Toward Best-Effort Information Extraction

Warren Shen[1], Pedro DeRose[1], Robert McCann[2], AnHai Doan[1], Raghu Ramakrishnan[3]
[1]University of Wisconsin-Madison, [2]Microsoft Corp., [3]Yahoo! Research

ABSTRACT

Current approaches to develop information extraction (IE) programs have largely focused on producing *precise IE results*. As such, they suffer from three major limitations. First, it is often difficult to execute partially specified IE programs and obtain meaningful results, thereby producing a long "debug loop". Second, it often takes a long time before we can obtain the first meaningful result (by finishing and running a precise IE program), thereby rendering these approaches impractical for time-sensitive IE applications. Finally, by trying to write precise IE programs we may also waste a significant amount of effort, because an approximate result – one that can be produced quickly – may already be satisfactory in many IE settings.

To address these limitations, we propose iFlex, an IE approach that relaxes the precise IE requirement to enable *best-effort IE*. In iFlex, a developer U uses a declarative language to quickly write an initial *approximate IE program* P with a possible-worlds semantics. Then iFlex evaluates P using an *approximate query processor* to quickly extract an approximate result. Next, U examines the result, and further refines P if necessary, to obtain increasingly more precise results. To refine P, U can enlist a *next-effort assistant*, which suggests refinements based on the data and the current version of P. Extensive experiments on real-world domains demonstrate the utility of the iFlex approach.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous

General Terms

Experimentation, Languages

1. INTRODUCTION

Over the past decade, the problem of information extraction (IE) has attracted significant attention. Given a collection of text or Web pages, many solutions have been developed to write programs that extract structured information from the raw data pages (see [1, 11] for recent tutorials).

Virtually all of these solutions, however, have focused only on developing *precise IE programs*: those that output exact IE results. As such, they suffer from several major limitations. First, during the course of developing an IE program, we often cannot execute a partially specified version of the program, and even if we can, it is not clear what the produced result means. This makes it hard to probe whether the extraction strategy pursued by the program is promising, and in general produces a long "debug loop", especially over complex IE programs.

Second, partly because of the above limitation, it often takes a long time (days or even weeks) before we can obtain the first meaningful result (by finishing and running a precise IE program). This long "lag time" is not acceptable to time-sensitive IE applications that need extraction results quickly.

Finally, by trying to write precise IE programs we may also waste a significant amount of effort. In many extraction settings, perhaps an approximate result – one that can be produced quickly – is already satisfactory, either because an exact result is not required (e.g., in mining or exploratory tasks), or because the approximate result set is already so small that a human user can sift through it quickly to find the desired answer.

To address the above limitations, in this paper we argue that in many extraction settings, writing IE programs in a *best-effort* fashion is a better way to proceed, and we describe iFlex (iterative Flexible Extraction System), an initial solution in this direction. In iFlex, a developer U quickly writes an initial *approximate extraction program* P. Then U applies the *approximate program processor* of iFlex to P to produce an approximate result. After examining this result, U can refine P in any way he or she deems appropriate. For this refinement step, U can enlist the *next-effort assistant* of iFlex, which can then suggest particular "spots" in P that can be further refined (to maximize the benefit of U's effort). Then, U iterates with the execution and refinement steps until he or she is satisfied with the extracted result. The following example illustrates the above points.

Example 1.1. *Given 500 Web pages, each listing a house for sale, suppose developer U wants to find all houses whose price exceeds $500000 and whose high school is Lincoln. Then to start, U can quickly write an initial approximate IE program P, by specifying what he or she knows about the target attributes (i.e., price and high school in this case). Suppose U specifies only that price is numeric, and suppose further that there are only nine house pages where each page contains at least one number exceeding 500000 as well as the word "Lincoln". Then iFlex can immediately execute P to return these nine pages as an "approximate*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

superset" result for the initial extraction program. Since this result set is small, U may already be able to sift through it and find the desired houses. Hence, U can already stop with the IE program.

Now suppose that instead of nine, there are actually 120 house pages that contain at least one number exceeding 500000 as well as the word "Lincoln". Then iFlex will return these 120 pages. At this point, U realizes that the IE program P is "underspecified", and hence will try to refine it further (to "narrow" the result set). To do so, U can ask the next-effort assistant to suggest what to focus on next. Suppose that this module suggests to check if price is in bold font, and that after checking, U adds to the IE program that price is in bold font. Then iFlex can leverage this "refinement" to reduce the result set to only 35 houses. Now U can stop, and sift through the 35 houses to find the desired ones. Alternatively, U can try to refine the IE program further, enlisting the next-effort assistant whenever appropriate.

This example underscores the key idea underlying iFlex: instead of requiring the developer to invest significant time *up front* to write relatively complete IE programs before being able to run them and obtain some results, iFlex can produce meaningful results from whatever approximate programs the developer has written so far. Then, as the developer devotes more time refining the programs, iFlex produces increasingly precise results.

While attractive, realizing the above key idea however raises several difficult challenges. The first challenge is to develop a language for writing approximate IE programs with well-defined semantics. Toward this goal, we extend Xlog, a Datalog variant recently proposed for writing declarative IE programs [21]. The extension, called Alog, allows developers to write approximate extraction rules as well as specify the types of approximation involved. We then show that Alog programs have possible-worlds semantics.

Given Alog programs, the second challenge is then to build an efficient approximate processor for these programs. Toward this goal, we first consider how to represent approximate extracted data. We show that current representations are not succinct enough for approximate IE results, and then develop a much more compact text-specific representation called *compact tables*. We then show how to efficiently execute Alog programs over compact tables.

Finally, when refining an Alog program P, we consider how the next-effort assistant can identify "spots" in P where the "next effort" of developer U would best be spent. We formulate this as a problem in which the assistant can ask U a question (e.g., "is price in bold font?"), and then U can add the answer to this question to P, as a "refinement". The key challenge is then to define the question space as well as a way to select a good question from this space. We describe an initial solution to this problem, one that efficiently simulates what happens if U answers a certain question, and then selects the question with the highest expected benefit.

To summarize, in this paper we make the following contributions:

- Introduce iFlex, an end-to-end "iterative approximate" solution for best-effort IE. As far as we know, this is the first in-depth solution to this important problem.
- Introduce a declarative language to write approximate IE programs with well-defined semantics.
- Develop techniques to efficiently represent and perform query processing on approximate extracted text data.
- Develop a novel simulation-based technique to efficiently assist the developer in refining an approximate IE program.
- Perform extensive experiments over several real-world data sets which show that iFlex can (a) quickly produce meaningful extraction results, (b) successfully refine the results in each iteration using the next-effort assistant, and (c) quickly converge to precise extraction results, in a relatively small number of iterations.

2. APPROXIMATE IE PROGRAMS

We now describe how to write approximate IE programs. Our goal is to extend a precise IE language for this purpose. Many such languages have been proposed, such as UIMA, GATE, Lixto, and Xlog [12, 5, 13, 21]. As a first step, in this paper we will extend Xlog, a recently proposed Datalog variant for writing declarative IE programs [21] (leaving extending other IE languages as future research). We first describe Xlog, and then build on it to develop Alog, a language for writing approximate programs for best-effort IE.

2.1 The Xlog Language

We now briefly describe Xlog (see [21] for more details).

Syntax and Semantics: Like in traditional Datalog, an Xlog program P consists of multiple *rules*. Each rule has the form $p :- q_1, \ldots, q_n$, where the p and q_i are *predicates*, p is called the *head*, and the q_i's form the *body*. Each predicate atom in a rule is associated with a relational table. A predicate is *extensional* if its table has been provided to program P, and *intensional* if its table must be computed using the rules in P. Currently, Xlog does not allow rules with negated predicates or recursion.

A key distinction of Xlog, compared to Datalog, is that it can accommodate inherently procedural steps of real-world IE with *p-predicates* and *p-functions*. A p-predicate q has the form $q(\overline{a_1}, \ldots, \overline{a_n}, b_1, \ldots, b_m)$, where the a_i and b_i are variables. Predicate q is associated with a procedure g (e.g., written in Java or Perl) that takes an input tuple $(u_1, \ldots, u_n)$, where u_i is bound to a_i, $i \in [1, n]$, and produces as output a set of tuples $(u_1, \ldots, u_n, v_1, \ldots, v_m)$. A p-function $f(\overline{a_1}, \ldots, \overline{a_n})$ takes as input a tuple $(u_1, \ldots, u_n)$ and returns a scalar value.

EXAMPLE 2.1. *Figure 1.a shows an Xlog program that extracts houses with price above $500000, area above 4500 square feet, and a top high school. This program contains two p-predicates, $extractHouses(\overline{x}, p, a, h)$ and $extractSchools(\overline{y}, s)$, and one p-function $approxMatch(\overline{h}, \overline{s})$. The p-predicate $extractHouses(\overline{x}, p, a, h)$, for example, takes as input a document x and returns all tuples (x, p, a, h) where p, a, and h are the price, area, and high school of a house listed in x, respectively. The p-function $approxMatch(\overline{h}, \overline{s})$, for example, returns true iff two text spans (i.e., document fragments) h and s are "similar" according to some similarity function (e.g., TF/IDF).*

Among all types of p-predicate, we single out a special type called *IE predicate*. An *IE predicate* is a p-predicate that extracts one or more output spans from a single input document or span. For example, $extractHouses(\overline{x}, p, a, h)$ in Figure 1.a is an IE predicate that extracts the attributes p, a, and h from x. Similarly, $extractSchools(\overline{y}, s)$ extracts a school name s from y.

