Declarative Information Extraction Using Datalog with Embedded Extraction Predicates

Warren Shen[1], AnHai Doan[1], Jeffrey F. Naughton[1], Raghu Ramakrishnan[2]
[1]University of Wisconsin-Madison, [2]Yahoo! Research

ABSTRACT

In this paper we argue that developing information extraction (IE) programs using Datalog with embedded procedural extraction predicates is a good way to proceed. First, compared to current ad-hoc composition using, e.g., Perl or C++, Datalog provides a cleaner and more powerful way to compose small extraction modules into larger programs. Thus, writing IE programs this way retains and enhances the important advantages of current approaches: programs are easy to understand, debug, and modify. Second, once we write IE programs in this framework, we can apply query optimization techniques to them. This gives programs that, when run over a variety of data sets, are more efficient than any monolithic program because they are optimized based on the statistics of the data on which they are invoked. We show how optimizing such programs raises challenges specific to text data that cannot be accommodated in the current relational optimization framework, then provide initial solutions. Extensive experiments over real-world data demonstrate that optimization is indeed vital for IE programs and that we can effectively optimize IE programs written in this proposed framework.

1. INTRODUCTION

Suppose an application must extract structured information (e.g., person names, paper titles, meetings) from a large set of text documents. How should we develop and execute such extraction (IE) programs? Given the wide variety of real-world applications that must extract information from text (e.g., [11, 13], see [3, 9, 12] for recent tutorials), providing efficient support for developing and executing IE programs has become increasingly important.

Unfortunately, even though several approaches have been widely used to develop IE programs, none of them has been very satisfying. Perhaps the most straightforward approach is to employ an off-the-shelf, monolithic IE "blackbox." This approach however severely limits the expressiveness of IE programs that can be developed. Hence, the most popular approach today is to decompose an IE task into smaller subtasks, apply off-the-shelf IE blackboxes (whenever appropriate) or write hand-crafted code to solve each subtask, "stitch" them together (e.g., using Perl, Java, C++), then perform any necessary final processing. This approach is very powerful, but generates large IE programs that are very difficult to understand, debug, modify, and optimize.

In response, a recent approach has proposed *compositional* frameworks for developing IE programs. A prime example of such frameworks is UIMA [14]. UIMA proposes an "object-oriented" language with standard object APIs. This language allows developers to code each IE subtask as an extraction object, then compose new extraction objects from existing objects. UIMA represents each IE program as a workflow whose nodes communicate via some communal "blackboards." UIMA-like languages can make writing, debugging, and modifying IE programs much easier. But optimizing IE programs written in such languages has proven to be difficult, and so far no solution has been proposed, to the best of our knowledge.

Optimization, however, is vital for any practical approach to developing and executing IE programs. Many real-world applications increasingly must run complex IE programs over large data sets, and unoptimized programs often take days or weeks to run, an unacceptably long time. As a concrete example, consider DBLife at *dblife.cs.wisc.edu*, a structured portal for the database community that we have been developing [11]. For certain IE tasks in DBLife our unoptimized IE programs (written by stitching together multiple modules) took more than a day over 110+ MB of data, an unacceptable runtime because we have to rerun the IE programs every single day (over newly crawled data) to keep DBLife fresh. Similar stories are reported at IBM Almaden [22] and Yahoo! Research, and the importance of IE optimization was also underscored in a recent KDD tutorial [3]. Optimization turns out to be crucial even when IE programs take only hours to run, because it enables fast response time for debugging and development.

Given the limitations of current approaches, in this paper we argue that developing IE programs using Datalog with embedded procedural predicates is a good way to proceed. By this we mean Datalog programs that allow user-defined predicates that are pieces of procedural code (e.g., in Perl, Java, C++). Figure 1.a shows a tiny such program, which extracts all titles and associated abstracts of talks from text documents. Here $docs(d)$ is an extensional predicate, as defined in traditional Datalog. The other predicates however are *procedural predicates*: pieces of code cleanly encapsulated

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

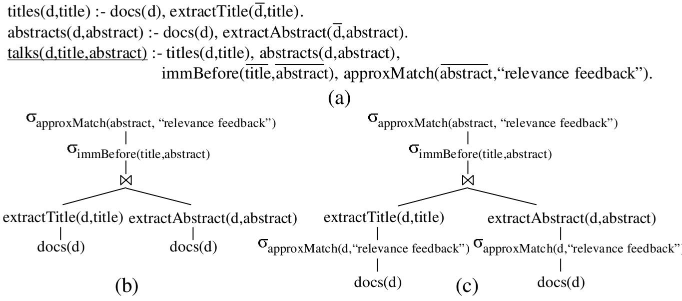


Figure 1: (a) An IE program in procedural Datalog, and (b)-(c) its execution plans on two different data sets.

and "plugged" into the Datalog rule.

We argue that Datalog with embedded procedural predicates is a good way to proceed for two important reasons. First, as we mentioned earlier, people already write their IE programs as collections of small programs that they stitch together in ad-hoc ways. In our framework, these small programs become the procedural predicates, and Datalog becomes a much cleaner and more powerful way to combine them than ad-hoc stitching. Thus, writing IE programs this way retains and enhances the important advantages of current approaches, such as being easy to understand, debug, and modify.

The second important reason is that once we write IE programs in this framework, we can apply query optimization techniques to them. This gives programs that, when run over a variety of data sets, are more efficient than any monolithic program because they are optimized based on the statistics of the data on which they are invoked. The following example illustrates this point.

EXAMPLE 1.1. *Consider again the IE program p in Figure 1.a. Program p may be compiled into either the execution plan q_1 of Figure 1.b or q_2 of Figure 1.c. Given a document d, plan q_1 extracts all titles and abstracts from d, then keep only those (title,abstract) pairs where the title occurs immediately before the abstract. Finally, q_1 retains only talks whose abstracts contain "relevance feedback" (allowing for misspellings and synonym matching). Plan q_2 is similar to q_1, except that it discards a document d as soon as it finds out that d does not contain "relevance feedback", on the ground that if d does not contain that phrase, then abstracts extracted from d also cannot contain that phrase.*

At first glance, it appears that q_2 is more efficient than q_1 because it can prune irrelevant documents early. However, this truly depends on the selectivity of the selection operator $\sigma_{approxMatch(d,''relevance\ feedback'')}$ and the runtime cost of approxMatch. If a data set mentions "relevance feedback" frequently (e.g., SIGIR proceedings), then the selection selectivity is low. Since approxMatch is expensive, q_2 can end up being significantly worse than q_1. On the other hand, if a data set rarely mentions "relevance feedback" (e.g., SIGMOD proceedings), then q_2 can significantly outperform q_1. Given IE programs written in our framework, we can generate such execution plans and select the appropriate one, depending on the statistics collected over the data set.

In the rest of the paper we elaborate on the above two reasons. We start by describing Xlog, an example of Datalog with embedded procedural predicates, tailored for information extraction. Next, we show that stitched-together IE programs can be quickly written in Xlog, yielding programs that mix procedural and declarative parts in a clean fashion. We then show how to compile such a program into an execution plan.

Next, we turn our attention to query optimization. Since we are dealing with text, our optimization challenges are somewhat different from the relational case in terms of (a) what optimizations should be considered, (b) what statistics should be gathered to support these optimizations, and (c) what cost model should be used to choose alternatives. To address these issues, we begin by showing that CPU time often dwarfs IO time and hence plays a significant role in IE programs. To reduce CPU time, we exploit three kinds of text-centric actions that we observe IE programs often execute. Specifically, we propose three optimization techniques: *pushing down text properties* prunes a text span as soon as we detect that it cannot be useful for subsequent extraction, *scoping extraction* significantly narrows text regions from which we extract certain data, and *indexing patterns* allows us to match a large number of text patterns efficiently.

We then describe a cost model that focuses on CPU time, show how to collect statistics to estimate plan cost, then show how to find a plan that employs the above optimization techniques to minimize runtime.

Finally, we present extensive experiments over real-world data that demonstrate that optimization is indeed vital for IE programs and that we can drastically optimize IE programs written in this framework, often cutting their runtime by orders of magnitude. We also briefly discuss experiments with these techniques in DBLife, our live data portal.

In summary, our contributions are as follows:

- While Datalog with embedded procedural predicates is not new, we show that it does provide a natural framework to write IE programs. This framework retains and enhances the advantages of current extraction approaches in ease of debugging, modifying, and understanding.
- Unlike current extraction approaches, we show that this framework is highly amenable to query optimization. In particular, we describe three optimization techniques that exploit text-centric actions that IE programs often execute.
- We show how to estimate the plan cost (runtime), collect statistics, and find a plan with minimal cost.
- We present extensive experiments that demonstrate that optimization is indeed vital for IE programs, and that we can significantly optimize IE programs written in our framework, often by orders of magnitude.

