The flow uses two input data sets: InsiderReportPerson (or, IRP in short) and JobChange. The first is a set of records extracted from XML insider reports. These reports are filed periodically by companies to state compensation-related aspects about their officers and directors. Each extracted record includes the person name, a central identification key (cik, a global SEC-assigned key for that person), a company identifier, the reporting date, and whether the person is an officer or a director. If the person is an officer, the title attribute contains the executive position (e.g., "CEO", "CFO", etc).

The second data set, JobChange, consists of records extracted from unstructured (text) reports that disclose job changes or new appointments in a company. These records exhibit high variability in the quality of data (e.g., people names, positions). A record

[1]http://en.wikipedia.org/wiki/Financial_data_vendor

in JobChange includes the extracted person name, the appointment date, the position (appointedAs), and the information about the appointing company. However, it does not include any *key* identifying the person. The attributes docid and span identify the document and the position within the document where the person name has been extracted from. Together, they serve as an identifier for the particular person occurence. (Note that the same real-world person may occur in many documents or many places in the same document.)

The first step in the flow is a mapping or transformation that constructs an initial instantiation of Person from insider reports. Since data from IRP is relatively clean and each person has a key, the resulting Person entities will form a reference data set to be used and further enriched in subsequent steps. For each person key (cik), we create a unique person entity that includes top-level attributes such as the person name and the key itself. Then, for each person, we must construct an employment history by aggregating from many input records. The employment history includes the companies for which the person worked over the years, and for each company, a list of positions held by that person. Since a position is a string that can vary from record to record (e.g., "CEO" and "Chief Exec. Officer"), normalization code must be used to identify and fuse the same position. Moreover, for each position, we must capture the earliest known date and the latest known date for that position. These attributes, earliest_date and latest_date, are the result of a temporal aggregation that considers all the IRP records that refer to the same person, company and position.

The second step in the flow starts the integration of the second data set, JobChange, by linking its records to corresponding Person entities. This entity resolution step produces a PeopleLink table that relates each person occurrence (identified by docid and span) in JobChange with an actual person (identified by cik) in Person. In the third step, we join JobChange with PeopleLink in order to "retrieve" the person cik, and then insert or "fuse" appropriate data into the employment history for that person entity. This fusion step is non-trivial since it may affect data at several levels in the Person structure. We may either insert a new company into the employment history or modify the set of positions in an existing company. In turn, the latter step either inserts a new position or modifies an existing one (in which case, earliest_date and latest_date may be changed, by reapplying the temporal aggregation described earlier).

In the subsequent sections, we will describe the HIL constructs that allow to specify, at a high-level, the above types of operations.

1.2 Contributions and Paper Outline

The main contributions of this paper are as follows:

- We give a high-level programming language (HIL) that covers the major components of structured information integration (mapping, fusion, entity resolution).
- We give algorithms to compile the HIL specification into efficient queries that run on Hadoop. These algorithms include non-trivial translations to joins and outer-joins, and implement logical indexes as binary tables.
- We evaluate the resulting framework, by applying it to real-world integration in the financial domain.

Outline of the Paper. Section 2 introduces HIL. We describe the main algorithms for HIL compilation in Section 3. An experimental assessment of the HIL framework appears in Section 4. In Section 5, we discuss the ability of HIL to express additional commonly used patterns in data integration such as conflict resolution, blocking, and score-based entity resolution. We then discuss additional related work and conclude.

2. SCRIPTING IN HIL

In this section, we give an overview of the HIL language, illustrated on our running example. The main ingredients of HIL are: (1) *entities*, defining the logical objects (including the input data sources), (2) *rules*, for either populating the entities or linking among the entities, and (3) *user-defined functions*, which accompany rules and perform operations such as string-similarity or cleansing and normalization of values. A special form of entities in HIL are *indexes*, which can be shared among the rules and facilitate the hierarchical, modular specification of the integration flow, as well as various forms of aggregation.

2.1 Entity Population Rules

We start by describing the first of the two main types of entity rules in HIL, namely the rules to *populate* the target entities. We use the mapping task from IRP to Person as an illustration. We will give the entity types that are used, as well as the rules that map between the various entities. There are multiple steps, which gradually increase in complexity as the specification progresses.

Top-level Mapping. We start by declaring the input and output entities needed at this point (IRP and Person), by giving a partial specification of their types. More entities will be added later, to describe the additional data structures (e.g., indexes) that are needed. We also express now a first rule that populates the top-level attributes of Person.

```
IRP: set [name: string, cik: int, ?];
Person: set [name: ?, cik: ?, emp: set ?, ?];
rule m1:   insert into Person
           select [name: i.name, cik: i.cik]
           from IRP i;
```

We first explain the entity declarations and then explain the rule. First, the data model of HIL allows for sets and records that can be arbitrarily nested. In the above, IRP and Person are both sets of records. An important feature of the type system of HIL is that one can give an unspecified type (denoted by ?) in any place where a type can appear (i.e., as the type of an attribute or as the type of the elements in a set). Moreover, records themselves can be left *open*, meaning that there can be additional fields that are either unknown or not relevant at this point. (See the ? at the end of the record types for IRP and Person.) Open records are especially useful when schemas are complex but only some fields are relevant to the current transformation. As more rules and declarations are added, HIL will dynamically refine the types of the entities, by inferring the most general types that are consistent with all the declarations.

An entity population rule uses a select-from-where pattern to specify a query over one or more input entities; this query extracts data that is then used to populate (partially) the output entity that is mentioned in the insert clause. For our example, rule m1 specifies that for each record i from IRP, we select the name and cik fields and use them to populate the corresponding attributes of a Person record. The select clause of a rule contains, in general, a record expression (possibly composite).

The semantics of an entity population rule is one of *containment*: for each tuple that is in the result of the select-from-where statement, there must be a tuple in the target entity (in the insert clause) with corresponding attributes and values. Thus, like types, entity population rules are open; for our example, Person may contain additional data (e.g., more records or even more attributes for the same record), that will be specified via other rules (or constraints, see later). This is consistent, in spirit, with the usual open-world assumption in data integration [28]. We also note that, since rules define only partially the target entities, it is the role of the HIL compiler (described in Section 3) to take all the declarations and create an executable set of queries that produce the final target entities.

Using Finite Maps (Indexes). We now introduce indexes, which are central to HIL and allow the modular and hierarchical construction of entities. The above rule m1 specifies how to map the top part of Person, but is silent about the nested set emp, which represents the employment history of a person. One of HIL design choices, motivated by simplicity, is that entity population rules can only map tuples into *one* target set. Any nested set (e.g., emp) is populated separately via a *finite map* or *index*. Similarly, any aggregated value that needs to appear in an entity will be computed by utilizing an index, which is populated separately. (We illustrate aggregation later in this section.)

An index is declared as a finite map: fmap $\mathbf{T_1}$ to $\mathbf{T_2}$, where $\mathbf{T_1}$ is the type of keys and $\mathbf{T_2}$ is the type of entries. In many cases, $\mathbf{T_2}$ is a set type itself. In our example, we declare an Employment entity to be an index that associates a person identifier (i.e., cik) with the employment history of that person (i.e., a set of companies, each with a set of positions):

```
Employment: fmap [cik: int]
                  to set [company: string, positions: set ?];
```

One can visualize this declaration in terms of a hash table where each key has the form [cik: <person_cik>] and whose associated value is a set of employment tuples for <person_cik>, each for a particular company. We now modify the earlier rule m1 to specify that the nested emp set of Person is the result of an index lookup on Employment (we use ! for the lookup operation):

```
rule m1′:  insert into Person
           select [ name: i.name, cik: i.cik,
                    emp: Employment![cik: i.cik] ]
           from IRP i;
```

Intuitively, the rule assumes that we have constructed (or we will construct) separately Employment and here we simply access its entry for the key i.cik.