One of the head predicates of an Xlog program P is called a *query*, and the result of P is the relation computed for the query predicate using the rules in P. We define this result by

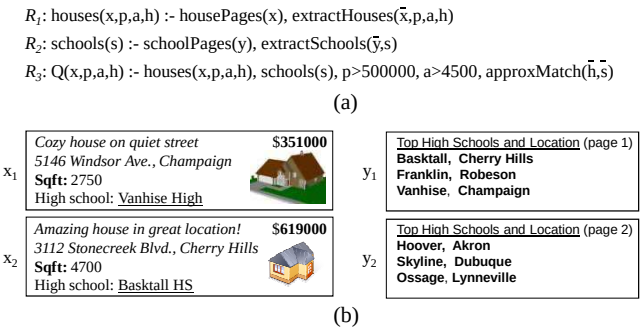

Figure 1: Example Xlog program.

applying traditional Datalog semantics (e.g., the least-model semantics) to the rules in P, using the associated relations for the p-predicates and p-functions in P.

EXAMPLE 2.2. *We illustrate Xlog by stepping through a conceptual execution of the program in Figure 1.a, assuming that* housePages *consists of the two documents x_1 and x_2, and that* schoolPages *consists of the two documents y_1 and y_2 (all in Figure 1.b). For each document x in* housePages*, rule R_1 invokes $extractHouses(\overline{x}, p, a, h)$ to extract the price, area, and high school of the listed house in x (e.g., to extract the tuple $(x_1, 351000, 2750,$ "Vanhise High"$)$ from x_1). Next, rule R_2 invokes $extractSchools(\overline{y}, s)$ for each document y in the* schoolPages *relation (e.g., to extract three tuples $(y_1,$ "Basktall"$)$, $(y_1,$ "Franklin"$)$, and $(y_1,$ "Vanhise"$)$ for y_1). Finally, rule R_3 joins the extracted* houses *and* schools *relations, keeping all tuples (x, p, a, h) where document x lists a house with price p above $\$500,000$, area a above 4,500 square feet, and high school h being listed in a school page y in* schoolPages*. In this example, rule R_3 produces the tuple $(x_2, 619000, 4700,$ "Basktall HS"$)$.*

Usage and Limitations: To write an Xlog program for an IE task, a developer U first decomposes the task into smaller tasks. Next, U writes an Xlog program that reflects this decomposition, "making up" p-predicates and p-functions as he or she goes along. Finally, U implements made-up predicates and functions, e.g., with C++ or Java procedures. For example, when writing the Xlog program in Figure 1.a, U makes up and then implements p-predicates $extractHouses$ and $extractSchools$, and p-function $approxMatch$.

To implement a procedure (e.g., for a p-predicate), U typically starts by examining the raw data and identifying domain constraints involving text features of the target attributes (e.g., "price is numeric", "school name is bold"). Next, U writes code (e.g., Perl modules) to extract text spans that satisfy these constraints, then writes additional code to further chop, merge, and filter the text spans, to eventually output the desired extraction result. Once done, U "plugs" the implemented procedures into the Xlog program, executes the program, and analyzes the results for debugging purposes.

As described, U often must spend a significant amount of time developing the procedures, and these procedures must be "fairly complete" before U can run the IE program to obtain some meaningful result. This in turn often makes the development process difficult, time-consuming, and potentially wasteful, as discussed in Section 1. To address these problems, we develop a best-effort IE framework in which we can quickly write approximate IE programs using Alog, an Xlog extension.

2.2 Best-Effort IE with Alog

Given our observations with Xlog, we propose that a best-effort IE framework provide support for the developers to:

- Declare any domain constraints regarding text features of the attributes they are extracting, with as little extra programming as possible.
- Write approximate IE programs and obtain well-defined approximate results, such that the more effort the developer puts in, the more precise the results are.

Our proposed language, an extension of Xlog called Alog, provides such support. To write an Alog program, a developer U can quickly start by writing an initial Xlog program, which is "skeletal" in the sense that it lists all the necessary IE predicates, but does not yet have any associated procedures for them.

In the next step, instead of implementing these procedures *in their entirety* – as is done during the process of developing an Xlog program – U instead implements them only *partially*, doing only as much work as he or she deems appropriate.

Specifically, consider a particular IE predicate q. To implement q, U partially writes one or more *description rules*, each declaring a set of domain constraints about the textual features of the (target) attributes that E extracts (e.g., "price is numeric"). Each description rule r therefore can be viewed as a way to extract approximate values for the target attributes. Next, U encodes the type of approximation by *annotating* r. Such an annotation in effect gives rule r a *possible-worlds interpretation*, by "translating" the single relation defined by r under Xlog semantics into a set of possible relations.

Finally, U executes the so-obtained partial IE program, examines the result, refines the program further by augmenting the description rules, executes the newly revised program, and so on, until satisfied. In the rest of this section we describe the above steps in more detail.

2.2.1 Writing an Initial Program

As discussed, to develop an Alog program, a developer U begins by writing an initial set of traditional Xlog rules. These rules form a "skeleton" of P, in the sense that each IE predicate (in the rules) is "empty": it is not associated with any procedure yet. Figure 2.a shows a sample skeleton program (which is the same program shown in Figure 1.a), where the IE predicates $extractHouses(\overline{a}, p, a, h)$ and $extractSchools(\overline{y}, s)$ do not have any associated procedure.

2.2.2 Writing Predicate Description Rules

Syntax: In the next step, instead of writing procedural code for each IE predicate q (as in traditional Xlog), U "partially implements" q by writing one or more *predicate description rules* (or "description rules" for short) to describe q using domain constraints.

For example, rule S_4 in Figure 2.b describes the IE predicate $extractHouses(\overline{x}, p, a, h)$. It asserts that p, a, and h are text spans from x, and that p and a are numeric. Thus, when provided with a document x as the input, this rule defines a relation (x, p, a, h) where p, a and h are extracted from x, and p and h take on numeric values. Similarly, rule S_5 in Figure 2.b describes $extractSchools(\overline{y}, s)$ and asserts that s is a bold-font text span from document y.

In general, a description rule r has the form $q :- q_1, \ldots, q_n$, just like a traditional Xlog rule, except that the head q is an IE predicate. Rule r's semantics are similar to that of an Xlog rule, except that r only defines a relation when it is provided with all of the required input. That is, whenever

we assign constant values to the input variables in the head of r, r defines a relation R such that for any assignment of constants to the other variables in r, if every predicate in the body of r is satisfied, then the tuple generated for the rule head (using the assignment) is also in R.

Using Description Rules to Capture Domain Constraints: Developer U typically builds description rules by adding *domain constraints* about the extracted attributes as predicates to the body of the rules. A domain constraint $f(a) = v$ states that feature f of any text span that is a value for attribute a must take value v. For example, rule S_4 in Figure 2.b imposes two domain constraints: $numeric(p) = yes$ and $numeric(a) = yes$. In each iteration U can refine each description rule by adding more domain constraints.

In general, *text features* capture common characteristics of text spans that we often are interested in extracting. Example features include numeric, bold-font, italic-font, underlined, hyperlinked, preceded-by, followed-by, min-value, and max-value. Each feature takes values such as yes, distinct-yes, no, distinct-no, and unknown. For example, bold-font(s) = distinct-yes means that s is set in bold font, but the text surrounding s is not.

iFlex currently uses a rich set of built-in features (Section 5.1.1 briefly discusses how to select good features), but more can be easily added, as appropriate for the domain at hand. To add a new feature f, a developer needs to implement only two procedures Verify and Refine. (Note that this is done only once, not per writing an Alog program.) Verify(s, f, v) checks whether $f(s) = v$, and Refine(s, f, v) returns all sub-spans t from s such that $f(t) = v$ (see Section 4.2 for more details).

Writing Safe Description Rules: Consider the following rule

$$extractHouses(\overline{x}, p, a, h) :- numeric(p) = yes, \; numeric(a) = yes$$

which states that p and a are numeric. This rule is not *safe* because it produces an infinite relation given an input document x (since p and a can take on any numeric value, and h can take on any value). Intuitively, this is because the rule does not indicate where p, a, and h are extracted from. To address this problem, iFlex provides a built-in from$(\overline{x}, y)$ predicate that conceptually extracts all sub-spans y from document x. Then, U can use this predicate to indicate that p, a, and h are extracted from document x. Figure 2.b shows the resulting safe description rules, using the from predicate.

In general, a description rule is safe if each non-input variable in the head also appears in the body, either in an extensional or intensional predicate, or as an output variable in an IE predicate. For example, rule S_4 in Figure 2.b is safe because the non-input variables p, a, and h all appear as output variables in from predicates in the body.

2.2.3 Encoding Approximation Types using Annotations

As described above, the combination of an Alog rule and related description rules defines a way to *approximately* extract some attributes. For example, rules S_1 and S_4 in Figure 2.b together describe a way to approximately extract p, a, and h.

