HIL: A High-Level Scripting Language for Entity Integration

Mauricio Hernández
IBM Research – Almaden
mahernan@us.ibm.com

Georgia Koutrika
HP Labs
koutrika@hp.com

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

Lucian Popa
IBM Research – Almaden
lpopa@us.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 clean, unified entities. Such flows typically include many stages of 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. A salient feature of HIL is the use of logical indexes in its data model to facilitate the modular construction and aggregation of complex entities. Another feature is the presence of a flexible, open type system that allows HIL to handle input data that is irregular, sparse or partially known.

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 can execute in parallel on Hadoop. We show how our framework is applied to real-world integration of entities in the financial domain, based on public filings archived by the U.S. Securities and Exchange Commission (SEC). Furthermore, we apply HIL on a larger-scale scenario that performs fusion of data from hundreds of millions of Twitter messages into tens of millions of structured entities.

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), movie databases (IMDB), knowledge bases (Wikipedia, DBPedia, Freebase), social media data (Twitter, blogs). Additionally, a number of specialized public repositories are starting to play an increasingly important role. These repositories include, for example, 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 [5, 7], where the important entities and their relationships are extracted and integrated from the underlying documents. We refer to the process of extracting data from documents, integrating the information, and then building domain-specific entities, as *entity integration*. Enabling such integration in practice is a challenge, and there is a great 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* [9], *schema matching* [20], *schema mapping* [11], *entity resolution* [10], *data fusion* [3]. 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 the identifying attributes of entities (hence, it requires entity resolution, that is, the identification of the same real-world entity across different records). Moreover, fusion 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 [6]) 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., Jaql [2], Pig Latin [18], XQuery).

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 [19], 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 declare properties on the desired output (e.g., one-to-one or one-to-many types of matches).

The entity population rules borrow features from schema mapping formalisms (e.g., s-t tgds [12] or second-order tgds [13]); however, our design is aimed at making the language intuitive, efficient and easy to use by the practitioners. 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, indexes are perhaps the most important differentiating aspect of HIL, as they allow the decorrelation of complex integration tasks into simple and efficient rules. It is the presence of indexes that enables the true scalability of HIL in terms of both the design (e.g., creating and maintaining the rules) and execution (e.g., on large amounts of data such as in social media).

With regard to entity resolution, HIL is not a replacement of the many physical algorithms or operators for matching of entities that have been developed in the literature over the years. Instead, HIL entity resolution rules are declarative statements that can express the common patterns used in practice (e.g., blocking, matching, constraints), while making use of specialized functions for string similarity, record matching, or for normalization of values. Furthermore, entity resolution rules in HIL are used in conjunction with the entity population rules in order to achieve fusion of the matched or linked entities into the final, integrated entities.

Our motivating example, which we introduce next, will show how the combination of entity resolution rules together with rules for mapping, fusion and aggregation of data gives HIL enough expressive power to achieve complex, end-to-end integration tasks. At the same time, we will use the example to show the relative ease and simplicity of HIL when compared to the alternatives.

1.1 A Motivating Example

As a motivating scenario, we use the financial integration from SEC that was described as part of the Midas system [5]. A different scenario that integrates entities from social media will be described and used for a larger-scale experimental evaluation in Section 6.

In the SEC 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 [6] 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 [2], a general data manipulation language for JSON, which in turn compiles into Map/Reduce jobs on Hadoop. As noted in [5], the integration process required a high cost of development and maintenance; the