Alternatively, a HIL *constraint* can specify, globally, the relationship between Person and Employment. This has the advantage that existing rules for Person do not need to be modified. Instead, the HIL compiler will ensure that the constraint is globally applied to all the rules that populate Person (and, in particular, that m1 will be rewritten into m1′).

```
constraint c1:  for (p in Person)
                assert p.emp = Employment![cik: p.cik];
```

As a parenthesis, we note that specifying the emp field of Person in terms of the lookup on Employment (in c1 or in m1′) enables HIL to infer a richer type for Person. Concretely, the type for emp changes from set ? to set [company: string, positions: set ?]. Internally, HIL uses type and row variables instead of ?, and applies unification to equate and evolve types as new declarations are given.

The above bits of specification do not state how to populate Employment but rather how it is used in Person. Separate rules can now be used to populate Employment. In particular, the following rule populates Employment based on data from IRP:

```
rule m2:  insert into Employment![cik: i.cik]
          select [ company: i.company,
                   positions: Positions![cik: i.cik, company: i.company] ]
          from IRP i
          where i.isOfficer = true;
```

Following the general pattern discussed above, to populate the positions field, rule m2 relies on a separate entity, Positions, that is indexed by person cik and by company. The other notable thing about m2 is that this is a rule that populates an index. For each record i in IRP where isOfficer is true, we insert a tuple in the entry of the Employment index that is associated with the key i.cik. Different entries in Employment, corresponding to different cik values, may be touched. Note also that multiple tuples may be inserted in the same Employment entry, corresponding to multiple input records with the same cik value but different company values.

Indexes are important data structures in themselves, and often reflect the natural way in which logical entities need to be accessed. In this example, employment histories need to be looked up by person key, while positions represent a finer-grained view that is indexed by both person key and company. Furthermore, indexes are a convenient mechanism that allows to *decorrelate* and *decompose* what would otherwise be complex rules into much simpler rules. In particular, the rules that populate a top-level entity (e.g., a person) are decorrelated from the rules that populate the associated substructures (e.g., employment of a person). In our example, we can have subsequent rules that further populate the Employment index, without affecting any of the existing rules for Person.

(No) Ordering of Rules. We remark that there is no intrinsic order among the entity population rules. Here we gave the rule to populate Employment after the rule for Person, but the order could be switched. It is up to the programmer to define the conceptual flow of entities and of rules. In contrast, it is the role of the compiler to stage the execution so that any intermediate entities are fully materialized before they are used in other entities (i.e., all rules for Employment must be applied before materializing Person). The main restriction in HIL is that we do not allow recursion among the entity population rules (see later Section 3 for more on this).

User-Defined Functions. We specify below the actual population of Positions from IRP, with the help of a UDF or user-defined function, normTitle, to *normalize* the title string associated with a particular position. Normalization is an operation that is frequently encountered in data cleansing, and often requires customization. From the point of view of HIL, all we need to provide is the signature of the function. The actual implementation of such function is provided (either in Java or Jaql) via a binding mechanism.

```
normTitle: function string to string;

rule m3:  insert into Positions![cik: i.cik, company: i.company]
          select [title: normTitle(i.title)]
          from IRP i
          where i.isOfficer = true;
```

Indexes and Aggregation. We now show how one can use an index in HIL to perform aggregation. Aggregation is similar to the way nested sets are constructed, except that we also need an actual function to reduce a set to a single value. We show here how to compute the earliest_date for a position (the latest_date is similar).

Intuitively, each position we generate (e.g., by rule m3) originates in some input document that contains a date (i.e., the reportingDate attribute of IRP). To compute the earliest date for a position, we need an auxiliary data structure to keep track of all the reporting dates for a position (of a given person with a given company). Thus, we define an "inverted" index PosInfo that associates a set of dates for each triple (cik, company, title). This set of dates represents a form of *provenance* for the triple.[2]

```
PosInfo: fmap [cik: int, company: string, title: string]
           to set [date: ?, ?];

rule m4:  insert into PosInfo![cik: i.cik, company: i.company,
                               title: normTitle(i.title)]
          select [date: i.reportingDate]
          from IRP i
          where i.isOfficer = true;
```

Rule m4 parallels the earlier rule m3: whenever m3 produces a normalized title for a given cik and company, rule m4 produces the reporting dates (for all the input records in IRP that have the same

[2]We could also include other source fields (e.g., docID).

cik, company and normalized title). In general, there may be additional rules to populate this inverted index, since there may be more data sources or more rules (beyond m3) to populate Positions. Since HIL currently does not provide automated support for computing provenance (as in [9]), we must explicitly write these rules.

Computing the earliest date for a position amounts then to obtaining the minimum date in a set of dates. First, we declare a user-defined function minDate for which we also give a simple implementation in Jaql.

```
    minDate: function set [date: t, ?] to t;
    @jaql { minDate = fn(a) min (a[*].date); }
```

We then change the earlier rule m3 to use the inverted index by adding the following to the select clause:

```
(*) earliest_date: minDate(PosInfo![cik: i.cik, company: i.company,
                                    title: normTitle(i.title)])
```

Alternatively, we can use a constraint to state, globally, that for each possible entry in Positions, the earliest_date is computed by such a call to minDate and PosInfo.

So far, we have given the main entity population rules that are needed to construct a Person entity, and some of the associated structure (e.g., employment and positions), from one input data source. We focus next on how to enrich this basic scenario, based on additional data sources. In particular, we look at entity resolution rules and the second step of our running example.

2.2 Entity Resolution Rules

An entity resolution rule takes as input sets of entities and produces as output a set of *links* between these entities. Each link entity contains references to the input entities and represents a semantic association or correspondence between those entities. For example, if the input entities contain information about people, the generated links will connect those entities that contain, presumably, information about the same real-world person.

An entity resolution rule uses a select-from-where pattern to specify how input entities are linked. The from clause specifies the input sets of entities that we want to link. The where clause describes all possible ways in which input entities can match. For example, one can specify that if the names of people in two lists are "similar", then a "candidate" link exists between the two people. Furthermore, additional clauses, including check, group by and cardinality clauses, specify constraints that filter the "candidate" links. For instance, if only one-to-one matches between people entities are allowed, candidate links that connect one person in one list with multiple persons in another list will be dropped. Next, we describe these clauses in detail using our running example (see Figure 1).

In our example, we want to match Person entities with JobChange entities using a person's name and employment history. If the name of the company that filed the job change already appears on the person's employment history, then we can use both the company and the person names to match the corresponding input entities. Otherwise, we only do a strong similarity match on the person names. In both cases, we do not want to create a match if a different birthday appears in both entities. Furthermore, in this particular entity resolution task, one Person entity can match multiple JobChange entities. However, multiple Person entities *cannot* match the *same* JobChange entity. When this conflict arises, we want to preserve the strongest links (e.g., those that match identical person names).