When writing such rules, a developer U often knows the *type of approximation* he or she is using. Hence, we want to provide a way for U to declare such approximation types. We then exploit this information to better process Alog programs. Currently, we focus on providing support for two common types of approximation: those about the *existence* of a tuple, and about the *value* of an attribute in a tuple, respectively. We allow a developer U to *annotate* an Alog rule to indicate these approximation types.

Specifically, an *existence annotation* indicates that each tuple in the relation R produced by a rule r may or may not exist.

DEFINITION 1 (EXISTENCE ANNOTATION). *Let p be the head of a rule r that produces relation R under the normal Xlog semantics. Then adding an existence annotation to r means replacing p with "p?". This produces a rule r' that defines a set of possible relations $\mathcal{R}$, which is the powerset of the tuples in R.*

An *attribute annotation* indicates that an attribute takes a value from a given set, but we do not know which value.

DEFINITION 2 (ATTRIBUTE ANNOTATION). *Suppose the head of rule r is $p(a_1, \ldots, a_i, \ldots, a_n)$. Then annotating attribute a_i means replacing the head with $p(a_1, \ldots, \langle a_i \rangle, \ldots, a_n)$ to produce rule r'. Suppose that r defines relation R under the normal Xlog semantics. Then r' defines the set $\mathcal{R}$ of all possible relations that can be constructed by grouping R by $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ and selecting one value for a_i in each group.*

We can easily generalize Definition 2 to annotating multiple attributes. In this case, each possible relation is constructed by first grouping R by all non-annotated attributes, then selecting one value for each annotated attribute in each group. The following example illustrates both existence and attribute annotations.

EXAMPLE 2.3. *Evaluating rule S_1 in Figure 2.a together with rule S_4 in Figure 2.b would produce the* houses *table in Figure 2.d, which contains all tuples (x, p, a, h) where x is in* housePages*, p, a, and h are extracted from x, and p and a are numeric.*

Now suppose we know that each document x in housePages *contains information about exactly one house (i.e., x forms a key in the true* houses *relation). Then we can annotate attributes p, a, and h to produce rule S_1' in Figure 2.c. Evaluating rule S_1' would produce the set of possible* houses *relations as represented in Figure 2.e, where each possible relation is constructed by selecting just one value (from the corresponding set of values) for each table cell (see Section 3 for more details). This way, each possible* houses *relation contains exactly one tuple for each document x, thus accurately reflecting our knowledge of the domain.*

Similarly, evaluating rules S_2 in Figure 2.a together with rule S_5 in Figure 2.b would produce the schools *table in Figure 2.d, which contains all tuples (s) where s is a bold span coming from a document y in* schoolPages.

Clearly, not all bold spans in each document y are schools. Thus, we can add an existence annotation to rule S_2 to produce rule S_2' in Figure 2.c. Evaluating S_2' would produce the set of possible schools *relations represented in Figure 2.e, where each possible relation consists of a subset of the extracted tuples.*

We represent the annotations for a rule r with a pair (f, A), where the Boolean f is true iff r has an existence annotation, and A is the set of attributes in the head predicate of r that have attribute annotations.

Alog Semantics: The result of an Alog program P is then the set of all possible relations we can compute for the query predicate in P. Defining this set reduces to defining the set of all possible relations that each Alog rule r in P computes.

Let r be $p :- q_1, \ldots, q_n$. Then we compute r as follows. We know that each predicate q_i is associated with a set of relations $\mathcal{R}_i$. If we select a relation $R_i \in \mathcal{R}_i$ for each q_i, then we can evaluate p (in the traditional Xlog semantics) to obtain a relation R.

S_1: houses(x,p,a,h) :- housePages(x), extractHouses($\bar{x}$,p,a,h)
S_2: schools(s) :- schoolPages(y), extractSchools($\bar{y}$,s)
S_3: Q(x,p,a,h) :- houses(x,p,a,h), schools(s), p>500000, a>4500, approxMatch($\bar{h}$,$\bar{s}$)

(a)

S_4: extractHouses($\bar{x}$,p,a,h) :- from($\bar{x}$,p), from($\bar{x}$,a), from($\bar{x}$,h) numeric($\bar{p}$)=yes, numeric($\bar{a}$)=yes
S_5: extractSchools($\bar{y}$,s) :- from($\bar{y}$,s), bold-font($\bar{s}$)=yes

(b)

S'_1: houses(x,<p>,<a>,<h>) :- housePages(x), extractHouses($\bar{x}$,p,a,h)
S'_2: schools(s)? :- schoolPages(y), extractSchools($\bar{y}$,s)
S'_3: Q(x,p,a,h) :- houses(x,p,a,h), schools(s), p>500000, a>4500, approxMatch($\bar{h}$,$\bar{s}$)

(c)

houses

x	p	a	h
x_1	**351000**	**351000**	"*Cozy*"
x_1	**351000**	*5146*	"*Cozy*"
x_1	**351000**	2750	"*Cozy*"
x_1	*5146*	**351000**	"*Cozy*"
		⋮	
x_2	4700	4700	"HS"

schools

s
"**Basktall**"
⋮
"**Champaign**"
"**Hoover**"
⋮
"**Lynneville**"

(d)

houses

x	p	a	h
x_1	{ **351000**, *5146*, 2750 }	{ **351000**, *5146*, 2750 }	{"*Cozy*", "*Cozy house*", ... , "Vanhise High", "High"}
x_2	{ **619000**, *3112*, 4700 }	{ **619000**, *3112*, 4700 }	{"*Amazing*", "*Amazing house*", ... , "Basktall HS", "HS"}

schools

s	
"**Basktall**"	?
⋮	
"**Champaign**"	?
"**Hoover**"	?
⋮	
"**Lynneville**"	?

(e)

Figure 2: Alog program and execution.

Let the annotations of r be (f, A). Then we must apply these annotations to R, in order to obtain the true set of relations presented by p (when each q_i receives a relation R_i as the input). To do so, we first apply attribute annotations A to R, as described in Definition 2. This produces a set of relations $\mathcal{R}_A$. Next, if r has an existence annotation (i.e., f is true), then we create the set of relations $\mathcal{R}_{Af}$, where each relation in this set is a subset of a relation in $\mathcal{R}_A$. The set $\mathcal{R}_{Af}$ is then the true set of relations presented by p (when each q_i receives a relation R_i as the input). If f is false, then $\mathcal{R}_{Af}$ is set to be $\mathcal{R}_A$.

The set of all possible relations that rule r computes is then the union of all sets $\mathcal{R}_{Af}$ that can be computed as described above (by assigning to the q_i's a different combination of the relations R_i's).

EXAMPLE 2.4. *The following is a conceptual procedure to evaluate rule S'_3 in Figure 2.a over the set of possible relations for* houses *and* schools *represented in Figure 2.e. Select one possible* houses *relation H and one possible* schools *relation S. Then, evaluate S'_3 using H and S to produce an intermediate set of possible relations (in this particular case, we produce a set with just one possible relation because S'_3 does not have any annotations). Repeat this process for every possible pair of* houses *and* schools *relations H and S. The output of S'_3 is the union of all intermediate sets of possible relations.*

2.2.4 Executing, Revising, and Cleaning Up Alog Programs

Once developer U has written an initial Alog program P (which consists of the "skeleton" Xlog rules, description rules for IE predicates, and possibly also annotations), he or she can execute P to obtain an approximate extraction result (see the next two sections).

Then, U can revise P by adding more domain constraints to the description rules. To find effective constraints, U can enlist the next-effort assistant, as Section 5 will discuss.

Eventually, either P has been revised to the point where it produces precise IE results (in which case U can stop), or to the point where U feels that adding more domain constraints will not help improve the extraction result. In this latter scenario, U may want to just write a "cleanup procedure" in a procedural language (e.g., Perl) rather than continue to revise P declaratively (by adding more domain constraints).

One such scenario occurs when an IE task involves a subtask that is hard to express declaratively. For example, suppose that U wants to extract publication citations and their last authors from DBLP. While it may be relatively easy to extract citations and *all* of their authors by asserting domain constraints declaratively, it may be cumbersome to extract the *last* author (since Alog does not naturally handle ordered sequences). Therefore, a more natural solution would be to assert domain constraints to extract citations and their *author list*, and then write a cleanup procedure to extract the individual authors and select the last author.

To incorporate a cleanup procedure g, U simply needs to declare a new p-predicate p and associate g with it. Afterward, U can use p in his or her Alog program just like any other p-predicate.

3. REPRESENTING APPROXIMATE DATA

We now describe a data model to represent the approximate extracted data that Alog rules produce.

Approximate Tables: Recall from Section 2 that Alog accounts for two types of approximation: the existence of a tuple and the value of an attribute in a tuple. To represent these types, we can extend the relational model by introducing *approximate tables*, or *a-tables* for short, where an a-table is a multiset of *a-tuples*. An *a-tuple* is a tuple $(V_1, \ldots, V_n)$, where each V_i is a multiset of possible values. An a-tuple may be annotated with a '?', in which case it is also called a *maybe a-tuple* [19]. Figure 2.e shows for example a-tables for the *houses* and *schools* relations (here for clarity we have omitted the set notation for the x attribute).