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

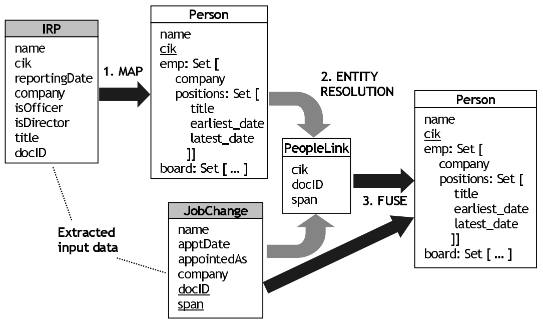

Figure 1: Example Integration Flow

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 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 functionality of HIL, and also to later evaluate HIL. 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 textual 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 in JobChange includes the extracted person name, the appointment date, the position (appointedAs), and 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 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. This 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 varies 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 aggrega-

```
1.     Person =
2.       join preserve Person, PeopleLink, JobChange
3.       where Person.cik == PeopleLink.cik
4.          and PeopleLink.docID == JobChange.docID
5.          and PeopleLink.span == JobChange.span
6.          and isOfficer (JobChange.appointedAs)
7.       into { name: Person.name,
8.               cik: Person.cik,
9.               emp: Person.emp,
10.              delta_emp:
11.                { company: JobChange.company,
12.                  title: normTitle (JobChange.appointedAs),
13.                  date: JobChange.apptDate }
14.            }
15.      → group by c = $.cik into {
16.          // copy name, cik, previous emp
17.          name: $[0].name,
18.          cik: c,
19.          emp: $[0].emp,
20.          // process new emp items
21.          delta_emp:
22.            ( $[*].delta_emp
23.              → group by comp = $.company into
24.                { company: comp,
25.                  positions:
26.                    ($ → transform { $.title,
27.                                     $.date })})
28.        }
29.      → transform { // copy name, cik, merge employments
30.          name: $.name,
31.          cik: $.cik,
32.          emp:
33.            ([$.emp, $.delta_emp]
34.             → expand // union of the two employment arrays
35.             → group by comp = $.company into
36.               { company: comp,
37.                 positions: ( [$[0].positions, $[1].positions]
38.                              → expand  // another deep union
39.                              ... // further processing
```

Figure 2: Snippet of fusion code in Jaql.

tion 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).

1.2 Advantages of HIL

We now discuss and contrast two approaches to express the above type of functionality. The first approach is based on general scripting languages for data manipulation that are currently available. We use Jaqll [2] here for illustration, but the same comparison applies for languages such as Pig Latin [18] or XQuery. The second approach, which we take in this paper, is based on HIL. We illustrate our discussion using the fusion step of our example.

Figure 2 gives a snippet of the Jaql code needed to fuse the new employment facts derived from our second data set, JobChange, into Person. We do not assume the reader's familiarity with Jaql syntax here, but assume some understanding of the basic operations to manipulate data (e.g., join, record projection, group by, etc.). Jaql, like many other scripting languages, is a functional language with no direct facilities for updates or fusion of data. Thus, the typical code that one writes takes as input a previous copy of the data (e.g., Person), and includes several stages of data processing, where the result of each stage is pipelined (using → in Jaql) to the next stage. The final stage produces a new version of Person.

```
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);


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);
```

Figure 3: Fusion via HIL rules.

In more details, the first stage (lines 2 - 14) has the role of joining the existing Person data with "delta" employment facts from JobChange. This stage exploits the links (PeopleLink) that were established by entity resolution, and extends the existing Person records to include a new delta_emp field with data from the corresponding JobChange record. The keyword preserve in front of Person ensures that all existing Person records are in the output, irregardless of whether there is a matching JobChange record. (This is similar to an outer-join.) Since the first stage breaks the normalization of the data (i.e., there may be multiple records per person, one for each delta_emp fact), the second stage (lines 15 - 28) groups first by cik to reconstruct Person entities. As a result of this operation, all the delta_emp facts for a person cik are accumulated together. These records are then processed in lines 22-27 into a hierarchical structure (i.e., grouped by company). We are now ready to go into the next stage, and merge the two employment sets (emp and delta_emp for each person (lines 33 and beyond). The merging process involves a deep union of hierarchical sets, to merge first the records at the company level, and then at the positions level.

As one can see from the example, the code is relatively low-level, with many physical operations involved. Since data needs to be carried around between stages, there is repeated copying of data, as well as grouping and renesting of the data. Fusion is accomplished by using union (which is often deeply nested and requires appropriate data alignment). As the code evolves and becomes more complex (i.e., while fusing more sources), the higher-level logic of the integration task is lost in the implementation details.

In contrast to the lower-level approach, in HIL we can use just two entity population rules (shown in Figure 3) to capture the logic of the fusion in this example, without going through any of the low-level physical operations (e.g., copy the data, group and regroup, deep-union, etc.). We give below a few high-level details on these rules, while leaving the full discussion on the HIL language (including the additional rules for the SEC example) for Section 2.