2. RELATED WORK

Information Extraction: Information extraction from text has received much attention in the database, AI, Web, and KDD communities (e.g., [24, 25, 27, 28], see [3, 9, 12] for recent tutorials). The vast majority of works improve extraction accuracy (e.g., with novel techniques such as HMM and CRF [9]), while some recent pioneering works improve extraction time (e.g. [6, 18]). Most works develop basic IE solutions that extract a single type of entity (e.g., person names, publication title) or relation (e.g., advising, giving talk). IE developers then commonly combine these solutions

– often as off-the-shelf IE "blackboxes" – with additional procedural code into larger IE programs. Since such programs are rather difficult to develop, understand, and debug, recent works have developed compositional IE frameworks, such as UIMA and GATE [10, 14]. UIMA for example models extraction tasks as objects and standardizes object APIs to enable "plug and play."

UIMA-like languages can make writing and understanding IE programs much easier. However, optimizing IE programs in such languages has proven to be difficult. A major reason is that their IE semantics is often not well-defined. For example, it is unclear whether an IE program in such languages produces the same result if we change the execution order of certain rules or parts of the program workflow. Sometimes it is also unclear what the program would produce if it involves certain recursions. This makes it difficult to optimize, because we do not know what types of optimization will have what effect on the program result. Our work avoids this problem by building on the well-understood semantics of Datalog.

Our work also relates to wrapper construction from Web pages [19, 24]. Like UIMA and GATE, many works here have also proposed rule-based or compositional solutions, to enable fast development of wrappers. However, they have not considered optimizing runtime, as our current work does.

Very recently, the work [22] also considers optimizing large-scale IE programs. It proposes an algebraic approach and considers text-centric optimizations that are complementary to those considered in this paper.

Datalog Extension: The LIXTO project [16] has also developed a Datalog extension to represent IE programs, but only in the context of wrapper construction. LIXTO focuses on rapidly developing wrappers, not on optimizing runtime, as we do. Its Datalog extension functions only as an internal language for the wrapper system, not as a language for the developers. Consequently, it does not allow embedded procedural predicates written by developers, as ours does. Recent works have also proposed Datalog extensions for a variety of problems, including networking [20], software analysis [26], diagnosis of distributed systems [1], and others [5, 21]. Our work shares the same spirit with these works in striving to make the applications more declarative, but focuses specifically on information extraction.

Optimization: Several recent works (e.g. [6, 18]) have developed optimization techniques for IE, and have considered optimizing extraction accuracy as well as runtime. However, they consider only restricted IE settings, such as those with a single IE "blackbox." In contrast, we consider optimizing IE programs that often involve multiple IE "blackboxes", with complex interaction among them, all within a declarative framework. The work [18] introduces a method to prune useless documents that do not contain certain words. Our technique of pushing down text properties is more general in that it can prune at finer granularity (e.g., text spans created during the IE process) as well as in many novel contexts (e.g., those involving length conditions). The work [8] quickly matches patterns against text documents by building an inverted index over the *documents* to reduce the number of documents considered for each pattern. In contrast, we index the *patterns* to reduce the number of patterns we consider for each document. Finally, our work also relates to optimization techniques for memory databases (since IE programs often operate largely in memory) [15, 23], and user-defined predicates (e.g. [7]). These works however do not consider text-centric optimization techniques, as we do in this paper.

3. DATA AND QUERY MODEL

We now describe Xlog, as an example of Datalog languages with embedded extraction predicates. We then use Xlog to highlight the advantages of such languages over current approaches for developing IE programs.

We start by briefly reviewing traditional Datalog [2]. A Datalog program P consists of multiple rules. Each rule has the form $p :- q_1, \ldots, q_n$, where the p and q_i are predicate atoms. Each predicate atom is a predicate symbol applied to its arguments, which are either constants or variables. The rule must be *safe* in that any variable appearing in the head (i.e., p) must also appear as an argument in the body (i.e., $q_1, \ldots, q_n$). Currently we do not consider rules with negated predicates nor with recursion, leaving such extensions as future work.

Each predicate atom in a rule is associated with a relation. We will use the terms "predicate atom", "predicate", "relation", "relation instance", and "table" interchangeably, when there is no ambiguity. A predicate is *extensional* if its table has been provided to the Datalog program P. Otherwise, it is *intensional*, and its table must be computed using the rules in P. One of the head predicates of P is called a *query*, in that its table is viewed as the output of P.

As an example, the following Datalog program returns conflict-of-interest (COI) cases where two authors have written a paper in the past two years:

$$\begin{aligned} coAuthor(a,b,y) \quad &:- \quad write(a,p), write(b,p), paper(p,y). \\ \underline{coi(a,b,y)} \quad &:- \quad coAuthor(a,b,y), y \geq 2005. \end{aligned}$$

Here predicates $write(a,p)$ and $paper(p,y)$ are extensional, whereas $coAuthor(a,b,y)$ and $coi(a,b,y)$ are intentional. The first rule returns tuples (a,b,y) where a co-authors with b in year y. The second rule retains only co-author pairs where $y \geq 2005$. Note that the query predicate $coi(a,b,y)$ is denoted as $\underline{coi(a,b,y)}$.

3.1 The XLog Language

We now describe Xlog, using the tiny example in Figure 2, which we will explain along the way.

Syntax: A key distinction of our setting, compared to traditional Datalog, is that many steps of real-world IE is inherently *procedural*. Furthermore, even when certain steps coded as procedural can be made declarative, developers are often reluctant to do so, because it would incur too much effort. Consequently, to capture such steps, we introduce the notion of a procedural predicate.

DEFINITION 1 (PROCEDURAL PREDICATE). *Such a 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 side-effect-free procedure g (e.g., written in C, Java) that takes as input a 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 $(u_1, \ldots, u_n, v_1, \ldots, v_m)$. Note that all tuples in this set agree on the first n elements: $u_1, \ldots, u_n$. Let the union of all such sets of tuples be r. Then we say that q is associated with relation r.*

We will often refer to q as a *p-predicate*, to the a_i as *bound* or *input arguments*, to the b_i as *free* or *output arguments*,

R_1: titles(x,d) :- docs(d), lines($\overline{d}$,x,n), allCaps($\overline{x}$)=true, n<5.
R_2: names(y,d) :- docs(d), seedNames(s), namePatterns($\overline{s}$,p), match($\overline{d}$,$\overline{p}$,y).
R_3: authors(y,d):- docs(d), titles(x,d), names(y,d), distLine($\overline{x}$,$\overline{y}$)<3.

(a)

docs
d_1
d_2

(b)

d_1: DATABASES / D. Smith / Dr. Smith is a professor at …

d_2: SQL QUERIES / Jane Brown / Brown, J. *Databases*. 1998

(c)

seedNames
David Smith
Jane Brown

(d)

Figure 2: (a) A sample Xlog program, and (b)-(d) sample data for the program.

and to g as a *p-procedure*. We will also use the terms "p-predicate" and "p-procedure" interchangeably when there is no ambiguity.

P-predicates are either built into Xlog, or provided by the developer, with the types of their arguments specified. For example, Figure 2 shows three such predicates: $lines(\overline{d}, x, n)$, $namePatterns(\overline{s}, p)$, and $match(\overline{d}, \overline{p}, y)$. The predicate $lines(\overline{d}, x, n)$ for instance takes as input a document d, and returns the set of all lines x in d, together with a number n indicating that the line is the n-th line.

We define procedural functions, or *p-function* for short, in a similar fashion. Figure 2 shows two such functions: $allCaps(\overline{x})$ and $distLine(\overline{x}, \overline{y})$. The function $allCaps(\overline{x})$ takes a line x and returns "true" if all characters in the line are capitalized. Given two text spans x and y in a document (see precise definitions below), $distLine(\overline{x}, \overline{y})$ returns the distance between x and y, measured in lines.

The above examples suggest that a rule can be properly evaluated only if its p-predicates and p-functions can obtain their inputs. We call such rules *well-formed.* Formally,

Definition 2 (Well-formed rule). *An* Xlog *rule $p :- q_1, \ldots, q_n$ is well-formed if we can order $q_1, \ldots, q_n$ such that for each literal q_i, $i \in [1, n]$ its bound arguments, if any, already appear as free arguments of some literals preceding q_i in the ordering. Note that extensional and intentional predicates contain only free arguments.*

Another distinguishing aspect of our setting is that we deal heavily with text. Hence, we make explicit several text-related notions: strings, documents, spans, containment, and IE predicates. Later we show how to exploit text properties involving these notions in optimizing an Xlog program.

Definition 3 (String, document, and span). *A string s is a sequence of characters. A document d is a tuple (id,content) where id is a key, and content is a string that represents the text content of the document. A span t is a string within a document. Formally, t is represented as a tuple (id,doc,start,end) where id is a key for t, doc is the id of the document, and start and end refer to the starting and ending positions of t in that document.*

In Figure 2, d is a document variable, x is a span variable (it refers to lines in documents), and so is y.