An a-table T represents the set of all possible relations that can be constructed by (a) selecting a subset of the maybe a-tuples and all non-maybe a-tuple in T, then (b) selecting one possible value for each attribute in each a-tuple selected in step (a).

Compact Tables: A-tables however are typically not succinct enough in our setting, where an Alog rule may produce a huge number of possible extracted values. For example, in Figure 2.e, the cells for attribute h in *houses* enumerate every sub-span in each house page, and *schools* contains one tuple for every bold sub-span in the school pages.

Therefore, iFlex employs *compact tables*, a much more compact version of a-tables specifically designed for approximate extracted text. The key idea is to exploit the sequential nature of text to "pack" the set of values of each cell into a much smaller set of so-called *assignments*, when possible.

Toward this goal, we define two types of assignments: exact and contain. An assignment exact(s) encodes a value that is exactly span s (modulo an optional cast from string to numeric). For example, exact("92") encodes value 92. As

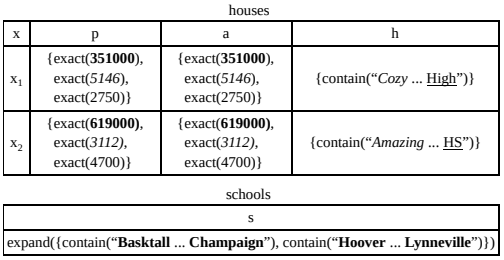

Figure 3: Compact tables.

shorthand, we will sometimes write $\mathsf{exact}(s)$ as simply s. An assignment $\mathsf{contain}(s)$ encodes all values that are s itself or sub-spans of s. For example, the assignment $\mathsf{contain}$("Cherry Hills") encodes all values that are spans inside "Cherry Hills" (e.g., "Ch", "Cherry", etc.).

Given this, each cell c in a compact table contains a multiset of assignments: $c = \{m_1(s_1), \ldots, m_n(s_n)\}$, where each m_i is either exact or $\mathsf{contain}$ and each s_i is a span. Let $\mathcal{V}(m_i(s_i))$ be the set of values encoded by $m_i(s_i)$. Then, the set of values $\mathcal{V}(c)$ encoded by cell c is $\cup_{i=1}^{n}\mathcal{V}(m_i(s_i))$. For example, consider the first a-tuple in the *houses* relation in Figure 2.e, where the cell for attribute h has possible values {"*Cozy*", "*Cozy house*",..., "Vanhise High", "High"}. Since these possible values are all the sub-spans of the span "*Cozy...*High" from document x_1, we can condense the cell for h to be {$\mathsf{contain}$("*Cozy...*High")}. Similarly, we can condense the possible values for h in the second a-tuple with another $\mathsf{contain}$ assignment. The compact table for *houses* in Figure 3 shows the result of condensing the *houses* a-table in Figure 2.e, using assignments.

Next, consider the *schools* table in Figure 2.e. We can condense the a-tuples of this table by representing them all with one compact tuple (expand({"Basktall", ..., "Lynneville")}). In general, if t is a compact tuple with cells $(c_1, \ldots, c_i, \ldots, c_n)$, where $c_i = \mathsf{expand}(v_1, \ldots, v_k)$, then t can be "expanded" into the set of compact tuples obtained by replacing cell c_i with an assignment $\mathsf{exact}(v_j)$: $(c_1, \ldots, \mathsf{exact}(v_j), \ldots, c_n)$, where $1 \leq j \leq k$. We call c_i an *expansion cell*.

Expansion cells can still be condensed further. For example, consider again the compact tuple (expand({"Basktall", ..., "Lynneville")}). Notice that the values {"Basktall", ..., "Lynneville")} is the set of all sub-spans of the two bold spans "Basktall ... Champaign" in document y_1 and "Hoover ... Lynneville" in document y_2. Thus, we can condense these values using two assignments $\mathsf{contain}(s_1)$ and $\mathsf{contain}(s_2)$, where s_1 ="Basktall ... Champaign" and s_2 ="Hoover ... Lynneville". As a result, we can further condense the compact tuple into an equivalent tuple (expand({$\mathsf{contain}(s_1)$}, $\mathsf{contain}(s_2)$)). Figure 3 shows the result of condensing the *schools* table in Figure 2.e, using expansion cells. We can now define compact tables as follows:

DEFINITION 3 (COMPACT TABLE). *A* compact table *is a multiset of* compact tuples. *A compact tuple is a tuple of cells* $(c_1, \ldots, c_n)$ *where each cell* c_i *is a multiset of assignments or an expansion cell. A compact tuple may optionally be designated as a* maybe compact tuple*, denoted with a '?'.*

A compact table T represents a set of possible relations $\mathcal{R}$. We can conceptually construct $\mathcal{R}$ by first converting T into an a-table T', then converting T' into the set of possible relations $\mathcal{R}$, as described earlier.

We convert T into the a-table T' as follows. Let $t \in T$ be a compact tuple with expansion cell c. Then, we replace t with the set of compact tuples $\mathcal{T}$, as described earlier. If t is a maybe compact tuple, then we modify each compact tuple $u \in \mathcal{T}$ to be a maybe compact tuple. We continue this process until T has no more expansion cells. Finally, we convert T into the a-table T' by converting each cell c into one with a set of possible values instead of assignments. That is, if cell c has assignments $\{m_1(s_1), \ldots, m_n(s_n)\}$, then we replace those assignments with the set of values $\cup_{i=1}^{n}\mathcal{V}(m_i(s_i)))$.

We note that the compact table representation is not a complete model for approximate data in that it is not expressive enough to represent any finite set of possible relations [19]. For example, compact tables cannot represent mutual exclusion among tuples (e.g., a relation contains either tuple t_1 or tuple t_2, but not both).

Nevertheless, in this paper we start with compact tables for two reasons. First, they can already express two common types of extraction approximation (the existence of a tuple and the value of an attribute), and thus can accommodate a broad variety of IE scenarios. Second, they exploit text properties, and hence enable efficient IE processing, as discussed in the next section. In future work we plan to explore complete, but non-text-specific representations (e.g., those in [19, 2]) for representing approximate IE results.

4. APPROXIMATE QUERY PROCESSING

We now describe the approximate query processor that generates and executes an execution plan for an Alog program P. First, we unfold the non-description rules in P to produce a program P'. Suppose the body of a rule r_1 in P has an IE predicate q, and that q appears in the head of a description rule r_2. Then we unfold r_1 by replacing q in r_1 with the body of r_2, unifying variables if necessary. We repeat this process until only IE predicates with associated procedures appear in the program. For example, in Figure 2.c, after unfolding rules S_1', S_2', and S_3' (using the description rules S_4 and S_5), rules S_1', S_2', and S_3' are transformed into the rules shown in Figure 4.a.

Next, we construct a logical plan fragment h for each rule r in P', as described in [21]. Initially, we ignore all annotations, compiling h as if it processes ordinary relations. Figure 4.b shows the plan fragments we compile for the rules in Figure 4.a (for simplicity, we ignore projections).

Then, we change each plan fragment h to work over compact tables instead of ordinary relations, in three steps. First, we convert each extensional relation into a compact table by changing the contents of each cell that has a document or span value v to instead have the value {$\mathsf{exact}(v)$}. Second, we modify each operator in h to take compact tables as input and produce a compact table as output. Third, we append an "annotation operator" ψ to the root of h, where ψ converts the set of relations output by h into another set of possible relations, taking into account the annotations of the rule r that h corresponds to.

Finally, we form a final execution plan g by "stitching" together the plan fragments, unifying variables if necessary (see [21]). Figure 4.c shows the result of stitching together the plan fragments in Figure 4.b based on the program in Figure 4.a.

In the rest of this section, we address the challenges involved to convert h to work over compact tables. We start by showing how to adapt relational operators and p-predicates to process compact tables. Then, we show how to efficiently evaluate domain constraints to extract text from compact

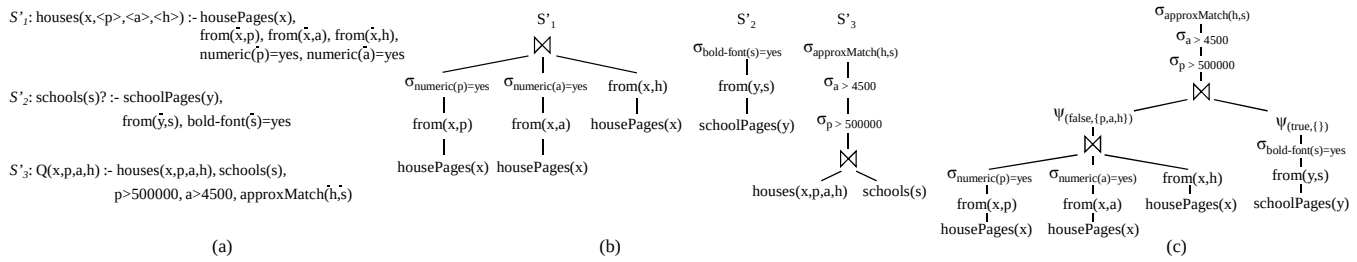

Figure 4: Compiling Alog programs.

tables. Finally, we define the ψ operator that converts sets of possible relations based on the annotations in the rules.