All these matching requirements are compactly captured in the following entity resolution rule (er1), which we analyze next:

```
  rule er1:  create link PeopleLink as
             select [cik: p.cik,  docid: j.docID,  span: j.span]
             from Person p, JobChange j, p.emp e
             where match1: e.company = j.company and
                           compareName(p.name, j.name),
                   match2: normalize(p.name) = normalize(j.name)
             check if not(null(j.bdate)) and not(null(p.bdate))
                   then j.bdate = p.bdate
             group on (j.docID, j.span) keep links p.name = j.name
             cardinality (j.docID, j.span) N:1 (p.cik);
```

The create clause specifies the name of the output set of entities (called PeopleLink here). The select clause restricts the attributes kept from the input entities to describe the link entities. For each link, we keep the key attributes of the input entities so that we can link back to them (along with any other information that may be required). In er1, we keep the (docid, span) from each JobChange and the person cik. Similarly to SQL, the create and select clauses are logically applied at the end, after processing the other clauses.

The from clause names the sets of entities that will be used to create links, which in our example are the sets Person and JobChange. Interestingly, this clause can also include other auxiliary sets, like the nested set p.emp that contains the employment of a person p. In this way, a user can link entities not only by matching attribute values but also by matching a value (such as a company name) to a set of values (e.g., the set of companies in a person's employment history). The from clause defines a set C of tuples of entities, corresponding, roughly, to the cartesian product of all input sets. However, if a nested set in the from clause is empty, C will still contain an entry that combines the other parts. In our example, if a particular p.emp is empty, the corresponding Person and JobChange entities will appear in C with a value of null in the p.emp part.

The where clause specifies the possible ways in which the input entities can be matched and essentially selects a subset of C. Each possible matching has a label (used for provenance of matches) and a predicate on the entities bounded in the from clause. Rule er1 specifies two matchings, labeled match1 and match2. A matching predicate is a conjunction of conditions that combine equality and relational operators (e.g., e.company = j.company), boolean matching functions (e.g., compareName(p.name, j.name)) and transformation functions (e.g., normalize(p.name)). For example, match1 states that a JobChange entity can match a Person if the company name in JobChange is in the Person's employment history and the person names match. For comparing person names, match1 uses compareName, a specialized UDF that we have built for this purpose. Note that match2 uses only an equi-join on the normalized person names to count for those cases that the company filing a job change for a person is not in the employment history of that person.

HIL filters out any tuple in C that *does not satisfy* any of the specified matchings. In effect, every matching $r_i (1 \leq i \leq n)$ results in a $C_i = \sigma_{r_i}(C) \subseteq C$. The result of the where clause is the union of all these subsets, $W = \bigcup_i^n C_i$, which we call the "candidate links". An important aspect is that *all* matchings in an entity resolution rule will be evaluated, regardless of their relative order and whether a matching evaluates to true or false.

Note that while rule er1 uses simple matching predicates with equi-joins and boolean matching functions, in reality, we can combine several, complex, matching predicates within a single entity resolution rule to count for the several variations in data and matching logic. HIL can in fact support other flavors of entity resolution. In Section 5, we will illustrate score-based matching and blocking.

Entity resolution rules can also specify semantic constraints that are required to hold on the output links and provide explicit resolution actions on constraint violations ensuring that the result is deterministic. The clauses check, group and cardinality serve this purpose and appear in a entity resolution rule in this order.

A check clause specifies further predicates that are applied to each candidate link. A check clause has the form if p_k then c_k, with p_k and c_k being predicates over the candidate links. For every candidate link in W, if p_k evaluates to true, then we keep the link only if c_k also evaluates to true. In our example, we want to enforce that if the entities for a person in a candidate link contain non-null birthdates, then the birthdates must match. In effect, a check clause specifies a global condition that must be satisfied by all candidate links matching p_k, regardless of the matching predicates. That is why although this condition could be "pushed-up" to each matching predicate, it is more convenient to specify it in a check clause.

The group on clause applies predicates to groups of candidate links. The clause specifies a list of attributes that serves as a grouping key and a predicate that is applied to all entities in a group. In our example, a person occurrence in a JobChange entity (identified by (docID, span)) may be linked to multiple entities in Person. We want to examine all these links together. Any link where the person name in both linked entities is exactly the same should be kept (while the other links are rejected) because having the same name provides stronger indication that we actually have a match. Of course, when there are no such "strong" links, in our example, we keep weaker links. Additional group and cardinality constraints can be specified to further refine the links. We could also specify that only the strongest links survive, by just changing the keep links part of the group clause to keep only links.

Additional types of group constraints are possible. For example, we can use aggregate functions on the attributes of a group to decide whether to keep the links or not. For example, the constraint

```
group on   (p.cik) keep links
                  e.company = j.company and
                  j.apptDate = max(j.apptDate)
```

keeps the most recent job change among all those filed by the same company for the same person (cik). As another yet example, we could use the provenance of the link to select links that are created by stronger matching predicates. For example, we could specify that if a JobChange matches several Person entities, then we should keep links created by match1 if they exist. The use of such group-based conditions justifies why HIL evaluates all matchings defined in the where clause.

Finally, a cardinality clause asserts the number of links a single entity can participate in (one or many). For example, the cardinality clause in er1 asserts that each (docID, span) pair should be linked to exactly one Person entity (but that Person entity can be linked to many JobChange entities). In the final result, if a (docID, span) pair maps to multiple ciks, then all these links are considered *ambiguous* and dropped from the output.

2.3 Additional Fusion Based on Links

We complete our running example with the rules for the fusion step (Step 3) in Figure 1. The following entity population rules fuse the new data from JobChange into the employment and positions of a person. These rules make use of a join with the PeopleLink table, which was computed by the previous entity resolution step.

```
rule m5:  insert into Employment![cik: l.cik]
          select [ company: j.company
                   positions: Positions![cik: l.cik, company: j.company] ]
          from JobChange j, PeopleLink l
          where j.docid = l.docid and j.span = l.span
            and  isOfficer (j.appointedAs) = true;

rule m6:  insert into Positions![cik:l.cik, company: j.company]
          select [title: normTitle(j.appointedAs)]
          from JobChange j, PeopleLink l
          where j.docid = l.docid and j.span = l.span
            and  isOfficer(j.appointedAs) = true;
```

The rules are similar to the earlier rules m2 and m3, except that the new data values (for company and title) come now from JobChange, while the cik of the person comes from PeopleLink. The join between JobChange and PeopleLink is based on docid and span, which form a key for JobChange. The rules also include a filter condition (and an UDF) to select only officers (and not directors).

Since HIL uses set semantics, the effect of m5 is that a new company entry will be inserted into the Employment index only if it did not exist a priori (e.g., due to rule m2) for the given person cik. If the company exists, then there is still a chance that the corresponding set of positions will be changed, since rule m6 may apply.

We must also ensure that the earliest and latest dates for a position will be adjusted accordingly, since we now have new data. For this, we make sure that the inverted index, PosInfo, that keeps track of all the reporting dates for a position, is also updated based on the new data. Thus, we need to write another rule (not shown here) that is similar to the earlier rule m4 except that we use JobChange and PeopleLink instead of IRP. The actual specification for earliest_date remains the same: the equation (*) and the discussion at the end of Section 2.1 applies here as well, with the difference that the minDate aggregation will now work on a larger set.

We note that we did not need to add any new target data structures (entities or indexes). The new rules simply assert new data into the same indexes declared by the initial mapping phase. This same pattern will typically apply when fusing any new data source: first, write entity resolution rules to link the new data source to the existing target data, then write entity population rules to fuse the new data into the target entities (and indexes).