Each HIL rule inserts data into a single entity, and these entities are part of the logical model that governs the integration task. In our scenario, Employment and Positions are entities, and they are used to index two types of information: employment (of a given person), and positions (of a given person and company). There is another top-level entity, Person (to be shown later), but the above fusion rules need not touch that; they need only insert the new employment facts and positions.

A HIL rule states where the data comes from, where it goes into (i.e., the target entity, and key if it is an index), and the actual construction of the tuples being inserted. In our example, the first rule joins JobChange with PeopleLink to retrieve the relevant cik, which is then used as a key to access the Employment index (via the ! operation) and insert new tuples. The rule relies on the Positions index, which is populated by a separate rule. In general, other rules may further populate into the same indexes as well as combine these indexes to form other entities (e.g., as we see later, Employment is a building block towards the main Person entity). The HIL compiler will take all the specified rules and evaluate them in the right order.

HIL rules are naturally decorrelated (via indexes), which, in turn, enables the scalability of the design. Each rule is an independent unit that captures a particular piece of the integration logic and that can be understood in isolation. For example, when writing the rule to insert new records into Employment based on JobChange and PeopleLink, one need not worry about other rules for the top-level Person entity, or for Positions, or for any other indexes that may participate in the final construction of a Person entity. Furthermore, the decorrelation of the integration task via HIL indexes plays an important role for the scalability of the runtime itself, as will be demonstrated later by our optimization techniques and by the experiments on social media data.

1.3 Contributions and Paper Outline

The main contributions of this paper are as follows:

- We give, in Section 2, a high-level language (HIL) that covers the major information integration components (mapping, fusion, entity resolution). HIL naturally decorrelates a complex integration task into multiple high-level rules, which offers both increased usability and potential for better optimization.
- We give, in Sections 3 and 4, algorithms to compile a declarative HIL program into lower-level, efficient operations.
    - We exploit general (i.e., platform-independent) optimization techniques that implement logical indexes as binary tables, and map large fragments of HIL rules (both entity population and entity resolution) into joins.
    - We also develop optimization techniques that are targeted towards Hadoop (Map/Reduce). In particular, we use a co-group technique to fuse multiple HIL indexes in parallel and access them as one, thus facilitating the fusion of complexly structured entities from large amounts of data.
- We evaluate the resulting framework, in Section 5, by first applying it to the SEC financial integration scenario.
- We further evaluate, in Section 6, the performance of HIL on a large-scale integration scenario that fuses data from hundreds of millions of Twitter posts into tens of millions of entities.

We note that HIL is extensively used within IBM for entity integration applications that are targeted towards public unstructured data (the SEC data and social media being two primary examples)
.

2. SCRIPTING IN HIL

In this section, we give an example-driven overview of the HIL language. The main ingredients of HIL are: (1) *entities*, defining the logical objects (including the 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 additional data structures (e.g., indexes). We also express now a first rule to populate 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 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 more fields that are 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 dynamically refines 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 entity 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 corresponding attributes of Person. 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 produced by 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. This is consistent with the usual open-world assumption in data integration [16]. 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 executable code that produces the final 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 via an index, which is populated separately. 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 Employment 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 ?];
```

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 Employment is constructed separately and here we simply access its entries.

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, indexed by person cik and company. The other notable thing about m2 is that it populates an index. For each record i in IRP where isOfficer is true, we insert a tuple in the Employment index entry that is associated with the key i.cik. Different entries in Employment, corresponding to different cik values, may be touched. Also, multiple tuples may be inserted in the same Employment entry, corresponding to multiple input records with the same cik 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 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. 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 its Jaql implementation.