Definition 4 (Containment). *We say a document d contains a span s iff s.doc = d.id (where notation a.b refers to element a of b). A span t contains a span s iff t.doc = s.doc, t.start $\leq$ s.start, and t.end $\geq$ s.end.*

Definition 5 (IE predicate). *An IE predicate q extracts one or more output spans from a single input span. Formally, q is a p-predicate $q(\overline{a_1}, \ldots, \overline{a_n}, b_1, \ldots, b_m)$, where there exist i and j such that (a) a_i is either a document or a span variable, (b) b_j is a span variable, and (c) for any output tuple $(u_1, \ldots, u_n, v_1, \ldots, v_n)$, u_i contains v_j (i.e., q extracts span v_j from span u_i).*

In Figure 2, $lines(\overline{d}, x, n)$ is an IE predicate which, when given a document d, returns all lines of d, together with their numbering n. Also, $match(\overline{d}, \overline{p}, y)$ is an IE predicate that extracts all spans y (in a document d) that match an input pattern p. On the other hand, $namePatterns(\overline{s}, p)$ is not an IE predicate, because the sole input variable s is not a document nor a span.

Finally, we define a special extensional predicate $docs(d)$ that we call the *document predicate*. The $docs(d)$ predicate contains all documents (e.g., Web pages, emails, reports, etc.) from which we want to extract information. We now can define Xlog as follows.

Definition 6 (Xlog program). *An* Xlog *program P consists of a set of Datalog rules where (a) rules can involve p-predicates and p-functions, (b) rules are safe and well-formed, (c) document predicate docs(d) appears in one or more rules, and (d) all arguments a_i of the query predicate $r(a_1, \ldots, a_k)$ are span variables. We say that P extracts relation $r(a_1, \ldots, a_k)$ from the collection of documents docs(d).*

Semantics: We can define the semantics of a rule in Xlog in the same manner it is defined in traditional Datalog, using associated relations. For instance, the least-model semantics of Datalog states that whenever we replace each variable in a rule r with a corresponding constant, if every tuple in the rule body (obtained via the replacements of the variables) belongs to the corresponding relation instance, then the tuple generated for the rule head (again via the replacements) also belongs to the corresponding relation. This semantics applies directly to Xlog, given that we have defined the notion of associated relations for p-predicates and p-functions (see Definition 1). We have

Proposition 1. *If p-predicates and p-functions are associated with finite relations, then the semantics of an* Xlog *program is well-defined, in that the set of output tuples it produces is finite and unique.*

Example 3.1. *To illustrate* Xlog*, we step through a conceptual execution of the* Xlog *program P in Figure 2. Section 4 describes how to generate execution plans for such* Xlog *programs. Consider executing P over the data of Figure 2. Figure 2.b shows a table docs with two documents d_1 and d_2, whose contents are shown in Figure 2.c. For each document d_i, rule R_1 extracts its title, if any. R_1 first calls p-predicate $lines(\overline{d}, x, n)$ (i.e., calls its procedure) to segment d_i into lines. It then retains only lines that are fully capitalized and are within the first four lines (i.e., $allCaps(\overline{x}) =$ true and $n < 5$). The result is a table $title(x, d)$ with two tuples ("DATABASES", d_1) and ("SQL QUERIES", d_2).*

Rule R_2 then extracts all person names from the documents. First, it takes each seed name s in table seedNames (Figure 2.d) and feeds the name into $namePatterns(\overline{s}, p)$ to generate possible variants of s. For example, given "David Smith", $namePatterns(\overline{s}, p)$ may generate "David Smith", "D. Smith", "Smith, D", "Dr. Smith", etc., each of which is referred to as a pattern. Next, for each document d_i,

```
people(d,personMention) :-  docs(d), personPatterns(personPattern),
                            match(d, personPattern, personMention).

conferences(d,conferenceMention) :- docs(d), confPatterns(confPattern),
                                    match(d, confPattern, conferenceMention).

chairType(d,chairType,chairPosition) :- docs(d), chairTypePatterns(chairTypePattern),
                                        match(d, chairTypePattern, chairType),
                                        match(d, "(?i)(vice\W+)?(co-)?chair", chairPosition),
                                        isBefore(chairType, chairPosition),
                                        distChar(chairType, chairPosition) < 20.

chair(d,personMention,conferenceMention,chairPosition,chairType) :-
                          people(d, personMention), conferences(d, conferenceMention),
                          chairType(d, chairType, chairPosition),
                          isBefore(conferenceMention, chairType),
                          isBefore(chairPosition, personMention),
                          distChar(chairPosition, personMention) < 20.
```

Figure 3: A sample Xlog program in our experiments.

$match(\overline{d}, \overline{p}, y)$ finds occurrences of the above patterns in d_i. In d_1 for example, $match(\overline{d}, \overline{p}, y)$ may find "D. Smith" and "Dr. Smith". These occurrences are returned as person names in variable y.

Finally, rule R_3 examines all pairs of titles and person names that occur in d_i, and retains only names that occur within 2 lines of a title. These names are returned as the authors of document d_i. The final output are tuples ("D. Smith", d_1) and ("Jane Brown", d_2).

Extension with Bulk P-Predicates: So far we have assumed that each p-predicate q executes in a *singleton* manner, in that each invocation of q takes as input a *single value* for each bound argument. In practice, however, for performance reasons developers often want to implement certain p-predicates so that each invocation takes as input *multiple values*, in a *bulk* manner.

To see why, consider evaluating rule R_2 in Figure 2. If the p-predicate $match(\overline{d}, \overline{p}, y)$ is singleton, then we must invoke *match* multiple times, once for each possible combination of document d and pattern p. In contrast, if *match* is a bulk p-predicate, then we only need to invoke *match* once, with the input being all documents d in $docs(d)$ and all patterns p generated by *namePatterns*, and the output being all occurrences of p in all documents d. In this case, since *match* gets access to documents and patterns all at once, it can execute certain optimizations that the singleton version cannot (e.g., indexing the set of patterns to avoid unnecessary pattern matching, see Section 5.4).

For these reasons, we allow developers to implement p-predicates as either singleton or bulk (or supply both implementations). Formally, we have

DEFINITION 7 (BULK P-PREDICATE). *Consider a p-predicate $q(\overline{a_1}, \ldots, \overline{a_n}, b_1, \ldots, b_m)$ with an associated procedure g. The bulk version of q is associated with a procedure g' such that g' takes input relations $r_1, \ldots, r_n$ and produces as output a relation $r = \cup_{(u_1,\ldots,u_n) \in r_1 \bowtie \cdots \bowtie r_n} g(u_1, \ldots, u_n)$.*

It is easy to show that under the above definition, the semantics of Xlog remains the same regardless of using singleton or bulk versions.

Per-Document IE Tasks: IE tasks can be classified into two groups. The first group extracts from *each document in isolation*, e.g., extracting authors of a document as in Figure 2, or publications from a Web page. The second group extracts *across documents*, e.g., finding pairs (a, b) where a is a professor homepage and b is the homepage of a class taught by a.

Per-document IE tasks are pervasive (e.g., constituting 94% of IE tasks in the current DBLife system). Hence, as a first step, in this paper we consider such IE tasks, leaving across-document IE tasks for future research.

3.2 Benefits of Xlog-like Languages

We have applied Xlog to several IE tasks in the DBLife system [11], and found it relatively easy to use. Given an IE task, we first decomposed it into smaller tasks. For example, we decomposed finding authors of documents into (a) finding titles, (b) finding names, then (c) combining titles and names and keeping those satisfying certain criteria.

Next, we wrote an Xlog program that reflects the above decomposition, utilizing a set of built-in predicates and functions, and "making up" other predicates and functions as we went along. For example, we wrote rules $R_1 - R_3$ in Figure 2, where *lines*, *allCaps*, *match*, and *distLine* are built-in predicates and functions, and *docs* and *seedNames* are extensional predicates, defined over some tables. We made up p-predicate *namePatterns*. Finally, we implemented made-up predicates and functions (e.g., *namePatterns* as a Java procedure). We iterated over the above steps until achieving a satisfactory solution. Figure 3 shows a small Xlog program that we wrote for DBLife and used in our experiments.

The above process suggests that Xlog is highly flexible. Clearly any IE task can be expressed in Xlog, because in the extreme we can just implement the entire task as a single p-predicate. However, the more we decompose the IE task (into smaller "pieces" and "stitching" them together using Xlog), the more declarative the code becomes, the more we can save on coding labor (e.g., by re-using built-in predicates and functions), the easier it is to debug and understand the code, and the more opportunities we have for optimization.

The current Xlog implementation already has a substantial set of built-in p-predicates and p-functions that capture common text related tasks. Figures 2-3 show examples of such predicates and functions, and we omit their descriptions for space reasons. However, we note that the current set already allows us to write powerful IE tasks (see the experiment section), and that this set is highly extensible, as new predicates and functions become available.

Besides ease of use, another benefit of Xlog is a clean semantics based on the well-understood Datalog semantics. This is important as we seek to optimize Xlog programs, or to extend the language. For instance, while it has been difficult to understand the semantics of many current IE languages in the presence of recursion (which occurs in many IE settings), extending Xlog to recursion should still result in a well-defined language, based on the recursion semantics of Datalog.

Finally, we note that, while not considered in depth in this paper, IE programs in Xlog-like languages can also potentially benefit from the wealth of relational technologies developed in the past thirty years, to handle storage, query processing, indexing, and optimization, all crucial issues to large-scale IE development.