2.4 HIL Syntax

Figure 2 gives the core syntax of HIL. The basic ingredients are the types and the expressions that can be built on these types. A HIL program is then a sequence of declarations of entities, rules and constraints. The entity population rules, the entity resolution rules and the constraints, although different in syntax, share the same building blocks: expressions, predicates, the shape of variable bindings that can appear in the from or for clause, etc. In the figure, we distinguished between two forms of entity population rules: the ones that insert into set-valued entities, and the ones that insert into indexes. An implicit restriction in HIL is that finite maps and functions are always top-level values (i.e., they must be declared as global entities, and cannot appear nested inside other values).

3. HIL COMPILATION

We now describe the compilation of HIL into efficient runtime queries. Since there are two main types of rules in HIL, of different nature, we first describe two compilation algorithms, one for the entity population rules, the other for the entity resolution rules. We then put these two components together in a single HIL compilation algorithm that takes into account the recursion that may exist between entity population and entity resolution rules.

3.1 Compilation of Entity Population Rules

We now highlight the main steps in the compilation of entity population rules. An initial step of parsing (which is standard and we do not describe here) is followed by type inference. We have given some intuition on how type inference refines entity types as new declarations are given. A full presentation of the inference algorithm is outside the scope of this paper and is given elsewhere.

The first step that we discuss is a phase where HIL rules are *enriched via the chase* with the constraints that relate the various entities. Recall that HIL constraints allow the programmer to specify global relationships between entities. The enrichment phase rewrites the rules so that it is guaranteed that their application, after enrichment, will satisfy all the global constraints.

Rule Enrichment via the Chase. Recall the rule m1 and the constraint c1 in Section 2.1. The rule m1 by itself does not guar-

```
                 Type variable   Unspecified type              Index type      UDF type
Types:   T ::=  int  |  string  | …  |  X  |  ?  |  set T  |  R  |  fmap T1 to T2  |  function T1 to T2
Record Types:    R :: = [ L1: T1, …, Ln: Tn ]         // closed record type
                      |  [ L1: T1, …, Ln: Tn, ? ]      // open record type

Entity Declaration:  E: set T;                       // set-valued entity
                     F: fmap T1 to T2;               // finite map (index)
                     f: function T1 to T2;           // UDF

Expressions:   e  ::= c | v | E                      // constants, variables, set-valued entities
                    |  v.L1 … Lm                     // record projection
                    |  [ L1: e1, …, Ln: en ]         // a record
                    | dom F                          // the set of all keys in the index F
                    | F ! e                          // index lookup
                    |  f (e)                         // UDF invocation

Predicates:    Pred ::= e op e'                      // comparison operation
                      | f(e)                         // used-defined predicate
                      | Pred1 and Pred2

Entity Population Rules:
           insert into E                          insert into F ! e'
           select  e                              select e
           from v1 in e1, … vk in ek              from v1 in e1, … vk in ek
           where Pred;                            where Pred;

Constraints:   for (v1 in e1, … vk in ek)
               where Pred
               assert e = e';

Entity Resolution Rules (binary case only):
           create link T as                          // Link table
           select [A: key, B: key']                  // identifying keys for the entities being matched
           from v1 in e1, … vk in ek
           where match1: Pred1, …, matchm: Predm    // matching rules
           check if Pred then Pred,  …
           group on (e1, …, el) keep links Pred , …
           cardinality key (1:1 | N:1) key';
```

Figure 2: HIL Syntax

antee the satisfaction of the constraint c1. In other words, a query that strictly implements m1 will create facts into Person but will not necessarily populate the emp field. We use a simple variation of the chase [2], to *build* constraints such as c1 *into* the rules. We note that the constraints we use are a form of equality-generating dependencies [2], where the right-hand side of the implication is an equality between two HIL expressions. (as in c1, for example).

We apply the constraints only on the target side of a rule. More precisely, the application of a constraint can change only the select clause specifying the records to be inserted in the target entity. In our example, c1 is applicable to m1, since c1 specifies that, for any record p in Person, the value of the emp field must equal to Employment![cik: p.cik]. At the same time, the *select* clause of m1 does not specify any value for the emp field. Thus, the chase step will try to add such field. To succeed, we must ensure that the cik field is actually defined by the select clause of the rule. For our example, this is the case, and we can replace Employment![cik: p.cik] by Employment![cik: i.cik]. The result is the rule m1′. If the select clause does not specify any expression for the cik field, then the chase step cannot succeed, and the rule will not be modified.

The chase process applies the constraints to all the possible rules until no longer applicable. It is possible, in general, to have a case where two constraints will try to assign different expressions for the same field in the select clause of a rule. If that happens, a compilation error is given and the HIL programmer must fix the conflict, by possibly removing one of the constraints. At the end of the chase, we can ignore all the constraints, since they were built into the rules. The next phase, which is query generation, starts from the enriched rules (e.g., we will use m1′ instead of m1).

Semantics of Entity Population Rules and Query Generation. The naive semantics of entity population rules is to identify all the applicable rules, that is, rules which generate new facts, and then to insert all the new facts into the target entities (either sets or indexes). This process then repeats, until no new facts can be generated. To avoid such iterative process, which can be inefficient, we use compilation to implement the semantics. The main assumption we make is that there is no recursion allowed among the entity population rules; hence, we can topologically sort the entities based on the dependencies induced by the rules, and then generate unions of queries (with no recursion) to populate the entities.

For presentation purposes, we break query generation into two steps. In the first step, the *baseline for HIL query generation*, indexes are implemented as functions and index lookups as function calls. In the second step, we transform the baseline queries into more efficient queries, where indexes are implemented as materialized binary tables and index lookups are implemented via joins.

3.1.1 Baseline Compilation Algorithm

We now describe the baseline algorithm for query generation. For each entity that appears in the insert clause of an enriched rule, we will generate a query term to reflect the effect of that rule. Since there may be many rules mapping into the same entity, the query for an entity will include a union of query terms (one per rule). In the additional case when the entity is an index, we encapsulate the union of query terms into a *function*. Furthermore, we parameterize the query terms by the argument of the function.

We illustrate on our running example. Assume first that m1′ and m2 from Section 2.1 are the only available rules. The following two queries (written here in an abstract syntax that resembles the rules) are generated for Person and Employment.

```
Person := select [ name: i.name, cik: i.cik,
                   emp: EmploymentFn (cik: i.cik) ]
          from IRP i;

EmploymentFn :=
  fn (arg). select [ company: i.company,
                     positions: PositionsFn ([ cik: i.cik,
                                               company: i.company]) ]
            from IRP i
            where arg = [cik: i.cik] and i.isOfficer = true;
```

The first query is immediate and reflects directly the rule m1′. The main difference is that, to compute the value of emp, we use a function call that corresponds to the index lookup on Employment. The second query, for Employment, is the actual function, with a parameter arg that represents possible keys into the index. The function returns a non-empty set of values only for a finite set of keys, namely those that are given by the rule m2 (assuming, for now, that this is the only rule mapping into Employment). More concretely, if the parameter arg coincides with an actual key [cik: i.cik] that is asserted by the rule m2, then we return the set of all associated entries. Otherwise, we return the empty set. To achieve this behavior, the body of the function is a *parameterized* query term, whose where clause contains the equality between the argument and the actual key. Finally, and similarly to the query for Person, the positions field in the ouput employment record is computed via a call to a function that implements the Positions index. We omit that function definition here.