Recall that the compact table representation is not a complete model. It turns out that compact tables also are not *closed* under relational operators, meaning that when applying a relational operator to compact tables, we cannot always exactly represent the output with another compact table.

Therefore, in the rest of this section we develop operators such that the execution plan produces an "approximate approximation" [19] for a program P. Specifically, we adopt a *superset semantics* for query execution, meaning that the result of the execution is guaranteed to be a superset of the possible relations defined by the program. In future work, we plan to explore other execution semantics (e.g., one that minimizes the number of incorrect tuples).

4.1 Modifying Relational Operators and P-Predicates

We now discuss how to modify relational operators (i.e., select, join, project) and p-predicates in execution plans to work over compact tables. Projection is straightforward since we ignore duplicate detection. Thus, we focus on selections and θ-joins.

Consider evaluating a selection on a compact table. Intuitively, to ensure superset semantics we want the results to contain *all* compact tuples that *may* satisfy the selection condition. To illustrate, consider applying a selection condition $p > 500000$ to the first compact tuple in the *houses* table in Figure 3. Since x_1 contains no possible price greater than 500000, we can safely drop this tuple. On the other hand, we must keep the tuple for page x_2 because it contains the possible price 619000.

In general, we evaluate a selection σ_f with selection condition f over a compact table T as follows. For each compact tuple $t \in T$ with cells $(c_1, \ldots, c_n)$, we evaluate the selection condition f over every possible tuple $(v_1, \ldots, v_n)$, where $v_i \in \mathcal{V}(c_i)$ for $1 \leq i \leq n$. If f is satisfied for *any* possible tuple, then we add t to T'. Also, if f is satisfied for some but not all of the possible tuples, we set t to be a maybe compact tuple. To reduce the number of possible tuples we consider, we enumerate possible values for only those attributes involved in f (e.g., when selecting on the price p, only enumerate possible values for the price).

We adapt θ-joins over compact tables $T_1, \ldots, T_n$ in a similar fashion except that we evaluate the θ condition on all compact tuples in the Cartesian product $T_1 \times \ldots \times T_n$. However, implementing certain other types of joins over compact tables, such as approximate string joins, is significantly more involved. In the full paper, we describe our solution for approximate string joins [20].

Finally, to evaluate a p-predicate p over a compact table T, we take the union of the results of evaluating p over each compact tuple $t \in T$. To evaluate p over a compact tuple t, we first convert t into an equivalent set of compact tuples U that have no expansion cells by repeatedly replacing each expansion cell c in t with $\mathcal{T}$ (Section 3). Then, for each compact tuple $u \in U$, we enumerate the possible tuples V that u represents, and invoke p for each $v \in V$. Let $p[v]$ be the tuples produced by p with input tuple v. We then convert $p[v]$ into a set of compact tuples by making each cell a set of assignments (i.e., by changing any cell with span value s to be $\{\mathsf{exact}(s)\}$). Finally, we set each tuple in $p[v]$ to be a maybe compact tuple if u represents more than one possible tuple (i.e., if $|V| > 1$).

4.2 Optimizing Alog Rules

Consider the plan fragment $\sigma_{bold\text{-}font(s)=yes}[from(\overline{y}, s)]$ of the plan from Figure 4.c. Conceptually, the IE predicate $\mathsf{from}(\overline{y}, s)$ extracts every possible span from an input document y. To speed this up, in practice the from predicate instead produces the compact tuple $(y, \mathsf{expand}(\{\mathsf{contain}(y)\}))$ to avoid enumerating all possible spans in y. In general suppose we apply $\mathsf{from}(\overline{y}, s)$ to an input compact tuple (y), where y is the set of assignments $\{m_1(s_1), \ldots, m_n(s_n)\}$ (representing possible values for y). Then, $\mathsf{from}(\overline{y}, s)$ produces compact tuple (y, s), where s is $\mathsf{expand}(\{\mathsf{contain}(s_1), \ldots, \mathsf{contain}(s_n)\})$.

However, this can still result in slow execution when applying domain constraints. For example, suppose that in the plan fragment $\sigma_{bold\text{-}font(s)=yes}[from(\overline{y}, s)]$, the from predicate produces the compact tuple $(y, \mathsf{expand}(\mathsf{contain}(y)))$. A naive method to apply the domain constraint $bold\text{-}font(s) = yes$ would be to to enumerate all possible spans in y and keep only those that are in bold font.

Instead, a more natural and efficient strategy would be to scan document y once and find all maximal spans $\{s_1, \ldots, s_n\}$ in y that are in bold font (e.g., by using a Perl regular expression). Then, we output the set of assignments $\{\mathsf{contain}(s_1), \ldots, \mathsf{contain}(s_n)\}$. This way, we avoid enumerating all possible spans in y.

In general, we optimize selections involving domain constraints as follows. First, in iFlex each text feature f is associated with two procedures Verify and Refine. $\mathsf{Verify}(s, f, v)$ returns true if $f(s) = v$, and false otherwise. $\mathsf{Refine}(s, f, v)$ returns all *maximal* sub-spans s' of s for which $\mathsf{Verify}(s', f, v)$ is true. A sub-span s' is maximal if we cannot extend it either to the left or the right to obtain a sub-span s'' for which $\mathsf{Verify}(s'', f, v)$ is true.

Then, to evaluate a selection σ_k where k is a domain constraint, we use Verify and Refine as follows. Let T' be the result of evaluating σ_k over a compact table T, where k is the constraint $f(a) = v$. Then T' will be a compact table with the same structure as the input T, except that each cell for attribute a has been transformed by applying k. In particular, let c and c' be two corresponding cells of T and

T' for attribute a, and let $c = \{m_1(s_1), \ldots, m_n(s_n)\}$. Then

$$c' = \cup_{i=1}^{n} \quad \mathcal{A}(k, m_i(s_i))$$

where $\mathcal{A}(k, m_i(s_i))$ denotes the set of assignments resulting from applying the constraint k to assignment $m_i(s_i)$. Also, if c is an expansion cell we set c' to be an expansion cell.

We now describe how to compute $\mathcal{A}(k, m_i(s_i))$. Recall that $m_i(s_i)$ encodes a set of values for attribute a. Our goal is to remove as many values as possible (from $m_i(s_i)$) that do not satisfy $\mathsf{Verify}(a, f, v)$. To do so, we consider two cases:

Case 1: If $m_i(s_i)$ is an exact assignment, then we can simply invoke $\mathsf{Verify}(s_i, f, v)$. Thus, $\mathcal{A}(k, m_i(s_i))$ returns the empty set if $\mathsf{Verify}(s_i, f, v)$ evaluates to false, and returns $m_i(s_i)$ otherwise.

Case 2: If $m_i(s_i)$ is a contain assignment, then we iteratively refine s_i. To do this we call $\mathsf{Refine}(s_i, f, v)$ to obtain all maximal sub-spans x in s_i such that $f(x) = v$. For each such region x, Refine produces an assignment $\mathsf{contain}(x)$ or $\mathsf{exact}(x)$. To see why, consider the Web page snippet "*Price: 35.99*. Only two left." Suppose k_1 is "*italics*(*price*) = yes". Then price is in italics and hence can be any sub-span of *"Price: 35.99"*. Consequently, applying Refine to the above snippet will produce contain(*"Price: 35.99"*). On the other hand, consider the snippet "Price: *35.99*. Only two left." Suppose k_1 is "*italics*(*price*) = distinct-yes". Then we know that price is in italics, but its surrounding text is not. Consequently, it can only be 35.99, and applying Refine to the above snippet will produce *exact*("35.99").

There is however a minor complication when evaluating Alog rules involving more than one domain constraint for an attribute a. Let $k_1, \ldots, k_n$ be the constraints we need to apply for an attribute a in an execution plan. Suppose refining an assignment with constraint k_1 yields an assignment $m(s)$, and further refining $m(s)$ with constraint k_2 yields an assignment $m'(s')$. Then it is possible that span s' does not satisfy k_1. However, each span that $\mathcal{A}(k_2, m_i(s_i))$ finally outputs must satisfy *all* constraints $k_1, \ldots, k_n$. Hence, we always check all sub-spans created with k_j for violation of $k_1, k_2, \ldots, k_{j-1}$. It is easy to prove that any order of applying $k_1, \ldots, k_n$ produces the same final set of assignments.

4.3 Defining the Annotation Operator

As the final step to generate an execution plan for Alog program P, we define the annotation operator ψ that converts a set of possible relations by applying the annotations of the rules in P. Suppose r is an Alog rule with annotations (f, A), and suppose h is a plan fragment we have produced for a rule r, ignoring the annotations of r. Then, the output of the plan $\psi_{(f,A)}(h)$ is the set of the possible relations defined by r (taking into account the annotations (f, A)).

Implementing ψ to handle existence annotations is trivial: we make every compact tuple that ψ outputs to be a maybe compact tuple. Hence, in the rest of this section we focus on handling attribute annotations.

Suppose we are evaluating the plan $\psi_{(f,A)}(h)$, and that plan fragment h produces compact table S. Our default strategy is to first convert S into an a-table T (see Section 3), evaluate ψ over T, and then convert the result back into another compact table. Thus, in the rest of this section, we describe BAnnotate, an implementation of ψ that takes as input an a-table T and outputs another a-table T'. Converting between a-tables and compact tables is straightforward and thus we do not discuss it further. In the full paper we discuss optimizing this process to evaluate ψ over compact tables directly, without converting them to a-tables [20].