4. GENERATING EXECUTION PLANS

Given an Xlog program P, we now discuss how to create a default physical execution plan h for P. (Section 5 shows how to optimize h.) We begin by creating a logical plan fragment for each rule in P, then combine these fragments into a logical plan f for P. Next, we elaborate on f to obtain a physical plan e. Finally, we modify e into h to

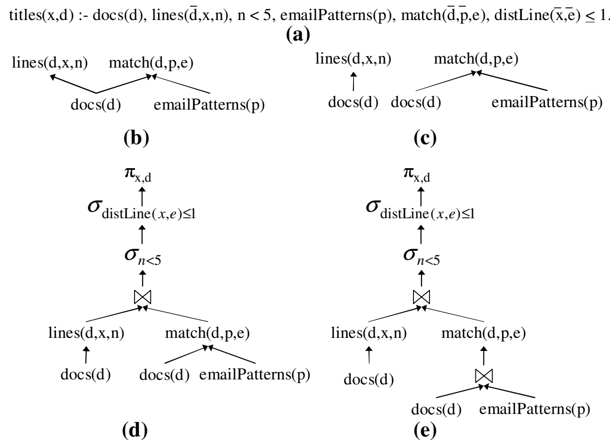


Figure 4: Generating a logical plan fragment for an Xlog rule.

take advantage of the per-document IE setting. Below we explain these steps, using the program of the single rule in Figure 4.a.

1. Create Plan Fragments for Rules: Let r be a rule $p : -q_1, \ldots, q_n$. A logical plan fragment for r is a tree T_r, whose nodes are predicates in $q_1, \ldots, q_n$ or relational operators, and whose edges denote the input dependencies across the nodes.

To construct T_r, first we create a graph G that captures the input dependencies of the p-predicates in r. G has n nodes, one for each predicate q_i, and a directed edge from q_i to q_j if q_j "needs" some data from q_i (i.e., there is a bound argument x in q_j that appears in q_i). Figure 4.b shows graph G for the rule in Figure 4.a. Note that we exclude from G all selection predicates, which are $(n < 5)$ and $(\text{distLine}(\overline{x}, \overline{e}) \leq 1)$ in this case.

Next, we convert G into a forest of trees, by duplicating nodes that have more than one parent. Figure 4.c shows the forest created for our example. We then join the trees in the forest (using join operators) in some arbitrary order, and add the necessary selection and projection operators, to form plan fragment T_r. Figure 4.d shows the plan fragment for our example.

2. Combine Plan Fragments into a Logical Plan f: Let $r_1, \ldots, r_m$ be the rules in program P, and $T_1, \ldots, T_m$ be the corresponding plan fragments. We add union operators to union all fragments whose rules share the same head predicate. Next, we compile the unions to form a logical plan. This compilation step "unfolds" the intentional predicates, by substituting them with their corresponding plan fragments, and unifying variables (i.e., renaming variables) if necessary. In our running example we have just one rule (Figure 4.a). So this step does not happen, and the logical plan f is the same plan mentioned earlier in Figure 4.d.

3. Compile f into a Physical Plan e: Next, we annotate the logical plan f. Since each document is read into memory only once (see below) and most IE operations operate on documents in memory, we select scan as the method to access the leaves, which are tables associated with extensional predicates or the document collection $docs(d)$, and select nested loop for joins.

Next, we select a physical operator for each p-predicate q, always selecting bulk when available, since the bulk version is often more efficient than the singleton one due to built-in optimizations. If we end up selecting a singleton version for q (because bulk is not available), then we must modify the subtree rooted at q. Specifically, let $z_1, \ldots z_k$ be the children of q in this subtree. Since q is now singleton, it cannot take as input $z_1, \ldots, z_k$, which are relations. Hence, we must first join the z_i, then apply q to the join. Figure 4.e shows how the subtree of $match(d,p,e)$ has been modified (compared to the same subtree in Figure 4.d) if $match(d,p,e)$ is singleton.

4. Modify e into h for the Per-Document Setting: Since we currently consider only IE tasks that process each document in isolation (Section 3.1), we next modify e into h to process documents *sequentially*. Plan h reads the first document d_1 from disk, applies the above physical plan e to d_1, outputs the result, reads the second document d_2, applies e to d_2, and so on.

We then improve h in several ways. First, we read the extensional tables in for the first document, then cache them in memory for subsequent ones. Second, e typically has several portions that can be executed independently of any document (e.g., any subtree not containing a document variable d). Hence, we execute these portions only once, for the first document, then cache the result for subsequent ones. In the vast majority of per-document IE tasks, the average document plus the document-dependent portion consume a relatively small amount of memory. Hence, it is reasonable to assume that the extensional tables and the document-independent portions will fit in memory. Finally, we return h as the default execution plan for the Xlog program P.

5. OPTIMIZING EXECUTION PLANS

We now present our current optimization solution. First, we describe several important characteristics of IE programs. Next, we propose three optimization methods that exploit these characteristics. Finally, we describe how we embed these methods in a cost-based optimization framework.

5.1 IE Program Characteristics

In many IE settings each document is read from disk into memory *only once* (Section 3.1), then processed by many expensive string operations (e.g., comparison, copy). As a result, in such settings CPU time often dwarfs IO time (e.g., constituting 84%-99% of total runtime in our experiments in Section 6). Consequently, we focus for now on reducing CPU time. To do so, we exploit three kinds of text-centric actions that we observe that IE programs often execute.

1. Creating Text Span Hierarchies: We observe that IE plans often start with a long text span (i.e., an entire document), then iteratively extract smaller and smaller text spans from it. For example, an IE program may extract titles and person names – smaller spans – from a given document. Another IE program may extract lines from a document, then lists (e.g., "D. Smith, M. Jones") from lines, then author names from lists (e.g., "D. Smith", "M. Jones"), and so on.

In such a text span "hierarchy", a property p_s of a "higher-level" span s often implies a property p_t for a "lower-level" span t that is related to s in some way. For example, suppose t contains s. Then "s is set in italics" implies "t must contain some italicized text." Similarly, "s contains word w" implies "t contains word w." As another example, "the length of s is 3" implies "the length of t is at least 3."

Consequently, during the IE process, if t does not have property p_t, we do not have to extract s from t, thus saving

• allCaps(s)
• italics(s)
• bold(s)
• underline(s)
• containWord(s,w)
• lengthWord(s)
• lengthLine(s)

(a)

contain(s,t), overlap(s,t), distWord(s,t,n)

(b)

• italics(s) $\wedge$ contain(s,t) $\rightarrow$ containItalics(t)
• italics(s) $\wedge$ overlaps(s,t) $\rightarrow$ containItalics(t)
• (lengthWord(s) = 3) $\wedge$ contain(s,t) $\rightarrow$ lengthWord(t) > 3
• containWord(s,w) $\wedge$ contain(s,t) $\rightarrow$ containWord(t,w)

(c)

Figure 5: Sample (a) p-functions that capture text span properties, (b) p-predicates that capture span relationships, and (c) mappings across properties.

time. We call this optimization *pushing down text properties*.

2. Pruning Span Pairs via Location Conditions: Components of real-world entities often appear in close proximity in text documents. For example, an author name typically appears close to a title. The room number of a meeting and the meeting time typically co-appear within a few sentences, and so on. Exploiting this phenomenon, IE plans often extract entity components as separate spans, then pair the spans and prune those pairs that do not satisfy certain location conditions, e.g., the spans must appear within the same sentence, or the same paragraph, or at most k words (or k lines) from each other, etc. For example, the plan in Figure 2 extracts titles and names, then keeps only those (title,name) pairs that occur within two lines of each other.

We can exploit such location conditions in optimization. For example, we can extract titles first, then extract names, *but only within two lines of titles*. If extracting names is expensive (as it often is, see Section 6), then this strategy can significantly reduce runtime, because it extracts names from a smaller amount of text. We call this optimization *scoping*.

3. Pattern Matching to Generate Text Spans: Real-world entities are often described using some patterns in text documents. Hence, pattern matching is pervasive in information extraction [6, 8, 11, 28]. It is also very expensive, and this expensive problem is seriously compounded in large-scale IE tasks, where we often must match tens of thousands of patterns against each document. For instance, in DBLife, each day we must match 48,806 name patterns, 99,462 paper titles, and 246 organizations against approximately 10,000 documents, a process that takes more than a day if executed in a straightforward fashion. The work [6] describes other large-scale pattern matching examples.

Consequently, we believe that any practical IE optimizer must optimize pattern matching. We describe one such optimization in this paper, which is called *pattern indexing* and is based on the observation that a document typically matches just a relatively small set of patterns.

We now describe the above three optimizations in detail.

5.2 Pushing Down Text Properties

In this optimization, recall that we push certain properties of higher-level spans to lower-level ones, so that we can prune lower-level spans that do not have such properties.