We now illustrate the case where multiple rules map into an entity and, hence, the expression defining the entity incorporates a union of query terms. If we consider the additional rule m5 for Employment, given in Section 2.3, the expression for EmploymentFn changes to the following function:

```
EmploymentFn :=
  fn (arg). select [ company: i.company,
                     positions: PositionsFn ([ cik: i.cik,
                                               company: i.company]) ]
            from IRP i
            where arg = [cik: i.cik] and i.isOfficer = true
        union
            select [ company: j.company
                     positions: PositionsFn ([ cik: l.cik,
                                          company: j.company]) ]
        from JobChange j, PeopleLink l
        where j.docid = l.docid and j.span = l.span
          and  arg = [cik:l.cik] and isOfficer (j.appointedAs) = true;
```

For a given parameter arg, there are now two query terms that can generate entries for the Employment index. The first query term is as before; the second query term, obtained from rule m_5, contains a similar condition requiring the equality between the parameter arg and the actual key [cik: l.cik].

As shown in these examples, during HIL compilation, we use an intermediate query syntax that is independent of a particular query language, although it is similar to an object-oriented or complex-value SQL. Translating from this syntax to a query language such as Jaql or XQuery is immediate. In our implementation and experiments, we use Jaql as our target execution language.

While the baseline algorithm gives rise to query expressions that map directly to the entity types and rules given in HIL, these query expressions can also be inefficient. In particular, indexes are not stored; an index lookup is computed, on the fly, by invoking the function associated with the index, which in turn executes the query terms inside the body. As a result, the query terms within a function will be executed many times during the evaluation of a HIL program. We next describe how to modify the baseline strategy to avoid such inefficiency.

3.1.2 Finite Maps as Binary Tables

For each HIL entity that is an index (or finite map), we generate a query that produces a binary table. This binary tables explicitly stores the *graph* of the finite map, that is, the set of all pairs of the form (k, v), where k is a key and v is the value associated with the key. Since v is typically a set (e.g., for each person cik, we have a set of employment records), the generated query consists of two parts. First, we generate a union of query terms that accumulates pairs of the form (k, e) where e is an individual value (e.g., a single employment record). Then, we apply a group by operation that collects all the entries for the same key into a single set.

To illustrate, instead of using a function for the Employment index, we can use the following query:

```
Employment := group by key
   ( select [ key: [cik: i.cik],
              val: [company: i.company,
                    positions: PositionsFn ([cik: i.cik,
                                             company: i.company]) ] ]
     from IRP i
     where i.isOfficer = true
   union
     select [ key: [cik: l.cik],
              val: [company: j.company
                    positions: PositionsFn ([ cik: l.cik,
                                              company: j.company]) ] ]
     from JobChange j, PeopleLink l
     where j.docid = l.docid and j.span = l.span
       and  isOfficer (j.appointedAs) = true);
```

The transformation from EmploymentFn to the actual query for Employment is not yet complete, since the Positions index is still accessed via a function call to PositionsFn. We will show how to change this shortly. The two inner query terms are similar to the ones in the earlier EmploymentFn; however, instead of being parameterized by the argument key, they explicitly output all the relevant (key, value) pairs. The outer group by is a nest type of operation that transforms set [key: t1, val: t2] into set [key: t1, val: set t2] and has the obvious semantics.

We now briefly describe how to modify the queries that refer to an index. For each reference to an index (earlier expressed via a function call), we will use a join with the binary table that materializes the index. Since an index is a finite map (i.e., it is defined for only a finite set of keys), the join must be an *outer* join, where the nullable part is with respect to the index that is being invoked. To illustrate, the earlier query for Person is replaced with:

```
Person := select [ name: i.name, cik: i.cik, emp: emptyIfNull (e.val) ]
          from IRP i left outer join Employment e
               on [cik: i.cik] = e.key;
```

In the above, the left outer join has a similar semantics to the SQL correspondent. Thus, the query always emits an output tuple for each entry in IRP. Furthermore, if there is a match with Employment, as specified by the on clause of the outer join, then e.val is non-null and becomes the output set of employment records. If there is no match, then e.val is null and we output the empty set for emp. The function emptyIfNull has the obvious meaning.

The actual query generation algorithm is more complex than hinted so far. Consider the query terms in the previous expression for Employment, which also require the addition of outer join, to access the binary table for Positions. In the case of the second query term, we cannot add an outer join directly, since the query term has multiple bindings in its from clause and also its own where clause. To account for such situation, we first construct a *closure* query that includes "everything" that the query term needs (except for the index lookup itself). This closure query is then outer-joined with the binary table representing the index. As an example, the closure query for the second query term in Employment is:

```
Emp_Cls_2 := select [ j: j, l: l ]
             from JobChange j, PeopleLink l
             where j.docid = l.docid and j.span = l.span
               and  isOfficer (j.appointedAs) = true;
```

The closure query has the same from and where clause as the original query term and returns a tuple with the values of the variables bound in the from clause. We use the names of the variables as the actual labels in the output record. The query term itself is then rewritten into an outer join as follows:

```
select [ key: [cik: x.l.cik],
         val: [company: x.j.company
               positions: emptyIfNull (p.val)] ]
from Emp_Cls_2 x left outer join Positions p
     on [cik: x.l.cik, company: x.j.company] = p.key;
```

Note that an expression such as l.cik in the original query term is automatically replaced by x.l.cik, where x is the variable bound to Emp_Cls_2, and where we use an extra projection to account for the variable that is needed (e.g., l).

3.2 Compilation of Entity Resolution Rules

We divide the query generation for entity resolution rules into two steps. The first step handles the where and check clauses. Since the effect of a check clause is local, i.e., it targets specific links that will be removed anyway, it is safe to apply it in conjunction with the matching predicates of the where clause. The group and cardinality clauses apply to groups of links, thus all links that belong to the same group need to be generated first before making a group-based decision on what links to drop. Therefore, these clauses are applied in the second step. Consequently, the queries generated by the first step construct candidate links between the input entities whereas the queries of the second step determine which links will be output. Our query generation algorithm aims at reducing the passes over the data and the amount of data passed from one query to the other.

Where and Check Clauses. While the semantics of an entity resolution rule is based on the cross-product of the inputs specified in the from clause, the query generation algorithm performs two optimizations to produce a more efficient query. First, we use each matching specified in the where clause to join and select entities from our inputs. Hence, we can re-write the where clause of er1 into the following query that generates a set of candidate links, PeopleLinkCand, that is the union of partial results from all matching predicates of the where clause:

```
  PeopleLinkCand:=
    select [p: p,  j: j,  emp: e,  provenance: 'match1']
    from Person p, JobChange j, p.emp e
    where e.company = j.company and compareName(p.name, j.name)
union
   select [p: p,  j: j,  emp: e,  provenance: 'match2']
   from Person p, JobChange j, p.emp e
   where normalize(p.name) = normalize(j.name);
```

The second optimization is rule enrichment by incorporating the conditions of the check clauses within each matching condition. A check clause has the form if p_k then c_k, which can be re-written as (not p_k or c_k). The check clause of er1 is then re-written as: null(j.bdate) or null(p.bdate) or j.bdate = p.bdate. Incorporating this constraint into the matching predicates leads to the following query:

```
  PeopleLinkCand':=
    select [p: p,  j: j,  emp: e,  provenance: 'match1']
    from Person p, JobChange j, p.emp e
    where e.company = j.company and compareName(p.name, j.name)
      and (null(j.bdate) or null(p.bdate) or j.bdate = p.bdate)
union
   select [p: p,  j: j,  emp: e,  provenance: 'match2']
   from Person p, JobChange j, p.emp e
   where normalize(p.name) = normalize(j.name)
     and (null(j.bdate) or null(p.bdate) or j.bdate = p.bdate) ;
```

While for simplicity the previous query outputs all entities (as well as a provenance attribute), the actual query will project on the attributes mentioned in the select clause of the entity resolution rule, and on any other the attributes used in the group and cardinality clauses. To achieve this, the algorithm performs a look-ahead and marks all attributes that need to be carried over.