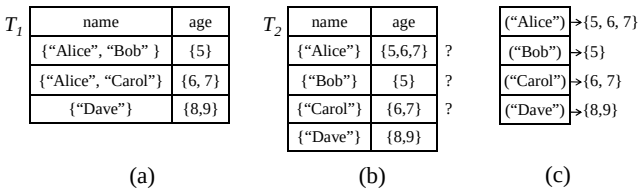

Figure 5: Handling attribute annotations.

The BAnnotate Algorithm: Consider evaluating the following Alog rule r: $person(name, \langle age \rangle) :- q_1, \ldots, q_n$. This rule r defines a set of possible relations $\mathcal{R}$, each with attributes $(name, age)$. Suppose that we have compiled a plan fragment h for rule r, ignoring the attribute annotation in r for *age*. Then, when evaluating the plan $\psi_{(false,\{age\})}(h)$, the BAnnotate algorithm for the ψ operator takes the a-table produced by h and outputs another a-table that represents a superset of $\mathcal{R}$.

Suppose that h produces the a-table T_1, shown in Figure 5.a. Intuitively, because of the attribute annotation for *age* in rule r, any possible relation in $\mathcal{R}$ will have at most one tuple for each of the four distinct possible *name* values ("Alice", "Bob", "Carol", or "Dave").

Therefore, BAnnotate outputs the a-table T_2, shown in Figure 5.b, which has one a-tuple for each of the possible distinct *name* values in T_1. In each a-tuple $(name, age)$, *name* has one possible distinct name value n, and *age* is the set of all age values a that can be associated with n (i.e., the tuple (n, a) appears in at least one possible relation in $\mathcal{R}$). The first three a-tuples in T_2 (for the three names "Alice", "Bob", and "Carol") are maybe a-tuples because not all possible relations have tuples for those three names. However, the last a-tuple $(\{\text{"Dave"}\}, \{8, 9\})$ is not a maybe a-tuple because every possible relation in $\mathcal{R}$ will contain a tuple for "Dave" (either ("Dave", 8) or ("Dave", 9)).

In general, suppose we are evaluating an operator $\psi_{(f,A)}$ over an input a-table T with attributes $(a_1, \ldots, a_n)$. For simplicity, for now assume that $f = false$, and that $A = \{a_n\}$ (i.e., we are only considering one attribute annotation for attribute a_n). Then, we proceed in two steps.

In the first step, we create an index I over T. Each entry in I is a pair (n, V) where the key n is a tuple of values $(v_1, \ldots, v_{n-1})$, and V is a set of all possible values of v such that the tuple $(v_1, \ldots, v_{n-1}, v)$ appears in at least one possible relation represented by T. To build this index, for each a-tuple $t \in T$, we iterate through every possible tuple $(v_1, \ldots, v_n)$ that t represents, and add v_n to the values associated with the key $(v_1, \ldots, v_{n-1})$ in I (creating a new entry if necessary). For example, Figure 5.c shows the index we build for the a-table T_1, given an attribute annotation for *age*.

In the second step, we use I to construct the output a-table T'. First, we set T' to be initially empty. Then, for each entry (n, V) in I, where n is the tuple $(v_1, \ldots, v_{n-1})$, we add a new a-tuple t to T', where $t = (\{v_1\}, \ldots, \{v_{n-1}\}, V)$. Finally, we set t to be a maybe a-tuple if not all possible relations represented by input a-table T has a tuple for $(v_1, \ldots, v_{n-1})$. That is, t is a maybe a-tuple iff T' does not contain any a-tuple $t = (\{v_1\}, \ldots, \{v_{n-1}\}, U)$ for some non-empty set of values U. For example, in T_2 in Figure 5.b, the first output a-tuple $(\{\text{"Alice"}\}, \{5, 6, 7\})$ is a maybe a-tuple because the input table T_1 has no a-tuple $(\{\text{"Alice"}\}, U)$

for any set of values U. On the other hand, the a-tuple ({"Dave"}, {8, 9}) in T_2 is not a maybe a-tuple because a-table T_1 has an a-tuple ({"Dave"}, U), where $U = \{8, 9\}$.

Suppose we are evaluating $\psi_{(f,A)}$ over a-table T with attributes $(a_1, \ldots, a_n)$, where A has multiple attribute annotations. Without loss of generality, suppose $A = \{a_1, \ldots, a_k\}$. Then, we perform a process similar to the one above, except that we construct an index I_i for each attribute $a_i \in A$. Note that each index will have the same set of keys. Then, for each key n in the indexes we construct an output a-tuple. Let $n = (v_1, \ldots, v_k)$. Then, we construct the a-tuple $(\{v_1\}, \ldots, \{v_k\}, U_{k+1}, \ldots, U_n)$, where U_i is the set of values associated with key n in index I_i for $k < i \leq n$.

5. THE NEXT-EFFORT ASSISTANT

The next-effort assistant can suggest ways to refine the current IE program by asking the developer U questions such as "is *price* in bold font?". If U chooses to answer the question, iFlex forms a new constraint from the answer and incorporates the constraint into the IE program. We now discuss this process in detail.

5.1 Question Selection Strategies

The Space of Questions: We consider questions of the form "what is the value of feature f for attribute a?", where f is a text span feature (see Section 2.2.2). Example features include (but are not limited to) bold-font, italic-font, hyperlinked, preceded-by, followed-by, min-value, and max-value. For the above question, suppose that U answers v. Then iFlex adds the predicate $f(a) = v$ to the description rule that that "implements" the IE predicate that extracts attribute a. Thus, at any point in time the *space of possible questions* contain all (feature,attribute) questions whose answers are still unknown. We now discuss two strategies to select the next best question from this space.

Sequential Strategy: This strategy selects questions based on a predefined order over the question space. Currently, we rank the attributes in decreasing "importance", in a domain-independent fashion, taking into account factors such as whether an attribute participates in a join, commonly appears in a variety of Web pages, etc. Then given the ranked list $a_1, \ldots, a_n$, we first ask questions related to attribute a_1. Once these are exhausted, we move to a_2, and so on.

Simulation Strategy: This strategy selects the question that is expected to reduce by the largest amount the size of the current result. Consider a question d regarding feature f of attribute a. Let V be the set of possible values for f, and P be the current Alog program being developed. Also, let $g(P, (a, f, v))$ be a new Alog program that is the result of adding $f(a) = v$ to the rule in P that is extracting a. Then the expected number of results after asking d is

$$\sum_{v \in V} Pr[X \text{ answers } v | \text{asks } d] \cdot |exec(g(P, (a, f, v)))|,$$

where $|exec(g(P, (a, f, v)))|$ is the size of the result that would be obtained if iFlex executes the modified Alog program. We compute this quantity by carrying out the execution, effectively simulating the case that U answers v.

Each probability $Pr[\text{answers } v | d]$ is set to be $(1 - \alpha)/|V|$, where α is the probability that U chooses not to answer the question (e.g., by selecting the option "I do not know"). This makes the simplifying assumption that U gives each answer $v \in V$ with equal probability. We are currently examining how to better estimate these probabilities from the data being queried.

Simulating $|g(P, (a, f, v))|$ for all feature/value pairs can be costly. Hence, we optimize this process using both subset evaluation and reuse, as described in Section 5.2.

Domain	Data	Tables	Table Descriptions	Num Pages
Movies	3 pages	Ebert	Roger Ebert's Greatest Movies List	1
		IMDB	IMDB Top 250 Movies	1
		Prasanna	Prasanna Top Movies	1
DBLP	85 pages	Garcia-Molina	Hector Garcia-Molina Pubs List	1
		SIGMOD	SIGMOD Papers '75-'05	31
		ICDE	ICDE Papers '84-'05	22
		VLDB	VLDB Papers '75-'05	31
Books	749 pages	Amazon	Amazon query on 'Database'	249
		Barnes	Barnes & Noble query on 'Database'	500

Table 1: Real-world domains for our experiments.

Notifying the Developer of Convergence: To provide more effective interaction, the assistant detects and notifies U when the result for a query appears to have "converged" to the correct result. To do so, in each iteration it monitors both the number of tuples in the result set as well as the number of assignments produced by the extraction process. If these numbers remain constant for k iterations (currently set to 3), the assistant notifies U that convergence appears to have happened. Then, U can either stop or continue in more iterations until he or she is satisfied. In Section 6.2 we evaluate the effectiveness of this notification method.

5.1.1 Discussion

We now discuss general issues regarding the assistant.

Ease of Answering Questions: First, the assistant should ask questions that the developer can answer quickly and accurately. Toward this end, we have carefully designed the features $f_1, \ldots, f_n$ so that the resulting questions focus on the *appearance* (e.g., bold-font), *location* (e.g., does this attribute lie entirely in the first half of the page?), and *semantics* (e.g., what is a maximal value for price?) of the attributes. In our experiments we found that developers were able to answer these questions quickly and accurately (after some visual inspection; if unsure, the developer answered "I do not know").