To do so, we first define $\mathcal{P}$, a set of p-functions that capture span properties. Examples include $allCaps(s)$ (true iff all characters in s are capitalized), $containWord(s, w)$ (true iff s contains word w), and $lengthWord(s)$ (returns the length of s in words), etc. Figure 5.a shows examples of property functions in the current Xlog implementation.

Next, we define $\mathcal{R}$, a set of p-predicates that capture relations between text spans. Examples include $contain(s, t)$ (true iff span s is contained within span t), and $overlap(s, t)$ (true iff s and t overlap), etc. Figure 5.b shows examples of relation functions and predicates in the current Xlog implementation.

Now consider a plan fragment $\sigma_{c(s)}[q(\ldots, t, \ldots)]$, where (a) q is an operator whose input includes a span t and whose output includes a span s, and (b) $\sigma_{c(s)}$ is a selection condition over s, being specified using text property functions in $\mathcal{P}$. Suppose that relationship $r(s, t)$ holds, where r is specified using predicates in $\mathcal{R}$. Suppose further that there exists a mapping $c(s) \wedge r(s, t) \rightarrow c'(t)$. Then we can push down text property $c(s)$, by inferring that t must satisfy the condition $c'(t)$.

Consequently, we can rewrite the plan fragment $\sigma_{c(s)}[q(\ldots, t, \ldots)]$ as $\sigma_{c(s)}[q(\ldots, \sigma_{c'(t)}t, \ldots)]$. This can potentially reduce the number of spans t fed into operator q, thereby reducing runtime. For example, consider plan fragment $F = \sigma_{allCaps(x)}[lines(d, x, n)]$. Given the mapping

$$allCaps(x) \wedge contain(x, y) \rightarrow containCaps(y)$$

we can rewrite F as $\sigma_{allCaps(x)}[lines([\sigma_{containCaps(d)}d], x, n)]$.

Thus, the optimization of pushing down text properties works as follows. Let $\mathcal{M}$ be a set of mappings. Given a triple $(\mathcal{P}, \mathcal{R}, \mathcal{M})$, for each plan fragment $\sigma_{c(s)}[q(\ldots, t, \ldots)]$ in the execution plan, check $\mathcal{M}$ to find an applicable mapping, then use the mapping to rewrite the fragment, as described above. Note however that just because we can rewrite a plan fragment by pushing down some text properties does not necessarily mean the new fragment will run faster. Our cost-based optimizer (see Sections 5.5-5.6) will decide if a rewriting can reduce runtime and should be carried out.

In the current Xlog implementation, we have defined a triple $(\mathcal{P}, \mathcal{R}, \mathcal{M})$ that covers all built-in functions and predicates (see examples in Figure 5). This set however is easily extensible: if a developer adds a new text property (as a p-function) or a new span relationship (as a p-predicate), he or she simply has to specify as many applicable mappings as possible. Furthermore, if a developer employs a p-predicate p in an IE program, he or she should specify span relationships between p's inputs and outputs, so that text properties can be pushed from the output of p to its input.

5.3 Scoping Extractions

In scoping, recall from Section 5.1 that if a plan imposes some location conditions on several spans, then we optimize by finding one span first, then using its location to narrow the region from which the other spans must be extracted. In what follows for ease of exposition we will consider only two-span cases, even though the optimization can be extended to work with more than two spans. We start by defining

Definition 8 (Location function and predicate). *A p-function (p-predicate) $f(\bar{s}, \bar{t})$ is a location function (predicate) if it can be used to impose a condition on the relative text locations of input spans s and t. We refer to such conditions as location conditions, and use $\mathcal{L}$ to denote the set of all location functions and predicates that have been provided to the optimizer.*

For example, a location function is $distLine(s, t)$, which measures the distance in lines between s and t. A location predicate is $samePara(s, t)$, which returns true if s and t are in the same paragraph. Next, we define

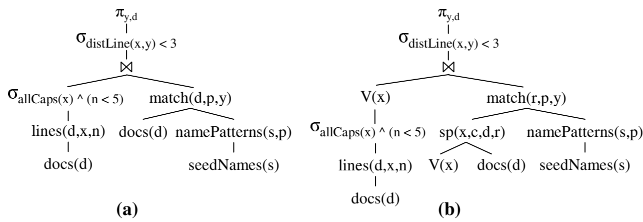


Figure 6: Scoping extractions.

Definition 9 (Scoping procedure). *The scoping procedure $sp(\overline{s}, \overline{c(s,t)}, \overline{r}, r')$ takes as input a span s, a condition $c(s,t)$ that involves only location functions and predicates in $\mathcal{L}$, and a span r. It outputs a minimal span $r' \subseteq r$, such that (a) for any span t that is contained in r, if $c(s,t)$ is true, then t is contained in r', and (b) there does not exist span r'' that is strictly contained within r' and yet satisfies condition (a).*

Now given a location condition $c(s,t)$ in an IE plan, we can either find s first, then scope t, or vice versa. Suppose we opt to find s, then scope t. Scoping t means "inserting" a scoping procedure as defined above into a location on the *extraction path* of t, which specifies how t is extracted, from document d (at a leaf of the IE plan) all the way to condition $c(s,t)$. Formally, we define

Definition 10 (Extraction path). *Let t be a span variable and $c(s,t)$ be a location condition in an IE plan e. Then the extraction path of t up to $c(s,t)$ is a sequence $(a_1, op_1, a_2, op_2, \ldots, a_n, op_n, t, c(s,t))$, such that a_1 is document d, operator op_1 extracts span a_2 from a_1, operator op_2 extracts span a_3 from a_2, and so on, until operator op_n extracts t from a_n, then feeds t into condition $c(s,t)$.*

Consider for example the IE plan in Figure 6.a. In this plan, $distLine(x,y) < 3$ is a location condition, and the extraction path of y is $(d, match(d,p,y), y, [distLine(x,y) < 3])$.

Given the extraction path $(a_1, op_1, \ldots, a_n, op_n, t, c(s,t))$ of t, we must insert the scoping procedure sp before an operator op_i. A reasonable solution is to insert sp before the *first* IE predicate in the extraction path, based on the heuristic that the sooner we can narrow the text region from which t will be extracted, the more the IE plan can save.

The above solution however may not produce correct results. For example, in Figure 6.a, consider again the path $(d, match(d,p,y), y, [distLine(x,y) < 3])$. Suppose we insert the scoping procedure sp right before $match(d,p,y)$, and suppose that in this particular case, *match* returns output *only if span d is longer than 300 characters.* If scope procedure sp extracts a span r from d, where d is longer than 300 characters but r is shorter than 300 characters, inserting sp into the plan may produce output that is different from the original plan. Consequently, to ensure sound insertion, we define

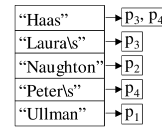
Definition 11 (Monotonic IE predicate). *Consider an IE predicate q that extracts spans $a_1, \ldots, a_n$ from span b. We say q is monotonic iff for any span b' contained in b, a span c contained in b' is extracted by $q(b')$ iff c is extracted by $q(b)$.*

We then insert the scoping procedure sp right before the first monotonic IE predicate on the extraction path of t. If such a predicate cannot be found, then we do not scope t.

p_1 = "(Jeff\s|Jeffrey\s)\s*Ullman"
p_2 = "(Jeff\s|Jeffrey\s)\s*Naughton"
p_3 = "Laura\s\s*Haas"
p_4 = "Peter\s\s*Haas"

d = "Homepage of Laura Haas"

(a)



(b)

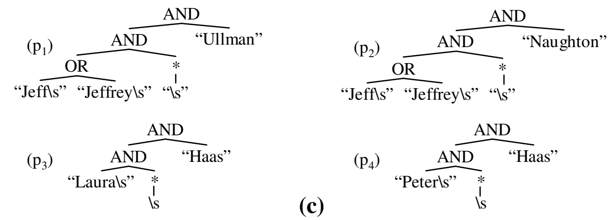


Figure 7: (a) A set of patterns $p_1 - p_4$ and a document d, (b) an inverted index over $p_1 - p_4$, and (c) parse trees for $p_1 - p_4$.

Theorem 1. *If sp is inserted before a monotonic IE procedure on the extraction path of t, as described above, then the modified IE plan is correct in that it produces the same output as the original IE plan.*

Note that if scope procedure *sp* is implemented in a singleton manner, it could potentially output many overlapping text spans. For example, suppose that we have extracted two adjacent lines x_1 and x_2 from document d. Then, invoking $sp(\overline{x}, \overline{distLine(x,y) < 3}, \overline{d}, d')$ with x_1 and x_2 as input values for x will produce two overlapping spans d'_1 and d'_2. Thus, if we insert *sp* into a plan such that another IE predicate q extracts from d' instead of d, q may process more text than if it had simply processed the original document d.

Thus, we implement the scope procedure in a bulk manner to merge overlapping output. First, given a set of spans S and R, we produce a set of output spans R' by invoking the singleton scope $sp(\overline{s}, \overline{c(s,t)}, \overline{r}, r')$ for each pair of spans $s \in S$ and $r \in R$ where r is contained in s. Then, for any two output spans r'_1 and r'_2 in R' that overlap, we replace r'_1 and r'_2 with a span $r'_{1,2}$, where $r'_{1,2}.start = min(r'_1.start, r'_2.start)$, $r'_{1,2}.end = max(r'_1.end, r'_2.end)$, and $r'_{1,2}.doc = r'_1.doc = r'_2.doc$. We repeat this process until no two spans in R' overlap.