Group and Cardinality Clauses. Each group and cardinality clause is re-written as a query. For example, the query for the group clause in rule er1 groups candidate links by the (docID, span) attributes and within each group checks if there are links that satisfy the condition p.name = j.name. Queries for group constraints are executed in the order specified in the entity resolution rule. The queries required for the cardinality constraints are executed last.

Cardinality clauses are more complex. For lack of space, we outline what happens if the cardinality constraint in rule er1 were 1:1 (checking for 1:N is similar but simpler):

```
                cardinality (docID, span) 1:1 (cik)
```

This clause requires mapping each pair (docID, span) to exactly one cik and vice versa. To enforce this constraint, we group links by their (docID, span) attributes and we count the number of distinct cik values within each group. Each group of links with more than one cik value is rejected as ambiguous. Then, we group the remaining links by cik and count the number of distinct (docID, span) pairs within each group. We again reject ambiguous groups of links. The remaining links comprise the final set of links that is output.

3.3 Integrated Compilation of HIL

As noted earlier, we do not allow recursion among entity population rules. The main reason for this is to avoid generation of recursive queries, which are not supported by the languages we target (e.g., Jaql or XQuery). In the absence of recursion, and provided that there are no entity resolution rules, the HIL compilation algorithm constructs a topological sort of all the entities in a HIL program; in this sort, there is a dependency edge from an entity E_1 to an entity E_2 if there is a rule mapping from E_1 to E_2. Queries are then generated, in a bottom-up fashion, from the leaves to the roots. The query generation algorithm for each entity E, which was already described, is based on all the rules that have E is as target.

However, when entity resolution rules are present, we do allow a limited form of recursion to take place. Often, in practice, entity resolution needs to use intermediate results in the integration flow, while the results of the entity resolution itself need to be used in the subsequent parts of the flow. Our running example, motivated from Midas, exhibits such behavior. The entity resolution performed in Step 2 of our flow (see Figure 1) makes use of the partial Person entities generated after Step 1. Subsequently, the fusion rules in Step 3 continue to populate into Person (and, in particular, their employment records), based on the result of entity resolution.

To achieve such behavior, we take the convention that entity resolution rules induce a staging of the overall program, where we force the evaluation of all the rules prior to a block of entity resolution rules. Thus, the order of the entity resolution rules in a HIL program becomes important. Concretely, for our example, the entity resolution small er1 in Step 2 requires the evaluation of all the entity population rules in Step 1 of the flow. To this end, we compile all the rules in Step 1 into a set P_1 of queries, using the compilation method for entity population rules. We then compile er1, using the method in Section 3.2, into a query P_2 that runs on top of the result of P_1 (and JobChange, which is source data). The PeopleLink table that results after P_2 is materialized and used as new source data into the next stage. This stage compiles together the union of all the entity population rules in *both* Steps 1 and 3, again using the compilation method for entity population rules. As an example, the query that is generated for Employment (see Section 3.1.2) incorporates rules from both Step 1 and Step 3. The resulting set P_3 of queries will produce the final data.

Note that, to achieve the fusion of the data produced by the rules in Step 3 with the data produced by the earlier rules in Step 1, we needed to recompile all these entity population rules together. In general, after the evaluation of a block of entity resolution rules, we compile and evaluate *all* the entity population rules (from the beginning of the HIL program) until the next block of entity resolution rules. Additional optimization is possible, in principle, where the materialized results from one stage (e.g., after P_1) are reused in the evaluation of the next stages (e.g., in P_3).

4. EVALUATION

In this section, we describe our experience in applying HIL to the financial integration scenario from SEC. We first give concrete details regarding the implementation and the execution of the three steps of the integration flow described in Figure 1, which we call the *Basic Person Integration Scenario*. We then add several other types of extracted data into the flow and we assess performance as the integration flow becomes increasingly more complex.

4.1 Basic Person Integration Scenario

The specification of the first step of the integration (illustrated in Figure 1) was along the lines described in Section 2.1, but included additional rules and entities to produce board memberships (in addition to employment), as well as rules to handle their provenance and temporal aggregation. The HIL code for this step comprised 6 target entity types, 6 HIL rules and 8 constraints.

The input IRP data consisted of $348,855$ records containing data from all the XML insider reports archived by SEC from 2005 to 2010. We compiled the HIL specification into Jaql, using the advanced compilation algorithm (with materialized binary tables), and ran it on a Hadoop cluster with 4 nodes (each an IBM System x3550, 2 CPUs, 32 GB). The total running time for the first step was 15 mins, and the result consisted of $32,816$ Person entities (15.77 MB of clean, aggregated person data).

The resulting Person entities were then used in the entity res-

Table 1: Characteristics of the data sources to integrate.

Data source	#Rcds	#Attr	#Links	#P-Rules	#ER-Rules
IRP (from XML)	348,855	9	no links	6	0
JobChange (from text)	1,077	7	697	5	2
Committee (from text)	63,297	10	50,355	3	2
Bios (from text)	23,195	9	29,822	5	2
Signatures (from text)	319,154	11	213,169	5	2

olution step of Figure 1. This step required two entity resolution rules, one as shown in Section 2.2 and a slight variation of that rule that exploited board membership history for people whenever available. The input JobChange data consisted of $1,077$ records extracted from text documents. The running time for the entity resolution was 3.7 mins, and the result consisted of 697 links.

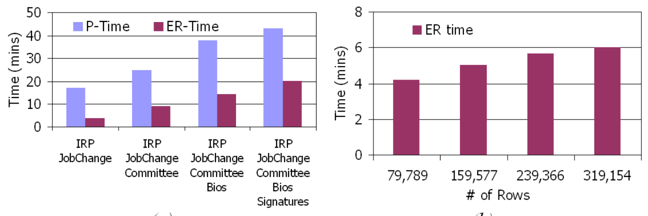
Finally, the third step included 5 more rules for fusion. These rules were along the lines described in Section 2.3 and covered the fusion of board membership data in addition to employment histories. There was no need to declare any new entities or constraints. These rules were compiled together with the rules of the first step (as discussed in Section 3.3) and applied to take into account the links generated by entity resolution. The resulting Jaql code ran in 17 mins, where the new running time includes the joins between JobChange and the links, in addition to the processing of IRP records and fusion of the results. We obtained the same number of Person entities as after the first step but each entity is now more complete with data aggregated from both IRP and JobChange.

We note that HIL is compiled to optimized Jaql code that, as expected, is more complex than the HIL specification itself. For example, the entity resolution rules in HIL are 2.32 KB on disk compared to the 12.54 KB of the corresponding compiled code. Additionally, the rules are more readable, since they give a clear and succint indication of what links are created, what fields they must contain and from what sources they were created. As another example, the fusion rules in the third step of this scenario were relatively simple in HIL. In contrast, the compiled code that fused the new rules with the rules in the initial mapping step is complex, as it required staging of the process based on the dependencies between entities, as well as many steps such as the materialization of the indexes, the use of outer joins, grouping, duplicate elimination, etc. Note that this complexity is not due to Jaql but to the inherent complexity of the problem and the data. A language such as XQuery, which is similar in many ways to Jaql, would also require complex structuring operations. In the HIL framework, all these low-level operations are hidden from the programmer and automatically handled via compilation that aims at generating optimized code.