More Types of Feedback: The assistant can be extended to handle more types of feedback, beyond question answering. For example, it can ask the developer to mark up a sample title. If this title is bold, then the assistant can infer that for the question "is title bold?", the answer cannot be "no" (but can be "yes" or "sometimes"). Hence, when searching for the next best question, the assistant does not have to simulate the case of the developer's answering "no" to this question, thus saving time. Exploiting other types of feedback effectively is an important future research direction.

5.2 Multi-iteration Optimization

Recall that during the development process, the developer iteratively refines and executes IE programs, and that in each iteration the assistant simulates program execution to search for promising questions for the developer. Consequently, optimizing program execution across iterations is critical to reduce development time as well as to maximize the utility of the assistant. For this goal, we employ two complementary solutions: reuse and subset evaluation.

The idea behind *reuse* is simple. When executing plan p in iteration n, we keep track of all intermediate results

Domain	IE Task	Description	Initial Program
Movies	T1	IMDB top movies with fewer than 25,000 votes	T1(title) :- IMDB(x), extractIMDB(x,title,votes), votes < 25000
	T2	Ebert top movies made between 1950 and 1970	T2(title) :- Ebert(x), extractEbert(x,title,year), 1950 ≤ year, year < 1970
	T3	Movie titles that occur in IMDB, Ebert, and Prasanna's top movies	T3(title1) :- IMDB(x), Ebert(y), Prasanna(z), extractIMDB(x,title1), extractEbert(y, title2), extractPrassana(z, title3), similar(title1,title2), similar(title2,title3)
DBLP	T4	Garcia-Molina journal pubs	T4(title) :- Garcia-Molina(x), extractPublications(x, title, journalYear), journalYear ≠ NULL
	T5	VLDB short publications of 5 or fewer pages	T5(title) :- VLDB(x), extractVLDB(x, title, firstPage, lastPage), lastPage < firstPage + 5
	T6	SIGMOD/ICDE pubs sharing authors	T6(title) :- SIGMOD(x), extractSIGMOD(x,title,authors1), ICDE(y), extractICDE(y,title,authors2), similar(authors1,authors2)
Books	T7	B&N books with price over $100	T7(title) :- Barnes(x), extractBarnes(x,title,b&n_price), b&n_price > 100
	T8	Amazon books whose list price equals the new price and used price is less than the new price	T8(title) :- Amazon(x), extractAmazon(x,listPrice,newPrice,usedPrice), listPrice = newPrice, usedPrice < newPrice
	T9	Books that are cheaper at Amazon than at Barnes	T9(title1) :- Amazon(x), extractAmazon(x,title1,newPrice), Barnes(y), extractBarnes(x,title2,b&n_price), similar(title1,title2), newPrice < b&n_price

Table 2: IE tasks for our experiments.

(e.g., intermediate compact tables). Let C_n be the set of these results. Then when executing p in iteration $(n+1)$, we reuse C_n to avoid re-extracting and re-executing from the scratch. To do so, we (a) examine the developer's feedback in iteration $(n+1)$ to find new attributes that this feedback "touches", (b) re-extract values for these attributes using C_n, then (c) re-execute the parts of plan p that may possibly have changed, again using C_n. We omit further details for space reasons.

We employ reuse in conjunction with a new technique, *subset evaluation*, for the purpose of selecting questions to suggest to the developer. The idea is to simply execute plan p over only a *subset* of the input documents, thus dramatically reducing execution time. Currently, the subset is created via random sampling (though more sophisticated creation methods are possible), and its size is set to be 5-30% of the original set, depending on how large this original set is. iFlex employs reuse as discussed above for both subset evaluation and full execution.

6. EMPIRICAL EVALUATION

We now describe experiments to evaluate the effectiveness of iFlex. We start with experiments on three real-world domains (Sections 6.1-6.2). Then, we describe experiments with the data of DBLife, a currently deployed IE application (Section 6.3).

Domains and IE Tasks: We considered three real-world domains for our experiments: Movies, DBLP, and Books. In each domain we first downloaded a set of Web pages (between 3-749). Next, for each domain we partitioned the pages into groups, where each group had related content. Then within each group we divided each page into a set of records (e.g., a fragment of an Amazon page describing information about one book) and stored the records as tuples in a table. Table 1 describes the characteristics of these tables.

Table 2 describes the nine IE tasks that we perform over the tables, and their initial Xlog program (before adding predicate description rules, annotations, and cleanup procedures). These tasks require extracting a variety of attributes, such as the title and year of movies from IMBD, Ebert, and Prasanna pages, the title, journal year, and page numbers of publications from various DBLP pages, and the title, and prices of books from Barnes and Amazon pages.

Methods: To evaluate iFlex, we compare it with Xlog, the method in which we write a precise Xlog program (and implement any necessary IE predicate using Perl code). As a sanity-check baseline, we also consider Manual, the method in which we manually inspect and collect the answers from the raw data.

Task	Num Tuples per Table	Manual	Xlog	iFlex
T1	10	1	28	1
	100	1	29	1
	250	3	29	1
T2	10	1	31	1
	100	1	31	1
	242	3	31	1
T3	10	1	58	1
	100	14	58	10 (8)
	242-517	80	58	16 (12)
T4	10	1	34	1
	100	2	34	1
	312	5	34	1
T5	100	4	37	1
	500	19	37	1
	2136	—	37	3
T6	100	76	55	6
	500	—	56	8
	1787-1798	—	57	23 (8)
T7	100	4	33	1
	500	20	33	1
	5000	—	33	8
T8	100	4	42	3
	500	19	43	4
	2490	—	43	5
T9	100	137	57	31
	500	—	57	34
	2490-5000	—	97	73 (6)

Table 3: Run time performance over 27 IE tasks.

6.1 Overall Performance

Table 3 shows the run time of the Manual, Xlog, and iFlex methods. The first column lists the nine IE tasks described earlier. The second column lists for each task three scenarios, where the first two scenarios require performing the IE tasks on a subset of the tuples (achieved via random sampling of the input pages), and the third scenario involves all tuples for each table.

For the 27 scenarios, the remaining columns show the run times of the three different methods. All times (averaged over 1-3 volunteers) are rounded to the minutes, and are counted from when the method is shown the Web pages, until when the correct result is produced (for non-iFlex methods) or when the next-effort assistant of iFlex notifies the developer that it has converged. In the iFlex column, the total run time is shown first, and the portion of that time that was spent writing cleanup code (if required) is shown in parenthesis.

Domain	Task	Num Tuples per Table	Num Correct Tuples	Num Tuples After Each Iteration (bold/italic entries when in reuse mode)										Num Questions Asked	Total Time (min)	Superset Size
				1	2	3	4	5	6	7	8	9	10			
Movies	T1	10	0	***10***	***0***									2	0.18	100%
	T2	100	31	18	18	18	***31***							4	0.37	100%
	T3	242-517	61	32	21	20	7	4	4	4	***98***			12	3.92	161%
DBLP	T4	10	5	***10***	***5***	***5***	***5***							6	0.48	100%
	T5	500	119	60	60	***119***	***119***	***119***						8	1.32	100%
	T6	500	318	199	199	66	66	1	1	1	***318***			12	7.72	100%
Books	T7	500	52	60	10	10	10	***52***						6	1.27	100%
	T8	2490	537	60	60	12	12	12	12	***537***			398	10	5.03	100%
	T9	100	45	900	865	515	473	471	398	18	18	18	***45***	16	30.81	100%

Table 4: Effects of soliciting domain knowledge in iFlex.

The results show that, as expected, Manual does not scale to large data sets. We stopped Manual in several cases (marked with "–" in the figure), after it became clear that the method was not scalable. Xlog scales better; it spent most of the time in writing and debugging the Perl code corresponding to the various IE predicates. However, iFlex achieves significantly better performance than Xlog, reducing run time by 25-98% in all 27 scenarios.

The run-time results without the cleanup time clearly suggest that iFlex can produce meaningful IE results quickly (much earlier than can be done with the precise-IE method Xlog).

The run-time results with the cleanup time suggest that iFlex can also produce precise IE results much faster than can be achieved with Xlog. This is due to two reasons. First, developers can declaratively write domain constraints quickly in iFlex, rather than spend time implementing these constraints in procedural languages, as is the case with Xlog. Second, the next-effort assistant can help the developers refine the IE programs effectively, as we discuss below.

6.2 Effectiveness of the Next-Effort Assistant

Recall that the next-effort assistant poses questions to the developer (to add constraints), and notifies the developer when the IE program has converged. We found that by just answering those questions, iFlex converged to the correct result in 23 out of 27 scenarios (described earlier). In the four remaining cases (not shown due to space limitation), it converged to 170%, 161%, 114%, and 102% of the correct result set, respectively. The cases of 170% and 161% occurred when the number of tuples returned was relatively small (22 and 98 tuples), which could be easily corrected with manual post-processing. The results suggest that the assistant can effectively help the developer refine a best-effort IE program and notify him or her of convergence.