Finally, we incorporate the bulk scope procedure in an IE plan with "virtual tables" that accumulate tuples in memory as input for bulk scope procedures. To illustrate, consider the plan in Figure 6.b. For each document d, we first evaluate the left subtree of the join, producing all tuples (d, x, n) where $[allCaps(x) \wedge (n < 5)]$. Next, the values for x are stored in memory in virtual table $V(x)$. Then, only after we have completely evaluated the left subtree, we continue evaluating the rest of the plan, where the bulk scope procedure $sp(x, c, d, r)$ now reads values for x from $V(x)$.

5.4 Optimizing Pattern Matching

We now discuss how to optimize pattern matching, as encapsulated in operator *match*. The singleton version of *match* takes a document d and a regular expression pattern p, then returns all spans in d that match p. As we have argued in Section 5.1, this version performs poorly when given a large set of patterns, because it will try to match all patterns in the set against an input document.

Hence, we design *iMatch*, a bulk version of *match* that can match a large set of patterns P against a document efficiently. The key idea underlying *iMatch* is that each docu-

	Statistics Collected			Parameters Estimated		
	num. output tuples	total runtime	total sum of span lengths	selectivity factor	time-per-unit factor	ave. output span length
p-predicate $p(R, S)$	$\mathcal{C}_p$	$\mathcal{T}_p$		$\mathcal{C}_p / (\mathcal{C}_R \cdot \mathcal{C}_S)$	$\mathcal{T}_p / (\mathcal{C}_R \cdot \mathcal{C}_S)$	
boolean p-function $f(R)$	$\mathcal{C}_f$	$\mathcal{T}_f$		$\mathcal{C}_f / \mathcal{C}_R$	$\mathcal{T}_f / \mathcal{C}_R$	
$R \bowtie S$	$\mathcal{C}_j$	$\mathcal{T}_j$		$\mathcal{C}_j / (\mathcal{C}_R \cdot \mathcal{C}_S)$	$\mathcal{T}_j / (\mathcal{C}_R \cdot \mathcal{C}_S)$	
$\pi_a(R)$	$\mathcal{C}_a$	$\mathcal{T}_a$		$\mathcal{C}_a / \mathcal{C}_R$	$\mathcal{T}_a / \mathcal{C}_R$	
condition (a=v) over R	$\mathcal{C}_c$	$\mathcal{T}_c$		$\mathcal{C}_c / \mathcal{C}_R$	$\mathcal{T}_c / \mathcal{C}_R$	
condition (a op v) over R	$\mathcal{C}_{c'}$	$\mathcal{T}_{c'}$		$\mathcal{C}_{c'} / \mathcal{C}_R$	$\mathcal{T}_{c'} / \mathcal{C}_R$	
IE-predicate $x(R, S)$ (where x extracts attribute a from attribute b)	$\mathcal{C}_x$	$\mathcal{T}_x$	$\mathcal{V}_x^a$	$\mathcal{C}_x / (\mathcal{C}_R \cdot \mathcal{C}_S)$	$\mathcal{T}_x / (l(b) \cdot \mathcal{C}_R \cdot \mathcal{C}_S)$	$\mathcal{V}_x^a / \mathcal{C}_x$

Table 1: Statistics collected to estimate parameters in the cost model.

ment d often matches only a small number of patterns in P. Hence, by reducing the number of patterns considered per document, *iMatch* can match patterns much more efficiently than the singleton *match*.

Toward this goal, *iMatch* starts by building an inverted index I over all patterns in P. Each entry of I has the form (k, U), where the key k is a string, and U is the set of all patterns in P such that k must appear in any string that matches a pattern in U. For example, Figure 7.b shows the inverted index for the four patterns $p_1 - p_4$ in Figure 7.a (note that we use "\s" to denote the space character). Here, the key "Haas" maps to patterns p_3 and p_4 because any string that matches these patterns must contain "Haas".

Once index I is built, given a document d, *iMatch* scans d to find K, the set of keys of I that appear in d. Finding K can be done in time linear in the length of d, by building a trie over the keys of I (see [4] for more details). Note that building this trie is done only once, after constructing I.

Once *iMatch* finds K, it obtains $P_d = \cup_{k \in K} lookup(k, I)$, where $lookup(k, I)$ returns the set of all patterns that I maps k to. Finally, *iMatch* searches d and returns spans that match patterns in P_d. To illustrate, consider document d in Figure 7.a. After scanning d, *iMatch* finds the keys "Laura\s" and "Haas". Looking up these keys in I yields two patterns p_3 and p_4. Hence, *iMatch* searches d only for occurrences of these two patterns, thus avoiding searching d against p_1 and p_2.

The only issue left is building index I. To do so, for each pattern $p \in P$, we construct a parse tree whose leaf nodes are the maximal strings in p that do not contain special regular-expression characters, and whose internal nodes correspond to Boolean connectives (e.g., AND and OR), or the "*" construct. Figure 7.c shows for example the parse trees of the four patterns in Figure 7.a.

Next, given the parse tree for p, we find all leaf strings S whose ancestors consist of only AND nodes. Since these strings must occur in any text span that matches p, we index p using these strings as keys in index I. If the parse tree for p has no leaves whose ancestors consist of only AND nodes (e.g. the parse tree for the pattern "A|B"), then we index p using all leaf nodes as keys.

5.5 Plan Cost Estimation

We now describe how to estimate the runtime of an execution plan. Section 5.6 describes how we enumerate and select a plan with minimal runtime.

The runtime of any plan p consists of the runtime of the internal nodes plus the runtime of the leaves. The runtime of the leaves is simply the IO time – the time to read in the documents and the extensional tables – and is the same for all plans in our settings. Hence, we only need to estimate the runtime of the internal nodes, each of which corresponds to a relational or procedural operator.

5.5.1 Statistics and Cost Model

Let q be a p-predicate with input relations $R_1, \ldots, R_n$, and output relation R_q. Then we estimate q's output size (i.e., the number of output tuples) as $|R_q| = s_q \cdot \prod_{i=1}^{n} |R_i|$, where s_q is a *selectivity factor* that we will compute for q. We also estimate q's runtime as $t_q = w_q \cdot \prod_{i=1}^{n} |R_i|$, where w_q is a *time factor*, the average time q takes to process a tuple in $R_1 \bowtie \ldots \bowtie R_n$. We estimate the output size and runtime for each p-function, and the relational operators equi-join and project in a similar fashion.

Estimating the output size and runtime for a selection is more involved, and proceeds as follows. First we compute (a) a selectivity factor s for all conditions of the form ($a = v$), where a is a variable in the plan p, (b) a selectivity factor s' for all conditions of the form (a op v) where op is $\geq, <, \leq$, or $>$, and (c) a time-per-unit factor t that estimates the average time it takes to process such a condition. We then compute s, s', and t for each p-function that can appear in a selection condition, in a similar manner.

Now consider a selection $f = \sigma_{c_1 \wedge \ldots \wedge c_n}$. Here each condition c_i has the form (a op v) where a is a relational attribute (e.g., n) or a procedural operation (e.g., $allCaps(x)$), op is $=, \geq, <, \leq$, or $>$, and v is a scalar or Boolean value. Examples of c_i include ($n < 5$) and ($allCaps(x) = true$). Let U and V be the input and output relations of f, respectively. Then, by making the simplifying assumption that the conditions c_i are independent, we estimate the size of V to be $|V| = \prod_{i=1}^{n} s_i \cdot |U|$, where s_i is the selectivity factor of condition c_i, estimated as above. Finally, we estimate the runtime of f to be $t(f) = |U| \cdot \sum_{i=1}^{n} t(c_i)$, where $t(c_i)$ is the time-per-unit factor of c_i.

Let q be a procedural operator. So far we have estimated the runtime and output size of q given its *singleton* version. Since the *bulk* version of q may behave very differently from the singleton version, if a user or developer supplies such a bulk version, he or she can optionally define a cost model specific for that version (as we do below with the bulk version of the *match* operator). In the absence of any such cost model, we employ the cost model of the singleton version to estimate the cost of a bulk version.

Cost Model of IE Predicates: If q is an IE predicate, then we know that q extracts span(s) a from some input span b. In practice, the runtime of q often correlates strongly with the length of b. Hence, instead of using the above cost model for generic procedural operators, we can use a more accurate cost model for q.

In this cost model, we compute w_{unit}, the average time that q spends processing one unit of the input span b. We then estimate the runtime of q as $t(q) = w_{unit} \cdot l(b) \cdot \prod_{i=1}^{n} |R_i|$, where the R_i are the input relations of q, as before, and $l(b)$ is the average length of span b.