```
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)])
```

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 a set of *links* between these entities. Each link represents a semantic association between the entities it refers to. 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 to be linked. 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. Additional clauses, including check, group on and cardinality, 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.

In our example (Figure 1), we match Person entities with JobChange records using a combination of person name and employment history. If the name of the company that filed the job change already appears in the person's employment history, then we use both company and person name for matching. Otherwise, we perform a strong similarity match only on person name. In both cases,

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

we do not want to create a match if different birthdates appear in the inputs. Furthermore, in this particular entity resolution task, one Person entity can match multiple JobChange records. However, multiple Person entities *cannot* match the *same* JobChange record. 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 a specialized UDF compareName. 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.

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 example, we could use the link provenance to select links that are created by the stronger matching predicates. For example, if a JobChange matches several Person entities, then we can give priority to the links created by match1 if they exist.

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 link to exactly one Person entity (but that Person entity can link 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 can complete now our HIL use case on the SEC scenario with the fusion step (Step 3 in Figure 1). We have already outlined in Section 1.2 the two main entity population rules (m5 and m6) that fuse the new data from JobChange into the employment and position indexes of a person. We give here a few more details.

The rules m5 and m6 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 person cik 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 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 are 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 with the new data. Thus, we need another rule (not shown here) that is similar to the earlier rule m4 except that it uses JobChange and PeopleLink instead of IRP. The actual specification for earliest_date, which must be incorporated in rule m6, remains the same: the equation (*) and the discussion at the end of Section 2.1 apply here too, with the difference that the minDate aggregation now works on a larger set.

Note that we did not need to modify the main rule m1′ for Person. Also, we did not need any new entities or indexes. The new rules simply assert new data into the same indexes declared by the initial mapping phase. This pattern typically applies when fusing any new data source: first, write entity resolution rules to link the new source to the existing target data, then write entity population rules to fuse the new data into the target entities (indexes).

2.4 Other HIL Features

For space reasons, we omit the syntax of HIL here and instead refer the reader to an earlier IBM research report [15]. We include next a discussion of a few other patterns, also expressible in HIL, which are common in complex data integration scenarios.

We have mentioned that user-defined functions can be used to cleanse and normalize the individual values that appear in a source attribute. A slightly different operation that is also 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. Rules must be added to explicitly populate the Aliases index, from the given data sources. Furthermore, the rule m1 for Person must be modified so that a unique name is selected, possibly via a user-defined function, from the list of aliases. 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.

Finally, we mention two aspects related to entity resolution that can also be expressed in HIL: blocking and score-based matching. Blocking is a common mechanism that is used to reduce the number of comparisons among input entities, by partitioning them according to some criteria (called blocking criteria or keys). Score-based matching, on the other hand, allows matching decisions to be made based on scores assigned to pairs of entities. This flavor of entity resolution is widely used in practice and appears in several commercial systems. A score-based entity resolution rule in HIL uses matching conditions in the where clause that are based on UDFs that compute the similarity of two records (e.g., based on distance, on features, or based on probabilistic similarity measures [4]). The scores computed by the matching conditions can then be used in the check clause (e.g., averaged and compared to a threshold value).

3. COMPILATION: ENTITY POPULATION

We now describe the compilation of HIL entity population rules into efficient runtime queries. We will discuss entity resolution rules in the next section, where we give the overall HIL compilation procedure that takes into account the recursion that may exist between entity population and entity resolution rules.

The naive semantics of entity population rules is to identify all the applicable rules, that is, rules which generate new facts, and to insert all the new facts into the target entities (either sets or indexes). This process repeats until no new facts are generated. To avoid such iterative and inefficient process, we use compilation (or query generation) to implement the semantics. The main assumption that 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.

We break query generation into several steps. In the first step, the *baseline for HIL compilation*, indexes are implemented as functions and index lookups as function calls. In a 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. A final optimization step, which is targeted at Hadoop, identifies multiple indexes that can be coalesced into *super-indexes*, which can then be accessed in a single join operation from the parent entity. Other Hadoop-oriented optimizations that are implemented in our compiler include pipelining of intermediate results whenever possible, and delaying duplicate elimination[3] until needed. Both optimizations allow the fusion of multiple map jobs into a single job, on a Map/Reduce platform.

3.1 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

[3]As already mentioned, HIL is based on set semantics.

actual key. Similarly to the query for Person, the positions field in the ouput employment record is computed via a call to a function (not shown here) that implements Positions.

In the case when multiple rules map into an entity, the expression defining that entity incorporates a union of query terms. For our example, if we consider the additional rule m5 for Employment, the expression for EmploymentFn changes to:

```
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 HIL entity types and rules, 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 modify next the baseline strategy to avoid such inefficiency.