In the next step, we evaluated the effectiveness of the next-effort assistant in iterating with the developer to solicit feedback. Table 4 shows iFlex's performance per iteration, in nine randomly selected IE scenarios. In each iteration we report (among others) the number of tuples in the result, and the execution method, i.e., subset evaluation, indicated by number in normal font, or reuse, by number in bold italic font. For example, task $T7$ over 500 tuples started in subset evaluation mode, producing 60 tuples for the first iteration. After interacting with the developer in only one iteration, iFlex reduced the result to 10 tuples. After two more iterations with no further reduction, iFlex converged, and switched to reuse mode to compute the complete result. Thus after only 6 questions in 5 iterations, iFlex produced the correct 52 tuples. The results in Table 4 suggest that the assistant solicited knowledge effectively, to converge in only a few iterations (2-10) to highly accurate result sets.

We also compared the sequential and simulation schemes for question selection. For each of the nine IE scenarios in Table 5, we measured the performance of iFlex using each scheme. In all cases sequential was faster, because question selection was very efficient (i.e., no simulation required). However, in four out of nine cases the questions asked via sequential selection were not nearly as useful for zooming in on the correct results as those asked via simulation. The extracted results were as much as 433 times larger than the correct results, suggesting that the better results obtained via simulation are well worth the additional cost when selecting questions.

Domain	Task	Tuples /Table	Correct Tuples	Guidance Scheme	Num Iterations	Questions Asked	Total Time (min)	Superset Size
Movies	T1	100	31	Seq	5	6	0.45	100%
				Sim	5	6	0.78	100%
	T2	100	31	Seq	4	4	0.28	100%
				Sim	4	4	0.37	100%
	T3	100	13	Seq	5	8	1.12	1762%
				Sim	6	10	2.18	170%
DBLP	T4	100	21	Seq	5	6	0.45	100%
				Sim	5	6	0.83	100%
	T5	500	119	Seq	4	6	0.70	100%
				Sim	5	8	1.32	100%
	T6	500	318	Seq	4	6	3.95	4243%
				Sim	8	12	7.72	100%
Books	T7	500	52	Seq	5	6	0.63	100%
				Sim	5	6	1.27	100%
	T8	500	155	Seq	5	6	0.82	233%
				Sim	7	10	3.72	100%
	T9	500	238	Seq	6	8	18.47	43299%
				Sim	10	16	33.82	100%

Table 5: Evaluating question selection strategies.

6.3 Evaluation on a Real-World System

Finally, we evaluated the practicality of iFlex by employing iFlex over one day's snapshot (10007 Web pages, 198 MB) of the crawled data of DBLife, a structured Web portal for the database community that we have been developing [7]. This data set was significantly more heterogeneous than those used in the above domains (Movies, DBLP and Books), and included a wide variety of Web pages such as personal homepages, conference homepages, and the DB-World mailing list.

We employed iFlex to write three IE programs (described in Figure 6) that correspond to similar programs in DBLife. We found that iFlex was flexible enough to accommodate relatively complex IE tasks over heterogeneous data such as those found in DBLife. A key reason for this is that iFlex provides a rich library of built-in text features that includes not only "syntactic" features (e.g., numeric, bold-font), but also "higher-level" features. Examples of such features include prec-label-contains which indicates whether the preceding label (i.e., a header of a section in the text) of a span contains a certain string, and in-list that indicates whether a span is part of a list. For instance, to complete the program for the panel task (Figure 6), we implemented the two following predicate descriptions:

$$extractPanelist(\overline{d}, x) : -match(\overline{d}, \overline{p}, x), personPattern(p),$$
$$prec\_label\_contains(x, \text{“}panel\text{”}) = yes,$$
$$prec\_label\_max\_dist(x) = 700$$

Task	Description	Query	iFlex (min)
Panel	Find (x,y) where person x is a panelist at conference y	onPanel(x,y) :- docs(d), extractPanelists(d,x), extractConference(d,y)	54 (5)
Project	Find (x,y) where person x works on project y	worksOn(x,y) :- docs(d), extractOwner(d,x), extractProjects(d,y)	44 (6)
Chair	Find (x,y,z) where person x is a chair of type y at conference z	chair(x,y,z) :- docs(d), extractChairs(d,x), extractConference(d,y), extractType(x,z)	60 (11)

Table 6: Experiments on DBLife data.

$$extractConf(\overline{d}, y) :- from(\overline{d}, y), inTitle(\overline{y}) = yes, starts\_with(\overline{y}, \text{“[A-Z][A-Z][A-Z]+”}), ends\_with(\overline{y}, \text{“0\d|19\d\d|20\d\d”}), max\_length(\overline{y}) = 18$$

Figure 6 shows that developing IE programs that produce exact IE results took 44-60 minutes each, using iFlex. In contrast, developing comparable precise-IE IE programs in DBLife, using Perl scripts, took 2-3 hours (we do not keep track of the exact amount of time, as they were developed several years ago). This result further suggests that iFlex can help reduce the time it takes to develop IE programs.

For the above three IE tasks, the final IE programs obtained with iFlex took 104, 351, and 107 seconds to run, respectively. These times were comparable to the execution times of the corresponding IE programs in DBLife (developed in Perl and tuned extensively several years ago). While anecdotal, this result does suggest that the approximate query processor proves quite efficient even on large data sets.

7. RELATED WORK

Information extraction has received much attention (e.g., [13, 12, 5, 21, 8, 18], see also [1, 11] for recent tutorials), and many solutions have been proposed to make developing IE programs easier. These include compositional frameworks [12, 5] as well as declarative languages [13, 21]. However, these solutions are difficult to use for best-effort IE because they provide little or no support for writing approximate IE programs with well-defined semantics, as iFlex does. Notable exceptions are [9, 13, 8]. These works propose declarative languages to write IE programs that may produce approximate results. These languages however do not have an explicit possible-worlds semantics, as Alog does.

Recent work has shown how to generate and represent approximate IE results from probabilistic graphical models (e.g. CRFs), using a model similar to a-tables [14]. In contrast, iFlex produces approximate IE results using a declarative language, and uses compact tables to efficiently represent highly approximate extracted data that often arises in best-effort IE. Many other works have studied representing and processing approximate or uncertain data in general (e.g., [3, 2, 6, 4, 15]). However, they have not considered the text-specific challenges we address, such as representing the large number of possible extracted values a best-effort IE program may produce.

The next-effort assistant of iFlex is similar in spirit to soliciting user feedback in a dataspace system, as described in [16]. Also, the iFlex system builds on an earlier work on approximate wrapper processing [17]. Finally, the CONTROL project [15] has also studied interactive query processing, but for data analysis in relational settings, not for IE over text as in our current work.

8. CONCLUSION AND FUTURE WORK

Despite recent advances, writing IE programs remains a difficult problem, largely because developers often try to obtain precise IE results. To address this problem, we have proposed iFlex, an approach that relaxes the precise-IE requirement to enable best-effort IE. We have described how developers can use iFlex to quickly write an initial approximate IE program P, obtain approximate results quickly, then refine P to obtain increasingly more precise results. As a near-term future work we plan to consider more expressive languages and data models for approximate IE. For the longer term, it would be interesting to consider how iFlex can be applied to other contexts, such as best-effort data integration [10] that seeks to provide approximate answers over a set of data sources. This context can potentially benefit from the ideas of best-effort IE, query processing, and developer interaction described in iFlex.

9. REFERENCES

[1] E. Agichtein and S. Sarawagi. Scalable information extraction and integration. In *KDD-06*.

[2] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE-08*.

[3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB-06*.

[4] S. Chaudhuri, G. Das, and V. R. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2):9, 2007.

[5] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *ACL-2002*.

[6] N. Dalvi and D. Suciu. Answering queries from statistics and probabilistic views. In *VLDB-05*.

[7] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured Web data portals: a top-down, compositional, and incremental approach. In *VLDB-07*.

[8] Y. Ding, D. W. Embley, and S. W. Liddle. Automatic creation and simplified querying of semantic Web content: An approach based on information-extraction ontologies. In *ASWC-06*.

[9] Y. Ding, D. W. Embley, and S. W. Liddle. Enriching OWL with instance recognition semantics for automated semantic annotation. In *ER Workshops*, 2007.

[10] A. Doan. Best-effort data integration (position statement). In *NSF/EPA/ONR/NARA/AHRQ/NCO Workshop on Data Integration*, 2006.

[11] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction: State of the art and research directions. In *SIGMOD-06*.

[12] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348, 2004.

[13] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The Lixto data extraction project: Back and forth between theory and practice. In *PODS-04*.

[14] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB-06*.

[15] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The CONTROL project. *Computer*, 32(8):51–59, 1999.

[16] S. Jeffery, M. Franklin, and A. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD-08*.

[17] R. McCann. “Efficient data integration: Automation, collaboration, and relaxation,” Ph.D. dissertation, University of Illinois, Urbana-Champaign, 2007. [Online]. Available: *http://dblife.cs.wisc.edu/people/mccann/papers/phd_thesis.pdf*

[18] B. Rosenfeld and R. Feldman. High-performance unsupervised relation extraction from large corpora. In *ICDM-06*.

[19] A. D. Sarma, O. Benjelloun, A. Y. Halevy, and J. Widom. Working models for uncertain data. In *ICDE-06*.

[20] W. Shen, P. DeRose, R. McCann, R. Ramakrishnan, and A. Doan. Towards best-effort information extraction. Technical report, 2008.

[21] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB-07*.