Let R_q be the output relation of q. We next estimate the size of R_q as before. However, we now must also estimate $l(a)$, the average length of extracted span a, so that we can use it to estimate the runtime of any other IE predicate that extracts from a.

Modeling the Cost of *match*: Let P and D be the set of input patterns and documents into *match*, respectively.

In the default cost model the runtime of *match* is estimated as $t(match) = w_{match} \cdot |P| \cdot |D|$, and the output size as $|R_{match}| = s_{match} \cdot |P| \cdot |D|$. These estimations however do not work well for the bulk version of *match* (Section 5.4). There, using an inverted index, *match* checks a document for a relatively small, on average constant, number of patterns. Hence, its runtime per document stays relatively constant, and so does its output size. For these reasons, we model the runtime of *match* as $t(match) = w_{match} \cdot |D|$, and the output size as $|R_{match}| = s_{match} \cdot |D|$.

In summary, Table 1 shows the statistics (selectivity, time-per-unit, average output span length) that we must estimate in order to estimate the runtime of the operators and hence the runtime of a plan. We now discuss how to estimate these statistics.

5.5.2 Estimating Statistics

We start by randomly selecting a set of k documents from the data set (to which later we want to apply the IE program). Next, we execute an IE execution plan p over this set to compute for each procedural or relational operator q the values $(\mathcal{C}_q, \mathcal{T}_q, \mathcal{V}_q^a)$, where $\mathcal{C}_q$ is the number of output tuples, $\mathcal{T}_q$ is the total processing time (excluding time spent in the descendant nodes), and $\mathcal{V}_q^a$ is the total sum of the lengths of each extracted text span attribute a (if q is an IE predicate).

We then compute the above statistics using these quantities, as shown in Table 1. For example, consider an IE predicate x that takes as input two relations R and S, and suppose that x extracts span a from span b. Then, we record $\mathcal{C}_x$, $\mathcal{T}_x$, and $\mathcal{V}_x^a$, the number of output tuples of x, total runtime of x, and the total sum of the span lengths of a, respectively. We then estimate the selectivity factor of x as $\mathcal{C}_x/(\mathcal{C}_R \cdot \mathcal{C}_S)$, and the time-per-unit factor of x as $\mathcal{T}_x/(l(b) \cdot \mathcal{C}_R \cdot \mathcal{C}_S)$, where $l(b)$ is the average length of b. Also, we estimate the average length of a as $\mathcal{V}_x^a/\mathcal{C}_x$.

Finally, if we do not know the average document length over all documents in the $docs(d)$ relation (e.g. from preprocessing the data), then we estimate this value as the average length of the k randomly selected documents.

5.6 Plan Enumeration

As the final piece in the optimization puzzle, we describe how to find a plan with minimal cost. The three optimization techniques described earlier give us a set of rewrite rules. A possible search strategy is to start with a default plan $p_{default}$, and exhaustively search the plan space S by repeatedly applying the rewrite rules. Then, we select the plan with the lowest estimated runtime from S.

However, the plan space S is huge (exponential wrt the number of nodes in the plan). Thus, we reduce the plan space size as follows. First, we assume that we use the *iMatch* algorithm for all *match* operators in the plan. Then, we search the remaining plan space in a phased approach. While this phased approach does not search the space exhaustively, in Section 6 we show empirically that this search strategy can still significantly improve performance.

In the first phase, we search the space of possible sets of scope operators that can be inserted into $p_{default}$. Specifically, for each condition $c(\overline{s}, \overline{t})$ in $p_{default}$ where s and t are extracted from r and $c(\overline{s}, \overline{t})$ involves only location functions and predicates, we can: (1) insert $sp(\overline{s}, \overline{c(s,t)}, \overline{r}, r')$, (2) insert $sp(\overline{t}, \overline{c(s,t)}, \overline{r}, r')$, or (3) insert neither scope operator. This results in $O(3^n)$ candidate sets of scope operators, where n is the number of location conditions in $p_{default}$. Then, we test each candidate set T by inserting the scope operators in T into $p_{default}$, estimating the cost of the resulting plan, and selecting the plan p' with the lowest cost. To avoid recomputing costs, we employ the dynamic optimization technique of memoizing the cost of subplans [17].

Then, in the second phase we search the space of text properties that can be pushed down. To do this, we repeatedly push down text properties in plan p' until no more properties can be pushed down. Then, for each new selection $\sigma_{c'}$ introduced by pushing down text properties, we remove $\sigma_{c'}$ from p' if doing so lowers the estimated cost of p'.

Data Set	Number of Documents	Size
Homepages	294	3.2 MB
DBWorld	90	5.5 KB
Conferences	142	2.5 MB

IE Programs	Description
affiliation	Find (X,Y) where person X is affiliated with organization Y.
confTopic	Find (X,Y) where topic X is discussed at conference Y.
confDate	Find (X,Y) where conference X is held during date Y.
advise	Find (X,Y) where person X is advising person Y.
chair	Find (X,Y, Z) where person X is a chair of type Y at conference Z.

Table 2: Data sets and IE programs.

6. EMPIRICAL EVALUATION

We now empirically show that optimization is vital for IE programs, and that we can effectively optimize IE programs written in our declarative framework.

Data Sets & IE Programs: Figure 2 describes the three data sets and five IE programs used in our experiment. The data sets Homepages, DBWorld, and Conferences are taken from DBLife [11], and consist of researcher homepages, DBWorld messages, and conference homepages, respectively. The five IE programs extract various academic relationships (e.g., chairing a conference, working for a university, etc.) and are converted from similar programs in DBLife. Additionally, we experimented with a large IE program running on a 116 MB one-day snapshot of DBLife data, and will report this experiment below.

The Need For Optimization: We first examine the need for optimizing extraction programs. Given the five IE programs and three data sets described earlier, Figure 8 shows all fifteen scenarios of possible execution. In each scenario we executed a program P on a data set D, and the four bars show (from left to right) the runtime of unoptimized P (i.e., the default execution plan in Section 4), P optimized on Conferences, P optimized on DBWorld, and P optimized on Homepages, respectively. A number on top of a bar indicates the runtime (which exceeds the figure). P optimized on Conferences for example means that we first run P over a random sample of 40 documents from Conferences to collect statistics, then optimize P using those statistics.

The results show that in all cases, optimized plans significantly outperform the unoptimized ones, cutting runtime by 52-99%, thus underscoring the crucial importance of optimization for IE programs. In addition, out of the 15 cases, optimizing on the same data set (e.g., optimized then run both on D) runs significantly faster than optimizing on a different data set in 8 cases, by 3-78%, runs comparably in 6 cases, and runs worse in only 1 case (see executing affiliation on Conferences). This result suggests that it is important to optimize specifically for a data set before running an IE program on that data set.

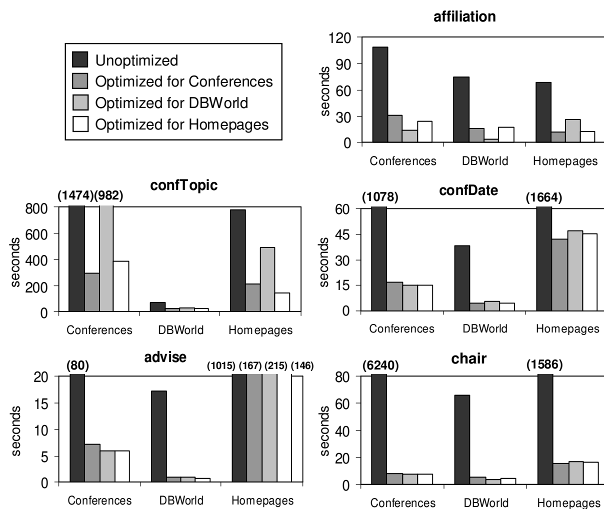


Figure 8: Runtime of programs optimized for different data sets.

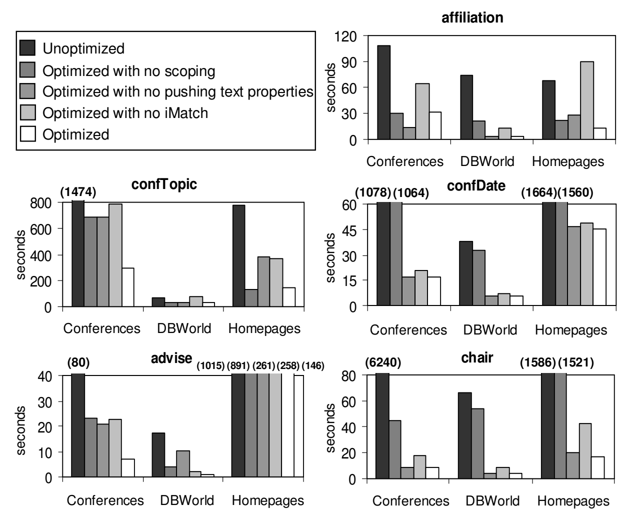


Figure 9: The runtime of programs optimized without one of the three optimizations.