3.2 Finite Maps as Binary Tables

For each HIL entity that is an index (finite map), we generate a query that produces a binary table representing the *graph* of the finite map, that is, the set of all pairs (k, v), where k is a key and v is the value for that 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 queries to accumulate 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 to collect 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. 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 an operation that transforms set [key: t1, val: t2] into set [key: t1, val: set t2] with 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 procedure for replacing index lookup operations with joins is slightly more involved, since it needs to account for the case when a query term has multiple bindings in its from clause and also has its own where clause. In 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.

3.3 Index Fusion with Co-Group

The final step in the compilation of entity population rules is an optimization that is targeted at the Map/Reduce (or Hadoop) platform. This optimization makes use of a *co-group* operation that is present in Jaql, and also in PigLatin, and has the ability to group in parallel multiple input sets by the same key. Applying this optimization has a significant impact when an entity needs to aggregate data from many indexes that share the same key. Rather than generating a sequence of binary joins between the parent entity and each of the indexes, we first generate a query to fuse all the contributing indexes into a single *super-index* using the same key. This super-index is then joined, via a single left-outer join operation, in the query for the parent entity.

As an example, assume that in addition to the earlier Employment index, we now have a few more indexes that accumulate further information about a person: Board (the board membership history of a person), Transactions (the most recent stock transactions by a person), and possibly others. These indexes use the same key (the cik of a person) to map to the corresponding entries for a given person. By using the outer join strategy outlined in the previous subsection, the main query for Person needs to include a sequence of left outer joins, one for each index that needs to be accessed. Instead, using the co-group strategy, we perform an index fusion operation to merge such indexes into one super-index. Index fusion is implemented as a single operation of the form[4]:

```
FusedIndex := cogroup Employment by x = Employment.key,
                      Board by x = Board.key,
                      Transactions by x= Transactions.key
```

[4]The actual Jaql syntax is slightly different, but the idea is the same.

```
into [ key: x,
       Emp_value: Employment.value,
       Board_value: Board.value,
       Transactions_value: Transactions.value];
```

Intuitively, all the participating indexes are partitioned based on their keys; then, for each key value (x), we stitch together the entries from all the participating indexes that actually have something for that key (or put null otherwise). The main entity for Person can then be obtained via a single outer join with FusedIndex that retrieves in one shot the employment, board, and transaction entries for a person. The implementation of this strategy requires three main steps: detection of when multiple indexes can be fused together, generation of the co-group queries that materialize fused indexes, and rewriting of the queries that can benefit from fused indexes.

While conceptually simple, index fusion can be very beneficial, especially when the number of indexes that can contribute to an entity becomes large. As part of our experiments, we will show in Section 6 how this optimization greatly improves the run-time performance of fusion of person entities from social media (Twitter).

4. COMPILATION: ENTITY RESOLUTION

We first give highlights of the compilation of entity resolution rules in isolation, and then discuss the integrated compilation of both types of rules (entity population and entity resolution).

4.1 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 *individual* links, 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 a group need to be generated before making a group-based decision on what links to drop. Therefore, these clauses are applied in the second step.

Where and Check Clauses. While the semantics of an entity resolution rule is based on the cross-product of the inputs in the from clause, the compilation algorithm performs two optimizations to produce a more efficient query. First, we use the matching conditions in the where clause to join and select entities from our inputs. Concretely, based on the where clause of er1, we generate the following query for candidate links, corresponding to the union of partial results from each of the matching predicates in er1:

```
  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);
```

In the same spirit, *blocking* conditions (see also earlier Section 2.4) are also pushed, whenever present, as join conditions in the where clauses of the above query terms.

The second optimization incorporates 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). As an example, the check clause of er1 is re-written as null(j.bdate) or null(p.bdate) or j.bdate = p.bdate, and then added as a conjunct in the where clause of both query terms in the above union.

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. We outline here what would happen 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.

4.2 Integrated HIL Compilation

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 4.1, 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 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 3.2) 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

Table 1: Characteristics of the SEC data 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	694	5	2
Committee (from text)	63,297	10	50,533	3	2
Bios (from text)	23,195	9	19,750	5	2
Signatures (from text)	319,154	11	215,848	5	2

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).