4.2 Scaling up Integration

We scale up the Basic Person Integration Scenario by adding several new types of extracted data. Each type of extracted data acts as a new data source, since it has its own format (schema), and its own set of records. Before the actual fusion, each data source is first linked to the initial Person entities created in Step 1 of the basic integration scenario, in the same way JobChange was linked before fusion. The characteristics of the data sources (together with the previous ones, IRP and JobChange) are summarized in Table 1. For each source, we give a count of the records, the relevant attributes (i.e., that are actually used in HIL rules), and of the links that are created by entity resolution. We also list the number of rules that each new data source requires. We distinguish between entity population rules, P-rules, and entity resolution rules, ER-rules.

Figure 3(a) shows the overall performance of the HIL-generated code with increasing number of data sources. We have described in Section 4.1 the initial computation of Person entities from IRP, as well as the addition of JobChange, which is represented by the first data point in Figure 3(a). With each increasing number of



(a) *(b)*

Figure 3: Integration times in the Financial Scenario

data sources, we include: (1) the total time, ER-Time, to generate the links between the new data sources and the Person entities (as generated in Step 1 from IRP), and (2) and the total time, P-Time, to fuse all the data (obtained by re-compiling and running all the entity population rules for all the data sources so far).

The second data point in Figure 3(a) corresponds to adding Committee into the flow. The entity resolution time (9.3 mins) includes now the previous entity resolution time (for JobChange) and the additional time to link Committee. P-Time (25 mins) accounts for running all HIL rules for fusing all three data sources together. It is interesting to observe the third data point, which corresponds to adding Bios into the flow. The cumulative entity resolution time, including now the time to link Bios, is 14.3 mins. The fusion time for all four data sources is 38 mins. Even though the number of records in Bios is not that large $(23,195)$, the integration flow processes now a lot more textual information (the biographies). This is also reflected in the fact that the resulting set of Person entities is significantly larger now, in size, than for the previous data point (43.3 MB vs. 19.2MB). Finally, the fourth data point corresponds to adding Signatures into the flow. These records are extracted from a special signature section of a certain type of input documents, and give additional information about key people and their employment (that may have been missed by the other types of extracted data). The cumulative entity resolution time, including now the time to link Signatures, is 20.3 mins while the fusion time for all five data sources is 43 mins. Overall, we observe that execution times scale smoothly as more data sources are added.

An additional experiment focuses specifically on entity resolution and its performance with respect to the number of entities that need to be resolved. In this experiment, we keep the number of Person entities constant $(32,816)$ and modify the number of Signatures that need to be resolved (from 1/4 of the initial Signatures file to 4/4 of the file). Figure 3(b) shows that execution times increases smoothly with the input size. In fact, even though the data size for Signatures doubles with each point, the increase in entity resolution time is at a much lower rate. This is credited to the ability to prune undesired links *early* during execution, by applying constraints to the link generation step, as well as by removing duplicate links when merging partial results. Both techniques help reduce the size of intermediate results.

4.3 Further Remarks

It is worth noting that the integration of entities from SEC forms was, in general, a complex process. Due to space limits, we have described only a core part of the SEC integration of entities. The full-fledged integration also included rules for generating company entities, investment entities, relationships (including lender/co-lender and parent-subsidiary types of relationships), as well as many user-defined functions for cleansing (of people names, company names, etc.) and for conflict resolution (see also Section 5). We have also left out rules for aggregating the stock transactions made by executives or directors of companies in various years, as well as temporal analysis rules to automatically determine how much such insiders have been holding in company stocks at any given time. All of these rules were expressed in HIL and used various indexes to access the transaction and holding information by various dimensions (company, person, year, type of security, type of ownership, etc).

To give an indication of the complexity of the SEC integration scenario, we used a total of 71 HIL rules, populating and linking 34 entity types, with the help of 21 UDFs, all split into 11 scripts. The UDFs ranged from very simple ones such as strToUpperCase, isNull, sortReverseByDate to more complex ones such as normalizeCompanyName and xmlToJson (a function to convert an XML document to JSON format). The size of the generated Jaql code is about 100 KB on disk, and the entire flow runs approximately 80 mins. The full-fledged HIL-based integration of entities from SEC documents was used to populate a commercial[3] master data management (MDM) system, and demonstrated to various financial analysts and regulators (including SEC itself). One of our immediate directions for experimentation with HIL is its application to other domains such as DBPedia, bibliographic sources (DBLP, Google Scholar, CiteSeer), or social media (Twitter, MySpace, blogs).

Finally, we note that we have left out any experiments on precision and recall from our evaluation. First, for the SEC domain, there is no golden standard for what a set of correct or complete entities means. In our case, we tested the results of the HIL integration by manually inspecting all the important company entities (e.g., “Bank of America", “Citigroup"), together with their people entities and relationships. Furthermore, the primary goal for HIL is to provide the *framework* for developing the integration rules, and not the rules themselves. In the end, it is up to the developers to improve the precision and recall by adding more rules, more matching clauses in an entity resolution rule, more cases in a normalization function, or by using a better string comparison function. Tools for helping to debug and refine HIL specifications are very important, but outside of the scope of this paper.

5. ADDITIONAL PATTERNS IN HIL

In this section, we describe additional functionality that one can express in HIL. We focus here on a few patterns that are common or important in the complex data integration scenarios we target.

Conflict Resolution. We have mentioned in Section 2 that user-defined functions can be used to cleanse and normalize the individual values that appear in a source attribute. In our example, titles of executives were normalized via a user-defined function that is written outside of HIL (i.e., in Java). Normalization is typically done before mapping to a target attribute or before entity resolution. A slightly different type of operation that is also very common and must involve user-defined functions is *conflict resolution*. Such operation is needed when the integration process yields multiple (conflicting or overlapping) values for an attribute that is required to be single-valued, if certain functional dependencies must hold.

To illustrate, consider the earlier rule m1 in Section 2.1. If a person with a given cik appears under different names in the data sources, then the resulting set of Person entities will contain duplicate entries (each with a different name) for the same cik. To avoid such duplication, the typical solution in HIL is to maintain a separate index, call it Aliases, which collects all the variations of a person's name across all known inputs. For example, the following rule collects aliases for each person cik that appears in IRP:

```
insert into Aliases![cik: i.cik]
select [name: i.name]
from IRP i;
```

Additional rules must be added to further populate the Aliases index from the other data sources (e.g., from JobChange). Furthermore, the initial rule m1 for Person must be modified so that a unique name is selected from the list of aliases for a person:

```
insert into Person
select [ name: chooseName (Aliases![cik: i.cik]),
         cik: i.cik ]
from IRP i;
```

The actual selection is done through an UDF that customizes the desired semantics. This process becomes more sophisticated if further attributes, such as the provenance of each alias, are also maintained in the index and then used in the selection function.