Component Contributions: Next, we analyze the relative contributions of the three optimizations discussed in this paper: pushing down text properties, scoping, and pattern indexing with *iMatch*. To do so, for each program Q and each data set D, we produced three optimized plans P'_1, P'_2, and P'_3, where each optimized plan was generated without one of the three optimizations (e.g. optimize P without considering pushing down text properties).

Figure 9 shows the runtime of these plans, as well as those of the unoptimized default plan and the fully optimized plan (i.e., with all three optimizations). The results show that in 35 out of 45 cases, removing one of the optimizations produces a plan that is inferior to the fully optimized plan. Overall, the results suggest that all three optimizations make important contributions to reducing runtime, and no optimization appears to dominate the others.

Zooming further into the results, we found that a very large amount of runtime of unoptimized plans, namely 84-99%, was CPU time, and most of this time was spent in pattern matching. This suggests that reducing CPU time, and in particular pattern matching time, is a promising direction for optimizing IE programs.

However, it is interesting to note that we can significantly reduce pattern matching time not just by optimizing the pattern matching operator, as expected, but also simply by reducing the amount of text that the matching operator must process. For example, consider executing chair over the Conferences data set, where pattern matching takes 99.7% of the execution time in the unoptimized plan. In Figure 9 we see that even without optimizing pattern matching (by using *iMatch*), we still managed to reduce runtime by 99%, simply by pushing down text properties and scoping the documents before matching patterns over them.

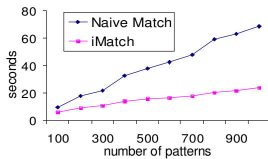
In the next step, we want to know how well our optimized pattern matching operator works. To do so, we compared *iMatch* with the naive match algorithm and showed the results in Figure 10.a. We evaluated both algorithms over Homepages, by varying the number of person name patterns that must be matched. The results show that, while the runtime of both algorithms grows linearly with respect to the number of patterns, the runtime of *iMatch* grows much more slowly, suggesting that *iMatch* can effectively match a large number of patterns.

Finally, we examine the importance of *text-based* parameters in our cost model. To do this, for each program P and each data set D, we produced an optimized plan P' using a *modified* cost model that did not differentiate IE predicates (i.e. the modified cost model ignored the length of input spans for IE predicates, see Section 5.5.1). In Figure 10.b, we see that in 12 out of 15 cases, modeling text span length results in a superior or comparable plan, suggesting that it is indeed critical to model text properties when optimizing IE programs.

Sensitivity Analysis: Figure 11 shows the runtime of optimized IE plans for affiliation, confTopic, and confDate, as we vary the number of documents over which we collect statistics (before optimizing). The results show that in all cases, we improve on the unoptimized plan by performing optimization with default parameter values (without collecting any statistics). However, in 7 out of 9 cases, collecting statistics over as few as 10 documents already helps us find a significantly better plan, further reducing runtime by 42%-98%. In all cases, collecting statistics over more than 10 documents does not appear to further improve runtime. This suggests that while collecting statistics is crucial for optimization, we can do so with a relatively small number of documents, thus incurring little overhead.

Declarative Extraction in DBLife: Finally, we tested the practicality of our framework by converting several extraction programs in DBLife into Xlog. These programs extract person and conference mentions and infer a variety of relationships between them (e.g. person X is giving a talk at conference Y). In their original imperative form, these programs took 7+ hours to run on a single day's snapshot of DBLife's crawled data (9572 web pages, 116 MB). In 2005, after 3 days of manually changing the code and evaluating different optimizations (by a team of 2 graduate students), we were able to *manually* optimize these programs and reduce their total runtime from 7 hours to 24 minutes.

Given the current declarative framework, we recently spent 3+ hours converting these programs into one Xlog program $\mathcal{Q}$. $\mathcal{Q}$ has 16 Xlog rules consisting of 47 predicates, out of which 18 are p-functions and 5 are p-predicates. We then optimized $\mathcal{Q}$ (using the current Xlog optimizer) and obtained an execution plan that took 61 minutes on the same set of



(a)

Runtime (seconds)	affiliation			confTopic			confDate			advise			chair		
Data set	C	D	H	C	D	H	C	D	H	C	D	H	C	D	H
Unoptimized	109	73.9	68.3	1474.4	68.7	780.4	1078.1	37.9	1663.7	79.6	17.1	1015.4	6239.5	66.2	1586.3
Optimized without modeling text span length	**10.0**	**3.7**	22.0	**240.9**	**33.2**	**143.0**	1056.0	33.2	1557.5	9.6	1.0	176.4	**8.1**	4.5	19.3
Optimized	30.7	**3.7**	**12.6**	297.7	**33.2**	147.0	**16.8**	**5.6**	**45.5**	**7.1**	**0.9**	**146.0**	**8.1**	**3.9**	**16.1**

(b)

Figure 10: (a) Comparison of Naive Match and *iMatch* (b) Runtime of programs optimized without modeling text span length. Data sets are abbreviated by their first letter.

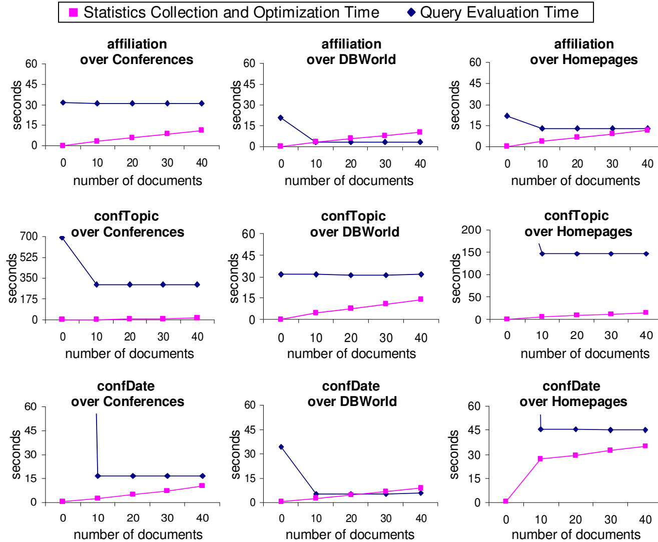


Figure 11: Runtime of programs optimized after collecting statistics over varying numbers of documents.

data, which we found acceptable in the current DBLife context. Thus, while anecdotal, this result does suggest that automatic optimization with declarative IE can drastically speed up development time, by eliminating tedious and labor intensive manual optimization.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a declarative framework for writing IE programs using Datalog with embedded extraction predicates. In addition to being easier to write and understand, we can also apply query optimization techniques to IE programs written in Datalog, a feature that is vital for any large-scale IE task. We provided initial solutions to optimizing IE programs, and described extensive experiments over real-world data to show the effectiveness of our approach. As future work, we plan to extensively evaluate the current framework, consider richer data models and query languages, develop new optimization techniques, and extend the current work to handle recursion and negation, which commonly occur in extraction scenarios. We would also like to explore implementation strategies for Xlog-like languages, e.g., on top of the Coral [21] deductive system. While such an implementation brings many potential benefits, it also raises non-trivial challenges, as the current optimization framework of Coral (and other open-source deductive systems) is not cost-based.

Acknowledgments: This work is partially supported by grants CAREER IIS-0347903 and ITR-0428168.

8. REFERENCES

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *PODS-05*.

[2] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] E. Agichtein and S. Sarawagi. Scalable information extraction and integration (tutorial). In *KDD-06*.

[4] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[5] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system LDL++. *TPLP*, 3(1):61–94, 2003.

[6] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE-06*.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 24(2):177–228, 1999.

[8] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *ICDE-02*.

[9] W. Cohen and A. McCallum. Information extraction from the World Wide Web (tutorial). In *KDD-03*.

[10] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *ACL-02*.

[11] 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*.

[12] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction: state of the art and research directions (tutorial). In *SIGMOD-06*.

[13] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD-05*.

[14] 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.

[15] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *TKDE*, 4(6):509–516, 1992.

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

[17] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE-93*.

[18] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl?: Towards a query optimizer for text-centric tasks. In *SIGMOD-06*.

[19] A. Laender, B. Ribeiro-Neto, A. Silva, and J. Teixeira. A brief survey of web data extraction tools. *SIGMOD Record*, 31(2), 2002.

[20] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: Language, execution and optimization. In *SIGMOD-06*.

[21] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The Coral deductive system. *VLDB Journal*, 3(2):161–210, 1994.

[22] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. Technical report, IBM Almaden, 2007.

[23] K. A. Ross. Selection conditions in main memory. *TODS*, 29:132–161, 2004.

[24] K. Vieira, A. S. da Silva, N. Pinto, E. S. de Moura, J. M. B. Cavalcanti, and J. Freire. A fast and robust method for web page template detection and removal. In *CIKM-06*.

[25] J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *WWW-03*.

[26] J. Whaley and M. S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *APLAS-05*.

[27] D. Williams and A. Poulovassilis. Combining information extraction and data integration in the ESTEST system. In *ICSOFT-06*.

[28] R. Y. Zhang, L. V. S. Lakshmanan, and R. H. Zamar. Extracting relational data from HTML repositories. *SIGKDD Explor. Newsl.*, 6(2):5–13, 2004.