5. EVALUATION: THE SEC SCENARIO

In this section, we describe our experience in applying HIL to the financial integration scenario from SEC. The goal here is to test less the scalability in terms of the data size, but rather the specification and execution of a complex scenario (in terms of the rules involved and the types of data), where the input data size is constrained by the domain itself (i.e., the number of relevant documents in SEC). In Section 6, we perform a different experiment to test the scalability of the system, in a parallel setting using a Map/Reduce cluster, and where the data scales to hundreds of millions of documents.

We start by describing the implementation and the execution of the three steps of the SEC integration flow 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.

5.1 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 and 6 entity population rules.

The input IRP data consisted of $348,855$ records, corresponding to all the XML documents with insider transactions of executives in the finance industry that were archived by SEC from 2005 to 2010. We compiled the HIL specification into Jaql, using the compilation algorithm described earlier, and ran it on an IBM System x3550 with 2 CPUs (4 cores each) and 32 GB main memory. The total running time for the first step was 1min 28s, and the result consisted of $32,816$ Person entities (15.77 MB). The resulting Person entities were then used in the entity resolution step of Figure 1. This step required two rules, one as shown in Section 2.2 and one that is a slight variation that exploits board membership history. The input JobChange data had $1,077$ records extracted from text documents. The running time for entity resolution was 12s, and the result consisted of 694 links. The third integration 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 in addition to employment histories. These rules were compiled together with the rules of the first step (as discussed in Section 4) and applied to take into account the links generated by entity resolution. The resulting Jaql ran in 1min 48s, 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 fused 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 two entity resolution rules mentioned above are 2.32 KB on disk compared to the 12.54 KB of corresponding Jaql code. In contrast to the Jaql code that consists of a staging of many different queries, with various intermediate results, the HIL rules are more readable, and they give a succint indication of what links are created, from what fields, and by which matching functions. As another example, the fusion rules in the third step of this scenario were relatively simple in HIL (as in Section 2.3). 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 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 spirit to Jaql, would also require complex operations. In the HIL framework, all these low-level operations are hidden from the programmer and automatically handled via the optimizing compiler.

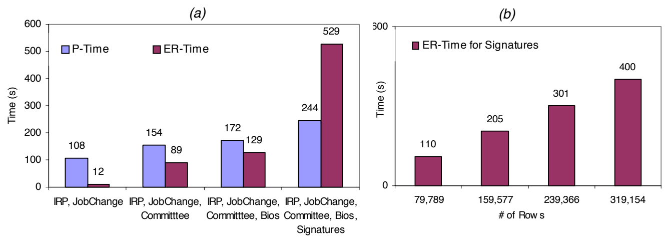

Figure 4: Integration times in the Financial Scenario

5.2 Adding More Data Sources

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 4(a) shows the performance of the HIL-generated code with increasing number of sources. We have described in Section 5.1 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 4(a). With each increasing number of data sources, we include: (1) the total time, ER-Time, to generate the links between the 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 sources so far). The second data point in Figure 4(a) corresponds to adding Committee into the flow. The entity resolution time (89s, or 1 min 29s) includes now the previous entity resolution time (for JobChange) and the additional time to link Committee. P-Time (154s, or 2min 34s) accounts for running all HIL rules for fusing all three data sources together. After adding Bios into the flow, the cumulative entity resolution time, including now the time to link Bios, is 129s, or 2min 9s. The fusion time for all four data sources is 172s, or 2min 52s. 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

Table 2: Data characteristics for social media integration.

# days	# Tweets (cumulative)	# Entities (cumulative)
1	21.72 millions	7.06 millions
2	44.13 millions	10.64 millions
3	68.25 millions	14.01 millions
4	91.13 millions	16.15 millions
5	116.24 millions	17.80 millions
6	140.17 millions	19.16 millions

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 extraction). The cumulative entity resolution time, is now 8min 49s. We notice a significant increase in the entity resolution time, which is explained by the large number of Signatures records that are compared with the Person entities, as well as by the large number of links that are generated and checked for the cardinality constraint. (These are N:1 links from Signatures to Person.) The total fusion time (P-time) for all five data sources is now 4min 4s.