Blocking and Score-based Entity Resolution. We now briefly illustrate two other important aspects of entity resolution that we can express in HIL: blocking and score-based matching. Blocking is a common mechanism used in entity resolution for reducing the number of comparisons among input entities according to some criteria. Score-based matching, on the other hand, allows matching decisions to be made based on a score assigned to each pair of entities. This flavor of entity resolution is widely used in practice and appears in several commercial systems. The following example shows how both mechanisms can be expressed in a HIL rule that matches a Person list with a Customer list.

```
rule er2:  create link CustomerLink as
           select [c_cik: c.cik,  p_cik: p.cik]
           from Customer c, Person p
           block c on [zip: c.zip]
           block p on [zip: p.zip]
           where match1: jaccard(c.lastname, p.lastname),
                 match2: jaccard(c.streetname, p.streetname),
                 match3: jaccard(c.strnumber, p.strnumber)
           check avg(score) >0.8;
```

The blocking criteria are specified in a HIL entity resolution rule using the block clause. Rule er2 contains one blocking clause for each input set of entities and specifies that Person and Customer entities will only be compared when their zip code values are the same. In general, blocking clauses can contain any HIL expression as long as their values are type-compatible.

There are many score-based similarity functions that we can use here as UDFs for computing how similar two records are (e.g., distance-based, feature-based, and probabilistic similarity measures [7]). For example, rule er2 makes use of Jaccard similarity on the last name, street address, and street number of Person and Customer. The score computed by each matching (the result of the Jaccard UDF) is then aggregated (averaged) in the check clause and compared to a threshold value. Notice that since these three scoring clauses appear in the where clause, they will be applied to all pairs of Person and Customer that satisfy the blocking condition.

As it can be seen, the main difference from the previous style of entity resolution captured in HIL (see earlier rule er1) is that now the predicates in the where clause return a non-negative score rather than true or false. Furthermore, for each candidate link, the scores associated with the predicates are combined by using numerical aggregation operators (rather than by using the boolean or). We leave additional details for an extended version of this paper.

6. RELATED WORK

Our HIL framework bridges two lines of data integration research: schema mapping [26] and entity resolution [16]. We have already summarized the high-level relationships between these two areas and HIL. We give here a few more details.

Although we draw inspiration from schema mapping formalisms (e.g., s-t tgds) [18] or SO tgds [19]), the mapping rules in HIL are designed to facilitate direct programming by a user. In addition to being able to abstract away unnecessary schema parts, HIL rules, by design, can map only into one entity (set or index) at a time, as

[3]We omit the name of the vendor for anonymity reasons.

opposed to s-t tgds which allow for a complex existentially quantified formula. Associations between different entities are explicitly given in HIL via indexes, which in effect replaces the need for Skolem functions as used in nested mappings [21]. Technically, HIL indexes are similar to the dictionaries that appear in the query optimization framework of [13], but they capture important logical steps in a data integration flow and not physical database structures. Furthermore, HIL indexes play a prominent role in summarization and fusion of data. Regarding entity resolution, Ajax [22] is another (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.

Model management [29] provides a high-level scripting framework, but operates mostly at the metadata (schema) level. Perhaps closer to HIL is iFuice [32], which combines mapping with fusion of data. However, iFuice has no compilation components, has no entity resolution (it assumes instead that the links are given), and fusion is focused on attributes (whereas fusion in HIL applies, more generally, in a hierarchy of entities and is driven by indexes).

Finally, there are several query/transformation languages with complex data processing capabilities (but not focused on data integration), including XQuery, XSLT, Jaql, Pig Latin, and the query language in ASTERIX [3], which, like HIL, also uses open types. Such languages are possible target languages for HIL compilation.

7. CONCLUSION

In this paper, we introduced HIL, a high-level scripting language for entity integration flows, we gave algorithms for compilation into runtime queries, and showed how we applied HIL for integration in the financial domain. As part of our future directions, we are exploring applications of the HIL framework to entity integration in other domains. Another future direction is to explore incremental compilation and evaluation algorithms, where the existing target data is incrementally modified when new data sources and rules are added. Finally, the use of a high-level language opens up many possibilities in the space of reasoning, debugging, and automatic maintenance (evolution) of the integration flow.

6. REFERENCES

[1] A. Arasu, C. Re, and D. Suciu. Large-Scale Deduplication with Constraints using Dedupalog. In *ICDE*, pages 952–963, 2009.

[2] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *JACM*, 31(4):718–741, 1984.

[3] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-World Models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[4] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In *VLDB*, 2011.

[5] M. Bilenko and R. J. Mooney. Adaptive Duplicate Detection using Learnable String Similarity Measures. In *KDD*, pages 39–48, 2003.

[6] J. Bleiholder and F. Naumann. Data Fusion. *ACM Comput. Surv.*, 41(1), 2008.

[7] S. Boriah, V. Chandola, and V. Kumar. Similarity Measures for Categorical Data: A Comparative Evaluation. In *SIAM*, 2008.

[8] D. Burdick, M. A. Hernández, H. Ho, G. Koutrika, R. Krishnamurthy, L. Popa, I. R. Stanoi, S. Vaithyanathan, and S. Das. Extracting, Linking and Integrating Data from Public Sources: A Financial Case Study. *IEEE Data Eng. Bull.*, 34(3):60–67, 2011.

[9] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1:379–474, April 2009.

[10] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An Algebraic Approach to Declarative Information Extraction. In *ACL*, pages 128–137, 2010.

[11] W. W. Cohen. Integration of Heterogeneous Databases without Common Domains Using Queries Based on Textual Similarity. In *SIGMOD*, pages 201–212, 1998.

[12] N. N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu. A Web of Concepts. In *PODS*, pages 1–12, 2009.

[13] A. Deutsch, L. Popa, and V. Tannen. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *VLDB*, pages 459–470, 1999.

[14] A. Doan, J. F. Naughton, R. Ramakrishnan, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. J. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong. Information Extraction Challenges in Managing Unstructured Data. *SIGMOD Record*, 37(4):14–20, 2008.

[15] M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios. Tailor: A Record Linkage Tool Box. In *ICDE*, pages 17–28, 2002.

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

[17] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236. Springer, 2009.

[18] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[19] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Composing Schema Mappings: Second-order Dependencies to the Rescue. *ACM TODS*, 30(4):994–1055, 2005.

[20] I. P. Fellegi and A. B. Sunter. A Theory for Record Linkage. *J. Am. Statistical Assoc.*, 64(328):1183–1210, 1969.

[21] A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.

[22] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *VLDB*, pages 371–380, 2001.

[23] S. Guo, X. Dong, D. Srivastava, and R. Zajac. Record Linkage with Uniqueness Constraints and Erroneous Values. *PVLDB*, 3(1):417–428, 2010.

[24] O. Hassanzadeh, A. Kementsietsidis, L. Lim, R. J. Miller, and M. Wang. A Framework for Semantic Link Discovery over Relational Data. In *CIKM*, pages 1027–1036, 2009.

[25] M. A. Jaro. Unimatch: A Record Linkage System: Users Manual. In *US Bureau of the Census*, 1976.

[26] P. G. Kolaitis. Schema Mappings, Data Exchange, and Metadata Management. In *PODS*, pages 61–75, 2005.

[27] N. Koudas, A. Marathe, and D. Srivastava. Spider: Flexible Matching in Databases. In *SIGMOD*, pages 876–878, 2005.

[28] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[29] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD*, pages 193–204, 2003.

[30] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[31] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.

[32] E. Rahm, A. Thor, D. Aumueller, H. H. Do, N. Golovin, and T. Kirsten. iFuice - Information Fusion utilizing Instance Correspondences and Peer Mappings. In *WebDB*, pages 7–12, 2005.

[33] V. Rastogi, N. N. Dalvi, and M. N. Garofalakis. Large-Scale Collective Entity Matching. *PVLDB*, 4(4):208–218, 2011.