An additional experiment focuses on entity resolution alone and its performance with respect to the number of entities to be resolved. Here, we keep the number of Person entities constant (32, 816) and modify the number of Signatures to be resolved (from 1/4 of the initial Signatures file to 4/4 of the file). Figure 4(b) shows that the execution time increases linearly with the input size.

5.3 Further Remarks

Integration of entities from SEC was, in general, a complex process, and we have described here only a core part. The full-fledged integration also includes rules for generating company entities, investment entities, relationships (including lender/co-lender and parent-subsidiary types of relationships), and many user-defined functions for cleansing (of people names, company names, etc.) and for conflict resolution. Other integration logic included rules for aggregating the stock transactions made by executives or directors of companies in various years, as well as temporal analysis rules to 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).

In total, we used 71 HIL rules, populating and linking 34 entity types, calling 21 UDFs, all split into 11 scripts. The UDFs ranged from very simple such as strToUpperCase, isNull, sortReverseByDate to more complex such as normalizeCompanyName and xmlToJson (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 in approximately 25 mins. The full-fledged HIL-based integration of entities from SEC was used in IBM to populate a commercial master data management (MDM) system, which was demonstrated to financial analysts and regulators (including SEC).

6. EVALUATION: LARGE-SCALE INTEGRATION OF SOCIAL MEDIA PROFILES

In this section, we give a different evaluation of HIL that is targeted at integration from social media, and in particular from Twitter. When compared to the SEC scenario, the data is now much larger, with the overall result consisting of tens of millions of entities, aggregated from hundreds of millions of Twitter messages. Each output entity represents the profile of a social media user, and

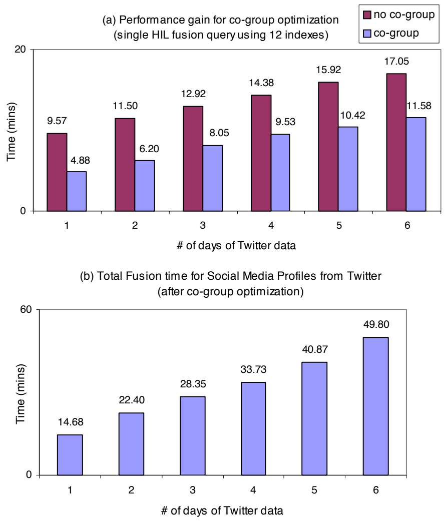

Figure 5: Performance of HIL fusion over Twitter data: (a) effect of co-group optimization, (b) total fusion time.

contains attributes whose values are extracted and fused from the Twitter messages involving that particular user over a time period.

The characteristics of the data are described in Table 2. The input size is measured in terms of the cumulative number of days (from 1 to 6) of Twitter messages that we process, where for each day we have access to 10% of the messages (obtained through Twitter decahose). For each such number of days, we first apply an extraction phase based on SystemT [6], where each input tweet (an unstructured document) can be annotated with any number of attributes ranging from personal information (address, name, marital status, etc.) to life events (birthdays, weddings, etc.), employment (occupation, employer, etc.), and shopping-related attributes. The latter attributes describe a person's purchase intent or sentiment towards products in the retail domain (e.g., intent to buy a camera at Best Buy, intent to buy groceries at Target, a positive comment about a Samsung smartphone, etc). The 2nd column in Table 2 shows the total number of tweets that were annotated, for each day. The set of annotated tweets is then passed as input to the fusion phase, which applies HIL rules to obtain a set of integrated profiles, each corresponding to a Twitter user. The last column in the table shows the resulting number of such user profiles. Note that the number of profiles does not increase linearly with the number of days. This is intuitive: as we analyze more Twitter messages, there are less new users to be discovered. Instead, the user profiles become more complete in terms of the data they accumulate.

The HIL specification includes a main entity population rule that constructs Person profiles. This rule relies on 12 auxilliary indexes that accumulate facts about each person across all tweets. The indexes all use the same key, namely Twitter userid, which is in the metadata associated with each tweet. Each index is populated by one or more HIL rules, based on the extracted annotations. For example, a HIL rule iterates over all tweets that have been annotated by a SystemT "Job" extractor, and inserts relevant facts (with attributes such as JobTitle, JobCategory, Date, the text of the tweet, etc.) into a HIL index called Occupation. Other indexes capture life events, purchase intents, sentiment towards products, etc. The main HIL rule accesses these indexes, via userid, to retrieve either an aggregated single value (e.g., the latest entry in the Occupation index for the given person) or the entire set of records for the given person (e.g., the history of purchase intents).

Figure 5 gives the run-time HIL performance for generating Person profiles, with increasing number of days of tweets. We highlight: (a) the impact of co-group optimization (described in Section 3.3) on the main HIL fusion rule, and (2) the resulting performance for the entire HIL fusion (including the population of all the indexes). The results were obtained by running the HIL-generated Jaql queries on a Hadoop cluster with 10 nodes, each an IBM System x3550, with 2 CPUs (4 cores each), and 32 GB memory.

As it can be seen, co-group optimization achieves a reduction of 30% to 50%, by replacing in this case 12 outer joins with a single co-group followed by one outer join. The effect of co-group is larger if more indexes are added to the integration flow (to incorporate additional extraction). We note that the main HIL fusion is a relatively large portion of the total integration time. For example, for 6 days of tweets, the time taken by the main HIL fusion (via co-group) is 11.58 mins, while the total time (including the creation of all the indexes) is about 50 mins. The gain in performance due to co-group becomes even more significant with further increase in both the time-period of analysis and complexity of analysis.

Full-fledged integration of entities from social media is more complex than what we have described here. To build user profiles with rich enough historical information, we have extracted and integrated data that spans *months* of tweets, with billions of messages. Going beyond Twitter, we have developed entity resolution rules to link with other social media sites such as blogs and forums. These HIL rules exploit combinations of attributes such as name, location, and other contextual clues, and were able to link with high accuracy over 2 million profiles across multiple social media sites.

7. RELATED WORK

Our HIL framework bridges two lines of data integration research: schema mapping [11] and entity resolution [10]. A key differentiating aspect between schema mapping formalisms (e.g., s-t tgds) [12] or SO tgds [13]) and the entity population rules in HIL is that the latter are designed to facilitate direct programming by a user. Associations between entities are explicitly given in HIL via indexes, which effectively eliminate the need for existential quantifiers or Skolem functions. Although technically similar to the dictionaries in the query optimization framework of [8], HIL indexes differ in that they capture the fusion logic in a data integration flow.

Regarding entity resolution, Dedupalog [1] is another declarative language that uses constraints to specify properties on the outcome of entity resolution. However, Dedupalog has no transformation or fusion (i.e., the equivalent of entity population in HIL), and it only allows expressing constraints (and no matching rules) on the candidate links. Furthermore, much of the execution logic is hidden behind a black-box system that attempts to minimize the number of constraint violations. In contrast, HIL rules provide explicit resolution actions on constraint violations, with deterministic results. Ajax [14] is an early data cleaning framework. However, it was focused on matching and clustering and less on mapping and fusion, and did not have a notion of logical entities. XClean [22] is a data cleaning framework for XML; however, its language is at a lower granularity than HIL, with clauses that correspond to physical-level operators that are manually orchestrated in a procedural flow.

Model management [17] provides a high-level scripting framework, but operates at the metadata (schema) level. Closer to HIL is iFuice [21], which combines mapping with fusion of data; however, it 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, and Pig Latin, which are all possible target languages for HIL compilation.

8. CONCLUSION

In this paper, we introduced HIL, a high-level language for entity integration, we gave algorithms for compilation into runtime queries, and showed applications of HIL to integration in the financial domain and from social media. An immediate direction is to explore incremental compilation and evaluation algorithms, where the target data is incrementally modified when new data sources and rules are added. More broadly, the use of a high-level language opens up many research directions in the space of reasoning, debugging, and maintenance (evolution) of integration flows.

9. REFERENCES

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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, 2009.

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

[13] 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.

[14] 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.

[15] M. Hernández, G. Koutrika, R. Krishnamurthy, L. Popa, and R. Wisnesky. HIL: A High-Level Scripting Language for Entity Integration. Technical Report RJ10499, IBM Research, June 2012.

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

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

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.

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

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

[21] 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.

[22] M. Weis and I. Manolescu. XClean in Action (Demo). In *CIDR*, pages 259–262, 2007.